Otto-von-Guericke Universität Magdeburg

Masterarbeit

Konzeptioneller Ordnungsrahmen für die domänengetriebene Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils

—

Demonstration am Beispiel einer Werbezeitvermarktungsplattform

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme
Arbeitsgruppe Wirtschaftsinformatik

Vorgelegt von:	Philipp Ernst
Erster Gutachter:	Prof. Dr. rer. pol. habil. Hans-Knud Arndt
Zweiter Gutachter:	Dr. Veit Köppen
Betreuer (Unternehmen):	Dipl.-Ing. Florian Stendel
Abgabetermin:	13. Februar 2019

Selbstständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Masterarbeit selbständig, ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Ort, Datum	Unterschrift: Philipp Ernst

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
ADM	Architecture-Driven Modernization
CIM	Computation Independent Model
DDD	Domain-Driven Design
DSRM	Design Science Research Methodology
IEEE	Institute of Electrical and Electronics Engineers
MVP	Minimum Viable Product
NIST	National Institute of Standards and Technology
OMG	Object Management Group
PIM	Platform Independent Model
POC	Proof-of-Concept
PSM	Platform Specific Model
SOA	Serviceorientierte Architektur

1. Einleitung

1.1 Motivation

> *„Good architecture makes the system easy to understand, easy to develop, easy to maintain, and easy to deploy. The ultimate goal is to minimize the lifetime cost of the system and to maximize programmer productivity.“*[1]
>
> *— Robert C. Martin, Clean Architecture*

Dieses Zitat verdeutlicht, welchen Einfluss der Bereich der Softwarearchitektur auf die Entwicklung einer Softwareanwendung und deren Lebenszyklus hat. Eine Softwarearchitektur kann als „gut“ bezeichnet werden, wenn ein System durch sie leicht verstanden, entwickelt, gewartet und bereitgestellt werden kann. Sie hat damit positive Auswirkungen auf die Produktivität der Entwicklungsteams. Insbesondere im Umfeld komplexer *monolithischer Systeme* können diese Vorteile jedoch oftmals nur bedingt realisiert werden. Zu den Hauptgründen zählt die Struktur dieser Monolithen in Verbindung mit der hohen Komplexität der zugrunde liegenden Problemdomäne. Langfristig kann die Situation durch die klare Abgrenzung und Erhöhung der Sichtbarkeit wesentlicher Programmteile verbessert werden. Dazu kann eine monolithische Anwendung in kleinere, besser handhabbare Teilprobleme aufgeteilt werden. Microservices stellen einen gegenwärtig vielversprechenden Ansatz zur Erreichung dieses Ziels dar.

Der *Microservice-Architekturstil* ist nach einer aktuellen Studie der Dimensional Research Inc. mit einem stark wachsenden, industriellen Interesse verbunden. Die Ergebnisse zeigen, dass 62% der 353 befragten Softwareexperten die Einführung des Microservice-Architekturstils planen und 29% Microservices bereits produktiv einsetzen.[2] Neben der Realisierung der positiven Eigenschaften guter Softwarearchitekturen, können Microservice-Architekturen maßgeblich zur Umsetzung von *DevOps*-Unternehmenskulturen, von *automatisierten Bereitstellungsverfahren* sowie zur Skalierung *agiler Softwareentwicklungsprozesse* beitragen. Viele Unternehmen haben diese Vorteile erkannt und versuchen sich durch die Nutzung des Microservice-Architekturstils weiterzuentwickeln und einen Wettbewerbsvorteil zu verschaffen. Sie sehen sich gegenwärtig und vor allem in der Zukunft mit der Herausforderung konfrontiert, ihre monolithischen Altsysteme durch den Einsatz des Microservice-Architekturstils zu transformieren und modernisieren.[3]

Doch wie können diese Unternehmen ihre monolithischen Systeme auf Basis des Microservice-Architekturstils transformieren, sodass sie von den positiven Eigenschaften guter Softwarearchitekturen profitieren können?

---

1 Martin 2017, Kap. 15 Abschn. 1.
2 Vgl. Global Microservices Trends 2018, S. 4.
3 Vgl. Lilienthal 2017, Vorwort.

1.2 Problemstellung und Zielsetzung

Der Themenbereich der *Migration* monolithischer Anwendungen auf Basis des Microservice-Architekturstils gehört gegenwärtig zu den bedeutendsten Forschungstrends im Kontext von Microservice-Architekturen.[4] Die Anpassung der Architektur komplexer monolithischer Anwendungen ist jedoch ein kompliziertes und aufwändiges Vorhaben, das fachliches und technisches Expertenwissen sowie eine gründliche Analyse und Planung voraussetzt. Aus Sicht der Softwarearchitektur stellt die *fachliche Dekomposition* eines Monolithen dabei die größte Herausforderung dar.[5] Dies ist häufig auf die enge Kopplung der Bestandteile des Monolithen und die Schwierigkeit der Identifizierung von abgrenzbaren Komponenten bzw. Services zurückzuführen.[6] Da Microservices die zugehörige Problemdomäne möglichst gut abbilden sollten, ist der fachliche Schnitt von großer Bedeutung für den nachhaltigen Erfolg der Dekomposition. Dies kann durch ein *domänenorientiertes Vorgehen* unterstützt werden. Eine Vielzahl der themenrelevanten Literatur bringt den Aufbau und die Strukturierung von Microservice-basierten Anwendungen daher mit den Mustern und Prinzipien des *Domain-Driven Designs (DDD)* in Verbindung. Die Mehrheit der Empfehlungen geht jedoch nicht über den grundlegenden Rat zur Nutzung von Domain-Driven Design und das damit in Zusammenhang stehende Konzept der Kontextgrenzen *(Bounded Contexts)* hinaus.[7] Zudem ist das domänengetriebene Vorgehen in der Theorie von DDD weder in Form eines expliziten Prozesses definiert, noch tiefgreifend in Verbindung mit dem Microservice-Architekturstil beschrieben.

Es existiert folglich ein Mangel an Hilfsmitteln und konkreten Handlungsanweisungen, die die domänengetriebene Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils effektiv unterstützen können. Um eine zukunftssichere Aufteilung der fachlichen Architektur gewährleisten zu können, ist eine systematische Vorgehensweise notwendig, die zu nachhaltigen Ergebnissen führt. Es bedarf sowohl der Orientierung an einem übergreifenden Prozessrahmen für die Transformation als auch einem konkreten Vorgehen für den domänengetriebenen Entwurf der fachlichen Architektur. Aufgrund der ausgeprägten Praxisorientierung des Themas sind die Erfahrungen und das Wissen über relevante Prinzipien, Praktiken und Strategien, die im Rahmen von Migrationsvorhaben von Bedeutung sind, breit in der heterogenen Literatur verteilt. Eine prozessorientierte Vorgehensweise kann daher zusätzlich von einer gezielten Aggregation dieser unterstützenden Informationen profitieren.

Die vorliegende Arbeit hat das Ziel einen konzeptionellen Ordnungsrahmen zu erarbeiten, der als Grundgerüst und Orientierungshilfe für die domänengetriebene Transformation der fachlichen Architektur monolithischer Anwendungen dienen kann. Der Ordnungsrahmen soll dabei auf wissenschaftlichen Erkenntnissen und Erfahrungswerten aus der industriellen Praxis aufbauen. Das grundlegende Ziel des Ordnungsrahmens ist das Vorgehen und die Komplexität von Migrationsvorhaben zu strukturieren und handhabbar zu machen. Zur Erfüllung dieser übergeordneten Aufgabenstellung und in Bezug auf die

---

[4] Vgl. Pahl und Jamshidi 2016, S. 145; Di Francesco et al. 2017, S. 25.
[5] Vgl. Fritzsch et al. 2019, S. 139.
[6] Vgl. Di Francesco et al. 2018, S. 36.
[7] Vgl. Zimmermann 2017, 302, 306.

identifizierte Problemsituation soll eine systematische Erarbeitung von Hilfsmitteln erfolgen, die den angestrebten Ordnungsrahmen konstituieren. Die vorliegende Arbeit soll dahingehend die folgende primäre Fragestellung beantworten:

> *Wie kann das Vorgehen und die Komplexität von domänengetriebenen, Microservice-basierten Migrationen monolithischer Anwendungen strukturiert und handhabbar gemacht werden?*

Zur weiteren Konkretisierung sollen die folgenden sekundären Fragestellungen beantwortet werden:

(1) Welche grundlegenden Vorbedingungen und Anforderungen sind mit dem Einsatz des Microservice-Architekturstils verbunden?
(2) Wie kann die domänengetriebene Transformation der fachlichen Architektur in einem Prozess abgebildet werden?
(3) Wie können die Muster und Prinzipien des Domain-Driven Designs zum domänengetriebenen Entwurf der fachlichen Architektur angewendet werden?
(4) Welche Prinzipien, Praktiken und Strategien können die Transformation der fachlichen Architektur unterstützen?

Die Ergebnisse sollen im Kontext von Microservice-Architekturen folglich einen Gesamtblick auf den Migrationsprozess ermöglichen als auch einen detaillierten Blick auf den Bereich des domänengetriebenen Entwurfs der fachlichen Architektur vermitteln. Die wesentlichen Aspekte der Architektur-Transformation, fachlichen Dekomposition und Service-Identifikation sollen dabei in Form von Prozessmodellen strukturiert und verallgemeinert werden. Zur Unterstützung dieser prozessorientierten Sichten soll darüber hinaus eine Zuordnung und Beschreibung von Prinzipien, Praktiken und Strategien erfolgen, die im Rahmen der transformativen Tätigkeiten relevant sind. Weiterhin sollen die grundlegenden Vorbedingungen und Anforderungen, die mit dem Einsatz des Microservice-Architekturstils verbunden sind, untersucht werden.
Auf Grundlage der theoretischen Ausarbeitung des konzeptionellen Ordnungsrahmens sollen die Ergebnisse zudem am praxisnahen Beispiel einer monolithischen Anwendung zur Vermarktung von Werbezeiten demonstriert und evaluiert werden.

1.3 Thematische Abgrenzung

Sämtliche Ausführungen der Arbeit orientieren sich an dem Leitgedanken „*Fachlichkeit vor Technik*“[8]. Im Mittelpunkt stehen die Aspekte der *fachlichen Architektur* monolithischer Anwendungssysteme. Es werden somit primär die von einer Software umgesetzten fachlichen Funktionen betrachtet. Die Arbeit geht nicht auf technische Aspekte der Migration monolithischer Systeme und der Realisierung von Microservice-Architekturen ein. Demnach werden keine Technologien und Konzepte zur Umsetzung der technischen Architektur von Microservice-basierten Anwendungen betrachtet. Da der Schwerpunkt auf dem fachlichen Schnitt monolithischer Systeme liegt, werden vorwiegend die Aspekte der *Dekomposition* (Aufteilung) und nicht der *Komposition* (Zusammensetzung) betrachtet.

[8] Lilienthal 2017, Kap. 7.2 Abs. 4.

Weiterhin konzentriert sich die Arbeit auf das *strategische Design* als Teilbereich des Domain-Driven Designs, da dieser für den fachlichen Schnitt und die Service-Identifikation von Bedeutung ist. Die taktischen Muster des Domain-Driven Designs, die zum Aufbau von DDD-spezifischen Domänenmodellen und zur Umsetzung einzelner Microservices verwendet werden können, finden nur grundlegende Berücksichtigung. Die Ausführungen der Arbeit nehmen folglich Abstand von der Implementierungsebene. Da auch die Daten einer Anwendung zur Implementierungsebene gehören, werden diese nicht in die Untersuchungen einbezogen. Zusammengefasst bezieht sich die Arbeit auf die Makroarchitektur und nicht die Mikroarchitektur von Softwaresystemen. Sie geht damit ausschließlich auf die übergreifende Architektur auf Ebene des Gesamtsystems ein.

Microservice-Architekturen sind mit den *serviceorientierten Architekturen (SOA)* verwandt, da beide Architekturstile einem servicebasierten Ansatz folgen. Im Detail haben sie jedoch nur wenige Gemeinsamkeiten, denn „*konzeptionell, architektonisch und organisatorisch haben die beiden Ansätze sehr unterschiedliche Auswirkungen*“[9]. Obwohl die Architekturstile wesentliche Unterschiede aufzeigen, können Microservices als nächste Evolutionsstufe serviceorientierter Architekturen bezeichnet werden.[10] In der vorliegenden Arbeit ist eine weiterführende Abgrenzung beider Architekturstile nicht vorgesehen.

1.4 Aufbau der Arbeit

Der grundsätzliche Aufbau der Arbeit orientiert sich an dem Vorgehen der Design Science Research Methodology (vgl. Kapitel 1.5).

Im Grundlagenteil (Kapitel 2) wird ein Überblick über die bedeutenden Themenbereiche in Bezug auf den Microservice-Architekturstil gegeben. Neben einführenden Informationen zu Softwarearchitekturen (Kapitel 2.1) werden die themenrelevanten Paradigmen und Architekturstile beschrieben und abgegrenzt (Kapitel 2.2). Weiterhin wird auf die bedeutenden organisatorischen und kulturellen Aspekte im Kontext von Microservice-Architekturen eingegangen (Kapitel 2.3). Ein Schwerpunkt der Arbeit ist der domänengetriebene Entwurf der fachlichen Architektur. Die Theorie des Domain-Driven Designs wird daher gesondert betrachtet (Kapitel 2.4). Abschließend werden die wesentlichen Inhalte zusammengefasst und grundlegende Zusammenhänge sowie Schlussfolgerungen herausgestellt (Kapitel 2.5).

Im Hauptteil werden zunächst die Ergebnisse des Literaturrecherche- und Analyseprozesses dargelegt (Kapitel 3). Auf dieser Basis erfolgt dann die Synthese des konzeptionellen Ordnungsrahmens (Kapitel 4). Jedes Unterkapitel (Kapitel 4.1 bis 4.5) kennzeichnet dabei einen spezifischen und konzeptionellen Teilbereich des Ordnungsrahmens. Die Teile des Ordnungsrahmens, die sich für eine szenario- bzw. fallstudienartige Untersuchung im Rahmen der Arbeit eignen, werden daraufhin demonstriert (Kapitel 5). Schließlich werden die Ergebnisse und Erkenntnisse zusammenfassend diskutiert und im Kontext bisheriger wissenschaftlicher Untersuchungen eingeordnet (Kapitel 6).

---

[9] Wolff 2016, Kap. 7.3 Abs. 3.
[10] Vgl. Salah et al. 2016, S. 318.

1.5 Methodisches Vorgehen

Design Science Research Methodology

Die vorliegende Arbeit orientiert sich an der *Design Science Research* Forschungsmethodik nach Hevner et al., die als Problemlösungsparadigma im Umfeld von Informationssystemen beschrieben werden kann.[11] Das konkrete Vorgehen wird dabei durch das Prozessmodell der *Design Science Research Methodology (DSRM)* bestimmt. Nach Peffers et al. setzt sich der Prozess aus sechs Phasen zusammen: (1) Problemidentifizierung, (2) Zieldefinition, (3) Entwurf, (4) Demonstration, (5) Evaluierung und (6) Kommunikation.[12]
Der Prozess nach DSRM ist zyklisch und iterativ ausgeprägt (vgl. Abbildung A-1; Anhang). Die Erkenntnisse aus der Evaluierungsphase haben z. B. rückwirkenden Einfluss auf den Entwurf des Artefaktes. Diese Wechselwirkung spiegelt auch den Prozess der Ergebnisfindung dieser Arbeit wieder. Weiterhin können die Prozessphasen (1) bis (4) als Einstiegspunkte für Forschungsvorhaben dienen. Der Einstiegspunkt für diese Arbeit ist problemorientiert, da die identifizierte Problemstellung der primäre Auslöser für die Durchführung der Forschungstätigkeiten ist. Die Anwendung von Design Science Research soll zur Lösung dieses Problems beitragen, indem ein Artefakt in Form eines konzeptionellen Ordnungsrahmens entwickelt und evaluiert wird.

Die Aktivitäten, die im Rahmen dieser Arbeit zur Lösungsfindung durchgeführt werden, können den DSRM-Prozessphasen (3) bis (5) wie folgt zugeordnet werden:

(3) *Entwurf*: Herleitung des Artefakts durch systematische Analyse der vorhandenen Wissensbasis
(4) *Demonstration*: Veranschaulichung des Artefakts durch Demonstration im Kontext einer realen Problemsituation
(5) *Evaluierung*: Bewertung der Eignung des Artefakts zur Lösung des Problems unter Berücksichtigung der definierten Zielsetzungen

Weiterhin definieren Hevner et al. sieben Richtlinien für die korrekte Anwendung von Design-Science Research (vgl. Tabelle A-1; Anhang).[13] Die vorliegende Arbeit versucht die Richtlinien (I) bis (VI) wie folgt zu adressieren:

I. *Design als Artefakt*: Das entstehende Artefakt des konzeptionellen Ordnungsrahmens besteht aus Teilartefakten, die nach der Definition von Hevner et al. als Modelle und Methoden klassifiziert werden können.[14]

---

[11] Vgl. Hevner et al. 2004, S. 75ff.
[12] Vgl. Peffers et al. 2007, S. 52ff.
[13] Vgl. Hevner et al. 2004, S. 82ff.
[14] Vgl. ebd., S. 77.

II. *Problemrelevanz*: Die Arbeit ist von Bedeutung für Organisationen, die ihre monolithischen Anwendungen auf Basis des Microservice-Architekturstils transformieren wollen und dabei ein domänengetriebenes Vorgehen anstreben.
III. *Design Evaluierung*: Die Evaluierung des Ordnungsrahmens erfolgt deskriptiv durch die Mittel der informierten Argumentation und der szenarioartigen Untersuchung. Zudem basiert sie auf Beobachtungen, die im Rahmen der fallstudienartigen Anwendung an einem realen System gesammelt werden. Diese Evaluationsmethoden eignen sich nach Hevner et al. zur Evaluierung von Artefakten.[15]
IV. *Forschungsbeitrag*: Die Ergebnisse der Arbeit bauen auf der vorhandenen Forschung auf. Neue Erkenntnisse und Zusammenhänge werden im Kontext der bestehenden Forschungsarbeiten verortet.
V. *Forschungsgenauigkeit*: Der Ordnungsrahmen basiert auf fundiertem Wissen, das durch Anwendung einer systematischen Literaturanalyse gewonnen werden konnte. Die Ergebnisse werden praxisbezogen demonstriert und evaluiert.
VI. *Design als Suchprozess*: Die Synthese des Ordnungsrahmens basiert auf der Analyse von wissenschaftlichen Erkenntnissen und Erfahrungswerten aus der industriellen Praxis. Es erfolgt eine iterative Anpassung der Teilartefakte auf Grundlage eigener Erkenntnisse bei der Demonstration.

Die DSRM-Prozessphasen haben grundlegenden Einfluss auf die Struktur dieser Arbeit. Nachfolgend wird eine Zuordnung der Kapitel dieser Arbeit zu den beschriebenen Prozessphasen und Richtlinien vorgenommen.

Nr.	Kapitel	Prozessphasen	Richtlinien
1	*Einleitung*	(1) Problemidentifizierung (2) Zieldefinition	(II) Problemrelevanz
2	*Grundlagen*	(3) Entwurf (vorbereitend)	(VI) Design als Suchprozess
3	*Literaturanalyse*		
4	*Synthese des Ordnungsrahmens*	(3) Entwurf	(I) Design als Artefakt (V) Forschungsgenauigkeit (VI) Design als Suchprozess
5	*Demonstration des Ordnungsrahmens*	(4) Demonstration	(III) Design Evaluierung (V) Forschungsgenauigkeit
6	*Evaluierung*	(5) Evaluierung	(IV) Forschungsbeitrag (V) Forschungsgenauigkeit
7	*Schlussbetrachtung*		(IV) Forschungsbeitrag

Tabelle 1: Einordnung der DSRM-Prozessphasen und Richtlinien

[15] Vgl. ebd., S. 86.

Systematische Literaturanalyse

Zur Feststellung des Forschungsstandes sowie zur Identifizierung relevanter Theorien und Konzepte basiert diese Arbeit auf der Durchführung einer systematischen Literaturanalyse als Methode der qualitativen Forschung. Die Ergebnisse dienen, neben den theoretischen Grundlagen, als Ausgangsbasis für die Entwicklung des konzeptionellen Ordnungsrahmens. Die angewandte Methodik orientiert sich am Literaturanalyseprozess nach Vom Brocke et al. Der Prozess setzt sich aus fünf Prozessphasen zusammen:

I. Festlegen des Untersuchungsumfangs
II. Erstellung eines Konzepts für die Recherche
III. Literaturrecherche (Suchprozess)
IV. Analyse der Literatur und Zusammenführung der Ergebnisse
V. Identifikation von Forschungslücke und -agenda[16]

Zur Definition des Untersuchungsumfangs der Literaturanalyse stützt sich diese Arbeit auf die Taxonomie nach Cooper. Dazu werden sechs Eigenschaften einer Literaturanalyse betrachtet, die jeweils unterschiedliche Ausprägungen annehmen können.[17] In Tabelle 2 wird eine Einordnung der Literaturanalyse anhand dieser Eigenschaften vorgenommen.

Eigenschaft		Einordnung			
(1)	*Fokus*	Ergebnis	Methoden	Theorien	Anwendungen
(2)	*Ziel*	Integration	Kritik	Problemidentifikation	
	>Integration	Verallgemeinerung	Konfliktlösung	Linguistische Brücke	
(3)	*Perspektive*	neutrale Repräsentation		Vertreten einer Position	
(4)	*Rahmen*	vollständig	vollst. + selektiv	repräsentativ	grundlegend
(5)	*Organisation*	historisch	konzeptionell	methodisch	
(6)	*Zielgruppe*	Fachpublikum	Allgemein	Praxis	Öffentlichkeit

Tabelle 2: Einordnung der Literaturanalyse in die Taxonomie nach Cooper [18]

(1) Die Analyse bezieht sich auf die Untersuchung von Forschungsergebnissen, vorhandenen Theorien sowie Praktiken und Anwendungen.
(2) Das Ziel der Literaturanalyse ist die Integration und Zusammenfassung von vorhandenem Wissen, mit dem Zweck der Verallgemeinerung von spezifischen Erkenntnissen und Aussagen.
(3) Die Ergebnisse sollen neutral dargestellt und reflektiert werden, das heißt ohne eine bestimmte Position bei der Interpretation einzunehmen.
(4) Der Ergebnisrahmen beschränkt sich sowohl auf Quellen, die den Untersuchungsumfang der Analyse repräsentieren, als auch auf Arbeiten, die von grundlegender Bedeutung für den Themenbereich sind.

[16] Vgl. Vom Brocke et al. 2009, S. 2211ff.
[17] Vgl. Cooper 1988, S. 109.
[18] In Anlehnung an: vgl. ebd., S. 109; Vom Brocke et al. 2009, S. 2212.

(5) Zur Organisation der Literaturanalyse werden konzeptionell und methodisch zusammenhängende Arbeiten gruppiert.
(6) Die Ergebnisse der Analyse sind, analog zu den Ergebnissen der vorliegenden Arbeit, primär an fachlich spezialisiertes Publikum und die Praxis gerichtet.

Die Literaturrecherche erfolgte auf Basis bestimmter Kriterien, die den Suchraum zielgerichtet einschränkten (vgl. Tabelle 3).

Kriterium	**Einschränkung**
Bibliotheken und Datenbanken	ACM Digital Library, AIS eLibrary (AISeL), Computing Research Repository (CoRR), dblp Computer Science Bibliography, Directory Of Open Access Journals (DOAJ), EBSCOhost Research Platform, Emerald Insight, IBM Research - Technical Paper Search, IEEE Xplore Digital Library, JSTOR, MIS Quarterly, Oxford Academic Computer Journal, ProQuest, Science Direct, Springer Link, Taylor & Francis Online, Wiley Online Library
Sprache	Die Quelle ist in deutscher oder englischer Sprache verfasst.
Literaturformen	Wissenschaftliche Beiträge (aus Journalen, von Konferenzen oder Workshops), Monografien, Artikel aus Fachmagazinen, Praxisberichte von Fachexperten
*Suchbegriffe**	Übergreifend: Monolith, Microservice(s), Domain-Driven Design Deutsch: fachliche Architektur, Migration, Architekturmigration, Transformation, Architekturtransformation, Extraktion, Dekomposition, domänengetriebener Entwurf Englisch: legacy, extract(-ing/-ion), migrat(-e/-ing/-ion), transform(-ing/-ation), decompos(-e/-ing/-ition) ** Die Suche basierte auf beliebigen Kombinationen der Suchbegriffe*

Tabelle 3: Einschränkungen der Literaturrecherche

Für die Recherche des Quellmaterials wurde ein strukturierter Ansatz nach Webster und Watson verfolgt.[19] Dafür wurden in erster Linie wissenschaftliche Bibliotheken und Datenbanken in die Suche einbezogen. Andere Literaturformen wie Monografien und Artikel aus Fachmagazinen wurden nur nachrangig und ergänzend untersucht.

[19] Vgl. Webster und Watson 2002, xvi.

2. Grundlagen

Diese Arbeit baut auf einer breiten theoretischen Grundlage auf, um die gestellten Forschungsfragen beantworten zu können. Zur Einführung in das Fachgebiet werden in diesem Kapitel daher die relevanten Themenbereiche vorgestellt.

2.1 Einführung in Softwarearchitekturen

2.1.1 Eigenschaften von Softwarearchitekturen

Nach einer am *IEEE-Standard 1471*[20] angelehnten Definition ist Softwarearchitektur:

> *„die grundsätzliche Organisation eines Systems, verkörpert durch dessen Komponenten, deren Beziehung zueinander und zur Umgebung sowie die Prinzipien, die für seinen Entwurf und seine Evolution gelten.“*[21]

Aufbauend auf dieser Definition können nach Starke sechs wesentliche Eigenschaften von Softwarearchitekturen beschrieben werden:

(1) Architekturen **enthalten unterschiedliche Strukturen**. Sie sind durch die Komponenten der Software sowie deren Schnittstellen und Beziehungen zueinander geprägt. In Softwarearchitekturen wird sowohl den Aufbau als auch den Ablauf von Software beschrieben. Sie sind daher statischer und dynamischer Natur.

(2) Softwarearchitekturen **beschreiben eine Softwarelösung** und stellen eine Dokumentation der Komponenten, der Schnittstellen sowie der übergreifenden Prinzipien und Prozesse dar. Sie umfassen somit planerische sowie entwurfsbezogene Konzepte und Strukturen, die zur Umsetzung der Software beitragen.

(3) Architekturen **basieren auf Entwurfsentscheidungen**. Diese Entscheidungen nehmen z. B. Einfluss auf den Entwurf der Komponenten oder beziehen sich auf die Technologieauswahl für die Umsetzung der Software.

(4) Softwarearchitekturen **fungieren als Bindeglied** zwischen den Anforderungen der Fachdomäne und der softwaretechnischen Implementierung. Die Anforderungen werden mit Hilfe der Softwarearchitektur auf die Komponenten der Software und deren gegenseitige Beziehungen abgebildet.

(5) Softwaresysteme können aus **unterschiedlichen Sichten** betrachtet werden. Sie können z.B. aus einer Kontextabgrenzungs-Sicht, Laufzeitsicht, Bausteinsicht oder Verteilungssicht beschrieben werden. Jede dieser Sichten dokumentiert einen anderen Aspekt des Gesamtsystems und stellt damit bestimmte Informationen für unterschiedliche Anspruchsgruppen bereit.

(6) Durch **Abstraktion** und Informationsfilterung bleibt eine bestimmte Sicht auf die Architektur verständlich und nachvollziehbar. Innerhalb der Sichten werden nicht benötigte Informationen daher bewusst ausgelassen.[22]

---

[20] Siehe: ISO/IEC/IEEE 42010:2011.
[21] Starke 2018, S. 16.
[22] Vgl. ebd., S. 16ff.

Der Begriff der Softwarearchitektur kann aus weiteren Blickwinkeln betrachtet werden. Die folgende Definition stellt die Bedeutung und Tragweite von architekturellen Entwurfsentscheidungen in den Vordergrund:

> *„Software architecture is the set of design decisions which, if made incorrectly, may cause your project to be cancelled."*[23]

Aus dieser Definition geht hervor, dass schlechte Architekturentscheidungen häufig nur mit hohem Aufwand und Kosten rückgängig gemacht werden können. Die Softwarearchitektur eines Systems lässt sich folglich nur bedingt anpassen, weshalb sämtliche Entscheidungen wohlüberlegt und vorausschauend getroffen werden sollten.

Nach Clements kann Softwarearchitektur auch wie folgt definiert werden:

> *„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."*[24]

Diese Definition legt den Schwerpunkt auf die Kommunikation und die gemeinsame Sprache zwischen allen Anspruchsgruppen (*Stakeholdern*) eines Softwaresystems. Die Softwarearchitektur als Kommunikationsgegenstand soll dabei zu einem einheitlichen Verständnis des Softwaresystems unter allen Anspruchsgruppen beitragen. Diesen Eigenschaften stehen im engen Bezug zu den zentralen Aufgaben und Zielen des Domain-Driven Designs (vgl. Kapitel 2.4).

Die Kommunikation kann durch eine ausführliche und treffende Architekturdokumentation unterstützt werden. Sie ermöglicht allen Beteiligten den Überblick zu behalten und die Softwarelösung zu verstehen. Die Komplexität des Softwaresystems kann besser beherrscht werden, wenn die verwendeten Konzepte und Strukturen übersichtlich dargestellt und dokumentiert werden. Im Allgemeinen kann Softwarearchitektur für die Dokumentation des Ist-Zustandes bestehender Systeme als auch für die Planung des Soll-Zustandes zukünftiger Systeme verwendet werden. Die Architektur bildet demnach das Grundgerüst für zukünftige Entwicklungen und Anpassungen. Sie hat maßgebenden Einfluss auf die Erweiterbarkeit und Flexibilität des Softwaresystems. Neben der Funktionalität der Software werden vor allem die Qualitätseigenschaften von den architekturellen Entwurfsentscheidungen beeinflusst. Dazu gehören Qualitätsmerkmale wie z. B. Sicherheit, Erweiterbarkeit, Robustheit oder Performanz. Weiterhin ist Architekturarbeit kein einmaliger Aufwand, sondern erstreckt sich für gewöhnlich über den gesamten Softwarelebenszyklus. Eine Softwarearchitektur entwickelt sich ständig weiter, denn bei jeder Anpassung oder Erweiterung des Systems müssen in der Regel auch architekturelle Entwurfsentscheidungen getroffen werden.[25]

---

[23] Eoin Woods, zitiert nach: Bass et al. 2013, S. 25.
[24] Clements et al. 2002, zitiert nach: Takai 2017, S. 8.
[25] Vgl. Starke 2018, S. 18ff.

2.1.2 Grundlegende Architekturbegriffe

Module und Komponenten

Software beinhaltet eine große Anzahl an Strukturen. Jedoch ist nicht jede Struktur ein Kandidat für die Architekturarbeit. Nach Bass und Clemens sind Strukturen für die Architektur relevant, wenn sich daraus Informationen über das Softwaresystem und dessen Eigenschaften ableiten lassen.[26]

Grundsätzlich kann in Softwarearchitekturen zwischen statischen und dynamischen Strukturen unterschieden werden (vgl. Kapitel 2.1.1). *Module* sind statisch und kapseln Funktionalität. Sie unterteilen Systeme in spezifische Implementierungseinheiten und können in weitere Module untergliedert und unterschiedlichen Teams zugewiesen werden. Über die so entstehenden statischen Modulstrukturen ist erkennbar, wie die Systemfunktionalität aufgeteilt wird und wer für die einzelnen Teile verantwortlich ist. Ein Beispiel ist die Aufteilung einer Anwendung in die Module Benutzeroberfläche, Geschäftslogik und Datenbank. Je nach Komplexität können diese Implementierungseinheiten beliebig weiter untergliedert werden.[27]
Eine *Komponente* ist eine spezifische Laufzeiteinheit, die aus einem oder mehreren Modulen bestehen kann. Komponenten sind folglich eine strukturelle Ebene höher angeordnet als Module. Sie sind außerdem dynamisch, da sie die Beziehungen und Interaktionen ihrer Elemente in den Vordergrund stellen. Im Mittelpunkt steht dabei die Ausführung von Systemfunktionen (Implementierungseinheiten) zur Laufzeit.[28]

Nach Turowski können Komponenten als Softwarebausteine beschrieben werden, die unter anderem folgende Eigenschaften besitzen:

- Wiederverwendbarkeit
- Abgeschlossenheit hinsichtlich der Implementierung
- Sie stellen Dienste über wohldefinierte Schnittstellen bereit
- Lose Kopplung
- Universell kombinierbar mit anderen Komponenten[29]

Die Weiterentwicklung einzelner Komponenten sollte demnach ohne großen Koordinationsaufwand erfolgen können. Weiterhin sollten einzelne Komponenten unabhängig dokumentierbar, nachvollziehbar und testbar sein.[30] Dies entspricht einigen grundlegenden Eigenschaften von Microservices (vgl. Kapitel 2.2.3).

---

[26] Vgl. Bass et al. 2013, S. 5.
[27] Vgl. ebd., S. 4f.
[28] Vgl. ebd., S. 5.
[29] Vgl. Turowski 2003, S. 16, 2003, S. 279.
[30] Vgl. Dowalil 2018, Kap. 1.1.1 Abs. 3.

Architekturstile und Entwurfsprinzipien

Die Begriffe Architekturstil und Architekturmuster können synonym verwendet werden. Diese Arbeit beschränkt sich auf die Verwendung des Begriffes „Architekturstil".

Ein *Architekturstil* ist ein grobgranulares Muster, das bestimmte Lösungen für häufig wiederkehrende Problemstellungen von Softwaresystemen bereitstellt. Diese Lösungen entsprechen einer Menge an Prinzipien, die die Architektur einer Anwendung beeinflussen.[31]

Nach Takai wird ein Architekturstil als eine „*Sammlung von Entwurfsprinzipien*"[32] beschrieben. Ein *Entwurfsprinzip* entspricht dabei einer allgemein akzeptierten und wohldefinierten Industriepraxis, beispielsweise die Zustandslosigkeit einer Schnittstelle oder die Wiederverwendbarkeit eines Service. Die konsistente Anwendung der Entwurfsprinzipien eines Architekturstils hilft bei der Analyse und dem Entwurf eines Softwaresystems und soll in erster Linie zur Erreichung bestimmter Qualitätsmerkmale beitragen.[33] Der Entscheidungsrahmen für das Treffen architektureller Entwurfsentscheidungen wird maßgebend durch die Auswahl und Nutzung von Architekturstilen bestimmt. Sie dienen durch ihre Entwurfsprinzipien, spezifischen Vorgaben und Regeln als Leitplanken für den Architekturentwurf. Ein Architekturstil kann beispielsweise Vorgaben für bestimme Aspekte wie Modularität oder Zyklen-Freiheit enthalten. Diese Regeln zur Systemstrukturierung beziehen sich primär auf die Elemente der Architektur und deren Beziehungen zueinander. Die Verwendung bestimmter Architekturstile hat demnach das übergeordnete Ziel, die Entstehung von technischen Schulden zu minimieren.[34]

Die Architektur einer Softwareanwendung basiert häufig auf einer Kombination aus mehreren Architekturstilen. Beispielsweise kann eine serviceorientierte Architektur (SOA) aus verschiedenen Services bestehen, die jeweils auf einer *Schichtenarchitektur* (vgl. Kapitel 2.2.2) aufbauen und den Architekturstil der Objektorientierung nutzen.[35] Softwarearchitekturen, die auf mehreren Architekturstilen basieren, werden folglich auch als Stil-basierte Architekturen bezeichnet.[36]

Nach Takai ist auch der *domänengetriebene Entwurf* ein Architekturstil und kann mit dem *Microservice-Architekturstil* (vgl. Kapitel 2.2.3) kombiniert werden.[37] Diese Kombination ist kennzeichnend für die vorliegende Arbeit.

---

31 Vgl. Microsoft Corporation 2009, S. 19.
32 Takai 2017, S. 8.
33 Vgl. ebd., S. 8.
34 Vgl. Lilienthal 2017, Kap. 6.1 Abs. 1.
35 Vgl. Microsoft Corporation 2009, S. 21.
36 Vgl. Oquendo et al. 2016, S. 168.
37 Vgl. Takai 2017, S. 8.

Entwurfsmuster

Entgegen einem Entwurfsprinzip ist ein Entwurfsmuster eine „*konkrete Entwurfsvorlage, die ein verbreitetes und wiederkehrendes Problem löst*“[38].

Während bei Architekturstilen die Architektur des gesamten Systems betrachtet wird, beziehen sich Entwurfsmuster in der Regel nur auf eine spezifische Problemstellung eines Systembestandteils. Entwurfsmuster finden primär im Bereich der Mikroarchitektur (vgl. Kapitel 2.1.3) Anwendung und lösen ein bestimmtes Teilproblem. Sie beeinflussen im Regelfall nicht die grundlegende Architektur der Softwareanwendung. Ein Architekturstil kann dagegen mehrere Entwurfsmuster enthalten. Hierarchisch gesehen, sind Architekturstile demnach höher angesiedelt als Entwurfsmuster. Architekturstile stellen grobgranulare Muster dar, während Entwurfsmuster feingranulare Muster sind. Die Unterscheidung zwischen Architekturstilen und Entwurfsmustern ist jedoch nicht immer eindeutig. Zum Beispiel kann das Konzept *Model-View-Controller (MVC)* sowohl den Architekturstilen als auch den Entwurfsmustern zugeordnet werden.[39]

2.1.3 Mikro- und Makroarchitektur

Im Kontext von Softwarearchitekturen werden zwei Abstraktionsebenen unterschieden: die *Mikro*- und die *Makroarchitektur*. Die Mikroarchitektur ist die Ebene niedriger Abstraktion und bezieht sich auf die Architektur einzelner Systemkomponenten. Die Makroarchitektur bezieht sich hingegen auf das Gesamtsystem, gemäß der Definition von Softwarearchitekturen (vgl. Kapitel 2.1.1) also auf alle Komponenten der Software und deren Zusammenspiel untereinander.[40] Abbildung 1 verdeutlicht diesen Zusammenhang:

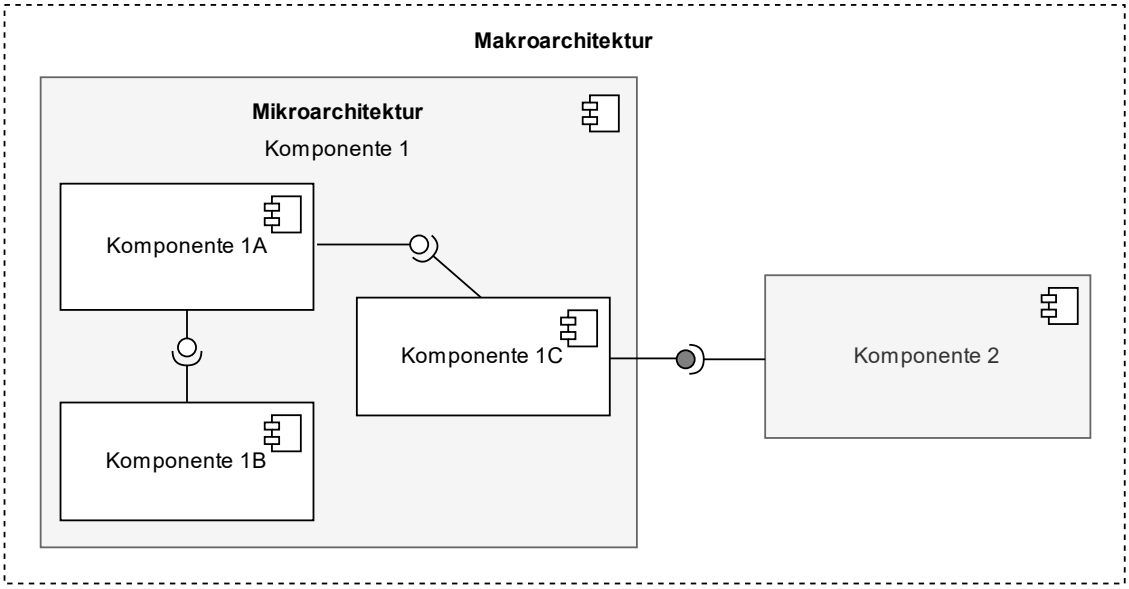

Abbildung 1: Mikro- und Makroarchitektur [41]

[38] Ebd., S. 7.
[39] Vgl. Goll und Dausmann 2013, S. 68f.
[40] Vgl. Dowalil 2018, Kap. 1.1.1 Abs. 3.
[41] In Anlehnung an: ebd., Kap. 1.1.1.

In Abbildung 1 sind zwei Komponenten eines Softwaresystems abgebildet. Das Zusammenspiel von Komponente 1 und 2 findet auf Ebene des Gesamtsystems, also auf Abstraktionsstufe der Makroarchitektur, statt. Wird Komponente 1 und dessen interner Aufbau nicht als Black-Box betrachtet, handelt es sich um die Ebene der Mikroarchitektur. Sie bezieht sich auf die Architektur einer einzelnen Komponente.

Die Untergliederung in Mikro- und Makroarchitektur ist in erster Hinsicht für Architekturentscheidungen von Bedeutung. Zur Ebene der Mikroarchitektur zählen Entscheidungen, die sich auf einzelne Komponenten des Systems beziehen und vom jeweils verantwortlichen Team getroffen werden können. Entscheidungen, die auf Makroarchitekturebene getroffen werden, betreffen dagegen das gesamte System und gelten für alle Komponenten und Teams. So können beispielsweise übergreifende Richtlinien definiert werden oder grundlegende Technologien für die gesamte Software vorgeschrieben und vereinheitlicht werden. Welche genauen Entscheidungen auf Ebene der Mikro- oder Makroarchitektur getroffen werden, ist stets von den individuellen Rahmenbedingungen der betrachteten Softwareanwendung abhängig.[42]
Aus Sicht einer Organisation ist die Unterscheidung in Mikro- und Makroarchitektur vor allem von der Teamstruktur abhängig. Ist nur ein Team für die Entwicklung und Wartung der Software verantwortlich, so deckt die Mikroarchitektur bereits den Großteil der Architekturentscheidungen ab. Erst bei größeren Systemen oder Systemlandschaften, bei denen mehrere Teams involviert sind, gewinnt das Thema der Makroarchitektur aus organisatorischer Perspektive an Bedeutung.[43]

2.1.4 Einordnung von Softwarearchitekturen

Softwarearchitektur schlägt die Brücke zwischen den Analyseaktivitäten und der Implementierung innerhalb eines Softwareprojektes (vgl. Kapitel 2.1.1). Sie nimmt die zentrale Rolle des Mediators ein und dient allen Anspruchsgruppen als Leitbild.[44]
Abbildung 2 vermittelt einen Überblick über die Zusammenhänge der Softwarearchitektur und den weiteren typischen Aufgabenbereichen der Softwareentwicklung.

Die Anforderungen bestimmen beispielsweise den Entwurf der Softwarearchitektur. Im Umkehrschluss wirken sich bestimmte Einschränkungen der Architektur auf die Anforderungsdefinition aus. Der Softwarearchitekt ist nicht nur für die Umsetzung bestimmter Anforderungen zuständig, sondern muss diese auch auf technische Realisierbarkeit prüfen und Verfeinerungen an den Anforderungsdefinitionen vornehmen. Softwarearchitek-

---

[42] Vgl. Wolff 2016, Kap. 13.3 Abs. 2-3.
[43] Vgl. Dowalil 2018, Kap. 3 Abs. 1.
[44] Vgl. Starke 2018, S. 29.

turen liefern außerdem entscheidende Informationen für die Projekt- und Aufgabenplanung und stellen das grobe Rahmengerüst für die Implementierung dar. Weiterhin helfen sie bei der Analyse und Steuerung von Risiken innerhalb von Softwareprojekten. Die Entstehung von Softwarearchitekturen ist auch durch die Organisation geprägt, z. B. durch den vorherrschenden Entwicklungsprozess oder Ressourceneinschränkungen. Im Umkehrschluss kann die Architektur auch maßgeblichen Einfluss auf die Gestaltung organisatorischer Prozessabläufe nehmen, indem Geschäftsprozesse an der Softwarearchitektur ausgerichtet werden.[45]

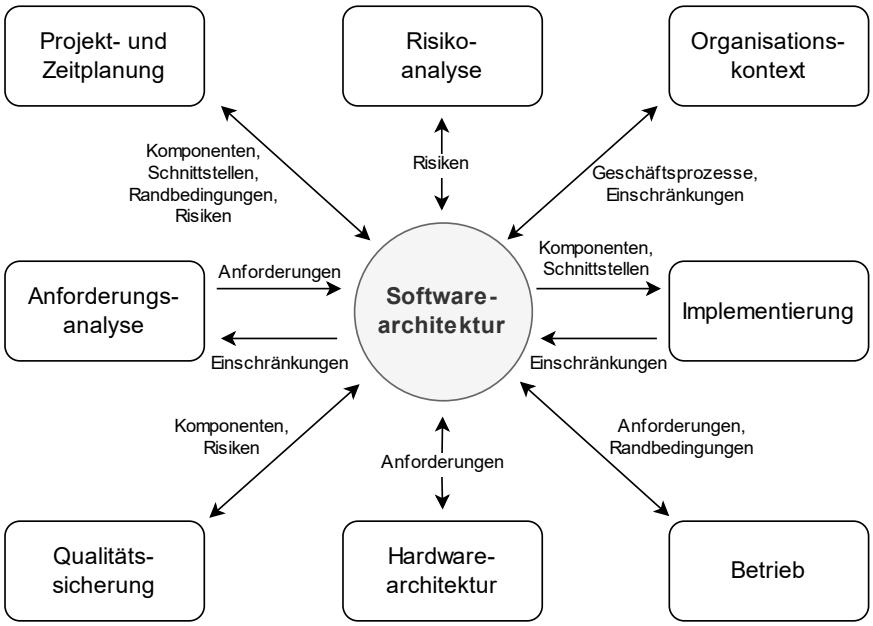

Abbildung 2: Architektur im Kontext anderer Entwicklungsaktivitäten [46]

Da sich die Software- und Hardwarearchitektur stark gegenseitig beeinflussen, müssen diverse Qualitätsanforderungen wie Verfügbarkeit und Sicherheit auch durch die Hardwarearchitektur entsprechend abgedeckt sein. Ebenso gibt die Softwarearchitektur gewisse Rahmenbedingungen für den Betrieb der Software vor. Die bereitgestellte Laufzeitumgebung muss daher die Vorgaben der Architektur hinsichtlich der bereitzustellenden Hardwareressourcen und Softwaredienste, z. B. Datenbanken, erfüllen. Darüber hinaus hat die Softwarearchitektur Einfluss auf die Test- und Prüfbarkeit von Anforderungen und ist insbesondere durch Tests einzelner Systemkomponenten von großer Bedeutung für die Qualität von Softwaresystemen.[47]

[45] Vgl. ebd., S. 29ff.
[46] Ebd., S. 29.
[47] Vgl. ebd., S. 31f.

2.2 Software-Architekturstile

Es existiert eine Vielzahl verschiedener Architekturstile, die auf die Erreichung bestimmter Qualitätseigenschaften abzielen. Dieses Kapitel gibt zunächst einen Überblick über die beiden primären Kategorien von Softwaresystemen – monolithische und verteile Systeme. Auf dieser Grundlage werden die Architekturstile, die für die vorliegende Arbeit relevant sind, vorgestellt und abgegrenzt.

2.2.1 Monolithische und verteilte Systeme

Ein Monolith bezeichnet streng genommen keinen Software-Architekturstil, sondern ein Softwaresystem, das bestimmte Eigenschaften besitzt und dadurch eine monolithische Architektur aufweist. Das gleiche gilt für Softwaresysteme mit verteilter Architektur. Monolithische und verteilte Systeme sind folglich als Klassifizierung von Architekturen anzusehen. In dieser Arbeit werden daher explizit nicht die Begriffe der monolithischen oder verteilten Architektur verwendet. Stattdessen kommen die Begriffe monolithisches und verteiltes System oder monolithische und verteilte Anwendung zum Einsatz.

Monolithische Systeme

Das Wort *Monolith* kommt ursprünglich aus dem Altgriechischen, wobei *Mono* für „ein“ und *Lithos* für „Stein“ steht.[48] Im übertragenen Sinn bedeutet Monolith „*aus einem einzigen großen Steinblock*“[49]. Nach Wagner ist eine Software monolithisch, wenn sie keine klar strukturierte Architektur vorweisen kann und technische Aspekte wie z. B. die Datenhaltung, Geschäftslogik oder die Nutzerschnittstellen eng miteinander verflochten sind.[50] Diese Definition bezieht sich auf frühe betriebliche Anwendungssysteme, die auf einem zentralen Großrechner *(Mainframe)* installiert wurden und keine klare Separierung von Zuständigkeiten vorsahen.
Mit der Einführung von technisch orientierten Schichten wurden klassische Monolithen später im Zuge von Dezentralisierungsbestrebungen in Komponenten aufgeteilt. Die Schichten der Datenhaltung und Präsentation wurden aus dem Monolithen herausgelöst und nach dem *Client/Server-Prinzip* an die Anwendungslogik angebunden (vgl. Kapitel 2.2.2).[51] Jedoch wird auch diese Art von Softwaresystemen heute noch als Monolith bezeichnet.

---

48 Vgl. Dowalil 2018, Kap. 7.1 Abs. 1.
49 Duden online, Suchbegriff: „Monolith“ 2018.
50 Vgl. Wagner 2014, S. 30.
51 Vgl. Herden et al. 2006, S. 25ff.

Nach einer zeitgemäßen Definition ist eine Softwareanwendung monolithischer Natur, wenn die gesamte Anwendungslogik in einer einzigen, zentralen Laufzeitumgebung bereitgestellt und betrieben wird. Heutzutage steht demnach die Anwendungslogik im Mittelpunkt, wenn von einem Monolithen die Rede ist.[52]
Die Frage – *Wann ist eine Anwendung ein Monolith?* – kann folglich nicht anhand der Größe eines Systems beantwortet werden, sondern in Bezug auf die Struktur und die Bereitstellung der Anwendung. In der Regel sind die einzelnen Module der Anwendungslogik nicht unabhängig voneinander ausführbar und teilen sich die Ressourcen eines Servers. Infolgedessen sind Monolithen häufig schwer zu warten und Änderungen nur mit großem Aufwand umsetzbar.[53]

Die **Herausforderungen**, die mit monolithischen Systemen einhergehen, können aus mehreren Perspektiven betrachtet werden:

Aus Sicht der Entwicklung besteht häufig ein begrenztes Verständnis über die Anwendung, wenn sie eine große und komplexe Code-Basis mit vielen Abhängigkeiten besitzt. Änderungen am Quellcode sind dann mit erhöhtem Abstimmungsaufwand und Risiko verbunden. Auch das Testen von Monolithen geht häufig mit großem Aufwand einher. Im Allgemeinen wirkt sich die Größe, Struktur und Komplexität des monolithischen Systems negativ auf die Produktivität der Entwickler und Tester aus.[54]

Aus Sicht des Betriebes muss immer die gesamte monolithische Anwendung als eine Einheit neu bereitgestellt werden, auch wenn nur ein kleiner Teil der Software geändert wurde. Darüber hinaus ist die Skalierung eines monolithischen Systems nur eingeschränkt möglich. Sie kann lediglich durch Vervielfältigen der gesamten Anwendung oder durch Hinzufügen von Hardware-Ressourcen erfolgen. Im Rahmen großer und komplexer Systeme sind meist mehrere Teams beteiligt, die nur bedingt unabhängig agieren können, da die Teile der Anwendung stark voneinander abhängen und alle Änderungen zwischen den Teams koordiniert werden müssen. Weiterhin unterliegt ein Monolith oftmals starken Einschränkungen hinsichtlich der eingesetzten Technologien, da diese häufig nur mit erhöhtem Aufwand und Risiko ausgetauscht werden können. Zur Minimierung von Risiken werden die eingesetzten Technologien daher oft weiterverwendet und die Chancen, die sich durch die Nutzung neuer Technologien ergeben, nicht ergriffen.[55]

Die beschriebenen Nachteile und Herausforderungen bedeuten jedoch nicht, dass monolithische Systeme generell schlecht sind. Für kleinere und weniger komplexe Anwendungen stellen sie oftmals eine angemessene Lösung dar und können bei Bedarf auch hochverfügbar und performant bereitgestellt werden.[56]

---

[52] Vgl. Daya 2015, S. 7.
[53] Vgl. Bucchiarone et al. 2018, S. 50.
[54] Vgl. Daya 2015, S. 7f.
[55] Vgl. Santis et al. 2016, S. 19f.
[56] Vgl. Balalaie et al. 2016b, S. 202.

Verteilte Systeme

Verteilte Systeme können als Gegenstück zu monolithischen Systemen betrachtet werden. Nach Starke ist ein verteiltes System wie folgt definiert:

> *„Ein verteiltes System zeichnet sich durch Zusammenarbeit von Verarbeitungs- und Speicherbausteinen über Kommunikationsnetze hinweg aus. Daten, Prozesse und auch Benutzer können damit räumlich verteilt voneinander arbeiten.“*[57]

In monolithischen Systemen sind alle Komponenten innerhalb einer einzelnen physischen Einheit zusammengefasst. Im Gegensatz dazu sind die Komponenten in verteilten Systemen auf räumlich getrennte Umgebungen verteilt. Das Ziel der Dezentralisierung ist die Verbesserung von Aspekten wie der Flexibilität, Skalierbarkeit, Resilienz oder Weiterentwicklung.[58] Durch lose Kopplung der Komponenten und einen hohen Grad an Modularität soll die Gesamtkomplexität des Systems besser gehandhabt und die Flexibilität verbessert werden. Die gewonnene Flexibilität dient der schnellen Reaktion auf Veränderungen der Umwelt und ist daher vorwiegend strategischer Natur.[59] Die Komponenten in verteilten Systemen können über unterschiedliche Technologien miteinander kommunizieren. Die Kommunikation kann direkt (synchron) oder indirekt (asynchron) erfolgen.[60]

Den Vorteilen von verteilten Systemen stehen verschiedene **Herausforderungen** gegenüber. Es existieren acht allgemeine Trugschlüsse hinsichtlich verteilter Systeme, die die wesentlichen Herausforderungen repräsentieren:

(1) *„Das Netzwerk ist ausfallsicher.*
(2) *Die Latenzzeit ist gleich Null.*
(3) *Der Datendurchsatz ist unendlich.*
(4) *Das Netzwerk ist sicher.*
(5) *Die Netzwerktopologie wird sich nicht ändern.*
(6) *Es gibt immer nur einen Netzwerkadministrator.*
(7) *Die Kosten des Datentransports können mit Null angesetzt werden.*
(8) *Das Netzwerk ist homogen.“*[61]

Da es sich um Trugschlüsse handelt, trifft in der Realität das Gegenteil zu. Beispielsweise sind Netzwerke nicht vor Ausfällen geschützt, die Kommunikation über Netzwerke unterliegt einer Latenz größer null, der Datendurchsatz ist limitiert und Netzwerke sind nie vollkommen sicher. Bei der Implementierung eines verteilten Systems sind diese acht Punkte zu berücksichtigen, was für zusätzliche Komplexität im Vergleich zu monolithischen Systemen sorgt. In Bezug auf die Punkte (1) und (2) gilt es z. B. entsprechende Maßnahmen zu ergreifen, falls eine angefragte Komponente nicht oder erst nach längerer Zeit antwortet.[62]

---

[57] Starke 2018, S. 118.
[58] Vgl. Mosleh et al. 2018, S. 125.
[59] Vgl. ebd., S. 126.
[60] Vgl. Starke 2018, S. 118.
[61] Dowalil 2018, Kap. 7 Abs. 1.
[62] Vgl. ebd., Kap. 7 Abs. 1.

2.2.2 Schichtenarchitekturen

Nach Esposito ist eine *Schicht* ein Bestandteil einer Software, der gemeinsam mit anderen Bestandteilen in einem Prozess läuft. Eine Schicht zielt auf die logische und nicht die physische Unterteilung einer Software ab.[63] Diese logische Unterteilung folgt dem Prinzip der Trennung von Zuständigkeiten *(Separation of Concerns)* und dient als Hilfsmittel für die Implementierung von einfachen als auch komplexen Softwareprojekten.[64]

Beim Architekturstil der *Schichtenarchitekturen* werden Softwaresysteme in Abstraktionsebenen strukturiert. Dabei wird das System in mehrere, aufeinander aufbauende Schichten unterteilt, in denen kohärente Funktionen der Software zu größeren Einheiten zusammengefasst werden. Die Aufteilung hat das primäre Ziel die Komplexität des Systems zu gruppieren. Die Schichten folgen einer hierarchischen Ordnung und unterliegen gewissen Regeln, um die Anzahl der Abhängigkeitsbeziehungen zu reduzieren und zyklische Abhängigkeiten zu vermeiden.[65]

Die *Drei-Schichtenarchitektur*, als verbreite Ausprägung des Schichtenmodells, sieht eine Unterteilung in die drei Schichten der Präsentation, Geschäftslogik und Datenhaltung vor. Die Präsentationsschicht ist für sämtliche Interaktionen des Benutzers mit der Software zuständig. Dahingegen fungiert die Schicht der Geschäftslogik als Bindeglied zwischen der Präsentation und der Datenhaltung. Sie ist für Aufgaben wie Berechnungen oder Validierungen zuständig und liefert, basierend auf den Eingabewerten, Daten an die Präsentationsschicht. Häufig greift die Präsentationsschicht jedoch auch direkt auf die Datenhaltung zu. Die Schicht der Datenhaltung kapselt Aufgaben der Kommunikation mit diversen Datenquellen wie Datenbanken oder Messaging-Systemen.[66]

Allgemein kann in Schichtenarchitekturen zwischen zwei Dimensionen unterschieden werden: der *technischen Schichtung* und der *fachlichen Schichtung*. Das Beispiel der Drei-Schichtenarchitektur ist durch drei technische Gesichtspunkte gekennzeichnet. Bei großen und komplexen Softwaresystemen ist eine technisch motivierte Unterteilung jedoch meist nicht ausreichend. Daher spielt in der Praxis die weitere Aufteilung nach fachlichen Aspekten eine bedeutende Rolle. Das System wird dabei zusätzlich in vertikal angeordnete, fachliche Schichten untergliedert. Die so entstehenden fachlichen Module oder Komponenten erstrecken sich über die horizontal angeordneten technischen Schichten. Das bedeutet, dass jede fachliche Einheit die Anteile der technischen Schichten enthält, die zum jeweiligen Geschäftsbereich bzw. zur Fachlichkeit gehören.[67] Abbildung 3 stellt diesen Sachverhalt dar.

---

[63] Vgl. Esposito 2016, Kap. 5 Abs. 3.
[64] Vgl. Familiar 2015, S. 22.
[65] Vgl. Goll und Dausmann 2013, S. 290f.
[66] Vgl. Fowler und Rice 2002, S. 19f.
[67] Vgl. Lilienthal 2017, Kap. 6.3.1 und 6.3.2.

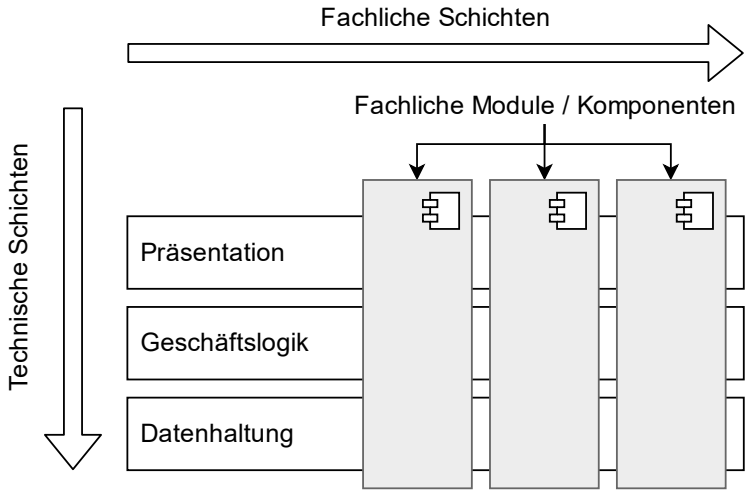

Abbildung 3: Technische und fachliche Schichtung [68]

Nach Lilienthal kann der gezeigte Zusammenhang als „*modulare Aufteilung eines Systems entlang der technischen Dimension*“[69] beschrieben werden.

Analog zu den technischen Schichten basieren auch die fachlichen Schichten auf bestimmten Beziehungen untereinander. Beziehungen zwischen fachlichen Schichten sind ebenfalls nur in eine Richtung erlaubt. Die fachlichen Schichten unterliegen somit gewissen Gebots- und Verbotsregeln, die in einer Hierarchie resultieren.[70]

Aus Sicht der Architekturdokumentation und unter der Annahme, dass es sich in Abbildung 3 nicht um räumlich verteilte, fachliche Komponenten handelt, können sowohl die Drei-Schichtenarchitektur als auch das Konzept der technischen und fachlichen Schichtung (vgl. Abbildung 3) im Grunde dasselbe Softwaresystem beschreiben. Es handelt sich vorwiegend um unterschiedliche Darstellungen. Die Drei-Schichtenarchitektur basiert auf einer technisch motivierten Unterteilung, wobei die technischen Schichten jeweils alle fachlichen Anteile zur Umsetzung des Softwaresystems enthalten. Dahingegen stellt Abbildung 3 die fachlichen Anteile explizit in den Vordergrund. Erst bei der eigentlichen Implementierung auf Basis dieser Architektur ist zu entscheiden, ob in erster Linie technisch oder fachlich strukturiert werden soll. Dies hat dann beispielsweise Einfluss auf die Struktur und die Benennung von Klassen oder Paketen. Weiterhin kann die Entscheidung über die technische und fachliche Motivation wesentlichen Einfluss auf die Teamstruktur haben (vgl. Kapitel 2.3.1).[71]

---

[68] In Anlehnung an: ebd., Kap. 6.3.2.
[69] Ebd., Kap. 6.3.1 Abs. 3.
[70] Vgl. ebd., Kap. 6.3.2 Abs. 5.
[71] Vgl. Zörner 2012, S. 110f.

2.2.3 Microservice-Architekturen

Einige Technologiekonzerne setzen bereits seit längerer Zeit auf Microservice-artige Architekturen, auch wenn der Begriff des *Microservice* erst später geprägt wurde. Amazon hatte im Jahr 2006 darüber berichtet wie das Unternehmen ihre Softwareanwendung auf Basis ihrer Cloud-Plattform entwickelt. Bereits zu dieser Zeit hatte Amazon seinen Online-Shop in mehrere Services unterteilt, die unterschiedliche Technologien verwenden konnten und von unabhängigen Teams verantwortet wurden.[72]
Nach Fowler wurde im Jahr 2011 im Rahmen eines Architektur-Workshops in der Nähe von Venedig ein Architekturstil thematisiert, den viele Software-Architekten zu dieser Zeit erforschten. 2012 hat sich diese Gruppe von Architekten dazu entschlossen, den Architekturstil als Microservice-Architekturstil zu bezeichnen.[73]

Definition und Eigenschaften

Microservice-basierte Anwendungen können der Kategorie der verteilten Systeme zugeordnet werden (vgl. Kapitel 2.2.1). Der Begriff der Microservice-Architektur ist nicht fest definiert. Es existieren jedoch verschiedene Ansätze, die den Begriff aus unterschiedlichen Blickwinkeln betrachten und bestimmte Eigenschaften hervorheben. In einer einfachen Definition von Newman werden Microservices wie folgt beschrieben:

> „*Microservices are small, autonomous services that work together.*“[74]

Microservices sind demnach klein, eigenständig und auf die Zusammenarbeit ausgerichtet. Nach einer Definition von Takai besitzen Microservices noch weitere Eigenschaften:

> „*Ein Microservice […] hat nur eine einzige Geschäftsaufgabe, aber erledigt diese besonders gut. Zusammen mit anderen Diensten lässt sich ein Microservice zu einer Microservice-Architektur komponieren.*“[75]

Aus dieser Definition lassen sich zwei wesentliche Eigenschaften ableiten:

(1) Ein Microservice erfüllt genau eine fachliche Funktion und ist darin besonders gut *(Single Responsibility Principle)*.
(2) Durch die Kombination von Services ergibt sich eine Microservice-Architektur.

Demnach führt der Einsatz des Microservice-Architekturstils zu einer fachlich orientierten Aufteilung der Softwareanwendung, die technische Aspekte in den Hintergrund stellt (vgl. Kapitel 2.2.2).[76]

---

[72] Vgl. O'Hanlon 2006, S. 14ff.
[73] Vgl. Lewis und Fowler 2014, Fußnote 1.
[74] Newman 2015, S. 2.
[75] Takai 2017, S. 17f.
[76] Vgl. Lilienthal 2017, Kap. 7.5 Abs. 3.

Eine ausführlichere und vielfach herangezogene Definition liefern Fowler und Lewis. Ihnen zufolge ist der Microservice-Architekturstil ein:

> *„Ansatz für die Entwicklung einer einzigen Anwendung in Form einer Reihe kleiner Services, die jeweils in einem eigenen Prozess laufen und die durch einfache Mechanismen kommunizieren – oft durch HTTP-Ressourcen-basierte APIs. Diese Dienste orientieren sich entlang [von] Business-Capabilities und sind durch vollautomatisches Deployment unabhängig voneinander deploybar. Es gibt nur ein Minimum an zentraler Verwaltung dieser Dienste, die unterschiedliche Programmiersprachen wie auch Datenspeicher-Technologien verwenden können.“*[77]

Aus dieser Definition lassen sich sechs grundlegende Eigenschaften von Microservice-Architekturen ableiten:

(1) Microservices sind *verteilte Dienste*, die jeweils eine eigene Laufzeitumgebung besitzen.
(2) Sie nutzen *leichtgewichtige Mechanismen zur Kommunikation*. Die Anwendungslogik befindet sich ausschließlich in den Service-Endpunkten und nicht in den Kommunikationskanälen.[78]
(3) Sie kapseln in ihrer Implementierung einzelne *Geschäftsfähigkeiten (Business Capabilities)* und sehen damit eine fachliche Modularisierung vor.
(4) Jeder Microservice ist *unabhängig* und (idealerweise) *automatisch deploybar*.
(5) Eine Microservice-Architektur ist auf ein *dezentralisiertes Management* ausgelegt, d. h. jeder Microservice wird von einem einzelnen Team unabhängig entwickelt und verantwortet.
(6) Die Microservice-Teams haben gewisse Freiheiten bezüglich der Auswahl der einzusetzenden Sprachen und Technologien.

Nach Starke können fünf weitere Eigenschaften hinzugefügt werden:

(1) Microservices können *unabhängig skaliert* werden.
(2) Sie nutzen eine *dezentrale Datenverwaltung*, d. h. jeder Microservice verwaltet seine eigenen Daten.
(3) Microservice-Architekturen benötigen eine *automatisierte Infrastruktur*.
(4) Die Kommunikation basiert auf dem gegenseitigen Auffinden der Services durch die Nutzung von *Service Discovery* Mechanismen.
(5) Microservices sind *evolutionär*, d. h. sämtliche Code-Änderungen sind klein und münden in schnellen und risikoarmen Releases.[79]

---

[77] Fowler und Lewis 2015, S. 14.
[78] Vgl. Gonzalez 2016, S. 9.
[79] Vgl. Starke 2018, S. 339f.

Abbildung 4 fasst die wesentlichen Eigenschaften des Microservice-Architekturstils in einer Übersicht zusammen:

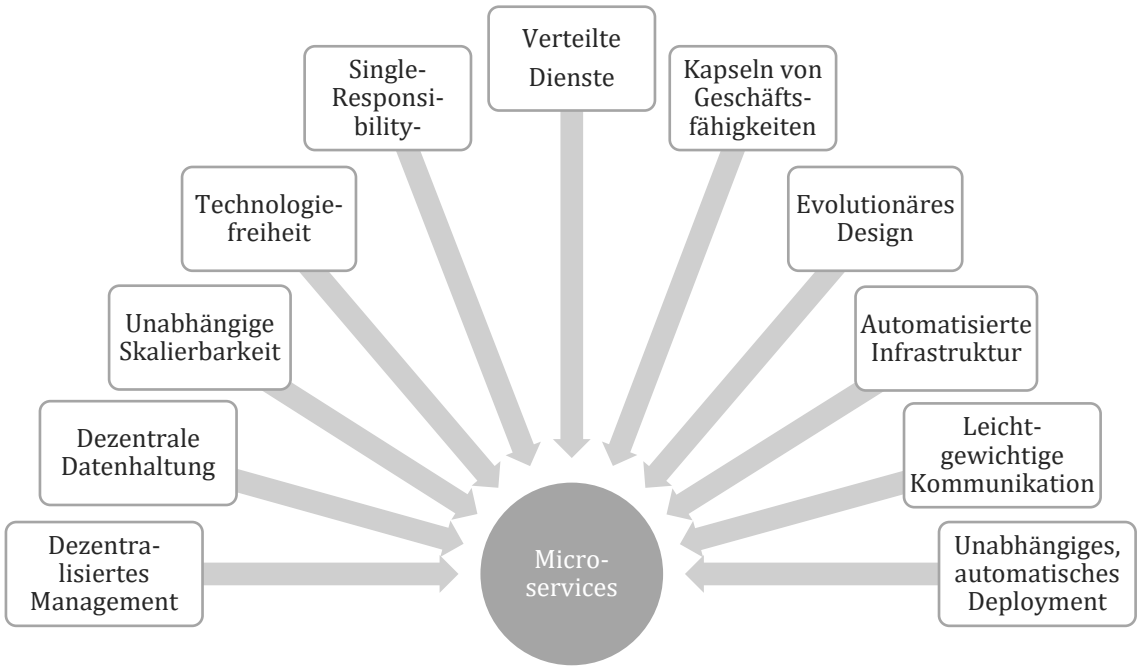

Abbildung 4: Eigenschaften des Microservice-Architekturstils

Eine nebenläufige Eigenschaft von Microservices ist deren Klassifizierung im Kontext der Architekturelemente „Modul" und „Komponente" (vgl. Kapitel 2.1.2). „*Ein Microservice [ist] sowohl Modul als auch Komponente, denn ein Microservice wird isoliert entwickelt und ist gleichzeitig eine Laufzeiteinheit, die unabhängig deployed werden kann.*"[80] Aufgrund dieser Besonderheit können Microservices zur Laufzeit komponiert werden. Für das Hinzufügen oder Ändern von Funktionalität muss einzig der betroffene Microservice und nicht die gesamte Softwareanwendung neu gebaut und bereitgestellt werden.[81]

Vorteile und Ziele

Grundsätzlich können die Herausforderungen von monolithischen Systemen als Motivation für den Einsatz von Microservice-Architekturen dienen (vgl. Kapitel 2.2.1). Es existieren zudem zahlreiche Gründe, die für den Einsatz verteilter Systeme sprechen, auch wenn deren Einsatz mit verschiedenen Herausforderungen verbunden ist. Mit Microservices lassen sich viele der Vorteile allerdings in einem größeren Umfang ausnutzen als mit alternativen Ansätzen verteilter Systeme.[82]

---

[80] Takai 2017, S. 20.
[81] Vgl. ebd., S. 20.
[82] Vgl. Newman 2015, S. 4.

Newman beschreibt sieben Hauptvorteile von Microservices, die für den Einsatz dieses Architekturstils sprechen können:

(1) Microservice-basierte Systeme profitieren von der **freien Technologiewahl**. Für jedes Teilproblem, das durch einen Microservice gelöst wird, kann die bestmögliche Technologie gewählt werden. So kann je nach Anforderung beispielsweise eine bestimmte Datenbanktechnologie oder besonders performante Programmiersprache zum Einsatz kommen. Auf Mikroarchitekturebene (vgl. Kapitel 2.1.3) sorgt dies für technologische Flexibilität innerhalb der einzelnen Microservice-Teams.
(2) Der Microservice-Architekturstil sorgt durch technische Konzepte wie z.B. das Schottwand-Entwurfsmuster *(Bulkhead)* oder das Sicherungs-Entwurfsmuster *(Circuit-Breaker)* ebenfalls für **verbesserte Resilienz (Robustheit)** und Verfügbarkeit, da im Gegensatz zu Monolithen keine ausfallkritische Komponente *(Single-Point-of-Failure)* mehr im System vorhanden ist.
(3) Ein weiterer Vorteil ist die individuelle und **unabhängige Skalierbarkeit** einzelner Microservices. Da jeder Service unabhängig vom Rest der Anwendung skalierbar ist, können sowohl Ressourcen als auch Kosten eingespart werden.
(4) Microservices können unabhängig, schnell und risikoarm bereitgestellt werden. Das **vereinfachte Deployment** ist daher ein ausschlaggebendes Argument für den Einsatz von Microservices.
(5) Microservices begünstigen eine gegenseitige **Ausrichtung der Organisation und der Architektur**. Dadurch können effizientere Teamgrößen ermöglicht und die Produktivität der Teams gesteigert werden. Dieser Vorteil korreliert mit dem Gesetz von Conway (vgl. Kapitel 2.3.1).
(6) Die **Kombinierfreudigkeit** von Microservices ermöglicht, dass die von ihnen bereitgestellte Funktionalität für unterschiedliche Zwecke und auf verschiedene Arten konsumiert und wiederverwendet werden kann.
(7) Die **einfache Austauschbarkeit** von Microservices ist ein bedeutender Vorteil. Aufgrund der kleinen Größe und Unabhängigkeit können einzelne Microservices mit geringem Aufwand ausgetauscht und beispielsweise durch eine neue Implementierung oder eine andere Technologie abgelöst werden.[83]

Nach Richards können drei weitere wesentliche Vorteile hinzugefügt werden:

(8) Durch die Anwendung des Microservice-Architekturstils kann im Gesamtsystemkontext eine **starke Modularität** erreicht werden. Je größer die Modularität einer Softwareanwendung ist, desto schneller können Änderungen vorgenommen, getestet und bereitgestellt werden.
(9) Aufgrund der Größe und Unabhängigkeit einzelner Microservices, verbessert sich außerdem die **Kontrolle von Änderungen**.
(10) Durch die starke Modularität wird auch die **Testbarkeit** positiv beeinflusst.[84]

[83] Vgl. ebd., S. 4ff.
[84] Vgl. Richards 2016, S. 38.

Auf Grundlage der zehn Vorteile können zwei übergeordnete strategische Ziele abgeleitet werden, die als Motivation für die Einführung von Microservices dienen können.

(1) Verbesserung der Agilität
(2) Nachhaltige Softwareentwicklung

Microservices können die Agilität verbessern, indem sie Unternehmen ermöglichen Produkte und Funktionalitäten schneller bereitzustellen und schneller auf veränderte Anforderungen zu reagieren.[85] Die erhöhte Geschwindigkeit resultiert folglich in einer verbesserten Produkteinführungszeit *(Time-to-Market)*. Weiterhin unterstützen Vorteile wie die hohe Modularität, leichte Ersetzbarkeit und Technologiefreiheit eine nachhaltige Softwareentwicklung, die Innovationen fördert und bei der die Produktivität des Entwicklungsteams, aufgrund des verminderten Risikos für das Auftreten von Architekturerosionen, weniger gefährdet ist.[86]

Die Vielzahl der Vorteile von Microservices impliziert jedoch nicht, dass sich dieser Architekturstil für jeden Einsatzzweck und -kontext eignet. Insbesondere die erhöhte Komplexität einer verteilten Anwendung erzeugt neue Probleme, die es zu beherrschen gilt und die in der Praxis oftmals unterschätzt werden.

Herausforderungen

Im Folgenden werden verschiedene Herausforderungen betrachtet, die mit dem Einsatz des Microservice-Architekturstils einhergehen.

(1) Im Microservice-Umfeld gilt es grundsätzlich die **Herausforderungen verteilter Systeme** zu berücksichtigen (vgl. Kapitel 2.2.1).
(2) Aufgrund der verteilten Datenhaltung wird auf Basis des *CAP-Theorems* häufig zugunsten der Verfügbarkeit und Ausfalltoleranz auf die vollständige Konsistenz der Daten verzichtet. Nach dem Konzept der ***Eventual Consistency*** wird die Konsistenz erst zu einem späteren Zeitpunkt sichergestellt.[87]
(3) In ausgeprägten Microservice-Architekturen mit einer Vielzahl an Services, die regelmäßig angepasst und bereitgestellt werden sollen, gilt es die **hohe Komplexität des Betriebes** zu beherrschen.[88]
(4) Auch die starke Modularisierung ist mit Nachteilen verbunden. Durch sie ist die nachträgliche **Restrukturierung *(Refactoring)*** und das Verschieben von Funktionalität zwischen verschiedenen Microservices häufig problematisch.[89]
(5) Microservice-Architekturen begünstigen **versteckte Beziehungen**, da die Aufrufe zwischen den Microservices nicht direkt erkennbar sind.[90]

---

[85] Vgl. Nadareishvili et al. 2016, S. 16.
[86] Vgl. Wolff 2016, Kap. 1.2 Abs. 1.
[87] Vgl. Carneiro und Schmelmer 2016, S. 13.
[88] Vgl. ebd., S. 13.
[89] Vgl. Wolff 2016, Kap. 1.2 Abs. 2.
[90] Vgl. ebd., Kap. 1.2 Abs. 2.

(6) Der **Entwurf der fachlichen Architektur** sollte deshalb mit besonderer Aufmerksamkeit erfolgen. Sie hat Auswirkungen auf die Organisation und die damit verbundene Struktur und Unabhängigkeit der Teams (vgl. Kapitel 2.3.1).[91]

(7) Für den Einsatz von Microservices ist somit eine **Anpassung der Organisation** erforderlich. Sie beinhaltet neben der Einführung kleiner, funktionsübergreifender Teams auch kulturelle Anpassungen wie z. B. die Einführung von DevOps-Praktiken (vgl. Kapitel 2.3.4).[92]

(8) Da Microservice-Architekturen für zahlreiche Organisationen ein neues Paradigma darstellen, benötigen die Anspruchsgruppen **neuartiges Wissen und neue Fähigkeiten** im Umgang mit Microservice-basierten Systemen. Neben entwicklungsbezogenen und betrieblichen Themen gilt dies vor allem für den Bereich der Domänenanalyse.[93]

Fachliche und technische Strukturierung

Eine bedeutende Eigenschaft von Microservices ist die Kapselung von Geschäftsfähigkeiten *(Business Capabilities)* und die damit verbundene fachliche Orientierung. Sie resultiert in einer fachlichen Schichtung, die durch die einzelnen Microservices repräsentiert wird (vgl. Kapitel 2.2.2). Die technischen Schichten gehen jedoch nicht verloren. Jeder Microservice beinhaltet jeweils alle für ihn relevanten technischen Aspekte. Gemäß den Eigenschaften von Microservices findet die Kommunikation zwischen den fachlichen Services durch leichtgewichtige Mechanismen statt. Abbildung 5 verdeutlicht diesen Sachverhalt in Ergänzung zu den in Kapitel 2.2.2 beschriebenen Zusammenhängen.

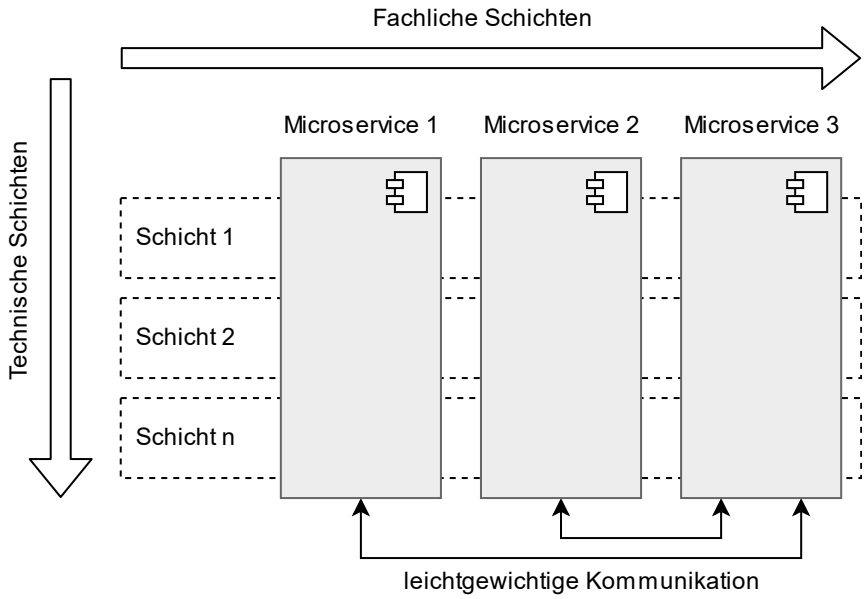

Abbildung 5: Technische und fachliche Schichtung bei Microservices

[91] Vgl. ebd., Kap. 1.2 Abs. 2.
[92] Vgl. Richards 2016, S. 39f.
[93] Vgl. Takai 2017, S. 22.

Das Prinzip der Schichtung in Microservice-basierten Systemen unterstützt den Leitgedanken, dass Softwaresysteme zuerst nach fachlichen Gesichtspunkten strukturiert werden sollten. Die Systeme werden dabei nach dem Prinzip der Trennung von Zuständigkeiten (*Separation of Concerns*) unterteilt (vgl. Kapitel 2.2.2). Eine Zuständigkeit im Kontext der fachlichen Schichtung kann als Teilaufgabe zur Erfüllung bestimmter fachlicher Anforderungen angesehen werden. Technische Strukturen werden erst innerhalb der jeweiligen Fachlichkeit bzw. auf niedriger Abstraktionsstufe gebildet.[94] Auf Ebene der Mikroarchitektur einzelner Microservices wird folglich der Architekturstil der Schichtenarchitektur zur technisch orientierten Strukturierung genutzt.

2.2.4 Abgrenzung von Microservices und Monolithen

Monolithische Systeme (vgl. Kapitel 2.2.1) und Microservice-basierte Systeme (vgl. Kapitel 2.2.3) basieren auf gegensätzlichen Konzepten und Zielen. Da sie kaum Gemeinsamkeiten besitzen, wird in diesem Kapitel ausschließlich auf die Unterschiede eingegangen.

Monolithen und Microservice-basierte Anwendungen sind durch unterschiedliche Eigenschaften geprägt. Microservices kennzeichnen ein Architekturstil für die Strukturierung verteilter Anwendungen, wohingegen eine monolithische Anwendung stark zentralisiert ist und keine verteilten Komponenten enthält. In der Regel korrelieren die Probleme und Herausforderungen des einen Ansatzes mit den Stärken und Vorteilen des anderen. Beispielsweise sind Monolithen im Gegensatz zu Microservices nicht mit den typischen Herausforderungen verteilter Systeme konfrontiert (vgl. Kapitel 2.2.1).

Abbildung 6 veranschaulicht die Aufteilung von vier unterschiedlichen Funktionalitäten in einem monolithischen System und einem Microservice-basierten System.

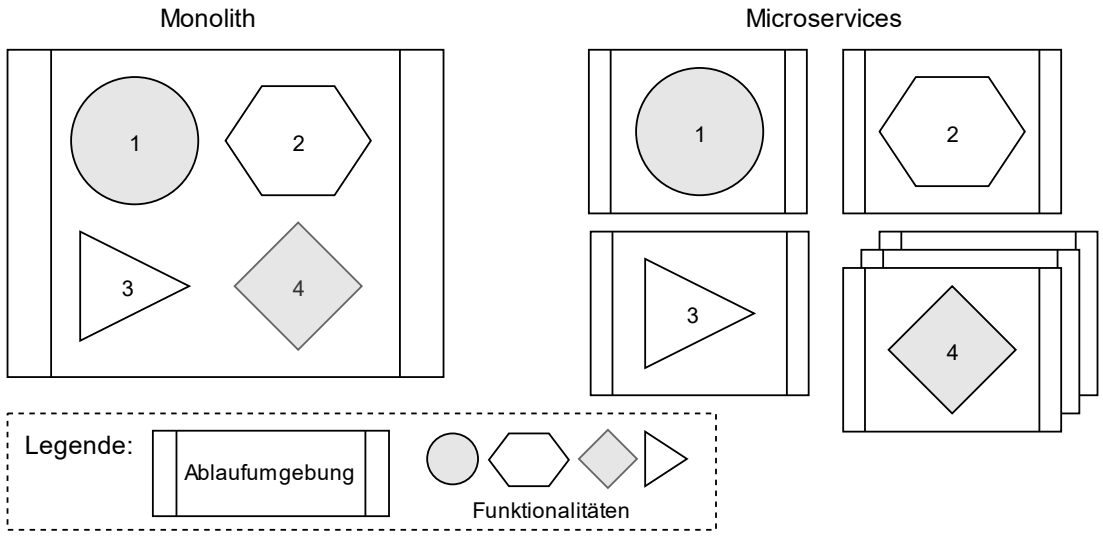

Abbildung 6: Funktionalität in Monolithen und Microservices [95]

[94] Vgl. Dowalil 2018, Kap. 2.7.
[95] In Anlehnung an: Starke 2018, S. 340.

Während in dem Monolithen die gesamte Funktionalität in einer *Laufzeitumgebung* vereint ist, sind die Funktionalitäten in der Microservice-Architektur in kleine, unabhängige Einheiten mit eigener Laufzeitumgebung verteilt.[96] Die vierte Funktionalität deutet zudem an, dass einzelne Microservices unabhängig skaliert werden können.

Neben der funktionalen Verteilung existieren noch weitere prägnante Unterschiede:

Die **Komplexität** einzelner Microservices ist im Vergleich zu monolithischen Systemen gering. Die Gesamtkomplexität des zu lösenden Problems bzw. der abgebildeten Geschäftslogik verringert sich durch Anwendung des Microservice-Architekturstils jedoch nicht. Vielmehr verteilt sich die Komplexität hin zur Ebene der Komposition einzelner Microservices.[97] Demzufolge ist die Komplexität in Monolithen nicht größer oder geringer, sondern zentralisiert in den monolithischen Strukturen.

Monolithen besitzen eine gemeinsame, meist große **Codebasis**. In Microservice-basierten Systemen besitzt jeder Service dahingegen seine eigene, unabhängige und kleine Codebasis. Weiterhin sind das **Verständnis und** die **Wartbarkeit** in Monolithen aufgrund der Größe und Struktur häufig negativ beeinflusst. Einzelne Microservices können durch ihre kleine Größe leichter verstanden und gewartet werden. Auch das **Deployment** komplexer monolithischer Anwendungen ist meist ineffizient und risikobehaftet, da sie nur als Ganzes bereitgestellt werden können. Im Gegensatz dazu kann jeder einzelne Microservice unabhängig und risikoarm bereitgestellt werden. Monolithen sind häufig auch an gewählte **Technologien und Programmiersprachen** gebunden, da ein Austausch die gesamte Anwendung betrifft und meist mit hohem Risiko verbunden ist. In Microservice-basierten Anwendungen können Technologien für jeden Service individuell gewählt und aufgrund der Größe und Unabhängigkeit leichter ausgetauscht werden.[98]

Die **Skalierung** monolithischer Anwendungen ist nur als Gesamtpaket möglich. In Microservice-Architekturen kann dagegen jeder Microservice einzeln und unabhängig skaliert werden.[99] Weiterhin erfolgt die **Kommunikation** in monolithischen Systemen für gewöhnlich über direkte Methodenaufrufe, wohingegen Microservices voneinander entkoppelt sind und ausschließlich über wohldefinierte Schnittstellen kommunizieren (vgl. Kapitel 2.2.3). Die **Datenhaltung** wird in Monolithen häufig über eine große, zentrale Datenbank realisiert. Microservices setzen dagegen auf eine dezentralisierte Datenhaltung, bei der jeder Service seine eigene Datenbank besitzt.[100]

Cloud-Umgebungen verfolgen Ziele wie z. B. die geografische Verteilung, schnelle Bereitstellung und Ressourcen-Effizienz. Aufgrund ihrer Komplexität und Struktur lassen sich Monolithen häufig nicht effizient genug ändern, bereitstellen und skalieren. Anwendungen auf Basis des Microservice-Architekturstils besitzen hingegen mehr Eigenschaften, die zu dem Betrieb in Cloud-Umgebungen passen.[101] Einzelne Microservices können

[96] Vgl. ebd., S. 339f.
[97] Vgl. Takai 2017, S. 22.
[98] Vgl. Daya 2015, S. 6.
[99] Vgl. Nadareishvili et al. 2016, S. 14.
[100] Vgl. Kalske et al. 2018, S. 37.
[101] Vgl. Kakivaya et al. 2018, S. 1.

schnell und unabhängig in die Cloud bereitgestellt werden und die Skalierungsvorteile ausnutzen.

In einem monolithischen System ist meist eine kleine Anzahl größerer Teams für jeweils umfangreichere Bereiche der Anwendung verantwortlich. Diese Bereiche überlagern und beeinflussen sich oft gegenseitig, sodass Konflikte zwischen den verantwortlichen Teams entstehen. Zur Sicherung der Qualität und Verfügbarkeit der Anwendung ist folglich ein erhöhtes Maß an Kommunikation- und Koordination notwendig. In einer Microservice-Architektur existieren viele kleinere Teams für eine große Anzahl von Services. Microservice-basierte Systeme zielen zudem darauf ab, dass sich die Verantwortlichkeiten einzelner Services nicht überlagern. Eine entsprechend angepasste **Organisationsstruktur** ermöglicht somit auch die klare Trennung von Zuständigkeiten und das unabhängige Arbeiten der einzelnen Teams.[102]

Tabelle A-2 (siehe Anhang) fasst die beschriebenen Unterschiede zusammen. Die aufgeführten Punkte kennzeichnen häufig auftretende Gegensätze zwischen Monolithen und Microservices. Sie stellen damit wesentliche Abgrenzungskriterien beider Ansätze dar, erheben jedoch keinen Anspruch auf Vollständigkeit.

2.3 Organisation und Kultur

Der Microservice-Architekturstil ist nicht nur ein Muster für die Architektur, er hat auch weitreichende Auswirkungen auf die Organisation. Ohne Anpassungen an der Organisation ist die Einführung und Nutzung des Microservices-Architekturstils nicht oder nur bedingt möglich. Microservice-Architekturen stehen demnach in enger Verbindung mit verschiedenen organisatorischen Aspekten.[103]
Dieses Kapitel gibt einen Überblick darüber, um welche organisatorisch und auch kulturell-geprägten Aspekte es sich dabei handelt und in welchem Bezug diese zum Microservice-Architekturstil stehen. Dazu gehören das Gesetz von Conway (Kapitel 2.3.1), der gedankliche Wandel von der Projektsicht zur Produktsicht (Kapitel 2.3.2), die Werte und Prinzipien der agilen Softwareentwicklung (Kapitel 2.3.3) sowie die Prinzipien und Praktiken des DevOps-Ansatzes (Kapitel 2.3.4).

2.3.1 Das Gesetz von Conway

In der Schrift mit dem Titel „How Do Committees Invent?“[104] begründete Melvin Conway bereits im Jahr 1968 eine Theorie, die bis heute Gültigkeit besitzt und auch im Rahmen der Betrachtungen zum Microservice-Architekturstil von hoher Relevanz ist. Nach Conway gilt das folgende Gesetz:

> *„Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations.“*[105]

---

[102] Vgl. Atchison 2016, S. 69ff.
[103] Vgl. Starke 2018, S. 337.
[104] Conway 1968.
[105] Ebd., S. 31.

Eine Organisation ist demnach nur in der Lage Systeme zu entwerfen, die die Kommunikationsstrukturen dieser Organisation abbilden. Änderungen an der Organisationsstruktur haben folglich Auswirkungen auf das Systemdesign. Diese Theorie bezieht sich nicht nur auf Softwaresysteme, sondern auf Systeme im Allgemeinen. Conways Ausführungen beschreiben außerdem, dass die Kommunikationsstrukturen einer Organisation nur bedingt flexibel sind. Die Flexibilität sinkt mit zunehmender Größe der Organisation, was die Auswirkungen auf den Systementwurf weiter verstärkt. Daher sollten selbst kleinere Organisationen versuchen die Kommunikation einzuschränken, um die Produktivität aufrecht zu erhalten.[106]

Nach Conway sollten jegliche Design-Tätigkeiten an dem Kommunikationsbedarf der Organisation ausgerichtet werden. Problematisch ist dabei, dass die Notwendigkeit zur Kommunikation selbst auch vom Systemdesign beeinflusst wird und das Design im Projektverlauf meist angepasst werden muss. Die Flexibilität und Anpassbarkeit einer Organisation ist daher entscheidend für die Umsetzung von guten und effektiven Systemdesigns.[107] An dieser Stelle kann ein Bezug zum Eingangszitat dieser Arbeit hergestellt werden (vgl. Kapitel 1.1). Die Realisierung guter Softwarearchitekturen wird nach dem Gesetz von Conway maßgebend von der Organisation beeinflusst.

Im Kontext von Softwarearchitekturen wird Conways Theorie durch Modelle wie z. B. das Modell vom *Architecture Influence Cycle* erweitert. Nach diesem Modell wird die Architektur eines Softwaresystems, analog zum Gesetz von Conway, durch bestimmte geschäfts- oder projektbezogene Faktoren der Organisation beeinflusst. Im Umkehrschluss können die Architektur und das entstandene System jedoch auch Rückwirkungen auf die Organisation haben. Eine Organisation kann die organisatorischen und erfahrungsbezogenen Auswirkungen der Entwicklung einer Architektur zur strategischen Ausrichtung der Organisations- und Kommunikationsstrukturen nutzen.[108]

Für die Abstimmung von Organisation und Architektur kann im weiteren Sinne zwischen zwei gegensätzlichen Strategien – dem *Conway Manöver* und dem *inversen Conway Manöver* – unterschieden werden. Beim Conway Manöver wird die Architektur bzw. das System so verändert, dass die Organisation passend abgebildet wird.[109] Die Organisation bestimmt folglich die Architektur. Analog dazu hat das IT-Beratungsunternehmen ThoughtWorks den Ansatz des inversen Conway Manövers definiert. Der Ansatz sieht vor die Organisationsstruktur so anzupassen, dass sie für die gewünschte Architektur des Systems förderlich ist. Im Idealfall bildet die angepasste bzw. optimierte Organisation dann die

[106] Vgl. ebd., S. 30f.
[107] Vgl. ebd., S. 31.
[108] Vgl. Bass et al. 2013, S. 57f.
[109] Vgl. Takai 2017, S. 115.

gleichen Strukturen ab, wie die Architektur.[110] Beim inversen Conway Manöver folgt die Organisation damit den strukturellen Vorgaben der gewünschten Architektur. Da die Flexibilität mit zunehmender Organisationsgröße sinkt, ist das inverse Conway Manöver in großen Unternehmen meist nur bedingt und langfristig umsetzbar.[111]

Aus dem inversen Conway Manöver ergeben sich zwei wesentliche Regeln:

(1) Für die Realisierung einer spezifischen Architektur wird eine daran ausgerichtete Organisation benötigt.
(2) Wenn die Architektur häufig angepasst wird, muss die Organisation ebenso häufig angepasst werden.[112]

Abbildung 7 stellt den gegenseitigen Einfluss von Organisationen und Architekturen im Kontext der zwei Strategien (Manöver) dar.

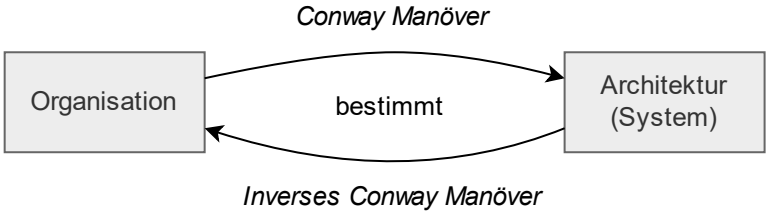

Abbildung 7: Einfluss von Organisation und Architektur

Was bedeutet dies nun für die Organisationsstrukturen in Microservice-Projekten?

Microservice-Architekturen basieren auf dem Zusammenspiel einer großen Anzahl kleiner und unabhängiger Microservices. Nach dem Gesetz von Conway sollte die Organisation für den Aufbau einer derartigen Architektur demnach aus vielen kleinen und unabhängigen Microservice-Teams bestehen.[113] Kommunikation und Abstimmung ist zwischen diesen Teams nur dann notwendig, wenn unterschiedliche Microservices miteinander kommunizieren müssen.[114] Ein Team ist dabei für ein oder mehrere Microservices verantwortlich, sofern die Services zum gleichen Kontext gehören (vgl. Kapitel 2.4.2). Abbildung 8 stellt diesen Zusammenhang als ER-Modell in Chen-Notation und mit [min., max.] -Kardinalitäten dar:

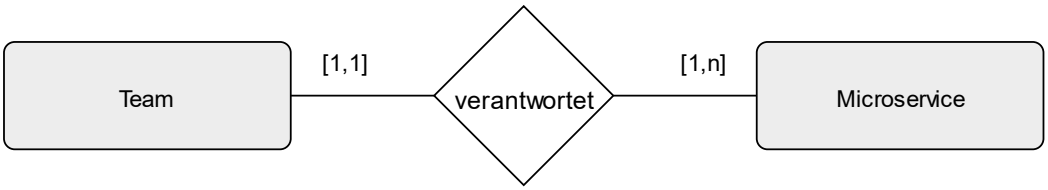

Abbildung 8: Teams in Microservice-Architekturen

[110] Vgl. ThoughtWorks Inc. 2015.
[111] Vgl. Takai 2017, S. 115.
[112] Vgl. DZone Inc. 2018, 22.
[113] Vgl. Fowler 2017, Kap. 1 Abschn. „Organizational Challenges".
[114] Vgl. Wolff 2016, Kap. 4.2.

Durch Anwendung des inversen Conway Manövers sollte die Organisation so aufgestellt werden, dass die Entwicklung der Microservice-Architektur unterstützt wird. In der Regel besitzt ein Unternehmen noch nicht die notwendigen organisatorischen Strukturen, wenn es beginnt erste Erfahrungen mit der Einführung des Microservice-Architekturstils zu sammeln. In diesem Fall ist die Anwendung des Conway Manövers, also die Anpassung der Architektur an die vorhandene Organisation, nur bedingt geeignet.
Das Ergebnis sollten fachlich orientierte Teams sein, die alle technischen Schichten innerhalb eines Microservice verantworten. Abbildung 9 zeigt am Beispiel eines Hotel-Buchungssystems[115], wie die Team-Organisation im Vergleich zu technisch orientierten Teams aussehen kann. Ausgehend von einer Organisation mit technisch ausgerichteten Organisationsstrukturen, repräsentiert der linke Bereich der Abbildung das Ergebnis eines durchgeführten Conway-Manövers. Die rechte Seite ist dagegen durch das inverse Conway-Manöver und die damit verbundene Anpassung der Organisation geprägt. Da es sich bei der Gästeverwaltung, Buchungsabwicklung und Zimmerverwaltung um spezielle Fachdomänen handelt, sind diese als abgegrenzte Kontexte anzusehen (vgl. Kapitel 2.4.2). Sie werden jeweils durch genau ein Microservice-Team umgesetzt.

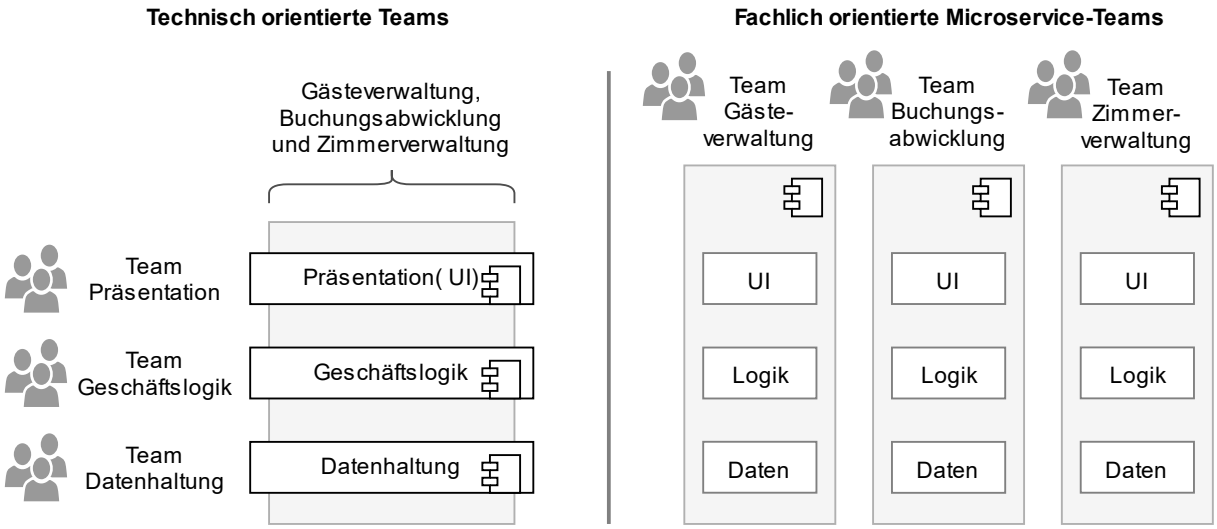

Abbildung 9: Technisch und fachlich orientierte Teams

2.3.2 Von der Projektsicht zur Produktsicht

Ein *Projekt* ist ein zeitlich begrenztes Vorhaben, um ein einzigartiges Produkt, Service oder Ergebnis zu schaffen. Es besitzt einen wohldefinierten Anfang und endet, wenn die Ziele des Projektes erreicht sind oder wenn das Projekt beendet wird, weil dessen Ziele nicht erreicht werden können oder keine Notwendigkeit mehr für das Projekt besteht. Die zeitliche Begrenzung eines Projektes bezieht sich generell nicht auf das Produkt, den Service oder das Ergebnis des Projektes. Das Resultat der meisten Projekte ist von fortwährender Natur.[116]

[115] In Anlehnung an: Oelmann 2018, Kap. 10.2.1 Abb. 10-4.
[116] Vgl. Project Management Institute 2008, 5.

Im Kontext von Softwareprojekten kann ein *Produkt* beschrieben werden als:

> *„ein klar definiertes Stück nutzbarer Software sowie alle notwendigen Artefakte und spezielle Hardware, die für den Betrieb dieser Software notwendig sind. Dabei ist es unerheblich, ob das Produkt autark lauffähig sein soll oder als Komponente nur im Zusammenspiel mit anderen Soft- und Hardwarekomponenten einsetzbar ist.“* [117]

Der unterschiedliche Zeithorizont von Projekten und Produkten bringt gewisse Folgen für die Architektur von Softwareprojekten mit sich. Steht das Projekt im Mittelpunkt, führt dies meist zu Architekturentscheidungen mit kurzfristigen Auswirkungen.[118]
Eine produktorientierte Ausrichtung der Organisation fördert hingegen die Entwicklung von Produkten mit wiederholbaren Prozessen und führt zu einem besseren Verständnis von Kundenbedarfen und nachhaltigen Geschäftsszenarien. Es entsteht ein gewisses Maß an Verantwortungsbewusstsein, das häufig zu besseren Geschäftsergebnissen führt.[119]

In Bezug auf Microservice-basierte Anwendungen sollte ebenfalls ein derartiges Umdenken stattfinden. Die Entwicklung und der Betrieb von Microservices sollte nicht in Form von Projekten organisiert werden, wobei die Entwicklung zum Projektende als abgeschlossen betrachtet wird und ein Wartungsteam die Software übernimmt. Vielmehr sollten die Microservice-Teams ihre Produkte bzw. Services über den gesamten Software-Lebenszyklus hinweg verantworten.[120] Amazon hat dahingehend bereits in frühen Zeiten servicebasierter Architekturen mit dem Leitgedanken „*You build it, you run it*“[121] einen Grundstein gelegt. Demnach kommt das Entwicklungsteam intensiv in Kontakt mit den betrieblichen Aufgaben der Software und dem Kunden. Dies ist von großer Bedeutung, da das regelmäßige Kundenfeedback ausschlaggebend für die Qualität des Produktes ist.[122]
In diesem Zusammenhang ist auch die enge Verbindung mit dem DevOps-Ansatz erkennbar (vgl. Kapitel 2.3.4).

Der Aspekt der Kontinuität, der sich in dem Produktgedanken wiederspiegelt, kann abschließend mit der Argumentation von Fowler und Lewis verdeutlicht werden:

> *„Die Produktmentalität ist mit der Aufteilung auf Business Capabilities verknüpft. Anstatt Software als eine Liste von Funktionen zu betrachten, die zu einem bestimmten Zeitpunkt fertig sind, geht es hier um eine kontinuierliche Beziehung.“* [123]

---

[117] Vigenschow 2015, S. 3.
[118] Vgl. Erder und Pureur 2016, S. 22.
[119] Vgl. ebd., S. 23.
[120] Vgl. Fowler und Lewis 2015, S. 15.
[121] O'Hanlon 2006, S. 16.
[122] Vgl. ebd., S. 16ff.
[123] Fowler und Lewis 2015, S. 15.

2.3.3 Agile Softwareentwicklung

Dieses Kapitel soll keinen Überblick über die Grundlagen der agilen Softwareentwicklung oder spezifische Methoden und Prozesse vermitteln. Es geht einzig auf relevante Werte und Prinzipien des agilen Ansatzes ein und dient der Beantwortung der Frage:

Inwiefern passen Microservices in die Welt der agilen Softwareentwicklung?

Das agile Manifest kann nach Tiemeyer folgendermaßen beschrieben werden:

> *„[…] die Menschen, die die Software entwickeln, [stehen] im Mittelpunkt des Manifests; sie sollen lauffähige Software produzieren und bei dieser Arbeit den Auftraggeber umfassend […] einbinden, um schnell und flexibel (agil) auf Anforderungsänderungen reagieren zu können.“*[124]

Das schnelle und flexible Reagieren auf Anforderungsänderungen ist auch das Ziel von Microservice-Architekturen und der damit in Verbindung stehenden Anpassung der Organisation (vgl. Kapitel 2.2.3 und 2.3.1). In Hinsicht auf die Flexibilität und Reaktionsfähigkeit passen Microservices als Architekturstil demnach gut zum agilen Manifest.

Das agile Manifest wird durch zwölf agile Prinzipien erweitert. Vier dieser Prinzipien können auf Aspekte der Architektur und Organisation zurückgeführt und durch den Microservice-Architekturstil direkt unterstützt werden:

(1) *„Die höchste Priorität besteht darin, den Kunden durch frühe und zahlreiche Lieferungen hochwertiger Software zufriedenzustellen.*
(2) *Funktionierende Software muss regelmäßig geliefert werden […].*
(3) *Mitarbeiter aus den Fachbereichen und Entwickler arbeiten täglich gemeinsam […].*
(4) *Die effizienteste und effektivste Methode zur Informationsübermittlung für und innerhalb eines Entwicklungsteams besteht in der direkten Kommunikation.“*[125]

Microservices haben das Ziel, dass Änderungen an einzelnen Services schnell und unabhängig umgesetzt und geliefert werden können (vgl. Kapitel 2.2.3). Die Lieferung von Produktinkrementen erfolgt dabei in Zyklen, sodass diese früh und häufig für den Anwender ausgeliefert werden können.[126] Entwickler und Anspruchsgruppen aus dem Fachbereich stehen währenddessen im engen Kontakt und stimmen sich regelmäßig ab (vgl. Kapitel 2.3.2). Die Mitglieder des Entwicklungsteams können dabei direkt und effizient miteinander kommunizieren, was der Größe und Unabhängigkeit einzelner Microservice-Teams zuzuschreiben ist (vgl. Kapitel 2.3.1). Der Microservice-Architekturstil und eine daran ausgerichtete Organisation unterstützen folglich alle vier agilen Prinzipien und können zur Skalierung agiler Prozesse und Methoden beitragen. Darüber hinaus ist die Verbesserung der Agilität ein strategisches Ziel von Microservices (vgl. Kapitel 2.2.3).

---

[124] Tiemeyer 2014, S. 75.
[125] Ebd., S. 78f.
[126] Vgl. Gruhn 2018, S. 52.

2.3.4 DevOps

Die *Entwicklung* und der *Betrieb* bezeichnen traditionell zwei getrennte Abteilungen einer Organisation, die unterschiedliche Aufgaben und Ziele verfolgen und verschiedene Sichtweisen auf eine Softwareanwendung einnehmen. Grundsätzlich soll die Entwicklung die erforderlichen Änderungen an der Software umsetzen und der Betrieb die Stabilität der produktiven Softwareanwendung sicherstellen. Diese unterschiedlichen Aufgaben- und Zielstellungen führen häufig zu Konflikten zwischen beiden Organisationseinheiten. Das Konfliktpotential ist zusätzlich erhöht, wenn beim Übergang von der Entwicklung in den Betrieb, also dem Deployment, kombiniertes Wissen aus beiden Bereichen benötigt wird.[127] Im agilen Umfeld wird dieser Konflikt weiter durch das agile Prinzip verschärft, dass funktionierende Software regelmäßig geliefert werden sollte (vgl. Kapitel 2.3.3).

Das Kunstwort *DevOps* setzt sich aus den englischsprachigen Begriffen *development* und *operations* zusammen und ist im Jahr 2009 auf einer Konferenz in Belgien entstanden. DevOps bezeichnet in erster Linie eine Organisationsform, die, entgegen der klassischen Trennung, die enge Zusammenarbeit von Entwicklung (Dev) und Betrieb (Ops) als Ziel hat. Diese Bereiche sollen im Rahmen von fachlich aufgeteilten Komponenten und Verantwortlichkeiten – den Microservices – zu einer Einheit zusammenwachsen.[128]

Amazon hat bereits im Jahr 2006, im Rahmen früher Microservice-artiger Bestrebungen, Pionierarbeit im Sinne von DevOps betrieben. Das Unternehmen setzte auf kleine Teams, die Wissen und Kompetenzen aus den Bereichen Entwicklung und Betrieb vereinten und die jeweils für einen fachlichen Service verantwortlich waren. So war jedes Team in der Lage seinen Service sowohl für die leichte Weiterentwicklung als auch den effizienten und stabilen Betrieb zu optimieren.[129]

Nach Sharma et al. sind für die erfolgreiche Umsetzung von DevOps sowohl Änderungen an der Kultur und den Prozessen als auch der Architektur notwendig.[130]

Eine zielorientierte Definition für DevOps liefern Bass et al.:

> „*DevOps is a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production, while ensuring high quality.*“[131]

---

[127] Vgl. Wolff 2015, Kap. 11.2.
[128] Vgl. ebd., Kap. 11.2.
[129] Vgl. ebd., Kap. 11.2.
[130] Vgl. Sharma et al. 2016, S. 269.
[131] Bass et al. 2015, Kap. 1.1 Abschn. „Defining DevOps“.

Neben dem organisatorischen und kulturellen Aspekt wird unter DevOps demnach auch eine *Menge von Praktiken* verstanden, die das Ziel haben Änderungen an Softwaresystemen schneller in Produktion zu bringen und dabei eine hohe Qualität zu gewährleisten. Diese Definition rückt die Kerngedanken von DevOps – *Geschwindigkeit und Qualität* – in den Vordergrund. Jegliches Vorhaben, das die Erreichung dieser Ziele unterstützt und fördert, ist demnach eine DevOps-Praktik, unabhängig davon auf welchen Methoden oder Werkzeugen sie beruht.[132]

Die Definition offenbart auch die enge Verbindung von DevOps und *Continuous Delivery*. Continuous Delivery bezeichnet ein technisches Konzept zur schnellen Auslieferung von Änderungen. DevOps ist dahingegen ein organisatorisches Konzept, das das gleiche Ziel verfolgt und daher gut mit Continuous Delivery harmoniert.[133] Continuous Delivery ist eine der bedeutendsten DevOps-Praktiken zur Unterstützung automatisierter Deployments. Eine weitere Praktik ist das *Continuous Monitoring*. Sie sieht im Kern die kontinuierliche Überwachung des Betriebs zur frühen Erkennung von Unregelmäßigkeiten vor.[134] Aufbauend auf Continuous Delivery wird auch *Continuous Deployment* als DevOps-Praktik beschrieben. Dabei werden Änderungen automatisiert und ohne Koordinationsaufwand bis in die Produktionsumgebung bereitgestellt. Bis dahin müssen verschiedene andere Umgebungen durchlaufen werden. Deshalb zählt auch das *Konfigurationsmanagement* zur effizienten Bereitstellung von Umgebungen zu den DevOps-Praktiken.[135] Der *Einsatz agiler Praktiken und Entwicklungsmethoden* kann ebenso den DevOps-Praktiken zugeschrieben werden (vgl. Kapitel 2.3.3).[136] Zusammengefasst ergeben sich vier Beispiele für bedeutende DevOps-Praktiken, die stark auf die Aspekte der Automatisierung und Effizienz ausgerichtet sind:

(1) Continuous Delivery / Continuous Deployment
(2) Continuous Monitoring
(3) Konfigurationsmanagement
(4) Einsatz agiler Praktiken und Methoden

Die vier aufgeführten DevOps-Praktiken kennzeichnen Prozesse und Methoden auf höherer Abstraktionsebene. Generell können noch weitere DevOps-Praktiken in detaillierter Form aufgeführt werden. Beispielsweise können die Praktiken des automatisierten Testens und des automatisierten Deployments den Prozessen Continuous Delivery und Deployment zugeschrieben werden. Weiterhin kann die automatisierte Bereitstellung von Infrastruktur mit dem Bereich des Konfigurationsmanagements in Verbindung gebracht werden.[137]

---

[132] Vgl. ebd., Kap. 1.1 Abschn. „Defining DevOps“.
[133] Vgl. Wolff 2015, Kap. 11.1.
[134] Vgl. Cerny et al. 2017, S. 32.
[135] Vgl. Zhu et al. 2016, S. 33.
[136] Vgl. Sharma et al. 2016, S. 609.
[137] Vgl. ebd., S. 269.

2.4 Domain-Driven Design

Zum Verständnis komplexer Softwaresysteme müssen dessen Ziele, Aufgaben und Umfeld genau verstanden werden. Es ist daher von Bedeutung ein gemeinsames, fachliches Verständnis über die Domäne zu erlangen und die geschäftliche und technische Seite aufeinander abzustimmen. Dabei kann eine gemeinsame Fachsprache zur effektiven Analyse und Entwicklung eines Softwaresystems genutzt werden.[138] Diese Zusammenhänge spiegeln die Grundgedanken von Domain-Driven Design wieder.

Domain-Driven Design (DDD) wurde von Eric Evans erstmals in seinem Buch *Domain-Driven Design, Tackling Complexity in the Heart of Software*[139] beschrieben. DDD ist ein Software-Entwicklungsansatz, der die effektive Entwicklung und Wartung von Softwarelösungen innerhalb komplexer Problemdomänen ermöglicht.[140]
DDD stellt eine Sammlung von Entwurfsprinzipien dar und hat damit grundlegenden Einfluss auf die Struktur einer Softwareanwendung, also den Komponenten und dessen Beziehungen zueinander (vgl. Kapitel 2.1.2). Der domänengetriebene Entwurf nach DDD kann daher als Architekturstil beschrieben werden. Weiterhin kann DDD auch als Mustersprache bezeichnet werden, da sich die Anwendung der zusammenhängenden Muster und Prinzipien auf die gesamte Codebasis ausdehnt.[141] Dies gilt jedoch nur auf Ebene der Mikroarchitektur bzw. für den Bereich des taktischen Designs (vgl. Kapitel 2.4.3).[142]

Domain-Driven Design unterscheidet das strategische und das taktische Design. Nach Evans sollten beide Bereiche kombiniert werden, um Softwareprojekte zum Erfolg zu führen.[143] Domain-Driven Design definiert folglich einen Werkzeugsatz aus strategischen und taktischen Mustern, dessen konsequente Anwendung den Entwurf und die Implementierung hochqualitativer Software unterstützt. Aus Unternehmenssicht dienen die strategischen Werkzeuge dem Entwurf von Softwaremodellen, die sich zur Erreichung von Wettbewerbsvorteilen auf die Kernkompetenzen der Organisation konzentrieren und auf optimalen Entwurfs- und Integrationsentscheidungen beruhen. Die taktischen Werkzeuge wiederrum setzen auf niedriger Abstraktionsebene des Softwareentwurfs an, indem sie das notwendige Vorgehen zur Umsetzung der Strategie abbilden.[144]
Abbildung A-2 (siehe Anhang) zeigt alle Werkzeuge des Domain-Driven Designs in einer Übersicht. Für die vorliegende Arbeit ist jedoch nur eine bestimmte Auswahl dieser Werkzeuge relevant (vgl. Kapitel 2.4.2 und 2.4.3).

---

[138] Vgl. Takai 2017, S. 3.
[139] Evans 2003.
[140] Vgl. Millett und Tune 2015, S. 3.
[141] Vgl. Lilienthal 2017, Kap. 6.5.
[142] Vgl. ebd., Kap. 6.5.2.
[143] Vgl. Evans 2015, S. 1.
[144] Vgl. Vernon et al. 2017, S. 1.

2.4.1 Modellierung von Domänen

> *„Modelle sind vereinfachte Darstellungen (Abstraktionen) der Realität. In komplexen Situationen oder Sachverhalten helfen Modelle, weil wir darin gezielt bestimmte Details der Realität aus- oder einblenden, um die Menge gezeigter Informationen zu kontrollieren.“*[145]

Das Hilfsmittel der Modellierung kann in der Softwarearchitektur verwendet werden, *„um die inhaltliche Komplexität fachlicher Sachverhalte zu reduzieren und [sich] auf die für eine IT-Umsetzung relevanten Teile zu fokussieren“*[146]. Neben der Darstellung fachlicher Situationen können sie außerdem zur Darstellung von Beziehungen und Zusammenhängen zwischen Systemen und Systemkomponenten eingesetzt werden.[147] Analog dazu können Modelle von Domänen genutzt werden, um die Komplexität fachlicher Sachverhalte einer Domäne zu strukturieren und deren Umsetzung im Softwaresystem zu unterstützen.

Eine *Domäne* ist ein Bereich, in dem bestimmtes Wissen, bestimmte Aktivitäten oder Einflüsse vorherrschen.[148] Ein *Domänenmodell* ist nach Evans ein *„Abstraktionssystem, das ausgewählte Aspekte einer Domäne beschreibt und dazu verwendet werden kann, Probleme in Bezug auf diese Domäne zu lösen“*.[149] Ein Domänenmodell beinhaltet daher das strukturierte und organisierte Wissen über das zu lösende Problem.[150] Fowler und Rice beschreiben ein Domänenmodell als Objektmodell einer Domäne, das Verhalten und Daten abbildet. Es setzt jene Objekte netzartig miteinander in Verbindung, die sowohl Daten als auch Geschäftsregeln erfassen.[151] Ein objektorientiertes Domänenmodell sieht einem Datenmodell häufig ähnlich, ist aber dennoch unterschiedlich. Es vermischt Daten und Prozesse, beinhaltet oft komplexe Assoziationen und nutzt das Prinzip der Vererbung. Generell nutzt ein objektorientiertes Domänenmodell diese Prinzipien umso ausgiebiger, je komplexer die Domänenlogik ist. Mit der zunehmenden Komplexität werden die Unterschiede zum Datenmodell offensichtlich. Das Domänenmodell lässt sich dann jedoch auch schwieriger auf die Daten abbilden.[152]

Subdomänen sind ein Konzept des Domain-Driven Designs, um die Komplexität von Domänen weiter zu unterteilen. Es handelt sich um Teilbereiche, die ein bestimmtes Fachgebiet, dessen Wissen, Aktivitäten und Einflüsse betrachten.[153] DDD unterscheidet drei

---

[145] Starke 2018, S. 185.
[146] Ebd., S. 185.
[147] Vgl. ebd., S. 186.
[148] Vgl. Evans 2003, S. 328.
[149] Übersetzt aus: ebd., S. 329.
[150] Vgl. Ariff und Steffens 2018, S. 27.
[151] Vgl. Fowler und Rice 2002, S. 116.
[152] Vgl. ebd., S. 116f.
[153] Vgl. Vernon et al. 2017, 141 (Glossar).

Arten von Subdomänen: die *Kerndomäne*, die *unterstützende Teildomäne* und die *allgemeine Teildomäne*. Die Kerndomäne ist der Teil der Domäne, der für die strategischen Ziele des Unternehmens von zentraler Bedeutung ist. Sie spiegelt den Wert der Softwareanwendung wieder und hebt sie vom Wettbewerb ab.[154] Zur Kerndomäne gehören demnach die Bereiche des Domänenmodells, die unverwechselbar und von zentraler Bedeutung für die Zwecke der beabsichtigten Softwareanwendung sind.[155] Eine Organisation sollte daher den größten Entwicklungsaufwand in die Kerndomäne investieren und diese genau abgrenzen. Unterstützende Teildomänen wirken unterstützend auf die Kerndomäne und müssen ebenfalls individuell entwickelt werden. Allgemeine Teildomänen sind strategisch bedeutungslos. Sie können durch Standardsoftware umgesetzt werden und sollten in Hinsicht auf die Entwicklung die geringsten Investitionen erhalten.[156]

Auf Modellebene besitzen komplexe Monolithen meist keine klar definierten Grenzen und vermischen diverse Modelle miteinander. Dabei verteilen sich verschiedene Konzepte über unterschiedliche Module und stehen in widersprüchlichen Beziehungen zueinander. Infolgedessen wird die Domänenlogik oft undurchsichtig in der Anwendung umgesetzt.[157] Die Nutzung der Muster und Prinzipien des Domain-Driven Designs kann dahingehend Abhilfe schaffen.

2.4.2 Strategisches Design

Das *strategische Design* betrachtet die strategisch für das Geschäft bedeutenden Bereiche und versucht die wesentlichen Aspekte zusammenzufassen und zu priorisieren.[158] Nach Evans umfasst das strategische Design die Modellierung sowie sämtliche Entwurfsentscheidungen der großen Systemteile. Es bezieht sich folglich auf die Ebene der *Makroarchitektur* (vgl. Kapitel 2.1.2) und hat starken Einfluss auf die gesamte Softwarelösung.[159] Die hierbei getroffenen Architekturentscheidungen können nur mit großem Aufwand rückgängig gemacht werden (vgl. Kapitel 2.1.1).

Das Herausfiltern der Kerndomäne ist ein zentrales Ziel des strategischen Designs. Evans spricht von strategischer Destillation des Domänenmodells, bei der das Modell mit zunehmender Erfahrung und Wissen über die Domäne verfeinert wird. Dadurch ergeben sich verschiedene Vorteile:

- Besseres Verständnis über die gesamte Architektur
- Erleichterte Kommunikation durch eine überschaubare Kerndomäne und gemeinsame Sprache
- Unterstützung von Restrukturierungen *(Refactoring)*
- Konzentration von Aufwänden auf die wertvollen Bereiche des Domänenmodells[160]

---

[154] Vgl. Evans 2003, S. 327; Vernon et al. 2017, 140 (Glossar).
[155] Vgl. Evans 2003, S. 258.
[156] Vgl. Vernon et al. 2017, S. 44f.
[157] Vgl. ebd., S. 16.
[158] Vgl. ebd., S. 7.
[159] Vgl. Evans 2003, S. 329.
[160] Vgl. ebd., S. 256.

Zwei bedeutende Konzepte des strategischen Designs sind die *Ubiquitous Language* und der *Bounded Context*. Ein Bounded Context bezeichnet eine „*semantische Grenze, innerhalb derer die Elemente des Softwaremodells eine bestimmte, konsistente Bedeutung haben*“[161]. Die Ubiquitous Language ist die gemeinsame und allgegenwärtige Sprache von Domänenexperten und Softwareentwicklern, die sowohl als Kommunikationsgrundlage zwischen Teammitgliedern dient als auch in der Software umgesetzt wird.[162] Ein Begriff kann in verschiedenen Teams unterschiedliche Bedeutungen haben, da sie in unterschiedlichen Fachkontexten tätig sind bzw. diese verantworten. Eine Komponente außerhalb eines Kontextes muss daher nicht der gleichen Definition folgen, die für einen Begriff innerhalb dieses Kontextes festgelegt wurde.[163] Ein Bounded Context kann demnach als sprachliche Begrenzung verstanden werden. Abbildung 10 stellt diesen Sachverhalt dar.

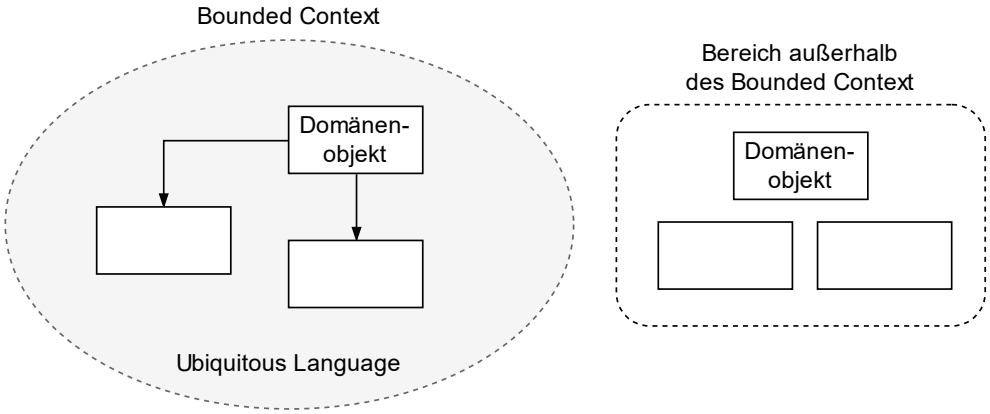

Abbildung 10: Bounded Context und Ubiquitous Language [164]

Ein Bounded Context ist genau einem Team zugeordnet und ein Team kann mehrere Kontexte verantworten. Um Konflikte zu vermeiden, sollte ein Bounded Context jedoch nicht von mehreren Teams verantwortet werden. Innerhalb eines Bounded Context existiert eine gemeinsame Sprache, d. h. eine Ubiquitous Language gehört zu genau einem Bounded Context. Ein Team, das an einem Kontext arbeitet, verwendet folglich die dafür vorgesehene Sprache. Außerdem sollten der Quellcode und die Daten für jeden Bounded Context sauber abgegrenzt werden. Sie gehören genau einem Team, das über wohldefinierte Schnittstellen festlegt, wie der zugehörige Bounded Context verwendet werden darf.[165] Dieses strikte Prinzip spiegelt die Abgeschlossenheit und Unabhängigkeit von Microservices wieder (vgl. Kapitel 2.2.3).

[161] Vernon et al. 2017, 140 (Glossar).
[162] Vgl. ebd., 141 (Glossar).
[163] Vgl. ebd., S. 14f.
[164] In Anlehnung an: ebd., S. 14.
[165] Vgl. ebd., S. 14.

Ein Bounded Context kann als Basis für die Implementierung von einem oder mehreren Microservices dienen. Bestimmte technische Anforderungen können beispielsweise dazu führen, dass ein Bounded Context von mehr als einem Microservice umgesetzt wird. Die Funktionalität bzw. Fachlichkeit eines Microservice sollte sich jedoch nicht über die Grenzen eines Bounded Context hinaus erstrecken.[166]

In Abbildung 11 werden die Zusammenhänge von Teams, Bounded Contexts und der Ubiquitous Language aufgezeigt. Die Darstellung basiert auf einem ER-Modell in Chen-Notation mit [min., max.] -Kardinalitäten.

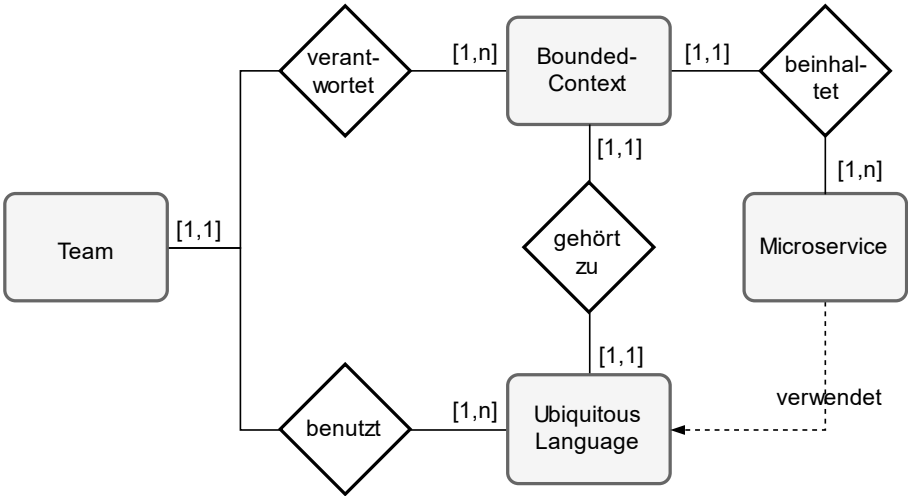

Abbildung 11: Relation von Team, Kontext, Sprache und Microservices

Ein weiteres bedeutendes Konzept des strategischen Designs ist die *Kontextlandkarte (Context Map)*. In einer Kontextlandkarte können unterschiedliche Bounded Contexts und deren Beziehungen untereinander abgebildet werden.[167] Weiterhin wird die Integration unterschiedlicher Kontexte im Domain-Driven Design durch verschiedene Muster unterstützt: *Shared Kernel, Customer/Supplier, Conformist, Anticorruption Layer, Separate Ways, Open Host Service* und *Published Language* (vgl. Abbildung A-2; Anhang). Da diese Konzepte nicht von zentraler Bedeutung für die vorliegende Arbeit sind, werden sie hier nicht näher beschrieben.

[166] Vgl. Wolff 2016, Kap. 4.3.
[167] Vgl. Vernon et al. 2017, 140 (Glossar).

2.4.3 Taktisches Design

Das *taktische Design* kennzeichnet eine Disziplin des Domain-Driven Designs, die sich mit der Modellierung von Domänenmodellen auseinandersetzt. Sie verfolgt das Ziel, die Komplexität der Domäne beherrschbar zu machen und verschiedene Qualitätsmerkmale von Domänenmodellen, wie z. B. die Klarheit und Verständlichkeit, sicherzustellen. Im taktischen Design werden dafür verschiedene Bausteine *(Building Blocks)* definiert. Sie stellen eine Menge von Mustern für die Umsetzung von möglichst nützlichen und aussagekräftigen Domänenmodellen bereit. Diese Muster basieren auf objektorientierten Prinzipien und dienen dem Zweck die Bedeutungen, die Beziehungen und die Geschäftslogik innerhalb einer Domäne abzubilden.[168]

Der größte Wert von Domain-Driven Design liegt vorwiegend in den Mustern des strategischen Designs begründet.[169] Sie beziehen sich auf sämtliche Entwurfsentscheidungen, die strategisch bedeutend sind und nur schwer rückgängig gemacht werden können (vgl. Kapitel 2.1.1 und 2.4.2). Im Gegensatz dazu bezieht sich das taktische Design in erster Linie auf die Ebene der Mikroarchitektur, also die Implementierungsebene einzelner Microservices (vgl. Kapitel 2.1.3). Der Bereich des taktischen Designs bildet daher im Gegensatz zum strategischen Design nicht den Schwerpunkt dieser Arbeit. Die im Rahmen der Arbeit relevanten Bausteine des taktischen Designs werden nachfolgend zusammengefasst:

- **Domänenereignisse *(Domain Events)*** repräsentieren Geschehnisse und Informationen über Aktivitäten innerhalb einer Domäne.
- **Entitäten *(Entities)*** sind Objekte, die eine eigene Identität besitzen. Sie werden anhand ihrer Identität und nicht nach ihren Attributen unterschieden.
- **Wertobjekte *(Value Objects)*** besitzen keine eigene Identität. Sie werden anhand ihrer Attribute und Logik unterschieden.[170]
- **Aggregate *(Aggregates)*** sind Domänenobjekte, die zusammengehörige Entitäten und Wertobjekte gruppieren. Sie enthalten eine Wurzelentität, die als einzige Schnittstelle für Objekte außerhalb des Aggregates fungiert. Ein Aggregat bildet die Transaktionsgrenze in einem Domänenmodell.[171] Weiterhin spiegeln Aggregate in einer Domäne das Verhalten und die Daten von Objekten wieder.[172]

[168] Vgl. Millett und Tune 2015, S. 310.
[169] Vgl. ebd., S. 123; Evans und Wolff 2015, S. 14.
[170] Vgl. Evans 2015, S. 11ff.
[171] Vgl. ebd., S. 16.
[172] Vgl. Takai 2017, S. 23.

2.5 Zusammenhänge und Schlussfolgerungen

Dieses Kapitel dient dem Zweck, entscheidende Zusammenhänge aus den vorigen Grundlagenkapiteln herauszustellen und wesentliche Schlussfolgerungen zu ziehen. Es soll einen Bogen um bedeutende Aspekte spannen und zu einem besseren Gesamtverständnis der Thematik führen. Als Grundlage dienen primär das Kapitel 2.2.3 über Microservice-Architekturen sowie die Themenbereiche der Organisation und Kultur aus Kapitel 2.3.

Die gezielte Entwicklung von Softwareanwendungen für die Cloud ist nicht Teil des direkten Untersuchungsumfangs dieser Arbeit. Die Prinzipien des *Cloud-Native* Ansatzes vermitteln jedoch grundlegende Zusammenhänge, die für diese Arbeit und im Kontext von Microservice-Architekturen von Bedeutung sind. Auf dieser Basis werden in Kapitel 2.5.2 wesentliche Einflussfaktoren und Abhängigkeitsbeziehungen herausgestellt.

2.5.1 Cloud-Native Anwendungsentwicklung

Cloud-Technologien als Grundlage

Nach der Definition des National Institute of Standards and Technology (NIST) ist Cloud-Computing:

> *„ein Modell, das es erlaubt jederzeit, bequem und bei Bedarf über ein Netzwerk auf eine geteilte Ansammlung konfigurierbarer Rechenressourcen (z.B. Netzwerke, Server, Speicher, Anwendungen und Services) zuzugreifen, die schnell und mit minimalem Managementaufwand oder Eingriff eines Serviceanbieters bereitgestellt werden können.“*[173]

Weiterhin ist Cloud-Computing durch fünf essenzielle Eigenschaften gekennzeichnet:

(1) Konsumenten können sich selbst und unabhängig mit den benötigten IT-Ressourcen versorgen *(on-demand self-service)*.
(2) Über ein Netzwerk kann von überall auf IT-Ressourcen zugegriffen werden *(broad network access)*.
(3) Konsumenten teilen sich einen großen, vereinheitlichten Pool aus IT-Ressourcen, die ihnen dynamisch und nach Bedarf zugewiesen werden *(resource pooling)*.
(4) Unter dem Aspekt der optimierten Ressourcen-Nutzung können IT-Ressourcen elastisch skaliert und bedarfsorientiert bereitgestellt werden *(rapid elasticity)*.
(5) Die Nutzung der IT-Ressourcen wird gemessen und überwacht, was für die Abrechnung von Bedeutung ist und Transparenz schafft *(measured service)*.[174]

[173] Übersetzt aus: Mell und Grance 2011, S. 2.
[174] Vgl. Fehling et al. 2014, S. 4f.

Aus der Perspektive eines Unternehmens können sich durch die Nutzung von Cloud-Technologien verschiedene Vorteile ergeben. Dazu zählen beispielsweise die Kostenreduzierung durch bedarfsorientierte Nutzung von Kapazitäten, die kürzere Produkteinführungszeit durch unmittelbare Bereitstellung von IT-Ressourcen, die Förderung der Innovationskraft durch niedrigere IT-Barrieren, die hohe Flexibilität durch leichte Skalierbarkeit von Services oder das Ermöglichen neuer Arten von Anwendungen und Services.[175]

Cloud-Native Anwendungen

Eine *Cloud-Native Anwendung* ist eine Anwendung, die mit dem Ziel entworfen wird die Vorteile von Cloud-Technologien möglichst umfassend auszunutzen.[176] Der Begriff Cloud-Native beschreibt demnach einen Ansatz zur Umsetzung von Softwareanwendungen für den Betrieb in der Cloud und unter Ausnutzung der Vorteile von Cloud-Technologien.

Nach Wilder besitzt eine Cloud-Native Anwendung mehrere kennzeichnende Eigenschaften. Sie nutzt Cloud-Technologien für eine zuverlässige und skalierbare Infrastruktur und setzt auf lose gekoppelte Architekturen. Neben der automatisierten und bedarfsorientierten Skalierung kann sie kostenoptimiert und ohne Ausfallzeiten oder Einschränkungen der Nutzererfahrung betrieben werden.[177] Eine Cloud-Native Anwendung kann zudem mit hoher Geschwindigkeit entwickelt werden, sie ist jederzeit und von überall zugänglich und fördert dadurch die Experimentier- und Innovationsfreudigkeit.[178]

Es lassen sich fünf Motivationsfaktoren von Cloud-Native Anwendungen ableiten:

(1) Lose Kopplung
(2) Elastische Skalierbarkeit
(3) Robustheit
(4) Geschwindigkeit
(5) Zugänglichkeit

Von traditioneller zur Cloud-Native Anwendungsentwicklung

Der Übergang von der traditionellen in die Cloud-Native Anwendungsentwicklung ist mit Herausforderungen verbunden. Nach Stine sind umfangreiche Änderungen in den drei Bereichen der (1) *Kultur*, (2) *Organisation* und (3) *Technologie* notwendig, die unter dem übergreifenden Aspekt der Dezentralisierung erfolgen.[179]

---

[175] Vgl. Avram 2014, S. 531f.
[176] Vgl. Wilder 2012, S. 10.
[177] Vgl. ebd., S. 10.
[178] Vgl. Stine 2015, S. 13f.
[179] Vgl. ebd., S. 15ff.

(1) Zum Kulturwandel zählt in erster Linie die Dezentralisierung von Fähigkeiten siloartig organisierter Teams in funktionsübergreifende Teams nach dem *DevOps*-Gedanken (vgl. Kapitel 2.3.4). *Continuous Delivery* ist dabei eine der bedeutendsten DevOps-Praktiken für die Dezentralisierung und Automatisierung von Bereitstellungsprozessen. In kultureller Hinsicht ist auch ein Wandel von zentraler Steuerungs- und Entscheidungsgewalt zu *dezentralisierter Autonomie* und Verantwortung notwendig.[180]

(2) Auf organisatorischer Ebene schlägt sich der Aspekt der Dezentralisierung in der Orientierung an zwei primären Teamstrukturen nieder. *Business Capability Teams* sind nach Geschäftsfunktionen aufgeteilt, vereinen ein breites funktionsübergreifendes Spektrum an Fähigkeiten und können unabhängig agieren. Die *Platform Operations Teams* versorgen die Business Capability Teams dabei mit den Werkzeugen, die sie zum Betrieb der Services benötigen.[181]

(3) Für die Dezentralisierung auf technologischer Ebene sollten monolithische Anwendungen in Microservice-basierte Anwendungen überführt werden, um die Verantwortung für einzelne Geschäftsfähigkeiten auf unabhängige Services zu verteilen. Das Konzept der Bounded Contexts (vgl. Kapitel 2.4.2) teilt konsistente Teilmengen der Geschäftsdomäne dabei auf entsprechende Microservices auf. Weiterhin ermöglichen Container-Technologien schlanke Deployment-Einheiten, die im Verantwortungsbereich einzelner Business Capability Teams liegen. Ein weiterer Punkt ist der Übergang von der Orchestrierung zur Choreografie. Die Service-Komposition mittels Choreografie impliziert die Verteilung der Integrationslogik in die Service-Schnittstellen. Sie sorgt für eine lose Kopplung zwischen den Services und Autonomie der Microservice-Teams.[182]

Tabelle 4 fasst die Anforderungen zur Erreichung der angestrebten Dezentralisierung und Autonomie in einer Übersicht zusammen:

Kultureller Wandel	**Organisatorischer Wandel**	**Technischer Wandel**
- DevOps - Continuous Delivery - Autonomie (dezentrale Entscheidungen)	- Business Capability Teams - Platform Operations Teams	- Microservices - Bounded Contexts - Container-Technologien - Choreografie

Tabelle 4: Anforderungen von Cloud-Native Anwendungen

Die Umsetzung dieser Anforderungen dient einem übergeordneten, gemeinsamen Ziel:

> *„All diese Veränderungen schaffen autonome Einheiten, die sich mit der gewünschten Innovationsrate sicher bewegen können.“*[183]

[180] Vgl. ebd., 15–21.
[181] Vgl. ebd., 21–23.
[182] Vgl. ebd., 23–27.
[183] Übersetzt aus: ebd., S. 28.

Microservices und Cloud-Native Anwendungen

Der Einsatz des Microservice-Architekturstils ist ein bedeutender Bestandteil des technisch notwendigen Wandels zur Umsetzung von Cloud-Native Anwendungen. Aus umgekehrter Sichtweise besitzen auch Microservice-basierte Anwendungen einen engen Bezug zu den übrigen Anforderungen des Cloud-Native Ansatzes. Sowohl DevOps und Continuous Delivery (vgl. Kapitel 2.3.4), an Geschäftsfähigkeiten ausgerichtete Teamstrukturen (vgl. Kapitel 2.2.3 und 2.3.1) als auch der Einsatz von Bounded Contexts (vgl. Kapitel 2.4.2) haben wesentlichen Einfluss auf die Realisierung von Microservice-Architekturen. Aufgrund des hohen Übereinstimmungsgrades beider Ansätze kann konkludiert werden, dass die Anforderungen von Microservice-basierten Anwendungen weitestgehend mit den Anforderungen von Cloud-Native Anwendungen übereinstimmen. Microservice-basierte Anwendungen stehen folglich in enger Verbindung mit den Prinzipien des Cloud-Native Ansatzes zur Entwicklung von Softwareanwendungen.

2.5.2 Einflussfaktoren und Abhängigkeiten

Ausgehend von den in Kapitel 2.5.1 beschriebenen Zusammenhängen können verschiedene Einflussfaktoren von Microservice-basierten Systemen abgeleitet werden. Nach Nadareishvili et al. spielen fünf Bereiche eine wesentliche Rolle: die gesamte Lösung bzw. Microservice-Anwendung auf Ebene der Makroarchitektur, die einzelnen Services auf Ebene der Mikroarchitektur, die Organisation und die Kultur sowie die Prozesse und Werkzeuge. Abbildung 12 vermittelt eine abstrakte Sicht auf diese Einflussbereiche.

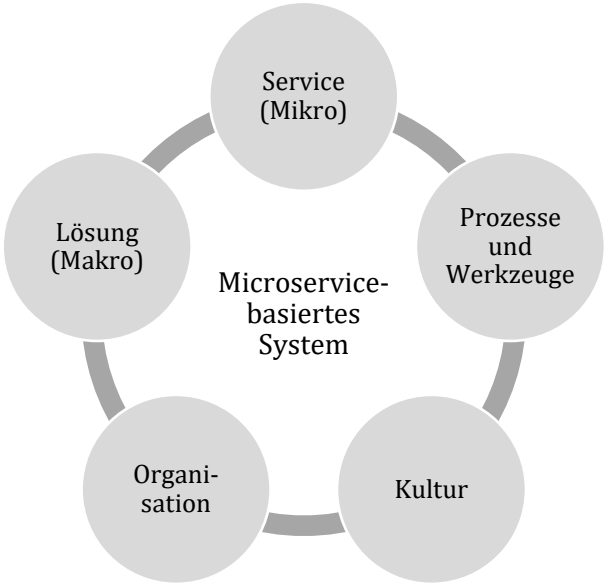

Abbildung 12: Einflussfaktoren von Microservice-basierten Systemen [184]

[184] In Anlehnung an: Nadareishvili et al. 2016, S. 27.

Die gezielte Beeinflussung dieser Faktoren steht in enger Verbindung mit der erfolgreichen Umsetzung von Microservice-basierten Systemen.[185] In Bezug auf Kapitel 2.3 können die Zusammenhänge genauer beschrieben werden:
Durch Anwendung des inversen Conway Manövers (vgl. Kapitel 2.3.1) können im Microservice-Kontext kleine, unabhängige, funktionsübergreifende und fachlich orientierte Teams entstehen, die Produkte statt Projekte entwickeln (vgl. Kapitel 2.3.2). Diese Teams profitieren vom Einsatz der Prinzipien und Praktiken der agilen Softwareentwicklung (vgl. Kapitel 2.3.3) sowie dem Wandel zu einer DevOps-Kultur (vgl. Kapitel 2.3.4). Die Bereiche der Entwicklung und des Betriebs können dadurch gemeinsam einen großen Teil des Produktlebenszyklus abdecken und die Unabhängigkeit der Microservice-Teams gefördert werden. Unterstützt werden sie dabei durch automatisierte Software-Bereitstellungsprozesse wie Continuous-Delivery sowie durch weitere DevOps-Praktiken.

In Anlehnung an Richardson können diese Zusammenhänge in einem Beziehungsdreieck dargestellt werden:

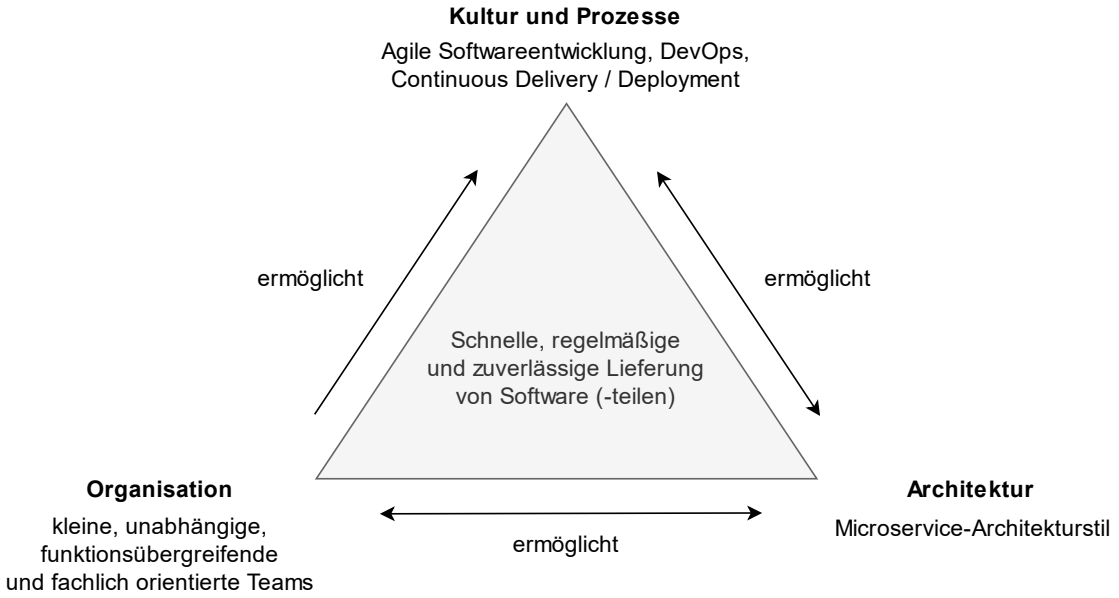

Abbildung 13: Einfluss von Architektur, Organisation, Kultur und Prozessen [186]

Nach dem Gesetz von Conway beeinflussen sich die Organisation und Architektur gegenseitig (vgl. Kapitel 2.3.1). Zudem wird die Etablierung auf Kultur- und Prozessebene durch eine entsprechende Anpassung der Organisationsstrukturen unterstützt. Zugleich steht auch die Architektur in Wechselwirkung mit der Kultur und den Prozessen. Aufgrund der Verteilung und Unabhängigkeit einzelner Services begünstigt der Microservice-Architekturstil die Realisierung von agiler Softwareentwicklung, DevOps und Continuous Delivery bzw. Continuous Deployment. Im Umkehrschluss unterstützen diese Kulturen und Prozesse die effiziente Umsetzung von Anwendungen auf Basis des Microservice-Architek-

[185] Vgl. ebd., S. 27.
[186] In Anlehnung an: Richardson 2018, S. 29.

turstils. Alle dargestellten Bereiche beeinflussen sich folglich stark gegenseitig. Die Abstimmung dieser Einflussbereiche aufeinander dient dem grundlegenden Ziel der schnellen, regelmäßigen und zuverlässigen Bereitstellung von Software bzw. Softwareteilen.

Microservices und DevOps

In Kapitel 2.3.4 wurde beschrieben, dass für die erfolgreiche Umsetzung von DevOps sowohl Änderungen an der Kultur und den Prozessen als auch der Architektur notwendig sind. DevOps kann dabei stärker vom Microservice-Architekturstil profitieren als dies im Umfeld monolithischer Systeme möglich ist. Das kann vor allem auf die strukturelle Anpassung der Organisation in kleine und unabhängige Teams zurückgeführt werden.[187]
Ein hoher prozessualer Automatisierungsgrad ist mit monolithischen Anwendungen nur bedingt realisierbar. Microservices sind daher häufig ein zentraler Bestandteil von DevOps-Bestrebungen. Zudem haben sich zahlreiche DevOps-Praktiken allein durch die Einbeziehung des Microservice-Architekturstils entwickelt.[188]

Im Umkehrschluss sind bestimmte DevOps-Praktiken wie Continuous Delivery und Deployment ab einem bestimmten Punkt nicht nur förderlich, sondern notwendig für die Umsetzung von Microservice-Architekturen. Das gilt vor allem für ausgeprägte Microservice-Architekturen mit einer Vielzahl an Microservices und einer großen Anzahl an Microservice-Teams. Die notwendige Automatisierung kann nur durch den Einsatz entsprechender Prozesse und Werkzeuge erreicht werden (vgl. Abbildung 12).[189] Ohne eine ausgeprägte Automatisierungskultur und die Anwendung von DevOps-Praktiken ist die Produktivität der Entwicklung negativ beeinflusst.[190]

Agile Softwareentwicklung und DevOps

In Kapitel 2.3.4 wurden agile Prinzipien und Praktiken als Teil der DevOps-Kultur beschrieben. Agile Softwareentwicklung und DevOps besitzen viele gemeinsame Eigenschaften wie die Konzentration auf die Menschen, deren Interaktionen und Zusammenarbeit. Obwohl DevOps von agilen Prinzipien geprägt wurde, handelt es sich um zwei unterschiedliche kulturelle Bewegungen. DevOps baut auf agilen Prinzipien und Praktiken auf und erweitert diese. Die DevOps-Kultur bezieht sich im Gegensatz zur agilen Softwareentwicklung nicht nur auf die Entwicklung, sondern auf die gesamte Organisation. Die Folgen des kulturellen Wandels sind im Kontext von DevOps daher weitreichender und gehen über das Ziel der schnellen Bereitstellung von Software hinaus.[191]

---

[187] Vgl. Balalaie et al. 2016a, S. 42.
[188] Vgl. Sharma et al. 2016, S. 269.
[189] Vgl. Daya 2015, S. 40.
[190] Vgl. Carneiro und Schmelmer 2016, S. 151.
[191] Vgl. Davis und Daniels 2016, S. 33.

Der Einfluss von Cloud-Technologien

Cloud-Technologien bilden die Grundlage für die Entwicklung von Cloud-Native Anwendungen. Aufgrund der engen Verbindung von Microservices mit den Prinzipien des Cloud-Native Ansatzes, profitieren auch Microservice-basierte Anwendungen stark von dem Einsatz von Cloud-Technologien (vgl. Kapitel 2.5.1). Des Weiteren wird die Umsetzung von DevOps durch Cloud-Technologien unterstützt. Dafür sind primär die Vorzüge von Cloud-Infrastrukturen, wie z. B. die gute Verfügbarkeit, breite Zugänglichkeit und automatisierte Bereitstellung von Ressourcen, verantwortlich.[192]

Zusammenfassung der Abhängigkeiten

In diesem Kapitel wurden die Zusammenhänge von fünf grundlegenden Einflussgrößen betrachtet: *Microservice-Architekturen (MSA)*, *agile Softwareentwicklung (AS), DevOps (DO)*, *Continuous Delivery (CD)* und *Cloud-Technologien (CL)*. Tabelle 5 fasst die Abhängigkeiten dieser Einflussbereiche in einer Abhängigkeitsmatrix zusammen. Die Matrix ist spaltenweise zu lesen. Beispiel: Microservice-Architekturen profitieren von einer DevOps-Kultur [Spalte MSA; Zeile DO].

	MSA	**AS**	**DO**	**CD**	**CL**
MSA	-	profitiert von	profitiert von	profitiert von	unabhängig
AS	profitiert von	-	nutzt AS als Praktik	profitiert von	unabhängig
DO	profitiert von	profitiert von	-	profitiert von	unabhängig
CD	profitiert von	profitiert von	nutzt CD als Praktik	-	unabhängig
CL	profitiert von	profitiert von	profitiert von	profitiert von	-

Tabelle 5: Abhängigkeitsmatrix der Einflussfaktoren

Die Matrix verdeutlicht, dass die unterschiedlichen Bereiche grundsätzlich positiven Einfluss aufeinander ausüben. Einzig der Bereich der Cloud-Technologien stellt eine unabhängige Einflussgröße dar und fungiert als Treiber für die Entwicklung bzw. die Etablierung der anderen Bereiche.
Das genaue Ausmaß der Einflüsse und Abhängigkeiten ist jedoch stets von dem individuellen Anwendungsfall und den Rahmenbedingungen abhängig. Die abgebildete Abhängigkeitsmatrix ist daher nicht als allgemeingültige Referenz zu verstehen. Zum Beispiel können die Abhängigkeitsbeziehungen der Matrix im Kontext einer ausgeprägten Microservice-Architektur mit vielen Microservices restriktiver sein und verschärft formuliert werden. Eine derartige Microservice-Architektur ist nicht praktikabel, wenn keine DevOps-Kultur etabliert ist, kein Continuous-Delivery Prozess existiert, die Entwicklung nicht auf agilen Prinzipien und Methoden basiert oder die technologischen Vorteile der Cloud nicht ausgenutzt werden.

[192] Vgl. Pahl et al. 2018, S. 5.

3. Ergebnisse der Literaturanalyse

Um ergänzend zu den theoretischen Grundlagen eine wissenschaftliche Ausgangsbasis für die Herleitung des konzeptionellen Ordnungsrahmens zu schaffen, basiert diese Arbeit auf der Durchführung einer systematischen Literaturanalyse. Die Durchführung der Recherche- und Analysetätigkeiten war mit zwei Zielen verbunden:

(1) Darstellung des Forschungsstandes
(2) Identifizierung relevanter Theorien und Konzepte

Weiterhin wurden zwei Recherchefragen (RF1, RF2) definiert, die den Rahmen für den Recherche- und Analyseprozess darstellten. Sie sollten einen gezielten Beitrag zur Beantwortung der Forschungsfragen leisten (vgl. Kapitel 1.2).

(RF1) Welche Beiträge befassen sich mit der Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils?

(RF2) Welche Beiträge befassen sich im Speziellen mit der Dekomposition der fachlichen Architektur monolithischer Anwendungen und welche davon basieren auf den Mustern und Prinzipien des Domain-Driven Designs?

Die identifizierten Beiträge wurden zunächst nach dem inhaltlichen Schwerpunkt kategorisiert. Anschließend wurden die Literaturquellen mit relevantem Schwerpunkt näher auf bedeutende Theorien und Konzepte untersucht. Sie dienten als Grundlage für das weitere Vorgehen zur Synthese des konzeptionellen Ordnungsrahmens (vgl. Kapitel 4). Die Rechercheergebnisse werden in Kapitel 3.1 (RF1) und Kapitel 3.2 (RF2) präsentiert.

3.1 Transformation der Architektur

Es konnten insgesamt 17 Beiträge identifiziert werden, die sich mit der Migration bzw. Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils befassen. Von diesen Beiträgen konnten 5 Arbeiten als *relevant* eingestuft werden. 7 Beiträge wurden als *nicht relevant* eingestuft, da sie durch ihre spezifische Ausrichtung nicht hinreichend zur Beantwortung der Forschungsfrage beitragen konnten. Weitere 5 Arbeiten wurden als *bedingt relevant* eingestuft, da ihr inhaltlicher Schwerpunkt nur eingeschränkte Relevanz in Bezug auf die Fragestellung besitzt, sie jedoch einige verwendbare Informationen liefern konnten. Somit ergab sich eine Auswahl von 10 wissenschaftlichen Vorarbeiten, die für die Synthese des konzeptionellen Ordnungsrahmens näher untersucht wurden. Die Rechercheergebnisse sind in Tabelle A-3 (siehe Anhang) dargestellt.

Trotz des gemeinsamen Bezugs auf die Transformation der fachlichen Architektur sind die inhaltlichen Schwerpunkte der identifizierten Beiträge unterschiedlich ausgeprägt. Im Kern liefern sie Informationen in den folgenden Bereichen:

- Aktivitäten und Herausforderungen im Migrationsprozess
- Anwendbare Muster und Prinzipien
- Praxiserfahrungen und Erkenntnisse *(Lessons Learned)*

Relevante Theorien und Konzepte

Die im Folgenden verwendeten Nummerierungen dienen als Referenz für die in Tabelle A-3 aufgeführten wissenschaftlichen Beiträge.

(1) Di Francesco et al. berichten im Rahmen einer empirischen Studie über einen allgemeinen Migrationsprozess, über typische Aktivitäten bei Migrationen, auftretende Herausforderungen und über verschiedene weitere Erfahrungen, die im Rahmen der untersuchten Migrationsprojekte gesammelt wurden.

(2) In Balalaie et al. werden 15 Muster und Prinzipien beschrieben, die in industriellen Software-Migrationsprojekten im Rahmen einer empirischen Studie identifiziert wurden. Die vorgeschlagenen Migrationsmuster basieren auf einem Kategoriensystem aus 12 Faktoren, die den Anwendungsbereich der Muster kennzeichnen. In den relevanten Kategorien der Dekomposition *(decomposition)* und des Verständnisses *(understanding)* konnten die folgenden vier Migrationsmuster identifiziert werden: (MP-2) Wiederherstellung der aktuellen Architektur, (MP-3) Dekomposition des Monolithen, (MP-4) Dekomposition des Monolithen auf Basis des Datenbesitzes, (MP-5) Ändern von Codeabhängigkeiten in Serviceaufrufe.

(3) Joselyne et al. präsentieren ein Rahmenwerk, dass diverse fachlich und technisch motivierte Aktivitäten von Migrationen in fünf Phasen zusammenfasst. Dabei sind einige Aktivitäten stark verallgemeinert dargestellt. Sie werden daher im weiteren Verlauf dieser Arbeit betrachtet. Der Großteil der aufgeführten Aktivitäten bezieht sich jedoch auf spezifische technische Aspekte und grenzt den Migrationsprozess durch bestimmte technologische Vorgaben ein. Sie werden daher nicht weiter betrachtet.

(4) In Taibi et al. konnten im Rahmen einer Studie 8 Beweggründe bzw. Motivationsfaktoren für Migrationen ausfindig gemacht werden. Die Arbeit berichtet über verschiedene Praxiserfahrungen wie Herausforderungen und Vorteile von Migrationen. Die Autoren propagieren ein Rahmenwerk, das drei mögliche Prozessalternativen vorsieht und in fünf Prozessphasen – System Structure Analysis, System Architecture, Prioritization of Feature Development, Coding, Testing – untergliedert ist.

(5) Razavian et al. schlagen mit dem „SOA Migration Framework“ ein Rahmenwerk vor, das eine ganzheitliche und abstrakte Darstellung eines Migrationsprozesses im Kontext serviceorientierter Architekturen umfasst. Das vorgeschlagene Rahmenwerk basiert dabei auf dem Hufeisen-Modell von Kazman et al., das auch von Di Francesco et al. (1) herangezogen wird. Weiterhin wird im Beitrag beschrieben durch welche konzeptionellen Elemente der Migrationsprozess bestimmt ist. Obwohl Microservice-Architekturen und serviceorientierte Architekturen nicht gleichzusetzen sind (vgl. Kapitel 1.3), ist das vorgeschlagene Rahmenwerk aufgrund der Abstraktheit und des Aspektes der Service-Basiertheit von hoher Relevanz für die vorliegende Arbeit.

(6) In Kalske et al. werden verschiedene technische und organisatorische Herausforderungen beschrieben, die bei der Migration monolithischer Anwendungen im Kontext von Microservice-Architekturen auftreten. Die Ausführungen basieren auf der Untersuchung mehrerer Publikationen und Fallstudien.

(7) Knoche et al. berichten über verschiedene Erfahrungen, die im Rahmen einer serviceorientierten Plattform-Migration eines Cobol-Systems gesammelt werden konnten. Neben der Beschreibung von Gründen für eine Modernisierung bzw. Migration schlagen sie einen Prozess vor, der auf die interne Umstrukturierung der bestehenden Anwendung abzielt, bevor einzelne Microservices erstellt werden können.

(8) In Bucchiarone et al. wird über Erfahrungen berichtet, die im Rahmen der Migration eines monolithischen Bankensystems gesammelt wurden. Der Schwerpunkt der Ausführungen liegt auf der Transformation der technischen Architektur, es wird jedoch auch unter allgemeinen Gesichtspunkten auf den Migrationsprozess eingegangen.

(9) Gouigoux et al. präsentieren verschiedene Erkenntnisse, die im Kontext eines industriellen Migrationsprojektes gesammelt werden konnten. Sie beziehen sich auf die drei Aspekte der Service-Granularität, des Deployments und der Orchestrierung.

(10) In Fan et al. wird ein Migrationsprozess vorgestellt, der auf einem spezifischen Entwicklungsprozess basiert und den Schwerpunkt auf technische Entscheidungen legt, die während einer Migration getroffen werden müssen. Der vorgeschlagene Prozess ist aufgrund der überwiegend technischen Ausrichtung nur bedingt relevant für die vorliegende Arbeit. Die Ausführungen liefern jedoch auch allgemeine Informationen zu Migrationen.

Zusammengefasst stellen die Beiträge relevante Inhalte aus fünf Kategorien bereit. Einerseits vermitteln sie aufbereitetes Wissen in Form von Prozessen, Modellen oder Rahmenwerken. Weiterhin beschreiben sie verschiedene Migrationsmuster, Aktivitäten und Vorgehensweisen, Gründe für Migrationen oder Herausforderungen bei der Durchführung von Migrationen. Tabelle 6 stellt eine Zuordnung der relevanten wissenschaftlichen Arbeiten zu den fünf Inhaltskategorien dar. Zutreffende Inhalte sind mit einem „x“ gekennzeichnet.

Relevante Inhalte	**Quellen**									
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Prozess / Modell / Rahmenwerk	x		x	x	x		x			x
Anwendbare Migrationsmuster		x							x	
Vorgehen / Aktivitäten bei Migrationen	x	x	x	x	x		x	x	x	x
Motivation / Gründe für Migrationen				x			x			
Herausforderungen in Migrationen	x			x		x				

Tabelle 6: Kategorisierung der relevanten Inhalte (RF1)

Mit dem *Hufeisen-Modell* (1) (5) und dem *SOA Migration Framework* (5) konnten zwei bedeutende Konzepte identifiziert werden. Sie besitzen besonders hohe Relevanz in Bezug auf die Erstellung eines allgemeinen Rahmens für die Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils.

3.2 Ansätze zur Dekomposition

Es konnten insgesamt 20 wissenschaftliche Beiträge identifiziert werden, die verschiedene Ansätze zur Dekomposition monolithischer Anwendungen beschreiben. Die Arbeiten wurden nach ihrem inhaltlichen Schwerpunkt klassifiziert und in 4 Kategorien eingeordnet. Die Klassifizierung der Arbeiten erfolgte unabhängig davon, ob die Neuentwicklung oder Migration eines monolithischen Altsystems betrachtet wird. Ein Teil der Ergebnisse basiert auf der Vorarbeit von Fritzsch et al.[193] über die Untersuchung verschiedener Dekompositionsansätze im Microservice-Umfeld. Tabelle A-4 (siehe Anhang) stellt eine Übersicht über die Rechercheergebnisse bereit.

Lediglich 5 Arbeiten stellen einen Bezug zur Domäne bzw. zum Geschäft her. Es verfolgt jedoch keiner der identifizierten Beiträge einen fachlich getriebenen Dekompositionsansatz auf Basis der Muster und Prinzipien des Domain-Driven Designs. Zwei Arbeiten betrachten Domain-Driven Design im Kontext der Implementierung Microservice-basierter Anwendungen, sie liefern jedoch auch grundlegende Informationen zur Anwendung von DDD und waren daher *bedingt relevant*. Zwei weitere Arbeiten betrachten die Elemente des taktischen Designs (vgl. Kapitel 2.4.3) und beziehen sich ausschließlich auf die Implementierungsebene. Sie wurden daher als *nicht relevant* eingestuft. Zur Vollständigkeit werden diese dennoch erwähnt. Weiterhin konnte eine Arbeit identifiziert werden, die einen geschäftlich getriebenen Dekompositionsansatz verfolgt, der jedoch nicht auf den Mustern und Prinzipien des Domain-Driven Designs basiert. Der Beitrag betrachtet das Konzept der Geschäftsfähigkeiten und wurde aufgrund der engen Verbindung mit den Eigenschaften von Microservices (vgl. Kapitel 2.2.3) als *bedingt relevant* eingestuft. Die übrigen Arbeiten konnten in die drei Kategorien *Quellcode und Daten*, *Schnittstellen* oder *Sonstige* eingeordnet werden. Sie lassen die Domäne als fachlichen Aspekt der Dekomposition außer Acht und betrachten ausschließlich die bestehende Implementierung monolithischer Systeme. Daher wurden sie nicht in die weiteren Untersuchungen einbezogen.

Relevante Theorien und Konzepte

Die im Folgenden verwendeten Nummerierungen dienen als Referenz für die in Tabelle A-4 aufgeführten wissenschaftlichen Quellen.

(1) (2) Steinegger et al. und Hippchen et al. berichten über die Prinzipien, Muster und Aktivitäten des Domain-Driven Designs, die den Aufbau von Microservice-basierten Anwendungen unterstützen können. Die Aktivitäten werden dabei in die gängigen Prozessphasen der Softwareentwicklung eingeordnet und in einer Fallstudie demonstriert. Die Betrachtungen finden nicht im Kontext der Transformation monolithischer Anwendungen statt. Weiterhin beziehen sich die Ausführungen größtenteils auf die Implementierungsebene. Dabei liegt der Schwerpunkt auf den Mustern des taktischen Designs (vgl. Kapitel 2.4.3) zur Umsetzung von Domänenmodellen in Microservices. Die Inhalte sind folglich nur bedingt relevant, weshalb sich die vorliegende Arbeit auf die Ausführungen zu den allgemeinen Aktivitäten im Domain-Driven Design beschränkt.

[193] Fritzsch et al. 2019.

(3) In Frey et al. wird ein Ansatz zur Dekomposition monolithischer Anwendungen beschrieben, der auf dem Konzept der Geschäftsfähigkeiten *(Business Capabilities)* basiert. Die Betrachtungen beziehen sich auf die geschäftliche Ebene und finden im Umfeld serviceorientierter Architekturen statt. Der Ansatz wurde aufgrund der geschäftlichen Ausrichtung, der Service-Basiertheit sowie der engen Verbindung von Geschäftsfähigkeiten und Microservices (vgl. Kapitel 2.2.3) in die näheren Untersuchungen einbezogen.

4. Synthese des Ordnungsrahmens

In diesem Kapitel werden die Theorien und Konzepte, die in Kapitel 3 im Rahmen der Literaturanalyse identifiziert wurden, genauer auf die Eignung für den konzeptionellen Ordnungsrahmen untersucht. Die Synthese des Ordnungsrahmens basiert jedoch nicht ausschließlich auf den identifizierten wissenschaftlichen Arbeiten. Sie baut zusätzlich auf Wissen aus Monografien, Artikeln aus Fachmagazinen, Praxisberichten von Fachexperten sowie dem theoretischen Grundlagenwissen dieser Arbeit auf (vgl. Kapitel 2). Abbildung 14 gibt einen Überblick über die fünf Teilartefakte, die den konzeptionellen Ordnungsrahmen in diesem Kapitel konstituieren.

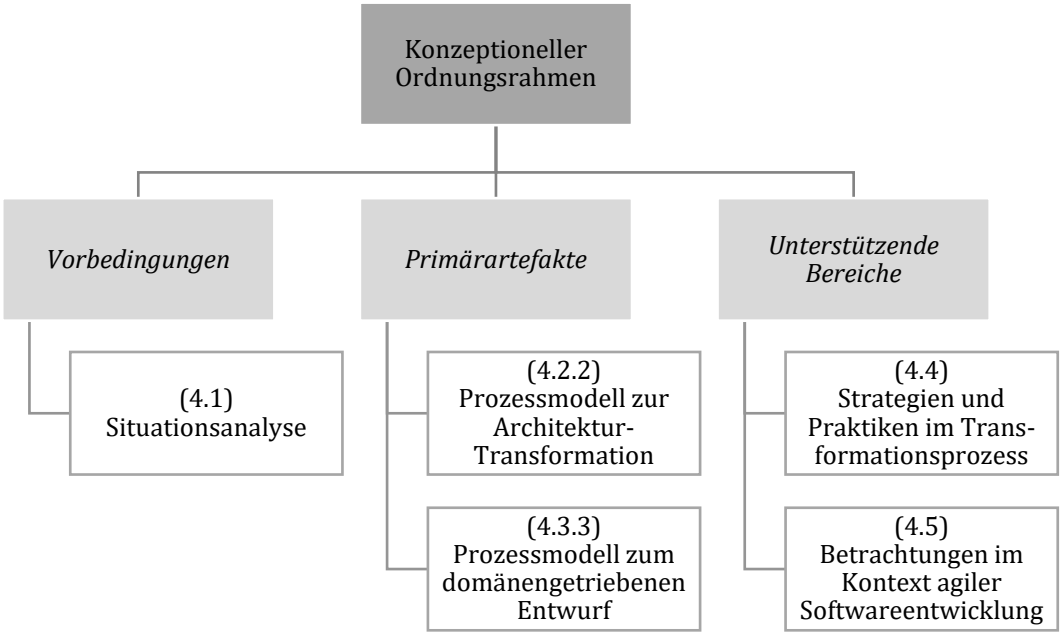

Abbildung 14: Ergebnisübersicht

4.1 Situationsanalyse

Bevor sich ein Unternehmen intensiv mit der Planung und dem Vorgehen zur Transformation der fachlichen Architektur auseinandersetzt, sollten die grundlegenden Vorbedingungen und Anforderungen, die mit dem Einsatz des Microservice-Architekturstils verbunden sind, im Rahmen einer Analyse der Ist-Situation überprüft werden. Die Situationsanalyse kann in Bezug auf den Bedarf und den Reifegrad des Unternehmens bzw. des betroffenen Projektes erfolgen.

4.1.1 Analyse des Bedarfs

Im Rahmen einer Bedarfsanalyse sollte zunächst der konkrete Bedarf des Unternehmens bzw. des Projektes für die Einführung einer Microservice-Architektur festgestellt werden. Wie bedeutend dieser vermeintlich triviale Schritt ist, zeigen die Ergebnisse der Studie

von Taibi et al. Demnach schlagen zahlreiche Unternehmen den Weg in Richtung Microservices ein, nur weil sie dem vorherrschenden Trend folgen, um zu einem späteren Zeitpunkt nicht den Anschluss zu verpassen.[194]

Grundsätzlich gelten die Vorteile von Microservice-Architekturen (vgl. Kapitel 2.2.3) als Motivationsfaktoren für die Einführung des Microservice-Architekturstils. Weiterhin korrelieren die Vorteile von Microservice-Architekturen mit den Nachteilen und Problemen monolithischer Systeme (vgl. Kapitel 2.2.4). Die Vorteile des Microservice-Architekturstils und die Herausforderungen monolithischer Anwendungen können folglich als die grundlegenden Treiber angesehen werden, in denen der Bedarf eines Unternehmens bzw. Projektes für die Einführung einer Microservice-Architektur begründet liegt.

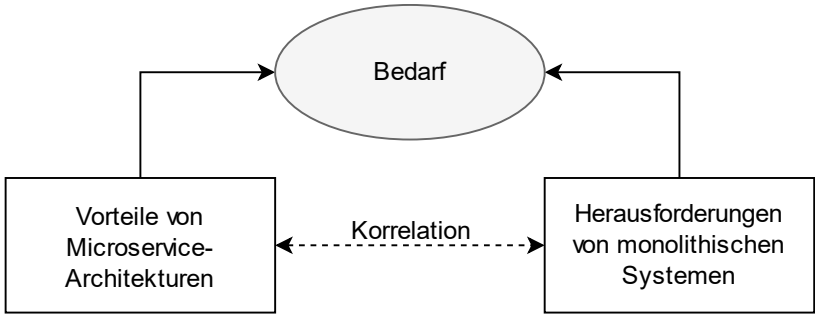

Abbildung 15: Zwei grundlegende Treiber der Bedarfsentstehung

Die Feststellung des Bedarfes kann demzufolge durch die Analyse der Ist-Situation im Kontext des monolithischen (Alt-) Systems und durch die Gegenüberstellung der Vorteile des Microservice-Architekturstils erfolgen. Der Vergleich sollte jedoch nicht ausschließlich auf der Ebene der Architektur erfolgen. Da der Microservice-Architekturstil auch weitreichende Auswirkungen auf die Organisation sowie die Kultur und Prozesse hat, sollten diese Bereiche in der Analyse der Ist-Situation berücksichtigt werden (vgl. Kapitel 2.5.2, Abbildung 13). Beispielsweise kann die Einführung einer Microservice-Architektur zur Umsetzung von DevOps, Continuous-Delivery oder der Skalierung agiler Prozesse beitragen (vgl. Kapitel 2.5.2; Tabelle 5). Dies spiegelt sich auch in den Ergebnissen der Studie von Taibi et al. wieder, in der die Unterstützung von DevOps-Vorhaben als verbreiteter Grund für die Einführung von Microservices identifiziert wurde. Die Studie zeigt zudem, dass die Wartbarkeit der Software, die Skalierbarkeit, die Verteilung von Verantwortlichkeiten in kleinere und unabhängige Teams sowie die verbesserte Fehlertoleranz zu den Hauptgründen für Migrationen zählen.[195]

Taibi et al. berichten weiterhin über die verschiedenen Vorteile, die praktizierende Unternehmen nach durchgeführten Migrationen erfahren haben. Neben der verbesserten Wartbarkeit und Skalierbarkeit ergaben sich z. B. die Vorteile der besseren Handhabbarkeit der Architekturkomplexität und der guten Investmentrendite *(Return-on-Investment)* aufgrund verringerter Wartungskosten. Des Weiteren konnte eine bessere Arbeitsverteilung, Performanz und Testbarkeit sowie eine verbesserte Modularität durch striktere

[194] Vgl. Taibi et al. 2017, S. 26.
[195] Vgl. ebd., S. 25f.

Aufteilung von Zuständigkeiten *(Separation of Concerns)* erreicht werden. Auch über Vorteile, wie z. B. eindeutigere Verantwortlichkeiten, das vorteilhafte Zusammenspiel mit der Projektmanagementmethode *Scrum* und das verbesserte Systemverständnis, wird berichtet.[196] Nach den Erfahrungen von Knoche et al. ist vor allem die geringe *Evolvierbarkeit* (engl. *evolvability*) einer Software ein Indikator, der für die Migration monolithischer Anwendungen spricht. Eine geringe Evolvierbarkeit ist dadurch gekennzeichnet, dass die Einführung von neuer Funktionalität nur mit hohem Aufwand möglich ist, was ein strategisches Risiko für Projekte darstellt. Weiterhin kann eine starke hersteller- und technologiebezogene Abhängigkeit die Einführung von Microservices motivieren, um einen höheren Grad an Freiheit zu erreichen.[197] Tabelle 7 fasst die beschriebenen Motivationsfaktoren zusammen. Da die Inhalte einzig auf den Erfahrungen der betrachteten Studien basieren, erheben sie keinen Anspruch auf Vollständigkeit.

Software	**Organisation, Kultur, Prozesse**
- Evolvierbarkeit - Wartbarkeit - Skalierbarkeit - Fehlertoleranz - Performanz - Testbarkeit - Technologiefreiheit - Modularität - Komplexität der Architektur - Systemverständnis	- Arbeitsverteilung - Teamstruktur und Aufteilung von Verantwortlichkeiten - Unterstützung von DevOps - Unterstützung agiler Methoden und Prinzipien (z. B. Scrum) - Return-on-Investment

Tabelle 7: Motivationsfaktoren für Migrationen

Neben den beschriebenen Motivationsfaktoren ist auch die Produktivität des Entwicklungsteams ein bedeutender Faktor für die Beurteilung des Bedarfs. Die Einführung des Microservice-Architekturstils geht mit neuen Herausforderungen, Komplexitäten und Risiken einher (vgl. Kapitel 2.2.3). Fowler hat in diesem Zusammenhang den Ausdruck *Microservice Premium* geprägt. Nach seiner Theorie ist die Produktivität des Entwicklungsteams in Abhängigkeit von der Komplexität der betrachteten Anwendung bzw. der zugrunde liegenden Problemdomäne zu betrachten. Dabei sollten Unternehmen bzw. Projekte den Microservice-Architekturstil nur in Erwägung ziehen, wenn das betrachtete System so komplex ist, dass es nicht effizient als Monolith entwickelt und betrieben wer-

[196] Vgl. ebd., S. 28.
[197] Vgl. Knoche und Hasselbring 2018, S. 45.

den kann. Infolgedessen sollte die Mehrheit der Softwareanwendungen zunächst als modular aufgebaute Monolithen implementiert werden, anstatt sie in Microservices aufzuteilen. Erst wenn eine monolithische Anwendung ein bestimmtes Maß an Komplexität erreicht hat, sodass die Produktivität der Entwickler negativ beeinflusst wird, lohnt es sich die Verwendung des Microservice-Architekturstils in Erwägung zu ziehen.[198] Dieses Vorgehen kann nach Fowler auch unter der Bezeichnung *Monolith First*[199] zusammengefasst werden. Abbildung 16 stellt den Zusammenhang von Produktivität und Komplexität dar.

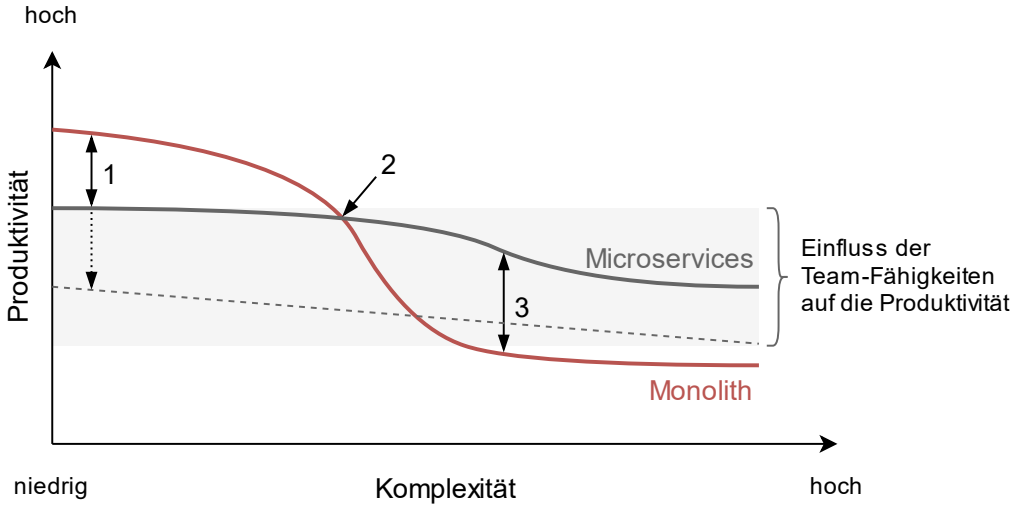

Abbildung 16: Einfluss von Produktivität und Komplexität [200]

Anhand der dargestellten Kurvenverläufe können die folgenden drei Zustandsindikatoren identifiziert werden:

(1) Aufgrund der spezifischen Herausforderungen von Microservices ist die Produktivität im Rahmen von wenig komplexen Anwendungen geringer als bei einer monolithischen Anwendung.

(2) Bei zunehmender Komplexität der Anwendung nimmt die Produktivität in einer monolithischen Umgebung schnell ab, wohingegen sich die Produktivität bei einer Microservice-Architektur nur marginal reduziert. Ab einem bestimmten Komplexitätsgrad kehrt sich die Situation um und der Einsatz einer Microservice-Architektur wird in Hinsicht auf die Produktivität attraktiver als ein Monolith.

(3) Die lose Kopplung und Unabhängigkeit einzelner Microservices wirkt sich bei hoher Anwendungskomplexität positiv auf die Abnahme der Produktivität aus. Sie sinkt deutlich weniger als in einem monolithischen System.

Nach Fowler hängt die tatsächliche Produktivität jedoch wesentlich von den Fähigkeiten der Entwicklungsteams ab. Ein eingespieltes Team, das bereits Erfahrung im Umgang mit Microservices besitzt, ist mit hoher Wahrscheinlichkeit produktiver als ein Team ohne entsprechende Erfahrung.[201] Weiterhin ist zu berücksichtigen, dass die werkzeugbasierte

---

[198] Vgl. Fowler 2015b.
[199] Fowler 2015a.
[200] In Anlehnung an: Fowler 2015b; Barcia et al. 2017, S. 6f.
[201] Vgl. Fowler 2015b.

Unterstützung im Microservice-Umfeld in den letzten Jahren stark zugenommen hat. Durch den schnellen technologischen Fortschritt werden die Herausforderungen des Microservice-Architekturstils daher zunehmend gemildert. Das hat zumindest positive Auswirkungen auf die technologisch bedingten Einstiegshürden.

4.1.2 Analyse des Reifegrades

Neben der Feststellung des Bedarfes sollte auch der Reifegrad des Unternehmens bzw. Projektes in Bezug auf die Einführung von Microservices untersucht werden. Dabei sind die Voraussetzungen in Verbindung mit dem Microservice-Architekturstil zu betrachten und mit der aktuellen Situation im Unternehmen bzw. im Projekt abzugleichen. Grundsätzlich erfordert die Einführung einer Microservice-Architektur auch Anpassungen an der Organisation, der Kultur und den Prozessen (vgl. Kapitel 2.5.2; Abbildung 13). Daher sollte der Reifegrad in Bezug auf diese Bereiche untersucht werden. Weiterhin sollten die Aspekte der Technologie und Infrastruktur sowie das vorhandene Wissen und die Erfahrung berücksichtigt werden. Die Anforderungen von Cloud-Native Anwendungen können dafür unterstützend in Betracht gezogen werden (vgl. Kapitel 2.5.1; Tabelle 4). Die Bereiche zur Bewertung des Reifegrades können auf Grundlage dieser Informationen wie folgt zusammengefasst werden:

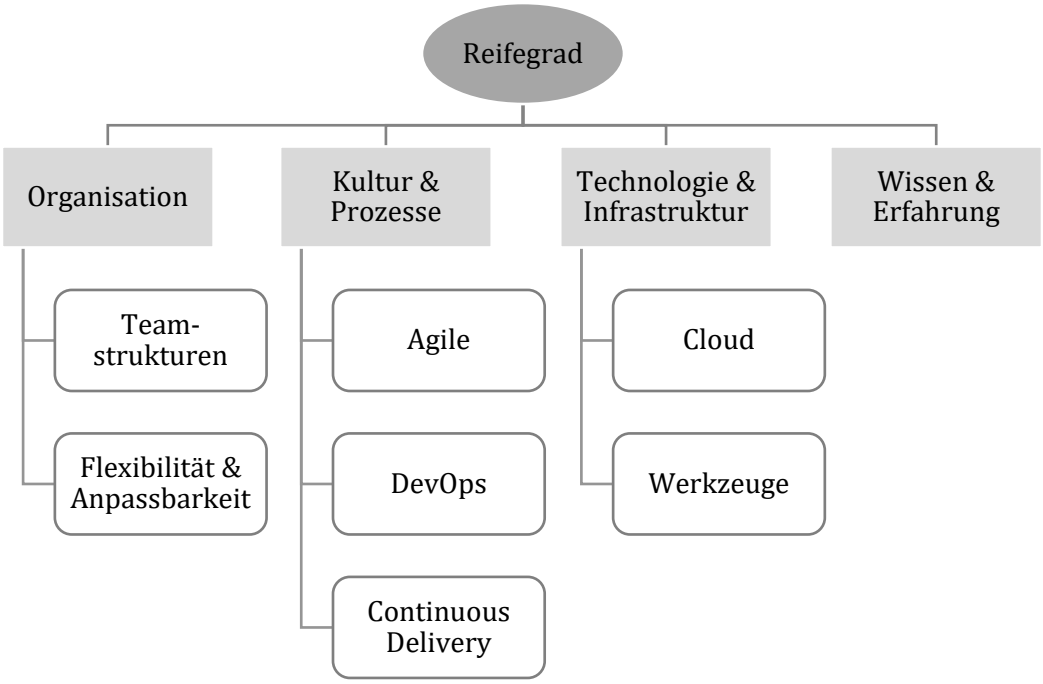

Abbildung 17: Bereiche und Faktoren zur Beurteilung des Reifegrades

Der nachfolgende Fragenkatalog bezieht sich auf die Bereiche aus Abbildung 17 und dient als erster Richtungsweiser für die initiale Beurteilung des Reifegrades.

Nr.	Reifegradfaktoren	Leitfragen / Beschreibung
(1)	Organisations- bzw. Teamstrukturen	- *Unterstützen die aktuellen Teamstrukturen (z. B. Größe und Zusammensetzung) den Einsatz von Microservices?* - *Wie flexibel ist das Unternehmen die Organisations- bzw. Teamstrukturen dahingehend anzupassen?* Das Gesetz von Conway, insbesondere das inverse Conway-Manöver, sind zu berücksichtigen (vgl. Kapitel 2.3.1).
(2)	Agile Softwareentwicklung	- *Werden bereits agile Methoden und Prinzipien eingesetzt?* - *Wie erfahren sind die Teams in deren Anwendung?*
(3)	DevOps-Kultur	- *Ist bereits eine DevOps-Kultur etabliert?* - *Welche DevOps-Praktiken werden angewendet?* - *Ist die Organisation flexibel genug, um die Bereiche der Entwicklung und des Betriebes zu vereinen? (siehe Punkt 1)*
(4)	Automatisiertes Deployment	- *Werden bereits Prozesse zur automatisierten Softwarebereitstellung eingesetzt?* - *Wie erfahren sind die Teams in dem Aufbau und der Anwendung derartiger Prozesse?*
(5)	Technologie & Infrastruktur	- *Werden bereits Technologien und Werkzeuge genutzt, die für die Entwicklung und den Betrieb von Microservice-Anwendungen relevant oder von Vorteil sind?* - *Wie flexibel ist das Unternehmen hinsichtlich der Einführung neuer Technologien und Infrastruktur?*
(6)	Wissen & Erfahrung	- *Welchen Wissens- und Erfahrungsstand haben die Entwicklungsteams hinsichtlich Microservice-Architekturen?* - *Welchen Stellenwert besitzen Weiterbildung und Wissensaufbau im Unternehmen bzw. Projekt?* Die tatsächliche Produktivität wird durch das vorhandene Wissen und die Erfahrung beeinflusst (vgl. Abbildung 16).

Tabelle 8: Fragenkatalog zur Beurteilung des Reifegrades

Bei der Analyse des Reifegrades kann zwischen der Bewertung auf Unternehmens- bzw. Projektebene und der Bewertung bereits bestehender Microservice-Anwendungen unterschieden werden. Die Bewertung von Microservice-Anwendungen ist im Kontext dieser Arbeit jedoch erst nach einer durchgeführten Migration relevant, weshalb dieser Themenbereich getrennt von einem Migrationsvorhaben zu betrachten ist. Anwendungsbezogene Aspekte, wie der Reifegrad der Codebasis, der Daten oder der Dekomposition, wurden daher nicht betrachtet. Die vorgestellten Ergebnisse beziehen sich somit ausschließlich auf die Beurteilung des Reifegrades von Unternehmen oder Projekten. Sie definieren einen groben Rahmen für die initiale Reifegradbewertung.

4.2 Architektur-Transformation

In diesem Kapitel wird die Herleitung eines Prozesses zur domänengetriebenen Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils beschrieben. Zunächst wird in Kapitel 4.2.1 das Hufeisen-Modell als Prozessgrundlage vorgestellt. Auf dieser Basis wird in Kapitel 4.2.2 ein Prozessmodell zur Architektur-Transformation vorgeschlagen, in dessen Kontext eine ausführlichere Beschreibung der einzelnen Prozessphasen erfolgt. Abschließend werden in Kapitel 4.2.3 weitere wissenschaftliche Ansätze im Kontext des Prozessmodells betrachtet.

4.2.1 Das Hufeisen-Modell als Prozessgrundlage

Kazman et al.[202] haben im Jahr 1998 das Hufeisen-Modell *(horseshoe model)* veröffentlicht, das für verschiedene architekturelle Umstrukturierungen herangezogen werden kann. Das Modell wurde seitdem häufig adaptiert, so auch vom *Architecture-Driven Modernization (ADM) Framework*. Das ADM-Framework ist eine Initiative der Object Management Group (OMG) und kann für die Realisierung verschiedener Modernisierungsziele im Umfeld von Softwareanwendungen eingesetzt werden. Neben Szenarien wie System-Konsolidierungen, Plattform-Migrationen, Quellcode-Umwandlungen oder der Integration von Anwendungen, unterstützt ADM auch servicebasierte Transformationen bestehender Anwendungsarchitekturen. ADM kann als Prozess beschrieben werden, der dem Verständnis und der Weiterentwicklung von Softwareanwendungen dient und dabei sowohl Funktionen, Komponenten und Anforderungen der Software als auch geschäftliche Aspekte und Ziele umfasst.[203]

Das ADM-Framework beschreibt drei Architektur-Perspektiven, die im Kontext von Modernisierungen eine Rolle spielen können: die technische Architektur, die Anwendungs- und Datenarchitektur und die Geschäftsarchitektur. Die drei Perspektiven können wiederum den Bereichen der Geschäftsdomäne und IT-Domäne zugeordnet werden. Die Geschäftsdomäne ist beispielsweise durch Geschäftsprozesse, Geschäftsregeln oder geschäftliche Semantiken gekennzeichnet, wohingegen sich die IT-Domäne auf die technische sowie die Anwendungs- und Datenarchitektur bezieht.[204] Abbildung 18 zeigt diese Aspekte des Hufeisen-Modells in Bezug auf die Modernisierung einer bestehenden Softwarelösung.

[202] Kazman et al. 1998.
[203] Vgl. Wasukar et al. 2013, S. 142f.
[204] Vgl. Khusidman und Ulrich 2007, S. 1.

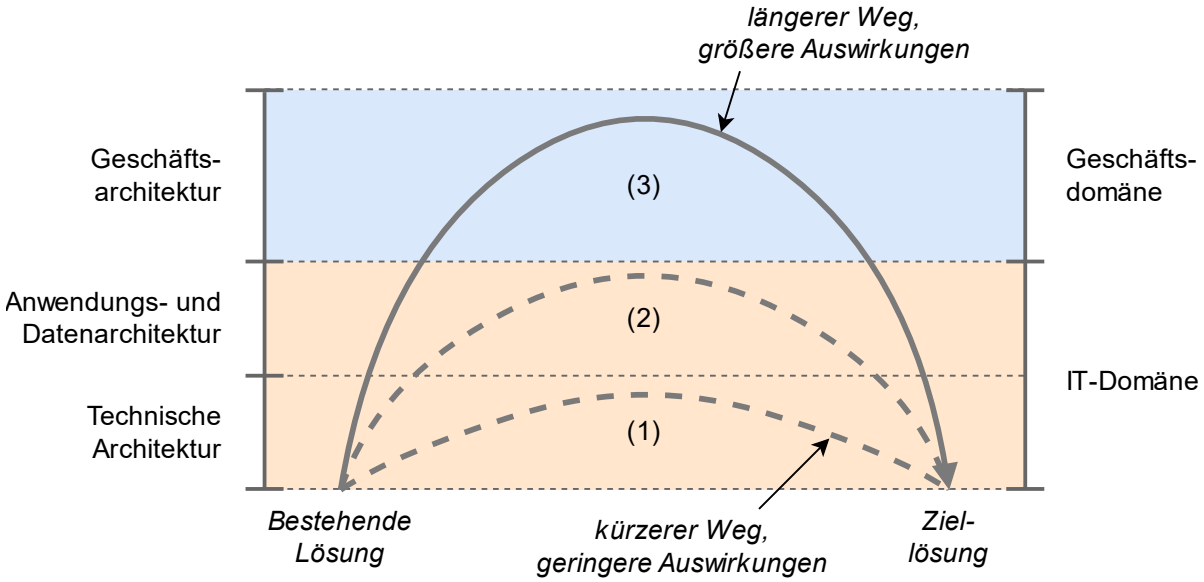

Abbildung 18: Hufeisen-Modell nach ADM [205]

Ein Modernisierungsvorhaben kann durch das Geschäft oder die IT motiviert sein. Im engeren Sinne kann es von jeder der drei Architektur-Perspektiven getrieben werden. Es ergeben sich folglich drei Szenarien:

(1) Eine von der technischen Architektur getriebene Modernisierung überführt die bestehende Lösung auf kürzestem Wege in die Ziellösung. Da ausschließlich technische Anpassungen vorgenommen werden, hat dieses Szenario die geringsten Auswirkungen auf die Softwarelösung.

(2) Von der Anwendungs- und Datenarchitektur getriebene Modernisierungen sind aufwändiger und haben umfangreichere Auswirkungen, da sie Anpassungen an der technischen sowie der Anwendungs- und Datenarchitektur beinhalten.

(3) Eine von der Geschäftsarchitektur getriebene Modernisierung kennzeichnet den längsten Weg und hat die größten Auswirkungen auf die Softwarearchitektur. Sie sieht eine Ausrichtung von Geschäft und IT vor und ist daher mit Anpassungen in der Geschäfts- und IT-Domäne verbunden.[206]

Primär lässt sich die Transformation einer monolithischen Anwendung auf Basis des Microservice-Architekturstils in die Architektur-Perspektive der Anwendungs- und Datenarchitektur einordnen. Das bedeutet, es finden in erster Linie Anpassungen an der Anwendungs- und Datenarchitektur sowie der zugrunde liegenden technischen Architektur statt. Die Grenze zur Geschäftsarchitektur wird jedoch überschritten, wenn die Prinzipien und Methoden des Domain-Driven Designs für die Analyse der bestehenden Lösung sowie dem Entwurf der Ziellösung zur Anwendung kommen. Unter Umständen werden im Rahmen der Transformation zwar keine Anpassungen an der Geschäftsarchitektur, wie z. B. den Geschäftsprozessen, vorgenommen, durch die Anwendung der strategischen Muster

[205] In Anlehnung an: ebd., S. 1ff.
[206] Vgl. ebd., S. 2ff.

des Domain-Driven Designs wird auf Geschäftsebene jedoch eine analytische Ausgangsbasis für die Transformation der Anwendungs- und Datenarchitektur geschaffen. Es wird folglich ein längerer Weg eingeschlagen, der durch die Ausrichtung von Geschäft und IT umfangreiche Auswirkungen mit sich bringt.

Die im Hufeisen-Modell dargestellten Kurvenverläufe veranschaulichen implizit, wie das vorhandene Wissen in Bezug auf die bestehende Lösung entdeckt, erweitert und verbessert wird und letztlich in die Ziellösung einfließt. Das ADM-Framework definiert diesbezüglich drei grundlegende Schritte, die den Weg einer Architektur-Transformation für jeden der drei Kurvenverläufe kennzeichnen:

(1) Wissensentdeckung in der bestehenden Lösung
(2) Definition der Zielarchitektur
(3) Umsetzung der Architektur

Die Tätigkeiten zur Wissensentdeckung haben das Ziel, den Ist-Zustand auf der jeweiligen Architekturebene abzubilden. Darauf aufbauend wird durch die Definition der Zielarchitektur der architekturelle Soll-Zustand beschrieben. Im dritten Schritt wird die Ist-Situation in den definierten Soll-Zustand überführt.[207]

Die fachliche Architektur spiegelt sich im Hufeisen-Modell in den Perspektiven der Geschäftsarchitektur sowie der Anwendungs- und Datenarchitektur wieder. Da sich die vorliegende Arbeit ausschließlich auf die Aspekte der fachlichen Architektur konzentriert, findet die Architektur-Perspektive der technischen Architektur im weiteren Verlauf der Untersuchungen keine Berücksichtigung.

[207] Vgl. ebd., S. 3.

4.2.2 Prozessmodell zur Architektur-Transformation

Di Francesco et al. schlagen einen allgemeinen Migrationsprozess vor, der an das Hufeisen-Modell von Kazman et al.[208] angelehnt ist. Der Prozess spiegelt ferner die drei Schritte von Architektur-Transformationen wieder, die im Hufeisen-Modell nach ADM identifiziert wurden (vgl. Kapitel 4.2.1). Er stellt demnach eine konkrete Ausprägung dieser Modelle im Kontext von Microservice-Architekturen dar. Die Schritte nach ADM werden im Migrationsprozess in drei Prozessphasen zusammengefasst:

(1) Reverse Engineering
(2) Transformation
(3) Forward Engineering

Das existierende System wird beim *Reverse Engineering* hinreichend analysiert, beispielsweise durch Code- und Daten-Analysen oder die Auswertung von Dokumentationen. Das Ziel dieser Analysen ist die Identifikation von Service-Kandidaten auf Basis des vorhandenen Wissens. In der *Transformationsphase* wird die bestehende Architektur auf logischer Ebene in die neue Microservice-Architektur überführt. Die Grundlage bilden die Informationen und das Wissen aus dem Reverse Engineering. Beim *Forward Engineering* wird die neu definierte Architektur schließlich implementiert.[209]

Der vorgeschlagene Migrationsprozess ist für eine vom Geschäft bzw. der Domäne getriebene Transformation der fachlichen Architektur jedoch nicht weitreichend genug definiert. Es fehlt die im Kontext von Domain-Driven Design relevante Perspektive der Geschäftsarchitektur. Das Domänenmodell fungiert dabei als zentrales Artefakt (vgl. Kapitel 2.4.1) für den Entwurf der Architektur und die Implementierung des neuen Systems.[210] Abbildung 19 zeigt eine entsprechend erweiterte Form des Prozessmodells. Sie legt dar, dass das Wissen über die bestehende Lösung auf Ebene der Anwendungs- und Datenarchitektur nicht als alleiniger Ausgangspunkt für die neue Architektur dient. In der Prozessphase des Reverse Engineerings wird die Ist-Situation unter Anwendung von Domain-Driven Design zusätzlich bis auf die logische Ebene der Geschäftsarchitektur analysiert. Sofern bereits ein Domänenmodell existiert, sollte dieses auf Aktualität überprüft, kritisch hinterfragt und nach den Prinzipien von Domain-Driven Design in serviceorientierte Domänenmodelle unterteilt werden. Falls kein Domänenmodell vorhanden ist, an dem sich die vorhandene Lösung orientiert, so können die serviceorientierten Domänenmodelle mit Hilfe von Domain-Driven Design auch direkt entwickelt werden. Die Domänenmodelle unterstützen dabei die Definition der Microservice-Architektur, die in der Prozessphase des Forward-Engineerings im neuen System implementiert wird.

Jegliche Ausführungen mit Bezug auf die Elemente der bestehenden Softwarelösung sind nachfolgend unter dem englischen Begriff *Legacy* zusammengefasst.

---

[208] Kazman et al. 1998.
[209] Vgl. Di Francesco et al. 2018, S. 29f.
[210] Vgl. Hippchen et al. 2017, S. 433.

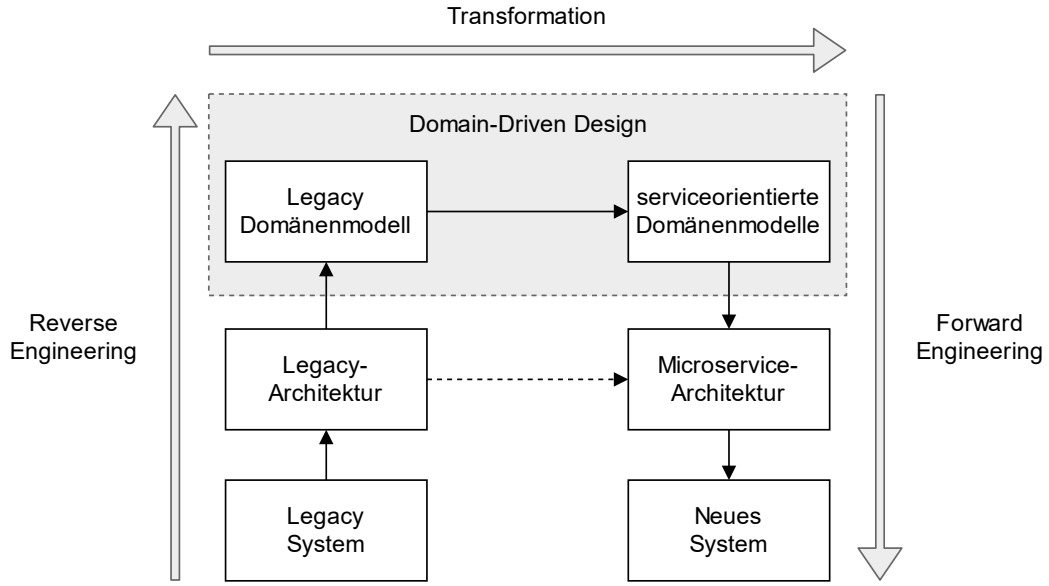

Abbildung 19: Allgemeiner Transformationsprozess [211]

Das in Abbildung 19 dargestellte Vorgehen besitzt in Bezug auf die durchzuführenden Aktivitäten und die beteiligten Elemente geringe Aussagekraft. Für eine ausführlichere Beschreibung dieser Aspekte kann das Rahmenwerk für SOA-Migrationen von Razavian und Lago[212] herangezogen werden. Es stellt eine Erweiterung des Hufeisen-Modells von Kazman et al. dar. Obwohl das Modell im Kontext von serviceorientierten Architekturen entwickelt wurde, ist es auch für Microservice-Architekturen anwendbar. Ausschlaggebend dafür ist die Service-Basiertheit beider Architekturstile und die hinreichend abstrakte Sicht des Rahmenwerkes auf den Transformationsprozess.
Die in Abbildung 18 und Abbildung 19 beschriebenen Ansätze werden in Abbildung 20 in einem *Prozessmodell* vereint und erweitert. Das Modell setzt sich aus Artefakten, Aktivitäten, Wissensbereichen sowie Prozess- und Informationsflüssen zusammen. Die *Artefakte* sind die zentralen Objekte, die in jeder der drei Prozessphasen entweder das Ergebnis oder die Grundlage einer bestimmten Aktivität darstellen. Die *Aktivitäten* spiegeln die notwendigen Schritte wieder, die während des Transformationsprozesses durchgeführt werden. Sie basieren auf vorhandenen Artefakten und zielen darauf ab, bestimmte Artefakte zu erzeugen oder zu modifizieren. Die *Wissenselemente* kennzeichnen Informationen, die in einem bestimmten Bereich des Prozesses benötigt werden. Das Wissen kann beispielsweise in Form von Daten, Modellen, Methoden oder Prinzipien vorliegen und hat im Kontext der durchzuführenden Aktivitäten maßgeblichen Einfluss auf den Ablauf des gesamten Prozesses.[213] Die *Informationsflüsse* kennzeichnen den Weg des Transformationsprozesses, der sich bis über die Ebene der Geschäftsarchitektur erstreckt, und spiegeln den Zusammenhang der Tätigkeiten wieder.

[211] In Anlehnung an: Di Francesco et al. 2018, S. 30.
[212] Razavian und Lago 2010.
[213] Vgl. ebd., S. 449.

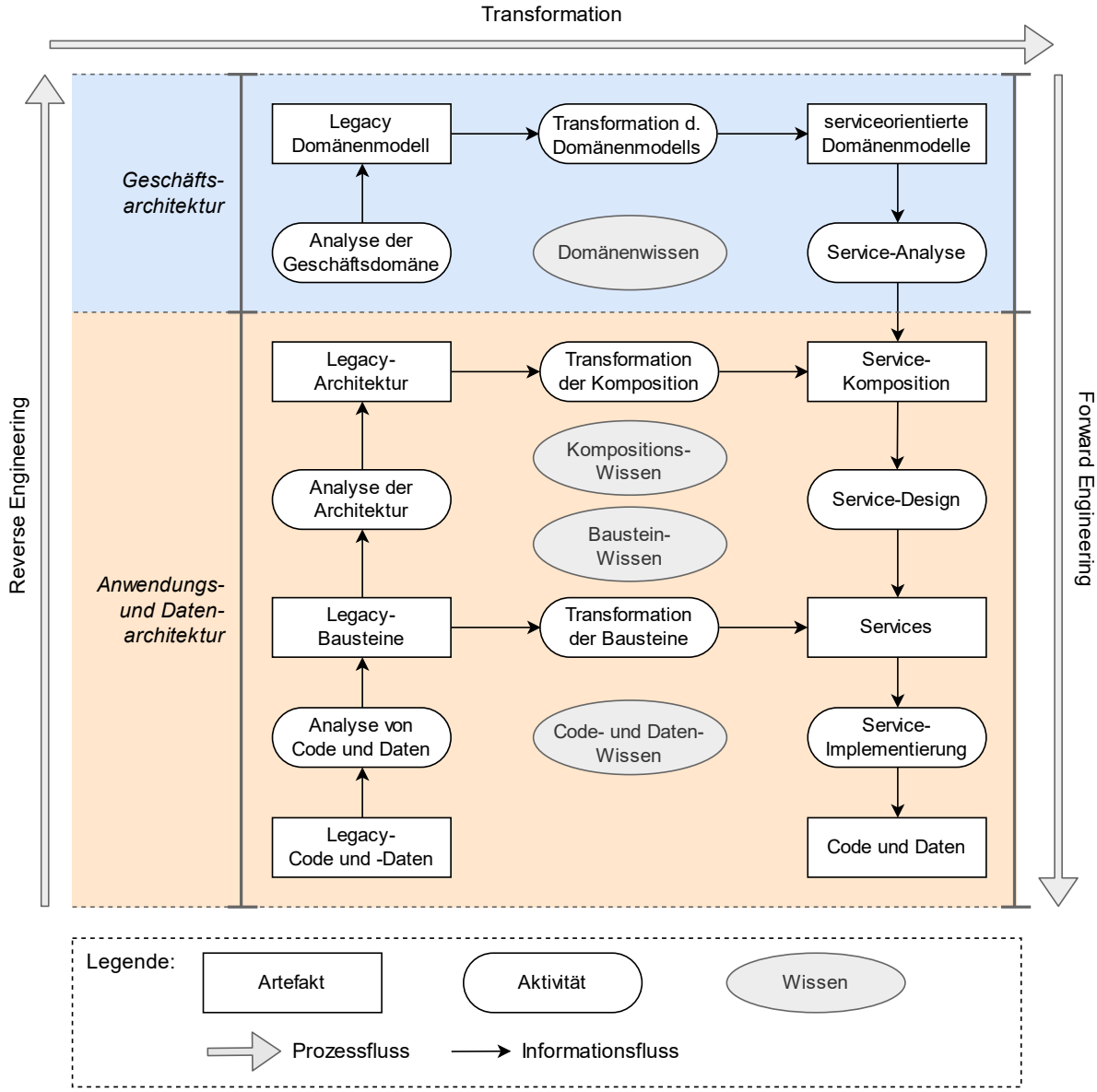

Abbildung 20: Prozessmodell zur Architektur-Transformation [214]

Da der Schwerpunkt dieser Arbeit auf der fachlichen Architektur liegt, werden im Prozessmodell ausschließlich die Abstraktionsebenen der Geschäftsarchitektur sowie der Anwendungs- und Datenarchitektur betrachtet. Sämtliche Aspekte im Kontext der technischen Architektur, z. B. die Transformation von Plattformen oder die Einführung bestimmter Technologien, finden keine Berücksichtigung.

Das Prozessmodell ist eine abstrakte Beschreibung des Transformationsprozesses, die als übergeordneter Rahmen für die Überführung monolithischer Legacy-Anwendungen in den Microservice-Architekturstil dient. Es stellt einen fachlich orientierten und von der Domäne getriebenen Transformationsprozess dar und vereint dafür die wesentlichen nicht-technisch motivierten Aktivitäten. Das Modell besitzt daher die Eigenschaft der

[214] In Anlehnung an: ebd., S. 447.

Technologieneutralität. Wie im Hufeisen-Modell beschrieben, verkörpert es einen langwierigen Transformationsprozess mit umfassenden Auswirkungen auf die Softwarearchitektur (vgl. Kapitel 4.2.1; Abbildung 18). Das Prozessmodell beinhaltet zwei valide Einstiegspunkte in der Prozessphase des Reverse Engineerings: die Analyse auf Ebene der Anwendungs- und Datenarchitektur und die Analyse der Geschäftsdomäne.

Reverse Engineering

In der Prozessphase des Reverse-Engineerings wird sowohl die Geschäftsarchitektur als auch die Struktur, die Funktionalität und das Verhalten des existierenden Systems analysiert. Dabei sollte ein möglichst umfangreiches Verständnis über die Domäne und das Legacy-System aufgebaut werden, um das Ziel der Identifizierung von Service-Kandidaten zu unterstützen.

Ausgehend von der bestehenden Implementierung werden zunächst die Bausteine des Altsystems – repräsentiert durch Module, Komponenten oder sonstige Code- und Daten-Bestandteile – durch Anwendung von Code- und Daten-Analysetechniken extrahiert. Die Extraktion der Legacy-Bausteine wird dabei sowohl durch Code- und Daten-bezogenes Wissen als auch durch Wissen höherer Ebenen beeinflusst. Auf dieser Grundlage kann die Analyse der Architektur erfolgen und das Zusammenspiel der einzelnen Bausteine untersucht werden. Dies geschieht durch Ausnutzung des vorhandenen Wissens zur Komposition wie z. B. den verwendeten Architekturstilen und Entwurfsmustern.

Als weiterer Einstiegspunkt wird die Geschäftsdomäne in Hinsicht auf die vorhandenen Geschäftsprozesse und die Fachlogik analysiert. Die Grundlage für diese Aktivität ist das vorhandene Wissen über die Geschäftsdomäne.[215] Im Kontext von Domain-Driven Design werden die geschäftlich relevanten Aspekte über Domänenmodelle repräsentiert. Sie stellen die zentralen Artefakte auf Ebene der Geschäftsarchitektur dar (vgl. Kapitel 2.4.1). Die Analyse der Geschäftsdomäne steht nicht in direkter Verbindung mit der Legacy-Architektur. Sie ist im Kontext von Domain-Driven Design eher als unabhängige Aktivität auf geschäftlicher Ebene einzuordnen. Dies wird durch die Erfahrungen von Knoche und Hasselbring bestätigt, nach denen die statischen Analysen von Quellcode und Daten auch nach der Analyse der Domäne und der Erzeugung des Domänenmodells stattfinden können.[216] Die Reihenfolge der Analysen ist folglich vernachlässigbar, solange das für die Architektur-Transformation benötigte Wissen auf allen Abstraktionsebenen vorhanden ist oder aufgebaut wird. Eine parallele Durchführung der Analysen ist daher ebenfalls möglich. In jedem Fall ist die Durchführung von allen im Prozessmodell dargestellten Aktivitäten des Reverse-Engineerings notwendig.

Das Ziel des domänengetriebenen Ansatzes ist die Ausrichtung der zu transformierenden Software an den geschäftlichen Gegebenheiten. Die Identifizierung von Service-Kandidaten ist dabei jedoch nicht ausschließlich durch die Aktivitäten auf Ebene der Geschäftsarchitektur bestimmt. Neben der Geschäftsdomäne können auch die vorhandenen Code-Bestandteile, Bausteine oder die Architektur entscheidende Informationen darüber liefern, welche Bestandteile des Legacy-Systems sich als Microservices eignen. Das Finden von

[215] Vgl. ebd., 448, 450.
[216] Vgl. Knoche und Hasselbring 2018, S. 45.

Service-Kandidaten kennzeichnet somit einen Prozess, der sich aus dem gewonnenen Wissen und den Erfahrungen auf allen Ebenen des Reverse Engineerings zusammensetzt.

Für das Reverse Engineering existieren generell keine Einschränkungen hinsichtlich der verwendbaren Informationsquellen. Es können z. B. der Quellcode, Datenmodelle, Tests, jegliche Dokumentationsformen wie Text- und Architekturdokumente, Modelle, Diagramme, Präsentationen sowie implizit im Team vorhandenes Wissen verwendet werden. Diese Informationen dienen dem allgemeinen Systemverständnis. Sie bilden die Basis für den Entwurf des neuen Systems und für sämtliche Entscheidungen, die während des gesamten Transformationsprozesses getroffen werden müssen.[217]

Nach Di Francesco et al. existieren verschiedene Herausforderungen in der Phase des Reverse Engineerings. Dazu zählen beispielsweise die folgenden Aspekte:

- Enge Kopplung der Legacy-Bausteine
- Unklare Grenzen zwischen den Bausteinen
- Fehlende oder schlechte Dokumentationen
- Geringes Verständnis von Quellcode und Programmiersprache[218]

Schlechte oder fehlende Dokumentationen sind in Bezug auf eine geplante Architektur-Transformation kritisch einzustufen. Eine Architekturdokumentation, die nicht den aktuellen Entwicklungsstand wiederspiegelt, ist bedeutungslos. Daher kann und sollte die Phase des Reverse Engineerings dazu genutzt werden, die Konformität von Quellcode und Architekturdokumentationen wiederherzustellen.[219]

Transformation

Die Prozessphase der Transformation wird durch drei Transformationsaktivitäten bestimmt: die *Transformation der Legacy-Bausteine*, die *Transformation der Komposition* und die *Transformation des Domänenmodells*. Die Legacy-Bausteine werden in Services mit einer bestimmten Granularität überführt. Dabei wird Wissen über den Code, die Daten und die Bausteine benötigt. Bei der Transformation der Komposition steht die Umwandlung der monolithisch verbundenen Bausteine in verteilte und sich gegenseitig aufrufende Services im Mittelpunkt. Hierbei spielt das Wissen über die Komposition und die Bausteine eine wesentliche Rolle. Dazu zählt zum einen das im Reverse Engineering gesammelte Wissen über das Legacy-System, z. B. welche Architekturstile oder Muster zur Service-Komposition zur Anwendung kommen. Zum anderen spielt das Wissen über die Ziellösung und damit verbundene Architekturstile eine grundlegende Rolle. Im Kontext

[217] Vgl. Di Francesco et al. 2018, S. 32.
[218] Vgl. ebd., S. 34f.
[219] Vgl. Hasselbring 2018, S. 180.

dieser Arbeit ist das notwendige Wissen vom Microservice-Architekturstil und dem domänengetriebenen Entwurf (vgl. Kapitel 4.3) geprägt. Auf Ebene der Geschäftsarchitektur wird die Transformation des Legacy-Domänenmodells primär durch die Motivationsfaktoren der Migration (vgl. Kapitel 4.1.1) und die Geschäftsanforderungen, die durch Geschäftsprozesse, Fachlogiken und Geschäftsstrategien gekennzeichnet sind, bestimmt. Sie bilden die Grundlage für den domänengetriebenen Entwurf der serviceorientierten Domänenmodelle (vgl. Kapitel 4.3) und die Implementierung des neuen Systems. Die Transformation wird dabei primär durch das Domänenwissen beeinflusst.[220]

Nach den Studienergebnissen von Di Francesco et al. wird die Transformation neben dem hohen Kopplungsgrad der Legacy-Bausteine vor allem durch die folgenden Herausforderungen und Aktivitäten bestimmt:

(1) Dekomposition des Legacy-Systems und der Domäne
(2) Identifizierung von Abhängigkeiten und Grenzen im Legacy-System
(3) Identifizierung von Service-Kandidaten und Service-Granularitäten
(4) Anwendung von Domain-Driven Design Praktiken
(5) Ausrichtung von Geschäft und IT
(6) Einbindung von Domänenexperten in den Transformationsprozess[221]

Die Ergebnisse der Studie bestätigen, dass die bedeutenden Aktivitäten und Herausforderungen mit dem Entwurf der fachlichen Architektur verbunden sind. Weiterhin ist ersichtlich, dass das tiefgreifende Domänenverständnis wesentlich für die Aufspaltung von Legacy-Systemen in Microservices ist. Die aufgeführten Punkte motivieren daher den Einsatz von Domain-Driven Design auf Ebene der Geschäftsarchitektur. Der domänengetriebene Entwurf der fachlichen Architektur auf Basis der Muster und Prinzipien des Domain-Driven Designs wird in Kapitel 4.3 beschrieben.

Weiterhin ist festzuhalten, dass die Identifikation von Service-Kandidaten nicht ausschließlich im Reverse Engineering erfolgt, sondern auch in der Transformationsphase eine Rolle spielt. Dies kann dadurch erklärt werden, dass die Transformationsphase als logischer und konzeptueller Schritt zu verstehen ist, in der keine tatsächlichen Umsetzungen im neuen System stattfinden. Weiterhin handelt es sich beim Entwurf der Zielarchitektur in der Regel um einen inkrementellen, iterativen und erfahrungsbasierten Prozess (vgl. Kapitel 4.5). Der Entwurf der Zielarchitektur wird dabei in mehreren Zyklen überprüft und schrittweise angepasst. Die Aktivitäten des Reverse Engineerings und der Transformationsphase sind folglich durch ein enges Zusammenspiel geprägt.

Forward Engineering

Beim Forward Engineering wird das Microservice-basierte System in einer *Top-Down* Vorgehensweise implementiert. Das geschieht sowohl auf Grundlage der Anforderungen und Ziele der neuen Microservice-basierten Umgebung als auch der entstandenen Artefakte aus der Prozessphase der Transformation. Auf Basis der serviceorientierten Domänenmodelle werden bei der *Service-Analyse* mögliche Service-Kompositionen identifi-

[220] Vgl. Razavian und Lago 2010, 448-451.
[221] Vgl. Di Francesco et al. 2018, 32–33, 35.

ziert, die die Geschäftsprozesse entsprechend abbilden. Eine *Service-Komposition* beschreibt dabei das Zusammenspiel verschiedener Services. Die Ergebnisse werden außerdem mit den kompositionsbezogenen Informationen, die während der Transformationsaktivitäten ausgehend von der Legacy-Architektur entstanden sind, abgeglichen und vereint. Im Rahmen der Aktivitäten des *Service-Designs* werden die einzelnen Microservices dann auf Grundlage der zuvor festgelegten Service-Kompositionen entworfen. Die Service-Kandidaten, die während der Transformation der Legacy-Bausteine identifiziert werden konnten, finden dabei ebenso Berücksichtigung. Im Rahmen der *Service-Implementierung* werden die einzelnen Microservices schließlich auf Basis der Service-Entwürfe im Quellcode umgesetzt.[222]

Die Artefakte der *Service-Komposition* und der *Services* nehmen im Modell eine Sonderrolle ein. Sie werden als einzige Artefakte jeweils durch zwei Aktivitäten bestimmt: einer transformierenden Aktivität auf der Ebene der Anwendungs- und Datenarchitektur und einer Top-Down-Aktivität, die auf dem Wissen aus der Ebene der Geschäftsarchitektur aufbaut. Beim Entwurf von Services und Service-Kompositionen findet somit ein Abgleich von Informationen statt, die sowohl auf der Domäne als auch der Anwendungs- und Datenarchitektur basieren. Die Anforderungen und Ziele auf Ebene der Geschäftsarchitektur werden dabei mit den Gegebenheiten und Lösungsmöglichkeiten auf Ebene der Anwendungs- und Datenarchitektur abgeglichen. Auf diese Weise können die Geschäftsdomäne und die IT-Domäne aneinander ausgerichtet werden (vgl. Kapitel 4.2.1).

Gemäß den Erfahrungen von Di Francesco et al. sind in der Phase des Forward Engineerings ein Großteil der Herausforderungen mit der Ebene der technischen Architektur verbunden. Da die technische Architektur im Kontext des Prozessmodells keine Berücksichtigung findet (vgl. Abbildung 20), werden diese Herausforderungen nicht betrachtet. In Bezug auf das Prozessmodell sind die folgenden Herausforderungen von Bedeutung:

(1) Neue Denkweisen und Zusammenarbeit im Entwicklungsteam
(2) Kommunikation und Wissensaustausch
(3) Vereinheitlichung durch Schaffen von Standards und Normen über Servicegrenzen hinweg[223]

Die Punkte (1) und (2) liegen in der Neuartigkeit des Microservice-Architekturstils und den notwendigen organisatorischen Anpassungen begründet. Diese Herausforderungen stehen außerdem in enger Verbindung mit der operativen Komplexität von Microservice-basierten Systemen und dessen Auswirkungen auf die Produktivität der Entwicklungsteams (vgl. Kapitel 4.1.1; Abbildung 16).

Servicebezogene Aktivitäten im Transformationsprozess

Das Prozessmodell aus Abbildung 20 beinhaltet sowohl implizit als auch explizit sichtbare Aktivitäten, die sich auf Microservices beziehen. Eine implizit im Modell enthaltene servicebezogene Aktivität ist die *Service-Identifikation*. Sie stellt ein übergeordnetes Ziel der Analysen in den Prozessphasen des Reverse Engineerings und auch der Transformation

[222] Vgl. Razavian und Lago 2010, 449, 451.
[223] Vgl. Di Francesco et al. 2018, S. 35f.

dar. Explizit im Modell sichtbar folgen darauf die Aktivitäten der *Service-Analyse*, des *Service-Designs* und der *Service-Implementierung*. In der Regel ist davon auszugehen, dass sich ein implementierter Service über die Zeit weiterentwickelt. Weiterhin müssen sich die gewählten Servicegrenzen im frühen Stadium der Service-Implementierung erst etablieren. Aufgrund unterschiedlicher Kriterien wie zu enger Kopplung oder zu niedriger Kohäsion bestimmter Microservices können weitere Restrukturierungen notwendig sein. Weiterhin können neue oder geänderte Anforderungen bestimmte Anpassungen am Service-Design und der Implementierung erfordern. Diese Dynamik im Transformationsprozess lässt sich unter dem Begriff der *Service-Evolution* zusammenfassen. In Abbildung 21 sind diese servicebezogenen Aktivitäten in einem zyklischen Modell zusammenhängend dargestellt, wobei die Service-Identifikation den Einstiegspunkt darstellt.

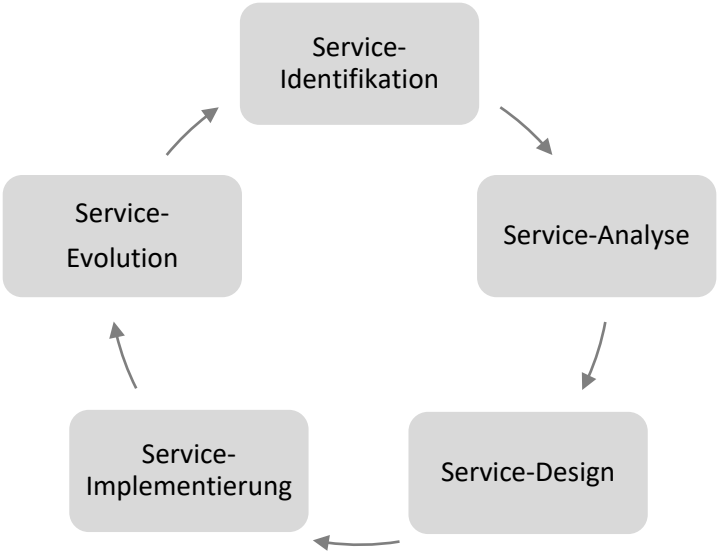

Abbildung 21: Servicebezogene Aktivitäten im Transformationsprozess

Modell-Artefakte im Transformationsprozess

Im Hufeisen-Modell nach ADM entstehen drei unterschiedliche Modell-Artefakte im Verlauf des Transformationsprozesses: das *Computation Independent Model (CIM)*, das *Platform Independent Model (PIM)* und das *Platform Specific Model (PSM)*. Das CIM ist ein Modell mit hohem Abstraktionsgrad, das unabhängig von den Details des Systementwurfs ist. Es kann als Informationsmodell bezeichnet werden und entspricht im weitesten Sinne dem *Domänenmodell*. Das PIM besitzt einen gewissen Grad an technologischer Unabhängigkeit, sodass es für eine bestimmte Menge von Plattformen anwendbar ist. Es bezieht sich auf den Entwurf der Anwendungsarchitektur und kann auch als *Design-Modell* bezeichnet werden. Das PSM ist ein Modell auf niedriger Abstraktionsebene und bezieht sich auf die Implementierungsebene eines Systems. Es trifft daher Aussagen über die Verwendung bestimmter Technologien bzw. Plattformen und kann unter dem Begriff *Implementierungs-Modell* zusammengefasst werden. Aufgrund der Entstehung und Ausnutzung

dieser Modelle in der Phase des Forward Engineerings kann der Transformationsprozess auch als *Modell-getriebener Ansatz* bezeichnet werden.[224]

Im Verlauf des Prozesses der Architektur-Transformation entstehen demnach drei unterschiedliche Modell-Artefakte auf verschiedenen Abstraktionsebenen. In Abbildung 20 ist von diesen Modell-Artefakten lediglich das Domänenmodell explizit abgebildet. Auf der Ebene der Anwendungs- und Datenarchitektur entstehen im Prozess außerdem Design-Modelle sowie Implementierungs-Modelle (Code-Modelle und Datenmodelle). Ausgehend vom Domänenmodell besteht eine sequenzielle Abhängigkeit der Modell-Artefakte, die sich über alle Ebenen erstreckt (vgl. Abbildung 22). Die Domänenmodelle dienen als Vorlage für den Entwurf der Architektur und die dabei entstehenden Design-Modelle, die den Architekturentwurf abstrahieren. Die Design-Modelle dienen wiederum als Grundlage für die Ableitung von Implementierungs-Modellen, die durch die Entstehung von Code- und Datenmodellen gekennzeichnet sind.

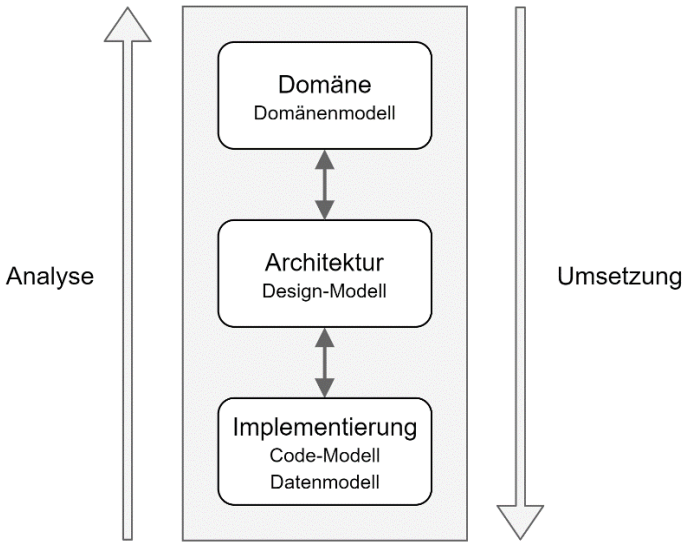

Abbildung 22: Modell-Artefakte im Transformationsprozess

Das Ziel während und nach der Transformation ist es, diese Modelle untereinander konsistent zu halten. Das Design-Modell sollte stets das Verhalten in der Software abbilden, das im Domänenmodell vorgegeben ist. Die Implementierungs-Modelle sollten wiederum die Fachlogik im Code und den Daten umsetzen, die aus der Domäne über das Design-Modell abgebildet wurde. Es besteht demnach eine transitive Abhängigkeit zwischen der Domäne und der Implementierung. Bezugnehmend auf die zentrale Rolle der Softwarearchitektur (vgl. Kapitel 2.1.4) wird zudem deutlich, dass das Design-Modell auf Ebene der Architektur als Bindeglied fungiert.

[224] Vgl. Pérez-Castillo et al. 2011, S. 80f.

4.2.3 Betrachtung weiterer wissenschaftlicher Ansätze

Im Rahmen der Literaturanalyse (vgl. Kapitel 3.1) wurden verschiedene Arbeiten identifiziert, die sich mit der Migration bzw. Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils befassen. In diesem Kapitel werden die Schnittbereiche des vorgestellten Transformationsprozesses (vgl. Abbildung 20) mit den bis hierhin nicht betrachteten Ansätzen beschrieben und diskutiert. Die Relevanz des vorgestellten Prozessmodells soll reflektiert werden, wobei der Schwerpunkt primär auf den Prozessphasen und -aktivitäten liegt.

In Balalaie et al. wurden vier für die Transformation der fachlichen Architektur relevante Muster (MP-2 bis MP-5) identifiziert.[225] Lediglich das Muster zur *Wiederherstellung der aktuellen Architektur (MP-2)* kann mit dem Prozessmodell (vgl. Abbildung 20) in Verbindung gebracht werden. Es zielt auf den Verständnisaufbau auf allen Abstraktionsebenen ab und kann daher der Phase des Reverse Engineerings zugeordnet werden. Die *Dekomposition des Monolithen (MP-3)* zielt auf eine domänengetriebene Dekomposition ab und ist auf Ebene der Geschäftsarchitektur im Prozessmodell enthalten. Die *Dekomposition des Monolithen auf Basis des Datenbesitzes (MP-4)* stellt einen alternativen Ansatz zum domänengetriebenen Entwurf dar, der im Umfeld von Domänen mit geringer Komplexität anwendbar ist und die Geschäftsarchitektur nicht einbezieht. Er findet daher keine Berücksichtigung in dieser Arbeit. Das *Ändern von Codeabhängigkeiten in Serviceaufrufe (MP-5)* ist eine Tätigkeit, die während der Transformation der Bausteine und der Service-Implementierung stattfindet. Sie ist damit implizit im Prozessmodell enthalten.

Das Rahmenwerk von Joselyne et al. fasst verschiedene in Migrationen auftretende Aktivitäten in fünf Phasen zusammen.[226] Dabei stellen einzig die ersten zwei Phasen einen Bezug zur fachlichen Architektur her. Die Aktivitäten der ersten Phase „Verständnis des existierenden Systems“ können in die Prozessphase des Reverse Engineerings eingeordnet werden. Der Prozess beginnt mit dem Aufbau von Wissen über die Domäne, was der Vorgehensweise in Knoche und Hasselbring[227] entspricht. Die Möglichkeit mit der Analyse der Geschäftsdomäne zu beginnen, ist auch im Prozessmodell (vgl. Abbildung 20) gegeben. In der zweiten Phase „Restrukturierung des existierenden Systems“ ist lediglich eine fachliche Aktivität „Extraktion von Geschäftslogik durch Code-Umstrukturierung“ aufgeführt, die im Prozessmodell in der Prozessphase der Transformation inkludiert ist. Alle weiteren aufgeführten Phasen und Aktivitäten sind technisch motiviert oder basieren auf technologischen Vorgaben. Sie sind für das fachlich ausgerichtete Prozessmodell nicht relevant.

[225] Vgl. Balalaie et al. 2018, S. 5ff.
[226] Vgl. Joselyne et al. 2017, S. 3ff.
[227] Knoche und Hasselbring 2018.

Taibi et al. schlagen ein Rahmenwerk zur Unterstützung des Migrationsprozesses vor, das in fünf Prozessphasen unterteilt ist und drei mögliche Prozessalternativen vorsieht.[228] Die Abgrenzung von drei Prozessalternativen ist vorwiegend strategisch bedingt und bezieht sich auf grundlegende Entscheidungen im Kontext von Migrationen. Unterscheidungen auf strategischer Ebene wurden bewusst nicht in das Prozessmodell integriert, da sie nicht zum Entwurf eines generisch anwendbaren Transformationsprozesses beitragen. Derartige Aspekte werden in Kapitel 4.4 im Kontext unterstützender Bereiche separat betrachtet. Die erste Prozessphase „Analyse der Systemstrukturen" entspricht dem Gedanken des Reverse Engineerings und enthält keine ergänzenden Aktivitäten in Bezug auf das Prozessmodell. Die Phase „Entwurf der System-Architektur" kann mit der Prozessphase der Transformation gleichgesetzt werden. Die dritte Phase „Priorisierung von Funktionen" bezieht sich erneut auf Strategien und Praktiken im Transformationsprozess, weshalb diese Aspekte unabhängig vom Prozessmodell betrachtet werden (vgl. Kapitel 4.4). Die zwei Phasen „Coding" und „Testing" beziehen sich auf die Implementierungsebene und werden im Prozessmodell über die Aktivität der Service-Implementierung abgebildet.

In Fan et al. wird ein Migrationsprozess vorgestellt, der auf einem spezifischen Entwicklungsprozess basiert und den Schwerpunkt auf technische Entscheidungen legt.[229] Der Prozess beginnt mit der Durchführung einer Domänenanalyse im Kontext von Domain-Driven Design. Die Aktivität zur Analyse der Geschäftsdomäne ist auch im Prozessmodell (vgl. Abbildung 20) vorhanden, wenngleich das Modell keine expliziten Vorgaben zur Anwendung von DDD macht. Die zweite Aktivität gibt die Analyse des Datenbankschemas vor. Entsprechende Tätigkeiten werden im Prozessmodell über die Aktivität „Analyse von Code und Daten" abgebildet. Alle weiteren im Migrationsprozess vorgeschlagenen Aktivitäten beziehen sich auf die technische Architektur und technisch motivierte Entscheidungen, die während einer Migration getroffen werden müssen. Sie sind für das fachlich ausgerichtete Prozessmodell daher nicht relevant.

Bucchiarone et al. berichten über Erfahrungen, die im Rahmen der Migration eines monolithischen Bankensystems gesammelt werden konnten.[230] Bei der Migration wurde ein domänengetriebenes Vorgehen verwendet, das jedoch nicht näher spezifiziert wird. Der Aspekt der Domänenorientierung spiegelt sich in der Abstraktionsebene der Geschäftsarchitektur im Prozessmodell (vgl. Abbildung 20) wieder. Alle weiteren Erfahrungen beziehen sich auf strategische Entscheidungen und angewendete Praktiken. Derartige Aspekte werden getrennt zum Prozessmodell betrachtet (vgl. Kapitel 4.4).

[228] Vgl. Taibi et al. 2017, S. 28ff.
[229] Vgl. Fan und Ma 2017, S. 110f.
[230] Vgl. Bucchiarone et al. 2018, S. 51.

4.3 Domänengetriebener Entwurf

Das in Abbildung 20 vorgestellte Prozessmodell beschreibt einen Prozess zur Architektur-Transformation, der sich bis über die Ebene der Geschäftsarchitektur erstreckt (vgl. Kapitel 4.2.2). Das Vorgehen kann folglich als domänengetrieben bezeichnet werden. Der domänengetriebene Entwurf ist eine fachliche Dekompositionstechnik, die eine Domäne in Domänenmodelle zerlegt und diese zu einem System komponiert.[231] Eine verbreitete Möglichkeit für den domänengetriebenen Entwurf der fachlichen Architektur ist die Anwendung der Muster und Prinzipien des Domain-Driven Designs (vgl. Kapitel 2.4).

4.3.1 Allgemeines Vorgehen mit Domain-Driven Design

Ein zentrales Konzept von Domain-Driven Design ist das *Knowledge Crunching*. Es hat das Ziel die relevantesten Informationen aus der Problemdomäne zu destillieren, um ein nützliches Domänenmodell zu erstellen, das die Anforderungen der geschäftlichen Anwendungsfälle erfüllen kann. Beim Knowledge Crunching soll durch enge Zusammenarbeit von Software- und Domänenexperten ein tiefes, gemeinsames Verständnis der Problemdomäne erreicht werden. Dies geschieht durch verschiedene Analysetechniken wie z. B. gemeinsames Brainstorming oder Aufstellen von geschäftlichen Anwendungsfällen *(Use Cases)* und Szenarien, die von der Softwarelösung umgesetzt werden sollen (vgl. Kapitel 4.3.1).[232] Es handelt sich um einen kontinuierlichen Lernprozess, der das Feedback aller Anspruchsgruppen iterativ und in abstrahierter Form in das Modell einfließen lässt. Ein Domänenmodell kann dadurch schrittweise optimiert werden.[233] Abbildung 23 stellt das Prinzip des Knowledge Crunchings als Bestandteil des allgemeinen Vorgehens im Domain-Driven Design dar.

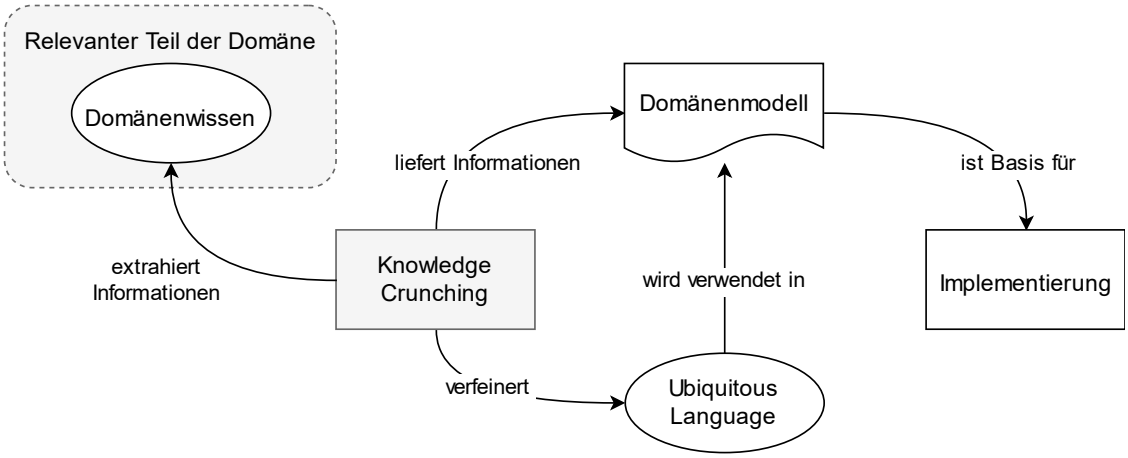

Abbildung 23: Allgemeines Vorgehen im Domain-Driven Design

[231] Vgl. Takai 2017, S. 23f.
[232] Vgl. Millett und Tune 2015, S. 15f.
[233] Vgl. Evans 2003, S. 9f.

Die Softwareentwicklung ist gemeinhin stark durch Analysetätigkeiten geprägt. Der analytische Ansatz des Domain-Driven Designs macht von den grundlegenden Konzepten des Problemraums *(problem space)* und des Lösungsraums *(solution space)* Gebrauch. Der Problemraum spiegelt die Domäne und damit die zu lösende strategische Herausforderung des Unternehmens wieder. Der Lösungsraum wird durch die Software repräsentiert, die es zu entwickeln gilt, um das spezifische Problem des Unternehmens zu lösen. Domain-Driven Design stellt im Problemraum die Subdomänen in den Mittelpunkt. Im engeren Sinne handelt es sich um die Kerndomäne und die umgebenden Subdomänen, die für die Lösung des spezifischen Problems relevant sind (vgl. Kapitel 2.4.1). Im Lösungsraum spielt das Muster der Bounded Contexts eine wesentliche Rolle (vgl. Kapitel 2.4.2). Bounded Contexts dienen als Rahmengerüst für die Implementierung einzelner Bereiche der Softwarelösung und grenzen das zugehörige Domänenmodell dafür fachlich und semantisch ab. Nach Vernon sollten Bounded Contexts und Subdomänen idealerweise eins-zu-eins aufeinander abgebildet werden. Dadurch werden Domänenmodelle in klar definierte Geschäftsbereiche unterteilt, um den Problemraum mit dem Lösungsraum zu verschmelzen. Bei Neuentwicklungen ist dies leichter umzusetzen als in Legacy-Systemen. Komplexe monolithische Legacy-Systeme besitzen meist große Bounded Contexts, die sich über verschiedene Subdomänen erstrecken. Weiterhin kann es vorkommen, dass eine Subdomäne durch mehrere Bounded Contexts umgesetzt wird.[234] Darüber hinaus kann es auch erwünscht sein mehrere Subdomänen durch einen Bounded Context abzubilden, wenn es sich um eine valide Kontextgrenze in Bezug auf die enthaltenen Domänenobjekte und deren Schnittstellen handelt.[235]

Abbildung 24 fasst die beschriebenen Zusammenhänge grafisch zusammen:

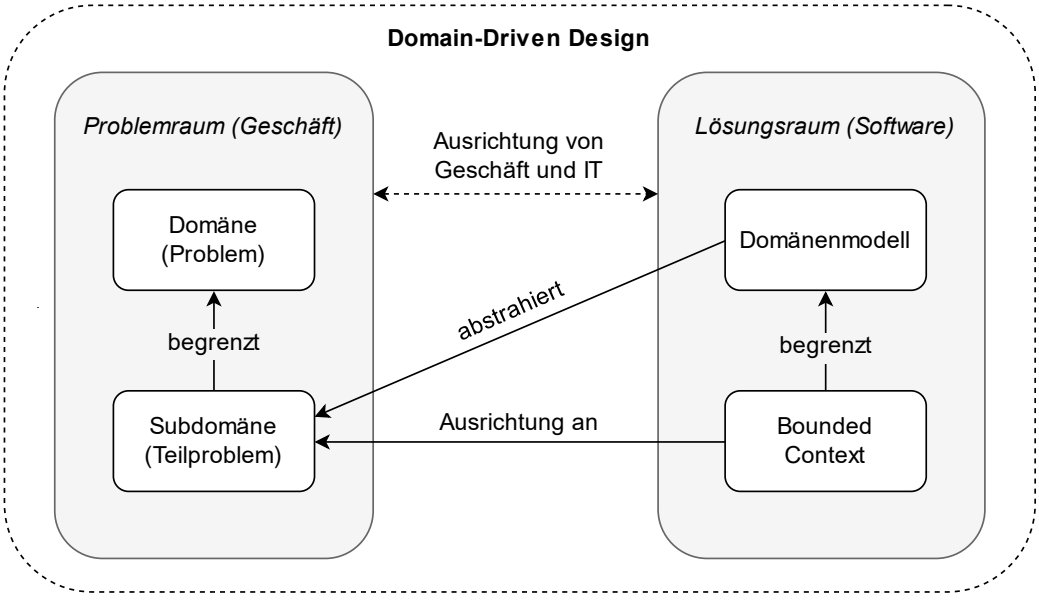

Abbildung 24: Domain-Driven Design im Problem- und Lösungsraum

[234] Vgl. Vernon 2013, Kap. 2 Abschn. „Real-World Domains and Subdomains".
[235] Vgl. Dowalil 2018, Kap. 4.3.

Die aufgeführten Zusammenhänge spiegeln eines der Kernziele von Domain-Driven Design wieder: *die Ausrichtung von Geschäft (Problemraum) und IT (Lösungsraum)*. Weiterhin ist es das Ziel eine Teamstruktur zu etablieren, bei der jeder Bounded Context von genau einem Team innerhalb einer Subdomäne verantwortet wird (vgl. Kapitel 2.4.2).[236] In diesem Zusammenhang ist die Bedeutung des Gesetzes von Conway erkennbar (vgl. Kapitel 2.3.1). Durch Anwendung des Conway Manövers folgt die Systemarchitektur den Strukturen der Organisation bzw. der Geschäftsarchitektur. Nach den Erfahrungen von Pautasso et al. kann eine entsprechende Umstrukturierung der Organisation sogar dazu führen, dass sich die Bounded Contexts und Services weitestgehend automatisch ergeben.[237]

[236] Vgl. Uludağ et al. 2018, S. 239.
[237] Vgl. Pautasso et al. 2017a, S. 96.

4.3.2 Orientierung an Geschäftsfähigkeiten

Die Theorie des Domain-Driven Designs nach Evans[238] beinhaltet keine klaren Handlungsanweisungen zur Identifizierung von Subdomänen und Bounded Contexts. Weiterhin werden keine konkreten Aussagen über die Größe von Subdomänen und Bounded Contexts getroffen. Grundsätzlich ist davon auszugehen, dass Subdomänen und Bounded Contexts verschiedene Größenformen annehmen können. Weiterhin wird die Granularität von der betrachteten Abstraktionsebene bestimmt. Grobgranulare Subdomänen und Bounded Contexts sind oftmals einfacher zu identifizieren. Sie können jedoch häufig weiter unterteilt werden. In der Theorie des Domain-Driven Designs wird nicht beschrieben, wie diese Unterteilung vorgenommen werden kann.

Gemäß der Microservice-Definition von Lewis und Fowler orientieren sich Microservices entlang von *Geschäftsfähigkeiten* (vgl. Kapitel 2.2.3). Durch die zusätzliche Betrachtung von Geschäftsfähigkeiten *(Business Capabilities)* innerhalb einer Domäne können die zuvor beschriebenen Limitierungen von Domain-Driven Design abgeschwächt werden.
Eine Geschäftsfähigkeit beschreibt was ein Unternehmen zur Erreichung eines Geschäftsziels tut und abstrahiert dafür die beteiligten Informationen, Prozesse, Menschen und Technologien. Sie repräsentiert damit eine Menge von zusammenhängenden Aspekten innerhalb einer Subdomäne, die zur Erfüllung eines bestimmten Geschäftsziels benötigt werden. Der von einer Geschäftsfähigkeit abstrahierte Geschäftsprozess beschreibt hingegen wie dieses Geschäftsziel erreicht wird.[239] Geschäftsfähigkeiten sind die Bausteine des Unternehmens, die überwiegend stabil sind. Sie sind weniger detailliert und verändern sich im Gegensatz zu Geschäftsprozessen über die Zeit nicht oder nur selten.[240]

Nach Richardson werden sowohl Subdomänen als auch Geschäftsfähigkeiten durch Analysen des Geschäfts identifiziert. Beide Ansätze verfolgen dabei die Identifizierung von fachlich abgrenzbaren Bereichen und die Analyseergebnisse sind häufig ähnlich.[241] Beim Vergleich der Definitionen von Geschäftsfähigkeiten und (Sub-) Domänen (vgl. Kapitel 2.4.1) fällt auf, dass beide Begriffe weitestgehend gleichbedeutend sind. Das Konzept der Geschäftsfähigkeiten wird im Rahmen dieser Arbeit folglich als gedankliche Erweiterung von Subdomänen verwendet. Analog zu Subdomänen sind Geschäftsfähigkeiten als Teil des Problemraumes anzusehen (vgl. Kapitel 4.3.1). Die Identifikation von Subdomänen kann demnach auch durch die Orientierung an vorhandenen Geschäftsfähigkeiten erfolgen. Durch die transitive Abhängigkeit können außerdem die dazugehörigen Bounded Contexts an den Geschäftsfähigkeiten ausgerichtet werden.

---

238 Evans 2003.
239 Vgl. Frey et al. 2015, S. 6.
240 Vgl. Richardson 2018, S. 51.
241 Vgl. ebd., S. 54.

Nach Newman spiegelt dies die präferierte Denkweise über Bounded Contexts wieder:

> *„you should be thinking [...] about the [business] capabilities those contexts provide the rest of the domain.“*[242]

Zusammenhängende Funktionalitäten werden folglich durch einen Bounded Context abgegrenzt und auf eine zugehörige Geschäftsfähigkeit abgebildet. Auf diese Weise wird ermöglicht, dass ein Microservice eine einzige Geschäftsfähigkeit implementiert.[243]
Abbildung 25 stellt die Zusammenhänge aus Abbildung 24 in einer entsprechend erweiterten Form dar. Im Rahmen dieser Abhängigkeitsbeziehung ist sowohl der Grundgedanke von DDD enthalten (vgl. Kapitel 4.3.1) als auch die Definition von Microservices erfüllt (vgl. Kapitel 2.2.3). Die dargestellte Abhängigkeitsbeziehung ist daher maßgebend für die weiteren Ausführungen in dieser Arbeit.

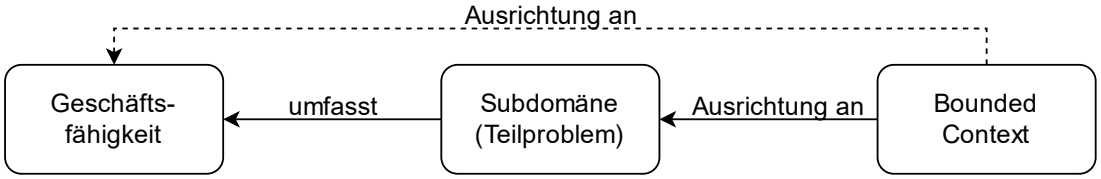

Abbildung 25: Orientierung an Geschäftsfähigkeiten

Nach Frey et al. kann auf Basis der Geschäftsanforderungen und der Geschäftsstrategie eine *Business Capability Map* erstellt werden, die die Geschäftsfähigkeiten eines Unternehmens bzw. einer Domäne abbildet. Analog zu Subdomänen, können Geschäftsfähigkeiten dabei unterschiedlichen Abstraktionsebenen angehören.[244] Durch die Zuordnung von Microservices zu Geschäftsfähigkeiten können sich folglich Services unterschiedlicher Granularitäten ergeben. Gemäß den in Abbildung 25 dargestellten Beziehungen ist dieses Prinzip auch auf Subdomänen und Bounded Contexts übertragbar. Im Demonstrationsteil der Arbeit ist ein Beispiel einer Business Capability Map aufgeführt, das verschiedene Geschäftsfähigkeiten unterschiedlicher Abstraktionsebenen enthält (vgl. Kapitel 5.2.1).

Die Orientierung von Bounded Contexts an Geschäftsfähigkeiten sollte mit besonderer Aufmerksamkeit in Hinsicht auf das Gesetz von Conway erfolgen (vgl. Kapitel 2.3.1). Die Geschäftsfähigkeiten eines Unternehmens sind nicht immer korrekt an der Problemdomäne ausgerichtet. Entsprechende Bounded Contexts würden folglich zu Microservice-basierten Systemen führen, die die Kommunikationsstrukturen der Organisation wiederspiegeln und die zugehörige Domäne nicht hinreichend repräsentieren.[245]

[242] Newman 2015, S. 34.
[243] Vgl. Dragoni et al. 2018, S. 95f.
[244] Vgl. Frey et al. 2015, S. 5ff.
[245] Vgl. Millett und Tune 2015, S. 83.

4.3.3 Prozessmodell zum domänengetriebenen Entwurf

Die Ausführungen in Kapitel 4.3.1 machen deutlich, dass durch ein domänengetriebenes Vorgehen unter Anwendung von Domain-Driven Design ein fachlicher Schnitt erreicht werden kann, der die Ausrichtung von Geschäft und IT fördert. Weiterhin kann die Entkopplung der identifizierten Softwarebestandteile und die damit einhergehende Unabhängigkeit der Teams unterstützt werden.

Aufbauend auf Kapitel 2.4 und den Ergebnissen in Kapitel 4.3.1 kann ein Prozessmodell für den domänengetriebenen Entwurf hergeleitet werden (vgl. Abbildung 26).

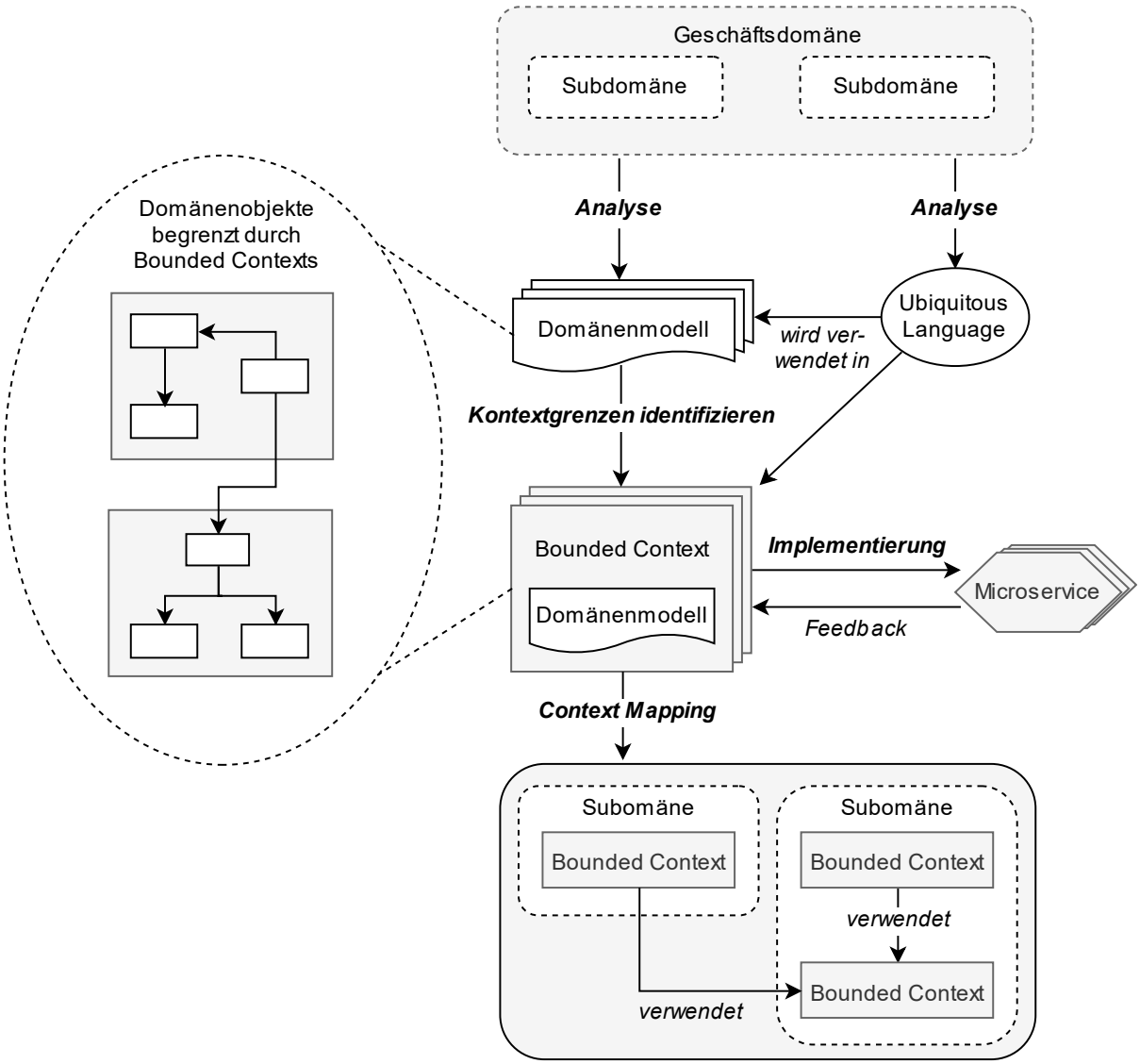

Abbildung 26: Prozessmodell zum domänengetriebenen Entwurf

Das vorgeschlagene Modell zum domänengetriebenen Entwurf legt den Schwerpunkt auf die Elemente des strategischen Designs (vgl. Kapitel 2.4.2). Die Muster des taktischen Designs dienen in erster Linie dem Aufbau von Domänenmodellen (vgl. Kapitel 2.4.3). Es handelt sich demnach um DDD-spezifische Konzepte, die sich auf die Implementierungsebene beziehen und für das Finden eines angemessenen fachlichen Schnittes von geringerer Bedeutung sind.

Die *Analyse der Geschäftsdomäne* dient dem Wissensaufbau und liefert Informationen für die Abbildung der Domänenlogik in Domänenmodellen. Gleichzeitig dient sie der Definition und Verfeinerung der allgegenwärtigen, gemeinsamen Sprache aller Beteiligten (Ubiquitous Language). Diese Sprache wird in jedem Domänenmodell durch einen Bounded Context eingegrenzt (vgl. Kapitel 2.4.2; Abbildung 11). Wird diese Sprache geändert, so ändert sich auch das Domänenmodell. Die Analyse der Domäne bzw. Subdomänen kann durch verschiedene Techniken erfolgen (vgl. Kapitel 4.3.4).

Bei der *Erstellung der Domänenmodelle* sollten die Aspekte der hohen Qualität und Konsistenz im Mittelpunkt stehen. Die Nutzung von Bounded Contexts ist daher von großer Bedeutung für die klare und eindeutige Abgrenzung der modellierten Konzepte und der damit verbundenen Aufteilung in separate, kohäsive Domänenmodelle. Die Konzepte bzw. Domänenobjekte, die in der Geschäftsdomäne mehrfach und inkonsistent zur Anwendung kommen, können innerhalb eines Bounded Context konsistent verwendet und unabhängig entwickelt werden.[246] Für jeden identifizierten Kontext sollte ein einziges Domänenmodell entwickelt werden, das an der dazugehörigen Subdomäne bzw. Geschäftsfähigkeit ausgerichtet ist (vgl. Kapitel 4.3.1 und 4.3.2).[247]

Ein *identifizierter Bounded Context*, der ein Domänenmodell fachlich sowie semantisch eingrenzt, gilt gemeinhin als Kandidat für die *Implementierung in einem Microservice*. Obwohl diese Relation kennzeichnend für den domänengetriebenen Entwurf ist, kann der Serviceschnitt von weiteren Faktoren beeinflusst werden (vgl. Kapitel 4.4.2).[248] Darüber hinaus kann ein Bounded Context durch mehrere Microservices implementiert werden (vgl. Kapitel 2.4.2; Abbildung 11).[249] Ausschlaggebend hierfür können beispielsweise verschiedene technische Aspekte und Qualitätsanforderungen oder die Größe eines Bounded Context sein. Weiterhin kann nicht nur das Domänenmodell innerhalb eines Bounded Context auf einen Microservice abgebildet werden, sondern auch ein einzelnes Aggregat, das als Wurzelelement logisch zusammenhängender Domänenobjekte fungiert (vgl. Kapitel 2.4.3).[250]

Die *Identifizierung von Kontext- und Servicegrenzen* ist häufig eine Herausforderung (vgl. Kapitel 4.2.2; Abschnitte „Reverse Engineering“ und „Transformation“). Die Grenzen müssen jedoch nicht bereits im ersten Entwurf optimal festgelegt werden. Vielmehr handelt es sich um einen Annäherungsprozess, bei dem die Grenzen mit zunehmendem Domänenwissen nachträglich angepasst werden können. Auch das regelmäßige Feedback aus

[246] Vgl. Stine 2015, S. 24f.
[247] Vgl. Vernon et al. 2017, S. 47.
[248] Vgl. Fan und Ma 2017, S. 110f.
[249] Vgl. Evans und Wolff 2015, S. 16.
[250] Vgl. Takai 2017, S. 29.

der Implementierung kann zur Stabilisierung der Bounded Contexts beitragen (vgl. Kapitel 4.5). Die folgenden drei Anhaltspunkte können die Identifizierung von Kontextgrenzen unterstützen:

(1) Mehrdeutigkeit in Terminologie und Konzepten der Domäne
(2) Ausrichtung an Subdomänen und Geschäftsfähigkeiten (vgl. Kapitel 4.3.2)
(3) Organisations- und Teamstrukturen[251]

Nachdem verschiedene Bounded Contexts identifiziert wurden, können deren Beziehungen untereinander mittels *Context Mapping* in einer Kontextlandkarte abgebildet werden. Dabei können unterschiedliche Formen der Integration zwischen Bounded Contexts zur Anwendung kommen (vgl. Kapitel 2.4.2). Weiterhin können über Kontextlandkarten implizit auch Teambeziehungen abgebildet werden, da ein Bounded Kontext von genau einem Team verantwortet wird.

Das in Abbildung 26 vorgeschlagene Prozessmodell verdeutlicht, dass ein Microservice nicht größer als seine kontextuelle Grenze – sein Bounded Context – werden kann. Aufgrund verschiedener weiterer Kriterien, die Einfluss auf die Servicegröße haben können (vgl. Kapitel 4.4.2), kann ein Microservice auch kleiner als der zugehörige Bounded Context sein. Der domänengetriebene Entwurf dient daher in erster Linie dem Finden von Grenzen auf Makroarchitekturebene. Er besitzt das Ziel die grobe fachliche Architektur zu entwerfen und eignet sich primär für die initiale Dekomposition monolithischer Anwendungen.[252] Innerhalb eines Bounded Context kann eine weitere Dekomposition nach verschiedenen Kriterien erfolgen, beispielsweise dem *Single Responsibility Principle* (vgl. Kapitel 2.2.3 und 4.4.2).
Das Prozessmodell zeigt weiterhin, dass beim domänengetriebenen Entwurf mehrere separate Domänenmodelle entstehen, die durch Bounded Contexts begrenzt sind. Durch diese fachliche Abgrenzung wird ein verteilter Ansatz im Sinne des Microservice-Architekturstils ermöglicht. Die Kontexte können als Basis für die Implementierung autonomer Microservices dienen, die von einzelnen Microservice-Teams unabhängig entwickelt und verantwortet werden können.

[251] Vgl. Millett und Tune 2015, S. 82.
[252] Vgl. Balalaie et al. 2018, S. 7.

4.3.4 Techniken zur Domänenanalyse

Die Analyse der Geschäftsdomäne kann durch die Anwendung verschiedener Techniken erfolgen. In diesem Kapitel wird eine Auswahl von Ansätzen beschrieben und klassifiziert, die in der Praxis häufig zur Anwendung kommen. Die Analysetechnik des *Event Stormings* wird dabei genauer betrachtet, da sie sich von dem Großteil der Techniken besonders abhebt und im Demonstrationsteil dieser Arbeit angewendet wird (vgl. Kapitel 5.2).

Die gängigsten Methoden zur Erforschung der Domäne sind nutzerzentriert, d. h. die Domäne wird ausgehend von den Anforderungen der Nutzer des Systems analysiert. Die Betrachtung von bestimmten Anwendungsfällen *(Use Cases)* ist eine Möglichkeit zur Erforschung einer Domäne anhand von Aktivitäten, die ein Nutzer durchführen muss, um ein bestimmtes Ziel zu erreichen. Einen Schritt weiter geht der Ansatz des *Behavior-Driven Developments*. Hier werden aus Sicht des Nutzers verschiedene *User Stories* und *Szenarien* formuliert, die einen Einblick in das erwartete Verhalten des Systems geben und dadurch die Anforderungen bzw. Akzeptanzkriterien des Nutzers offenlegen.[253] Eine mit Use Cases verwandte Technik ist das *Domain Storytelling*. Im Gegensatz zu Use Cases konzentriert sich diese Technik auf den intensiven Austausch zwischen dem Domänenexperten als Erzähler und dem Zuhörer, um möglichst schnell ein gemeinsames Verständnis über die Domäne und dessen Sprache aufzubauen.[254]

In Kontrast zu den nutzerorientierten Analysetechniken kann eine Domäne auch in einer Art Rückwärtsgang erforscht werden. Die Technik des Event Stormings legt den Schwerpunkt dabei nicht auf die Domänenobjekte, sondern auf die fachlichen Ereignisse der Domäne *(Domain Events)*. Diese Ereignisse verkörpern Tatsachen bzw. Dinge, die bereits passiert sind und sich nicht mehr ändern oder rückgängig machen lassen. Im Zentrum von Event Storming steht dabei die Offenlegung der bedeutenden fachlichen Ereignisse innerhalb der Domäne sowie deren auslösende Aktionen *(Commands)*. Dadurch können wesentliche Abhängigkeiten sowie Prozess- und Datenflüsse identifiziert werden.[255] Event Storming kann sowohl als initialer Schritt für die Entwicklung von Domänenmodellen als auch zur Identifizierung von Bounded Contexts angewendet werden.[256] Beim Event Storming werden in erster Linie die Elemente des taktischen Designs zur Modellierung verwendet, um die Strukturen und das Verhalten der Domäne abzubilden. Darauf aufbauend kann mit den Bounded Contexts und dem Context-Mapping eine Verbindung zum strate-

[253] Vgl. Millett und Tune 2015, S. 20ff.
[254] Vgl. Workplace Solutions GmbH.
[255] Vgl. Bonér 2017, S. 17ff.
[256] Vgl. Familiar 2015, S. 17.

gischen Design hergestellt werden. Außerdem kann Event Storming zum Ausbau der gemeinsamen Sprache und der Identifizierung der betroffenen Subdomänen sowie der Kerndomäne beitragen.[257]

Neben den Nutzer- und Tatsachen-zentrierten Analysetechniken existieren auch Vorgehensweisen zur Erforschung der Domäne, die als intermediäre Ansätze klassifiziert werden können. Dazu zählt beispielsweise die Analyse bereits existierender Artefakte und Modelle.[258] Diesen Ansätzen können ebenso die Modellierung von Sub- und Kerndomänen sowie die Technik des Context Mappings zugeordnet werden.

Abbildung 27 stellt die beschriebenen Techniken und Zusammenhänge grafisch dar. Die Abbildung zeigt sowohl das Domänenwissen als zentralen Gegenstand der Analysen als auch die Zuordnung der verschiedenen Analysetechniken, die dieses Wissen aus unterschiedlichen Richtungen und Ansatzpunkten untersuchen.

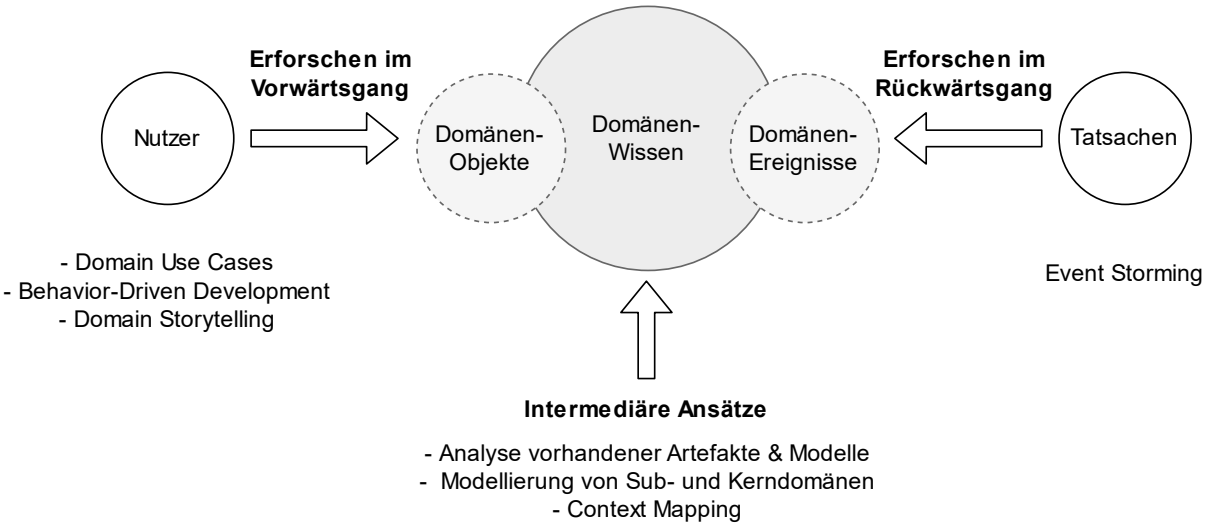

Abbildung 27: Analysetechniken im Domain-Driven Design

[257] Vgl. Millett und Tune 2015, S. 25.
[258] Vgl. ebd., S. 24.

4.4 Strategien und Praktiken im Transformationsprozess

Dieses Kapitel stellt eine Sammlung von grundlegenden Praxiserfahrungen dar, die das Prozessmodell zur Architektur-Transformation (vgl. Kapitel 4.2.2) unterstützen und erweitern. Sie kennzeichnen bedeutende Entscheidungsbereiche des Transformationsprozesses. Dabei wird auf Strategien und Vorgehensmuster, Dekompositionspraktiken in Bezug auf die Servicegröße, Priorisierungsaspekte der Service-Implementierung und die Bedeutung von Qualitätseigenschaften eingegangen.

4.4.1 Vorgehensmuster und Strategien

In komplexen monolithischen Systemen ist eine Transformation der Architektur, die in einem einzigen großen Schritt erfolgt, nur mit hohem Aufwand und Risiko durchführbar. Dieses Vorgehen entspricht einer Neuentwicklung und Abschaltung des Legacy-Systems, im Englischen auch *cut-and-run* genannt. Gemäß der *parallel operations strategy* können das Legacy-System und das neue System auch nebeneinander betrieben werden. Beide Systeme arbeiten dabei solange parallel zueinander, bis das neue System alle Funktionen des alten Systems abdeckt und das Altsystem abgeschaltet werden kann. Neue Funktionen müssen in diesem Fall jedoch in beiden Systemen oder ausschließlich im neuen System implementiert werden. In der Mehrheit der Anwendungsfälle produktiv laufender Systeme sollte daher die Methode der schrittweisen Transformation der Architektur gewählt werden.[259] Zu den Hauptgründen dafür zählen das verminderte Risiko, das mit den Änderungen einhergeht, die allgemeine Stabilität des Gesamtsystems sowie die Möglichkeit schrittweise Erfahrungen sammeln und bei auftretenden Problemen flexibler reagieren zu können. Darüber hinaus können Änderungen leichter wieder rückgängig gemacht werden. Insgesamt besteht hinreichender Konsens darüber, dass ein inkrementelles bzw. phasenweises Vorgehen in den meisten Fällen die sinnvollste Strategie darstellt.[260] Nach den Praxiserfahrungen von Di Francesco et al. ist dies zudem die am Häufigsten angewendete Vorgehensweise.[261]

Di Francesco et al. berichten weiterhin, dass während einer Migration häufig neue Funktionalitäten umgesetzt werden.[262] Um einen großen und komplexen Monolithen nicht weiter auszubauen, sollte neue Funktionalität, wenn möglich, direkt als Microservice implementiert werden, statt diese weiter in den Monolithen einzubauen.[263] Die Entscheidung, ob neue Funktionalität als Microservice oder im Monolithen umgesetzt wird, ist von

---

[259] Vgl. Wagner 2014, S. 42f.
[260] Vgl. Zhu et al. 2016, S. 33; Kalske et al. 2018, S. 39; Fan und Ma 2017, S. 110.
[261] Vgl. Di Francesco et al. 2018, S. 34.
[262] Vgl. ebd., S. 33.
[263] Vgl. Stine 2015, S. 30.

verschiedenen Faktoren abhängig. Gehört die Funktionalität beispielsweise zu einer neuen Geschäftsfähigkeit, kann sie problemlos als Microservice umgesetzt werden. Auf der anderen Seite kann eine neue Funktion auch lediglich darin bestehen, eine bestimmte Geschäftsfähigkeit im Monolithen durch kleine Änderungen zu erweitern. In diesem Fall sollte die Funktion zunächst im Monolithen implementiert werden. Zu einem späteren Zeitpunkt kann die Geschäftsfähigkeit dann als Microservice extrahiert werden.[264]
Nach den Erkenntnissen von Taibi und Lenarduzzi sollten bei der Umsetzung von Microservices zudem nicht nur die Funktionalitäten bzw. Geschäftsfähigkeiten betrachtet werden, sondern auch Geschäftsprozesse, die vom Monolithen isoliert werden können.[265]

In Konsequenz ist eine *Integration der Microservices mit dem monolithischen Altsystem* notwendig. Dabei können zwei Formen von Integrationen unterschieden werden. Die Abhängigkeit eines Monolithen von einem Microservice ist unproblematisch und sogar erwünscht. Umgekehrt wirkt sich die Abhängigkeit eines Microservice von einem Monolithen negativ aus, da sich der Microservice nicht unabhängig entwickeln kann. In diesem Fall kann die Integration über eine *Antikorruptionsschicht (Anticorruption Layer)* im Sinne des Domain-Driven Designs erfolgen. Mittels wohldefinierter Schnittstellenkontrakte und Übersetzungsmechanismen sorgt diese Schicht dafür, dass sich die Domänenmodelle des Microservice und des Monolithen nicht gegenseitig beeinflussen und keine enge Kopplung entsteht.[266] Das grundlegende Ziel besteht folglich darin die Abhängigkeiten der Microservices zum Monolithen zu minimieren.[267]

Ein weiterführendes Vorgehensmuster, das eine Kombination aus den zuvor genannten Ansätzen darstellt, ist das *Strangler Application Pattern*[268] nach Fowler. Bei diesem Muster wird die Funktionalität schrittweise aus dem Monolithen extrahiert und an diesen angebunden.[269] Eine Funktionalität bleibt dabei solange im Monolithen erhalten, bis sie durch einen Microservice vollständig abgelöst werden kann. Neue Funktionalität wird ebenfalls in Microservices umgesetzt und an das monolithische System angebunden. Entgegen der Strategien „parallel operations“ und „cut-and-run“ handelt es sich um ein schrittweises Ersetzen und Abschalten der Implementierung im Altsystem. Die sogenannte *Strangler Application* wächst über die Zeit parallel zur monolithischen Anwendung weiter, wohingegen die Größe des Monolithen kontinuierlich abnimmt.[270] Das monolithische System kann so entweder vollständig in Microservices aufgeteilt werden oder

[264] Vgl. Richardson 2018, S. 434f.
[265] Vgl. Taibi und Lenarduzzi 2018, S. 61.
[266] Vgl. Stine 2015, S. 30f.
[267] Vgl. Dehghani 2018, Abschn. „Minimize Dependency Back to the Monolith“.
[268] Fowler 2004.
[269] Vgl. Stine 2015, S. 32.
[270] Vgl. Richardson 2018, S. 430ff.

es bleibt ein kleiner Teil des Monolithen bestehen, beispielsweise aufgrund eines schlechten Aufwand-Nutzen-Verhältnisses.[271]
Analog zur Idee des *Walking Skeleton* bietet die Anwendung des Strangler Application Patterns eine vergleichsweise risikoarme Möglichkeit, möglichst schnell und früh Erfahrungen mit dem Microservice-Architekturstil zu sammeln und die Transformation der Architektur iterativ und schrittweise voranzutreiben.[272] Dabei kann es hilfreich sein die Microservices zunächst als *Proof-of-Concept (POC)* -Services oder *Minimum Viable Products (MVPs)* zu entwickeln.[273]

Von Bedeutung ist auch die *Atomarität* der durchgeführten Transformationsschritte. Unabhängig von der Größe der Inkremente sollte jeder Schritt entweder abgeschlossen oder rückgängig gemacht werden. Im Sinne von *evolutionären Architekturen* sollte daher in jedem Schritt eine Annäherung an den gewünschten Zielzustand erfolgen. Dabei ist jede atomare Einheit durch das Vorgehensmuster der „Implementierung, Verwendung und Abschaltung" bestimmt. Nach der Implementierung und aktiven Verwendung eines Microservice folgt somit die Abschaltung der alten Bestandteile im Monolithen.[274]

Extraktion oder Neuentwicklung

Das Herauslösen einer Geschäftsfähigkeit bzw. eines Bounded Context aus dem Monolithen kann durch die Wiederverwendung der vorhandenen Implementierung (Extraktion) oder durch eine Neuentwicklung erfolgen. Die *Neuentwicklung* einer Geschäftsfähigkeit kann verschiedene Vorteile mit sich bringen. Sie bietet beispielsweise Gelegenheit, über die Zeit getroffene Annahmen und Einschränkungen kritisch zu hinterfragen und die eingesetzten Technologien zu erneuern. Die *Wiederverwendung* der alten Implementierung sollte nur unter bestimmten Voraussetzungen in Erwägung gezogen werden. Der Wert der Implementierung sollte hoch sein, d. h. der Code sollte intellektuell bedeutungsvoll sowie komplex sein und die Implementierung sollte wenig Altlasten und technische Schulden beinhalten. Weiterhin sollte die zu extrahierende Geschäftsfähigkeit an klaren Domänenkonzepten ausgerichtet sein. Für die Mehrheit der komplexen monolithischen Systeme treffen diese Kriterien für gewöhnlich nicht zu. Daher ist die Neuimplementierung in der Regel der Extraktion vorzuziehen.[275] Beide Ansätze schließen sich jedoch nicht gegenseitig aus, sodass in der Praxis eine *Kombination aus Extraktion und Neuimple-*

[271] Vgl. Stine 2015, S. 33.
[272] Vgl. Monson-Haefel 2009, S. 120f.
[273] Vgl. Di Francesco et al. 2018, S. 32; Sharma et al. 2016, S. 616.
[274] Vgl. Dehghani 2018, Abschn. „Migrate in Atomic Evolutionary Steps".
[275] Vgl. ebd., Abschn. „Decouple Capability and not Code".

mentierung denkbar ist. Beispielsweise können wertvolle und komplexe Codebestandteile wiederverwendet werden, während Codebestandteile mit niedrigem Wert neu implementiert und technische Schulden abgebaut werden.

Die Extraktion ist häufig mit *Anpassungen am Monolithen* verbunden, um die Geschäftsfähigkeit als Microservice herauslösen zu können. In Knoche und Hasselbring wird zunächst eine interne Umstrukturierung im Monolithen vorgenommen, bevor die Extraktion der Funktionalitäten in Microservices erfolgt. Dabei werden Module identifiziert, die über das spezielle Konzept externer und interner Service-Fassaden servicefähig gemacht werden, um sie letztendlich als Microservices extrahieren zu können.[276] Diese Vorgehensweise kann mit dem Konzept der Modularisierung und dem Muster der Anti-Korruptionsschicht zur Entkopplung von Komponenten in Verbindung gebracht werden.

4.4.2 Größe von Microservices

Beim Entwurf von Microservices, vor allem in der Phase der Transformation (vgl. Kapitel 4.2.2), stellt sich die Frage nach der Servicegröße bzw. -granularität. Die Bestimmung von Servicegrößen ist in erster Linie ein Findungsprozess. Die Granularität der Microservices ergibt sich dabei primär durch die Anforderungen an die Anwendung und die Erfahrungen, die während des Transformationsprozesses gesammelt wurden.[277] Die Servicegröße ist allerdings kein ausschlaggebendes Kriterium. Der Schwerpunkt während der Transformation sollte vielmehr auf der Realisierung der Eigenschaften und Vorteile des Microservice-Architekturstils liegen. Es ist daher sicherzustellen, dass die entstehenden Services im Sinne der Eigenschaften des Microservice-Architekturstils entworfen werden. Dieser Entwurfsprozess kann durch verschiedene Entwurfsmuster und Richtlinien zur Dekomposition unterstützt werden. Eine bedeutende Richtlinie besteht darin, die entstehenden Microservices nach dem *Single Responsibility Principle* zu entwerfen. Dabei sollte die Grenze eines Microservice an einer Geschäftsfähigkeit ausgerichtet werden (vgl. Kapitel 2.2.3 und 4.3.2).[278] Zudem ist auch der Aspekt der losen Kopplung einzelner Microservices beim Entwurf zu berücksichtigen (vgl. Kapitel 2.2.3).

Im Kontext von Domain-Driven Design und dem domänengetriebenen Entwurf (vgl. Kapitel 4.3) kann der *Bounded Context* als obere Grenze und das *Aggregat* (vgl. Kapitel 2.4.3) als untere Grenze für die Servicegröße herangezogen werden. Ein Microservice kann demnach nicht größer als ein Bounded Context und nicht kleiner als ein Aggregat sein.[279]

Nach Wolff können sechs Faktoren unterschieden werden, die Einfluss auf die Größe eines Microservices nehmen. Das Auffinden der unteren und oberen Grenze wird durch jeweils drei dieser Faktoren beeinflusst. Abbildung 28 stellt die Einflussfaktoren in einer Übersicht dar.

---

[276] Vgl. Knoche und Hasselbring 2018, S. 45ff.
[277] Vgl. Pautasso et al. 2017b, S. 97.
[278] Vgl. Venugopal 2017, S. 23201.
[279] Vgl. Schwartz 2017, S. 592.

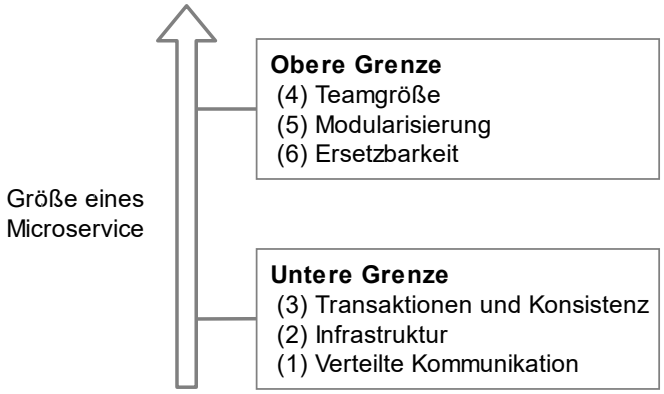

Abbildung 28: Größe eines Microservice (Einflussfaktoren) [280]

(1) Je mehr Microservices vorhanden sind, desto mehr **verteilte Kommunikation** findet statt, was die Performanz der Anwendung negativ beeinflusst. Die verteilte Kommunikation nimmt mit der Anzahl der Services zu, weshalb die Größe der Microservices nicht zu klein gewählt werden sollte.

(2) Für jeden Microservice muss eine eigene **Infrastruktur** für alle Umgebungen bereitgestellt werden. Je nach eingesetzter Technologie für den Service und für die Bereitstellung der Infrastruktur kann dies unterschiedlich aufwendig sein. Bei hohem Bereitstellungsaufwand sollte die Anzahl der Microservices geringgehalten und die Services größer geschnitten werden. Bei der Nutzung automatisierter Infrastruktur-Bereitstellungsverfahren relativiert sich der Aufwand jedoch, sodass die Infrastruktur kein entscheidendes Kriterium darstellt.

(3) Microservices sind zu klein gewählt, sofern Microservice-übergreifende Transaktionen genutzt werden und dabei Datenkonsistenz über mehrere Microservices notwendig ist. Die Servicegröße sollte demnach so festgelegt werden, dass **Transaktionen und Konsistenz** nicht mehrere Microservices gleichzeitig betreffen.[281]

(4) Die **Teamgröße** hat Einfluss auf die obere Grenze der Servicegröße. Ein Microservice darf nicht so groß werden, dass mehrere Teams sich an der Arbeit beteiligen müssen oder die Unabhängigkeit eines Microservice-Teams nicht gewährleistet werden kann.

(5) Ein Microservice sollte nur so groß werden, dass eine angemessene **Modularität** vorherrscht und die Wahrscheinlichkeit für das nachträgliche Verschieben von Funktionalität in andere Microservices gering ist. Die Größe sollte ebenfalls so gewählt werden, dass ein einzelner Entwickler den Microservice verstehen und weiterentwickeln kann.

[280] In Anlehnung an: Wolff 2016, Kap. 4.1 Abb. 4-1.
[281] Vgl. ebd., Kap. 4.1.

(6) Microservices sollten einfach zu ersetzen sein. Da die **Ersetzbarkeit** mit zunehmender Größe abnimmt, sollte ein Microservice nie so groß werden, dass er nicht mehr leicht zu ersetzen ist. Die Grenze liegt dabei unterhalb der Modularisierung, da ein Microservice verständlich sein muss, um ersetzt werden zu können.[282]

Nach Gouigoux und Tamzalit kann die Servicegröße auch durch das *Verhältnis der Kosten für die Qualitätssicherung und für das Deployment* beeinflusst werden. Die Kosten sind dabei stets mit dem geschaffenen Mehrwert für das Geschäft in Verbindung zu setzen. Die Erfahrungen bestätigen zudem, dass es sich bei der Festlegung von Servicegrößen um einen erfahrungsbasierten Annäherungsprozess handelt.[283]

Die Faktoren zur Bestimmung der Servicegröße, die in diesem Abschnitt aufgeführt wurden, können als Orientierungshilfe im Transformationsprozess (vgl. Kapitel 4.2.2) dienen. Das Festlegen geeigneter Servicegrößen hängt allerdings auch vom speziellen Einsatzkontext und den eingesetzten Technologien ab.[284]

Eine mögliche *Strategie* besteht darin, *mit grobgranularen Services zu beginnen* und diese später in kleinere Microservices aufzuteilen. Beispielsweise kann die hohe operationale Komplexität von Microservice-Architekturen für dieses Vorgehen sprechen. So besteht die Möglichkeit, erst nach dem Erreichen einer gewissen operationalen und prozessualen Reife zu feingranularen Microservices überzugehen.[285]

4.4.3 Priorisierung der zu entwickelnden Microservices

Nachdem die Service-Kandidaten im Reverse Engineering identifiziert und der Sollzustand in der Transformationsphase definiert wurde, gilt es die Zielarchitektur in der Phase des Forward Engineerings zu implementieren (vgl. Kapitel 4.2.2). Die einzelnen Microservices sollten dabei in einer entsprechend priorisierten Reihenfolge umgesetzt werden.[286]

Ein zentraler Aspekt für die Priorisierung ist die Betrachtung des Wertes, den die Umsetzung eines Microservice für das Geschäft liefert. Die Umsetzung wird umso höher priorisiert, je größer der *Geschäftswert* ist. Ergänzend dazu kann die *Anzahl der Abhängigkeiten* eines Microservice für die Priorisierung herangezogen werden. Besitzen zwei Microservices einen ähnlichen Geschäftswert, so ist der Microservice mit weniger Abhängigkeiten höher zu priorisieren. Er ist unabhängiger und kann in der Regel schneller und mit geringerem *Risiko* bzw. Auswirkungen auf die Architektur des bestehenden Systems umgesetzt werden. Weiterhin können jene Microservices höher priorisiert werden, die in Verbindung mit Geschäftsfähigkeiten stehen, die einer hohen *Änderungshäufigkeit* unterliegen oder viele *Fehler* beinhalten.[287]

---

[282] Vgl. ebd., Kap. 4.1.
[283] Vgl. Gouigoux und Tamzalit 2017, S. 62f.
[284] Vgl. Wolff 2016, Kap. 4.1.
[285] Vgl. Dehghani 2018, Abschn. „Go Macro First, then Micro“.
[286] Vgl. Larrucea et al. 2018, S. 97.
[287] Vgl. Taibi et al. 2017, S. 30.

Nach Di Francesco et al. kann die Priorisierung auch durch die *Anforderungen des Minimum Viable Products (MVPs)* bestimmt werden. Demnach werden die Microservices zuerst umgesetzt, die Bestandteil eines definierten MVPs sind. Da ein MVP an den grundlegenden Geschäftsfunktionen ausgerichtet ist, orientiert sich dieser Ansatz ebenfalls am Aspekt des Geschäftswertes. Darüber hinaus können auch problembehaftete Geschäftsfunktionen, deren Fehler sich nur schwer im Monolithen beheben lassen, als Kandidaten für höher priorisierte Microservices herangezogen werden.[288]

Grundsätzlich sollte bei der Priorisierung eine *Abwägung von Kosten und Nutzen* erfolgen. Der Nutzen bzw. Geschäftswert kann beispielsweise darin bestehen die Entwicklungsgeschwindigkeit zu verbessern, indem die Geschäftsfähigkeiten höher priorisiert werden, die in der Vergangenheit häufig geändert wurden oder in der Zukunft vielen Änderungen unterliegen werden. Weiterhin kann der Nutzen in der Lösung einer Problemsituation bestehen. Dies betrifft z. B. die Teile der Anwendung, die Probleme in Hinsicht auf die Aspekte der Performanz, Skalierbarkeit oder Zuverlässigkeit besitzen. Eine entsprechend priorisierte Umsetzung hat daher unmittelbaren Einfluss auf die Qualitätseigenschaften der Softwareanwendung (vgl. Kapitel 4.4.4). Letztlich kann der Nutzen auch in der Schaffung von Möglichkeiten begründet liegen. Beispielsweise kann die Extraktion eines Microservice die Extraktion weiterer Microservices durch verbesserte Abhängigkeitsbeziehungen begünstigen.[289]

Zusammenfassend können die beschriebenen Aspekte zur Priorisierung sowohl aus der Sicht des Geschäfts als auch der Anwendung betrachtet werden:

Geschäftssicht	Anwendungssicht
- Geschäftswert - Risiko - MVP-Anforderungen	- Abhängigkeiten - Änderungshäufigkeit - Enthaltene Fehler

Tabelle 9: Aspekte der Priorisierung

Die Entwicklung einer Priorisierungsstrategie basiert in den meisten Fällen nicht auf der isolierten Betrachtung einzelner Aspekte, sondern auf der Kombination verschiedener Priorisierungsaspekte. Nach Dehghani ist die *kombinierte Berücksichtigung der Aspekte des Geschäftswertes und der Änderungshäufigkeit* von besonders großer Bedeutung. Demnach sollten Teile der Anwendung, die einen großen Wert für das Geschäft besitzen und häufigen Änderungen unterliegen, zuerst in Microservices extrahiert werden.[290]

---

[288] Vgl. Di Francesco et al. 2018, S. 34.
[289] Vgl. Richardson 2018, S. 442.
[290] Vgl. Dehghani 2018, Abschn. „Decouple What is Important to the Business and Changes Frequently".

4.4.4 Konzentration auf die Qualitätseigenschaften

Die Architektur eines Softwaresystems hat maßgebenden Einfluss auf die Umsetzung qualitativer Eigenschaften in der Software. Zudem lassen sich getroffene Architekturentscheidungen zu einem späteren Zeitpunkt nur schwer rückgängig machen (vgl. Kapitel 2.1.1). Folglich sollte der Schwerpunkt auch im Architektur-Transformationsprozess auf den Qualitätseigenschaften der Software liegen. Sämtliche Architekturentscheidungen sollten daher mit Bedacht getroffen werden und zur Erreichung der vereinbarten Qualitätseigenschaften beitragen.[291]

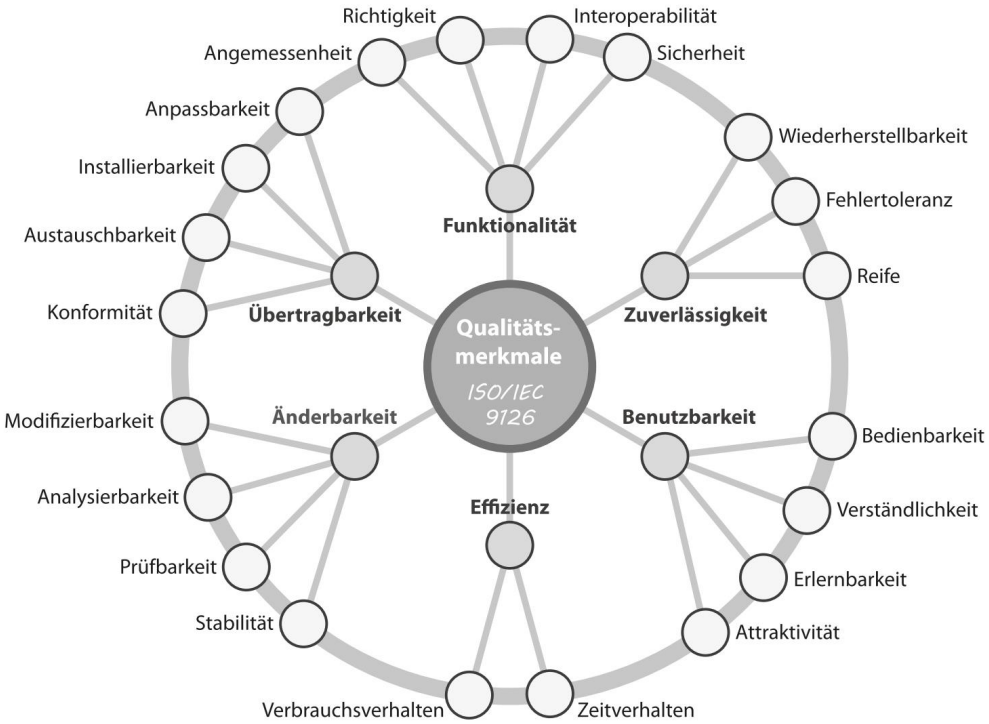

Abbildung 29: Qualitätseigenschaften nach ISO/IEC 9126 [292]

Abbildung 29 zeigt die Qualitätseigenschaften von Softwaresystemen nach der *ISO/IEC 9126*[293] Norm. Der Standard kann während des Transformationsprozesses als Orientierungshilfe für die Realisierung verschiedener Qualitätseigenschaften verwendet werden. Einige dieser Qualitätsmerkmale korrelieren mit den Vorteilen von Microservices (vgl. Kapitel 2.2.3) und den Motivationsfaktoren für den Einsatz des Microservice-Architekturstils (vgl. Kapitel 4.1.1). Beispielsweise sind Microservices durch die geringe Größe und Modularität leicht verständlich, änderbar und austauschbar. Weitere Qualitätseigenschaften sind die hohe Zuverlässigkeit aufgrund der hohen Fehlertoleranz und Verfügbarkeit sowie die gute Skalierbarkeit einzelner Services.

---

[291] Vgl. Hasselbring 2018, S. 176.
[292] Toth 2014, S. 46.
[293] Siehe: ISO/IEC 9126-1:2001.

4.5 Betrachtungen im Kontext agiler Softwareentwicklung

Die Ausführungen in Kapitel 2.3.3 haben gezeigt, dass sich der Microservice-Architekturstil mit den Werten und Prinzipien agiler Softwareentwicklung vereinen lässt. Weiterhin wurde in den Kapiteln 2.5.2 und 4.1.2 dargelegt, dass die Einführung des Microservice-Architekturstils durch die Prinzipien der agilen Softwareentwicklung unterstützt werden kann. In diesem Kapitel werden daher die Tätigkeiten der Architektur-Transformation (vgl. Kapitel 4.2) und des domänengetriebenen Entwurfs (vgl. Kapitel 4.3) im Kontext der agilen Softwareentwicklung betrachtet.

Nach den Erfahrungen von Di Francesco et al. werden bei Migrationen, neben der Extraktion und Neuimplementierung existierender Funktionalitäten (vgl. Kapitel 4.4.1), häufig auch neue Geschäftsfunktionen implementiert. Der Aspekt der Agilität ist daher auch für den Prozess der Architektur-Transformation von großer Relevanz.[294]
Der agile Software-Entwicklungsprozess ist auf Iterationen und schnelle Feedbackzyklen ausgelegt. Der Grund dafür ist die kontinuierliche Anpassung von Zielen und Anforderungen an die Software. Folglich ist auch die Softwarearchitektur ständigen Änderungen unterworfen. Es existiert kein fest definierter Zielzustand. Daher sollte die Softwarearchitektur iterativ angepasst werden und schrittweise entstehen. Während des Prozesses der Architektur-Transformation entstehen so mehrere Zwischenlösungen der Architektur, die stets das Ziel verfolgen sich an die aktuell definierte Ziellösung anzunähern. Der Ist-Zustand der Softwarearchitektur entfernt sich dadurch zunehmend von der ursprünglich geplanten Lösung. Im Projektverlauf und mit jeder Zwischenlösung wird der Spielraum für Architekturentscheidungen dabei immer kleiner.[295]
Abbildung 30 stellt die beschriebenen Zusammenhänge zur Entstehung von Softwarearchitekturen dar:

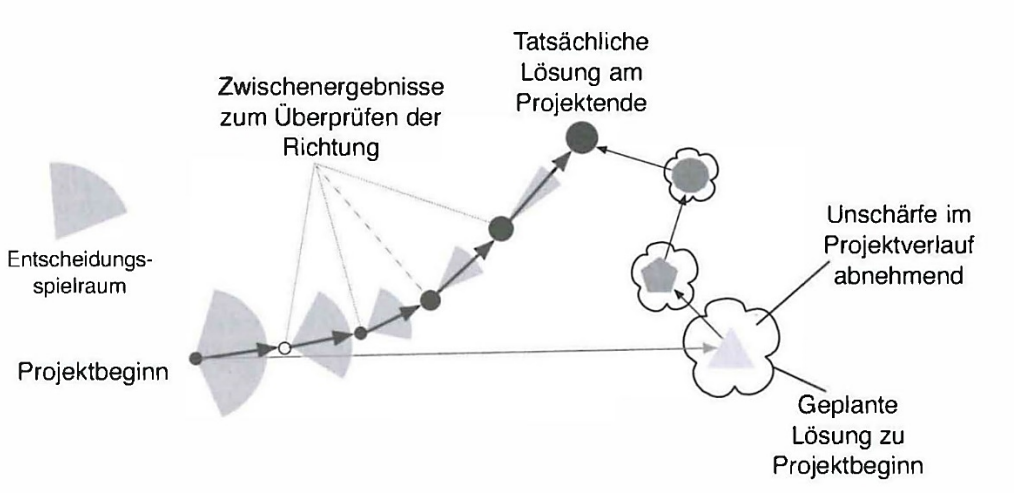

Abbildung 30: Architektur in der agilen Softwareentwicklung [296]

[294] Vgl. Di Francesco et al. 2018, S. 36.
[295] Vgl. Starke 2018, S. 26f.
[296] Ebd., S. 27.

Durch das Herauszögern bedeutender Architektur- und Designentscheidungen kann verhindert werden, dass unnötige Komplexität in der Softwareanwendung aufgebaut wird. Dabei ist zu berücksichtigen, dass die Entscheidungen in einem späten, aber verantwortungsvollen Moment und nicht dem letztmöglichen Moment getroffen werden. Weiterhin sollte die Architekturarbeit ein integraler Bestandteil aller Entwicklungsaktivitäten sein. Die Entwicklung der Architektur erfolgt dabei inkrementell und sollte im Rahmen der Formulierung von Anforderungen, beispielsweise in Form von User Stories und Use Cases, vorangetrieben werden.[297]

Die Ausführungen in den Kapiteln 2.3.3 und 2.5.2 belegen, dass die Werte und Prinzipien der agilen Softwareentwicklung eine bedeutende Rolle im Umfeld von Microservice-Architekturen einnehmen. Sowohl Microservices als auch die agile Softwareentwicklung verfolgen das Ziel der regelmäßigen Auslieferung von Software- bzw. Produktinkrementen. In diesem Zusammenhang spielt die Verbindung der Architektur und Umsetzung eine wesentliche Rolle. Die Umsetzung basiert auf den Vorgaben der Softwarearchitektur (vgl. Kapitel 2.1.4) und die Entwicklung der Architektur ist im Gegenzug auf stetiges Feedback von den Umsetzungsaktivitäten angewiesen. Dadurch ergibt sich im Kontext agiler Softwareentwicklung eine enge Verzahnung der Architektur mit der Umsetzung, die auf einem ständigen Abgleich und Erfahrungsaustausch beider Bereiche beruht. Ein Entwicklungsvorgehen, das auf agilen Werten und Prinzipien beruht, kann daher maßgeblich zu dem Erreichen der notwendigen Flexibilität beitragen. Abbildung 31 stellt die beschriebenen Zusammenhänge dar.

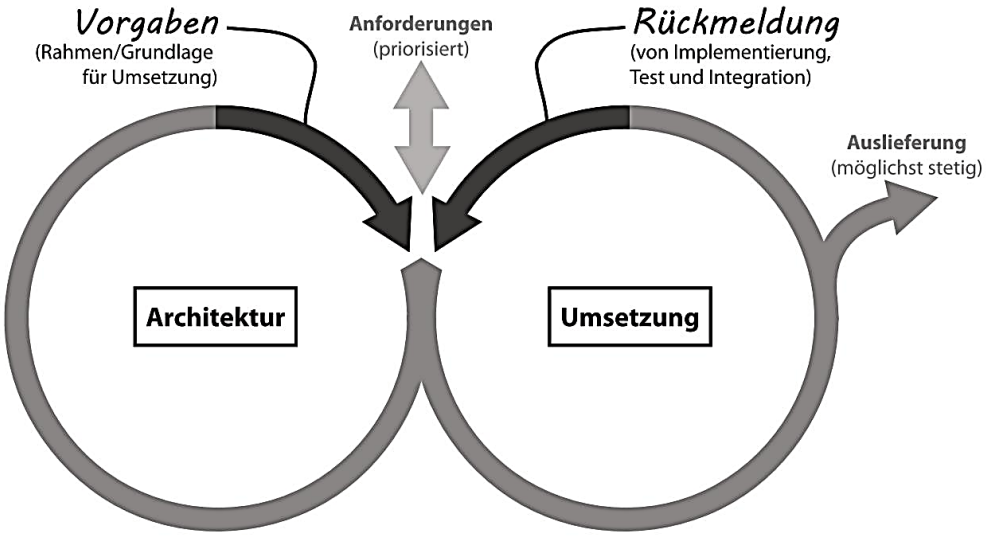

Abbildung 31: Architektur und Umsetzung im agilen Umfeld [298]

[297] Vgl. Hasselbring 2018, S. 177.
[298] Toth 2014, S. 19.

In Kapitel 4.2.2 wurden fünf servicebezogene Aktivitäten beschrieben, die im Prozess der Architektur-Transformation auftreten. Die Tätigkeiten der Service-Identifikation, der Service-Analyse, des Service-Designs, der Service-Implementierung und der Service-Evolution (vgl. Abbildung 21) werden ebenfalls schrittweise und iterativ durchgeführt. Im Rahmen der Anwendung des Prozessmodells zur Architektur-Transformation sind traditionelle Entwicklungsvorgehen, wie z. B. das *Wasserfallmodell*, folglich nur bedingt geeignet. Das Prozessmodell ist daher nicht als linearer Prozess zu interpretieren.

Domain-Driven Design basiert auf dem Grundgedanken der iterativen und explorativen Zusammenarbeit aller Anspruchsgruppen und dem damit verbundenen schrittweisen Erfahrungsaufbau (vgl. Kapitel 4.3.1). Demnach kann die Anwendung von Domain-Driven Design auch die Weiterentwicklung der Architektur auf agile Weise unterstützen.[299] Dies bestätigen die Erfahrungen von Hippchen et al. über die Anwendung von Domain-Driven Design in Kombination mit der agilen Projektmanagement-Methode Scrum.[300] Werden Softwaresysteme mit Domain-Driven Design sowie agilen Methoden entworfen und entwickelt, so entstehen auch die Domänenmodelle schrittweise und iterativ. In Bezug auf das Prozessmodell zur Architektur-Transformation (vgl. Kapitel 4.2.2) und das Prozessmodell zum domänengetriebenen Entwurf (vgl. Kapitel 4.3.3) werden die Domänenmodelle folglich über den gesamten Transformations- und Entwicklungsprozess kontinuierlich angepasst und verbessert.[301] Jedes unabhängige und agil vorgehende Microservice-Team ist währenddessen für sein eigenes Domänenmodell verantwortlich.

[299] Vgl. Uludağ et al. 2018, S. 244f.
[300] Vgl. Hippchen et al. 2017, S. 443.
[301] Vgl. Sharma et al. 2016, S. 60.

5. Demonstration

In diesem Kapitel werden die Teilartefakte aus Kapitel 4, die sich für eine Demonstration im Rahmen dieser Arbeit eignen, veranschaulicht. Der konzeptionelle Ordnungsrahmen wird dabei im Kontext einer realen Problemsituation betrachtet. Das Ziel ist es, die Zusammenhänge sowie den Nutzen und die Anwendbarkeit der Ergebnisse darzustellen.

Der in Kapitel 4 erarbeitete konzeptionelle Ordnungsrahmen setzt sich aus fünf Teilartefakten zusammen (vgl. Kapitel 4.1 bis 4.5). Aufgrund der eingeschränkten Anwendbarkeit einiger Teilartefakte, wird in diesem Kapitel ausschließlich auf die Situationsanalyse (Kapitel 4.1) und den domänengetriebenen Entwurf der fachlichen Architektur (Kapitel 4.3) eingegangen. Die Ursachen für diese Beschränkungen werden in Kapitel 6.2 diskutiert.

5.1 Situationsanalyse

5.1.1 Organisations- und Systemkontext

Als Betrachtungsgegenstand dient die real existierende, *monolithische Werbezeitvermarktungsplattform* eines Großkonzerns aus der Medienbranche. Das System gehört zur Sales-Domäne des Unternehmens und stellt einer Vielzahl von Werbekunden verschiedene Funktionen zur Buchung von TV-Werbung bereit. Die Buchung erfolgt in erster Linie durch externe Werbeagenturen über eine webbasierte Nutzeroberfläche. Sie kann jedoch auch durch Angestellte des Konzerns über einen separaten Java-basierten Client erfolgen. Die gesamte Anwendungslogik basiert auf einer objektorientierten Implementierung und wurde mit der Programmiersprache Java entwickelt.
Das monolithische System ist von der Anlage von Kundenaufträgen bis zur Rechnungsstellung für den gesamten *Auftragslebenszyklus* verantwortlich. Darüber hinaus wird über die Anwendung die gesamte *Produktkonfiguration* abgewickelt. Das zentrale produktbezogene Element des Systems ist das Konzept der Werbeblöcke, in denen die Kunden ihre Werbespots buchen können. Innerhalb der Anwendungslandschaft der Sales-Domäne nimmt die monolithische Anwendung eine zentrale Rolle ein und besitzt eine Vielzahl von Schnittstellen zu verschiedenen Umsystemen. Beispielsweise muss die Anwendung mit einem Preisberechnungssystem kommunizieren, um eine Zuordnung von Preisen zu Werbeblöcken vornehmen zu können. Sie ist außerdem mit einem Rabattberechnungssystem verbunden, das verschiedene Formen von Rabatten für Kunden und Aufträge bereitstellt.

5.1.2 Überprüfung des Bedarfs

Nachfolgend wird der konkrete Bedarf des Unternehmens für den Einsatz des Microservice-Architekturstils überprüft. Dazu wird auf die in Kapitel 4.1.1 beschriebenen Motivationsfaktoren eingegangen.

Die monolithische Werbezeitvermarktungsplattform ist über einen Zeitraum von etwa fünfzehn Jahren durch verschiedene Dienstleister und Teams entwickelt und verantwortet worden. Sie ist währenddessen zu einem *komplexen Softwaresystem* mit etwa einer Millionen Zeilen Programmcode angewachsen. Aufgrund der großen und unübersichtlichen Codebasis sowie der Verwendung veralteter Technologien sind viele *technische Schulden* in der Anwendung vorhanden. Das System verwendet zum Beispiel ein selbst entwickeltes Persistenz-Modul, um Objekte auf Relationen bzw. Datenbank-Tabellen abzubilden. Dieser objektrelationale Mapper wird bereits seit mehreren Jahren nicht mehr weiterentwickelt. Der vollständige Umstieg auf eine aktuellere und frei verfügbare Technologie ist nur mit hohem *Aufwand* und *Risiko* durchführbar. Neben der *Technologiefreiheit* wird auch die *Testbarkeit des Systems* sowie das *Systemverständnis* negativ beeinflusst. Ausschlaggebend hierfür sind unter anderem die *vielen Abhängigkeiten* im Quellcode und die *schlechte Modularität*. Aufgrund der hohen Komplexität der monolithischen Anwendung ist die *Weiterentwicklung und Wartung* des Systems sehr aufwendig geworden. Jedes Deployment oder Release erfordert ein hohes Maß an Kommunikation und Koordination, was die *Produktivität* der Teams wesentlich einschränkt und stets mit erhöhtem Risiko für das Auftreten von Fehlern verbunden ist. Folglich werden nur wenige größere Änderungen vorgenommen und selten neue Funktionen implementiert. In Bezug auf Abbildung 16 (vgl. Kapitel 4.1.1) kann die Situation zwischen den zwei Zustandsindikatoren (2) und (3) eingeordnet werden, mit einer Tendenz zum dritten Indikator. Das bedeutet, dass die Produktivität der Entwicklung durch die Komplexität der Anwendung bereits merklich abgenommen hat.

Aufgrund der vergleichsweise geringen Nutzerzahl bestehen keine besonderen Anforderungen an die *Skalierbarkeit* des Systems. Dahingehend ist auch die *Performanz* der Anwendung kein ausschlaggebender Motivationsfaktor für die Nutzung des Microservice-Architekturstils. Die Weiterentwicklung erfolgt bereits unter Anwendung agiler Methoden und Prinzipien, die jedoch durch die monolithischen Anwendungs- und Teamstrukturen nicht weitreichend unterstützt werden. Weiterhin sind die Bereiche der Entwicklung und des Betriebs streng voneinander getrennt. Die Etablierung einer DevOps-Kultur ist unter den gegebenen Bedingungen nur schwer umsetzbar.

Insgesamt ist die Situation stark durch die Herausforderungen geprägt, die mit der Entwicklung und dem Betrieb der monolithischen Anwendung einhergehen. Sowohl die Anwendungsentwicklung als auch die Bereiche der Organisation, der Prozesse und der Kultur können von einer Transformation der Architektur profitieren.

5.1.3 Bewertung des Reifegrades

Zur weiteren Beurteilung der Ist-Situation erfolgte die Bewertung des Reifegrades anhand der Reifegradfaktoren und des Fragenkataloges aus Kapitel 4.1.2. Im Folgenden werden diese Fragen in der entsprechenden Reihenfolge und im Kontext des betrachteten Unternehmens beantwortet.

(1) *Organisations- bzw. Teamstrukturen*: Die monolithische Werbezeitvermarktungsplattform wird von einem einzigen großen Team entwickelt. Diese Teamstruktur ist für den Einsatz des Microservice-Architekturstils nicht geeignet. Das Unternehmen ist jedoch flexibel in der Anpassung der Organisations- und Teamstrukturen, sodass keine Hindernisse für die zukünftige Unterstützung durch die Organisation bestehen.

(2) *Agile Softwareentwicklung*: Das Team besitzt tiefgreifende Erfahrung in der Anwendung von Scrum, einem agilen Vorgehensmodell zum Projektmanagement. Scrum wird bereits seit über zwei Jahren im Projekt angewendet.

(3) *DevOps-Kultur*: Es ist keine DevOps-Kultur im Projekt etabliert und sie befindet sich gegenwärtig auch nicht im Aufbau. Obwohl das Unternehmen flexibel in der Anpassung der Organisationsstrukturen ist (siehe Punkt 1) und mit agilen Vorgehensweisen im Projekt bereits erste DevOps-Praktiken aktiv anwendet (vgl. Kapitel 2.3.4 und 2.5.2), kann die generelle Umsetzbarkeit von DevOps nicht hinreichend beurteilt werden.

(4) *Automatisiertes Deployment*: Im Projekt werden keine Prozesse zur automatisierten Bereitstellung der Software verwendet. Das Deployment des Monolithen erfolgt manuell und mit hohem manuellen Test- und Koordinationsaufwand. Das Unternehmen besitzt jedoch Erfahrung mit automatisierten Bereitstellungsverfahren und kann dieses Wissen kurzfristig bereitstellen und nutzen.

(5) *Technologie & Infrastruktur*: Die Organisation ist flexibel hinsichtlich der Einführung neuer Technologien. Im Projekt sind bereits erste Erfahrungen mit Cloud-Technologien wie *OpenShift* und *Amazon Web Services* vorhanden. Die verwendeten Werkzeuge und Technologien sind für die Entwicklung und den Betrieb von Microservices relevant und vorteilhaft.

(6) *Wissen & Erfahrung*: Im Projektkontext wurden neue Funktionen bereits in ersten Microservices implementiert. Die aktive und regelmäßige Weiterbildung in den essenziellen Bereichen – Cloud, Microservices, Agile und DevOps – gehört zur Lernkultur des Unternehmens. Das Team sammelt somit erste Erfahrungen im Umgang mit Microservices und bildet sich kontinuierlich weiter.

Die Ausführungen der Punkte (1) bis (2) und (4) bis (6) sprechen für die Einführung des Microservice-Architekturstils aus Sicht der betrachteten Reifegradfaktoren. Einzig in Bezug auf Punkt (3) konnte aufgrund der organisatorischen Rahmenbedingungen der Untersuchung keine eindeutige Beurteilung der Situation vorgenommen werden. Insgesamt ist der Reifegrad anhand dieser initialen Analyse positiv zu bewerten. Die unternehmens- und projektbezogene Reife begünstigt demnach die Architektur-Transformation der betrachteten monolithischen Anwendung.

5.2 Domänengetriebener Entwurf

Der domänengetriebene Entwurf (vgl. Kapitel 4.3) ist ein wesentlicher Bestandteil des Prozessmodells zur domänengetriebenen Transformation der fachlichen Architektur (vgl. Kapitel 4.2). Die in Kapitel 5.1.1 beschriebene Domäne beinhaltet eine große Menge an komplexer Fachlogik. Der domänengetriebene Entwurf und die Muster und Prinzipien des Domain-Driven Designs können dabei helfen diese Komplexität zu strukturieren und beherrschbar zu machen. Das Ziel ist den Monolithen in fachlich unabhängige Einheiten zu zerlegen, sodass die Vorteile des Microservice-Architekturstils (vgl. Kapitel 2.2.3 und 4.1.1) entsprechend ausgenutzt werden können. Der Aspekt der Unabhängigkeit einzelner Microservices ist dabei von zentraler Bedeutung.

5.2.1 Betrachtung vorhandener Geschäftsfähigkeiten

In Kapitel 4.3.2 wurde beschrieben, dass Geschäftsfähigkeiten stabil sind und sich im Gegensatz zu Geschäftsprozessen nur in seltenen Fällen ändern. Weiterhin wurde dargelegt, dass die Konzepte der Geschäftsfähigkeiten und Subdomänen weitestgehend gleichbedeutend sind. Sofern entsprechendes Vorwissen über Geschäftsfähigkeiten aus dem Bereich des *Enterprise Architecture Managements* vorhanden ist, kann dieses Wissen zur Unterstützung des domänengetriebenen Entwurfsprozesses genutzt werden. In diesem Zusammenhang kann sich die Verwendung von *Business Capability Maps*, die einen Überblick über die in einer Domäne vorhandenen Geschäftsfähigkeiten geben, als hilfreich erweisen. Je nach Detaillierungsgrad können sich diese Business Capability Maps über mehrere Abstraktionsebenen erstrecken.

Im Rahmen der Untersuchungen wurde eine Business Capability Map erstellt, die sich über drei Ebenen erstreckt (vgl. Abbildung 32). Dafür wurden unterschiedliche Architekturdokumentationen aus dem Bereich der Sales-Domäne untersucht. Darunter befand sich auch eine Business Capability Map mit Geschäftsfähigkeiten der höheren und mittleren Abstraktionsebenen (siehe Ebene 1 und 2). Die dritte Ebene wurde exemplarisch für die Geschäftsfähigkeit der Auftragsabwicklung ergänzt.

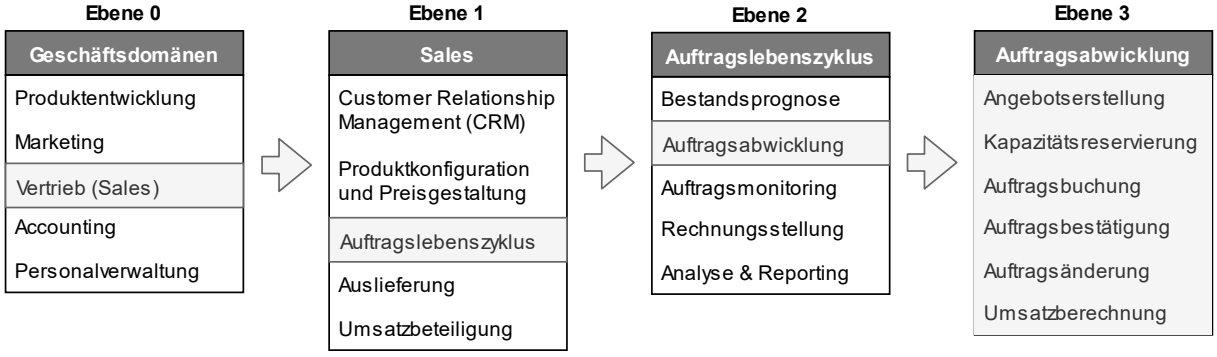

Abbildung 32: Business Capability Map der Sales-Domäne (Auszug)

Die abgebildete Business Capability Map stellt nur einen von mehreren möglichen Abstraktionspfaden dar. Auf Ebene 1 werden die grobgranularen Geschäftsfähigkeiten der

Produktkonfiguration und des Auftragslebenszyklus durch die monolithische Werbezeitvermarktungsplattform abgedeckt. Ebene 2 zeigt die vorhandenen Geschäftsfähigkeiten aus dem Bereich des Auftragslebenszyklus, während auf Ebene 3 die feingranularen Geschäftsfähigkeiten für den Bereich der Auftragsabwicklung abgebildet sind. Aufgrund der Granularität sind für die fachliche Aufteilung in Microservices in der Regel erst die Geschäftsfähigkeiten ab der zweiten Abstraktionsebene relevant. Die Geschäftsfähigkeiten der höheren Abstraktionsebenen beinhalten meist eine große Menge an Domänenlogik. Daher kann eine Orientierung an diesen Ebenen schnell zu grobgranularen Services führen, die den Eigenschaften von Monolithen nahekommen.

5.2.2 Analyse der Geschäftsdomäne

Für die Analyse der Geschäftsdomäne bzw. des Problemraumes wurde die Workshop-basierte Methode des Event Stormings angewendet (vgl. Kapitel 4.3.4). Durch Event Storming kann eine Domäne schnell und effektiv erforscht werden. Das ereignisgetriebene Vorgehen vereinfacht dabei die Modellierung von komplexen Geschäftsprozessen. Weiterhin können die Ergebnisse zum Ermitteln und Validieren von Kontextgrenzen (Bounded Contexts) verwendet werden. Die Methodik bietet sich vor allem in Situationen an, in denen wenig Vorwissen über die Domäne oder noch kein Domänenmodell vorhanden ist. Beides war im Rahmen der vorliegenden Arbeit der Fall.

Für den Workshop wurden zwei wesentliche Ziele definiert:

(1) Aufbau von einem gemeinsamen Verständnis über die Domäne
(2) Identifizierung von Kontextgrenzen (Bounded Contexts)

Der Workshop wurde mit dem Architekten der Sales-Domäne, zwei Entwicklern des Projektes und einem unabhängigen Entwickler als neutrale Person durchgeführt. Der Architekt übernahm aufgrund von tiefgreifendem Wissen über die Domäne eine Schlüsselrolle als Domänenexperte. Im Rahmen der initialen Erforschung der Domäne wirkte der Architekt damit vertretend für den Fachbereich bzw. die Anwender der Software. Da Event Storming auf einem iterativen Vorgehen basiert, sollte in anschließenden Workshops die Beteiligung der betroffenen Fachbereiche erfolgen. Eine weitere Besonderheit war die Einbindung eines projektfremden Entwicklers. Dieser nahm ohne Domänenwissen am Workshop teil und konnte das erarbeitete Wissen stets kritisch hinterfragen. Dabei wurde die Intention verfolgt, dass die Ergebnisse auch für neue Projektmitglieder und außenstehende Personen verständlich präsentiert werden.

Den zentralen Bestandteil des gesamten Workshops bildete eine *Legende*, die die verwendeten Elemente des Domain-Driven Designs sowie deren Beziehungen untereinander darstellt und den Teilnehmern als grundlegende Orientierungshilfe diente (vgl. Abbildung A-3; Anhang). Die Legende setzt sich aus *Domänen-Ereignissen, Aktionen, Aggregaten, externen Systemen, Richtlinien, Nutzern* und *lesenden Modellen* zusammen. Eine Aktion kann dabei auf ein Aggregat oder externes System wirken. Aufgrund einer Aktion können Aggregate und externe Systeme ein oder mehrere Domänen-Ereignisse erzeugen. Ein Domänen-Ereignis kann außerdem durch ein anderes Domänen-Ereignis ausgelöst werden und sich auf ein Aggregat auswirken. Weiterhin kann ein Domänen-Ereignis eine bestimmte Richtlinie auslösen, die wiederum eine oder mehrere Aktionen verursacht. Auch lesende

Modelle können durch Domänen-Ereignisse beeinflusst werden. Ein lesendes Modell liefert dabei bestimmte Daten bzw. Informationen, aufgrund dieser ein Nutzer eine Aktion durchführen oder auslösen kann. Das kann auch unabhängig von Informationen aus einem lesenden Modell erfolgen. Bei den Domänen-Ereignissen und Aggregaten handelt es sich um bedeutende Muster des taktischen Designs (vgl. Kapitel 2.4.3). Die übrigen Elemente sind ebenfalls dem taktischen Design zuzuordnen, gehören jedoch nicht zu den Kernmustern des taktischen Designs.

Im ersten Schritt wurden die *Domänen-Ereignisse* innerhalb der Sales-Domäne identifiziert und zeitlich geordnet. Dabei standen die bedeutenden fachlichen Ereignisse im Mittelpunkt, weshalb nicht jedes denkbare Ereignis aufgeschlüsselt wurde. In der Sales-Domäne konnten beispielsweise die Ereignisse „Spot gebucht", „Motiv zugeordnet" und „Ausstrahlung fakturiert" identifiziert werden. Im nächsten Schritt wurden die Auslöser für diese Ereignisse identifiziert. Im Sinne der Ursachenforschung zählen bestimmte *Aktionen*, *externe Systeme*, *Richtlinien* oder andere *Ereignisse* zu den möglichen Auslösern. Basierend darauf können Ereignisse auch zu *Aggregaten* zugeordnet werden, da diese ebenfalls Aktionen entgegennehmen und Ereignisse auslösen können. Auf diese Weise konnten die wesentlichen Geschäftsprozesse sukzessiv im Kontext der monolithischen Werbezeitvermarktungsplattform modelliert werden. Bei der Modellierung hat sich durch die verwendeten Begriffe außerdem die Fachsprache *(Ubiquitous Language)* der Domäne herauskristallisiert.

Zum Ende des Workshops konnten die Ergebnisse in vier grobe Bereiche untergliedert werden (vgl. Abbildung A-4 bis Abbildung A-7). Drei der Ergebnisbereiche konnten in weitere abgetrennte Abschnitte untergliedert werden. Dadurch konnten verschiedene Aspekte der Domäne getrennt betrachtet und besser mit der Komplexität umgegangen werden. Die entstandenen Modelle spiegeln erste nicht-spezifizierte Formen von *Domänenmodellen* wieder. Sie sind unter Verwendung der formlosen Elemente der Event Storming Legende (vgl. Abbildung A-3) entstanden und spiegeln in Form und Farbe die zur Modellierung verwendeten Klebezettel wieder. Da die Domänenmodelle unabhängig von den Ebenen der Architektur und Implementierung sind, können sie auch als *Informationsmodelle* bzw. *Computation Independent Models* bezeichnet werden (vgl. Kapitel 4.2.2).

Eine Besonderheit ist, dass die entstandenen Modelle aufgrund der gewählten Analysetechnik prozessorientiert sind. In Bezug auf das Konzept der *Domänensichten* können die Ergebnisse daher als Prozess-Sicht auf die Domäne interpretiert werden.[302] Auf Basis der Ergebnisse sollte die Ausarbeitung von formalisierten Domänenmodellen erfolgen, damit diese für die Entwicklung von *Design-Modellen*, *Code-Modellen* und *Datenmodellen* verwendet werden können (vgl. Kapitel 4.2.2). Dabei sollten auch jene Domänensichten berücksichtigt werden, die die Beziehungen der Domänenobjekte wiederspiegeln. Für die Identifizierung von initialen Kontextgrenzen war der Formalitätsgehalt der Domänen- bzw. Informationsmodelle jedoch ausreichend. Die Modelle wurden daher nicht weiter verfeinert und spezifiziert.

[302] Vgl. Hippchen et al. 2017, S. 437f.

5.2.3 Identifizierung von Kontextgrenzen

Auf Grundlage der Workshop-Ergebnisse aus Kapitel 5.2.2 konnten erste potenzielle *Kontextgrenzen* abgeleitet werden. Durch Abstraktion der modellierten Geschäftsprozesse wurden dafür zunächst die *Geschäftsfähigkeiten* niedriger Granularität extrahiert. Dazu zählen z. B. die Geschäftsfähigkeiten der Sendeplan-Aktivierung oder der Spotbuchung. Nachfolgend wurden die identifizierten Bereiche zu Geschäftsfähigkeiten bzw. Bounded Contexts höherer Granularität zusammengefasst. Diese konnten wiederum den Geschäftsfähigkeiten noch höherer Ebene zugeordnet werden. Dieses *Bottom-Up Vorgehen* spiegelt den Charakter von Event Storming wieder. Während der Nachbereitung der Workshop-Ergebnisse wurde die zuvor erstellte Business Capability Map hinzugezogen (vgl. Kapitel 5.2.1). Durch die Kombination beider Ansätze konnten die Ergebnisse entsprechend abgeglichen und verbessert werden.

Abbildung 33 liefert einen Überblick über die identifizierten Bereiche. Alle dargestellten Geschäftsfähigkeiten bzw. Kontexte konnten auf Basis der Workshop-Ergebnisse identifiziert werden. Innerhalb der Sales-Domäne haben sich damit unterschiedliche Bereiche von hoher, mittlerer und niedriger Abstraktionsebene herausgebildet. Die zum Bereich der *Produktkonfiguration* gehörigen Kontexte sind stark durch Umsysteme beeinflusst, da die Produktkonfiguration überwiegend von den Daten externer Systeme abhängig ist. Dies trifft insbesondere für die Kontexte der *Preis-Verarbeitung*, *Sendeplanung* und *Motiv-Disposition* zu. Die extern bereitgestellten Informationen dienen innerhalb der Beispielanwendung zur Anreicherung von Daten im Sinne der Produktkonfiguration. Das Produkt ist dabei der Werbeblock, in den sich ein Kunde einbuchen kann und der mit gebuchten Werbespots komponiert wird. Im Kontext der Preis-Verarbeitung werden z. B. Preisinformationen aus dem externen Preisberechnungs-System importiert, die im Werbeblock hinterlegt und bei einer Buchung sowie der Rechnungsstellung berücksichtigt werden.

Ein mögliches Beispiel für zwei Kontextgrenzen, die mit hoher Wahrscheinlichkeit valide sind, kann anhand des Werbeblock-Domänenobjektes demonstriert werden. Im Kontext der *Bestandsprognose* besitzt das Konzept des Werbeblocks eine andere Bedeutung als im Kontext der *Buchungsabwicklung*. Die Bestandsprognose betrachtet die Freilängen und Kapazitäten von Werbeblöcken. In der Buchungsabwicklung sind diese Attribute hingegen nicht von Bedeutung. Bei einer Buchung ist z. B. von Interesse, ob ein Spot in einem Werbeblock gebucht werden kann und welche kosten die Buchung verursacht. Diese Mehrdeutigkeit in der Terminologie und den Konzepten der Domäne kann als bedeutungsvolles Indiz dafür herangezogen werden, dass es sich um zwei unterschiedliche Kontexte handelt (vgl. Kapitel 4.3.3).

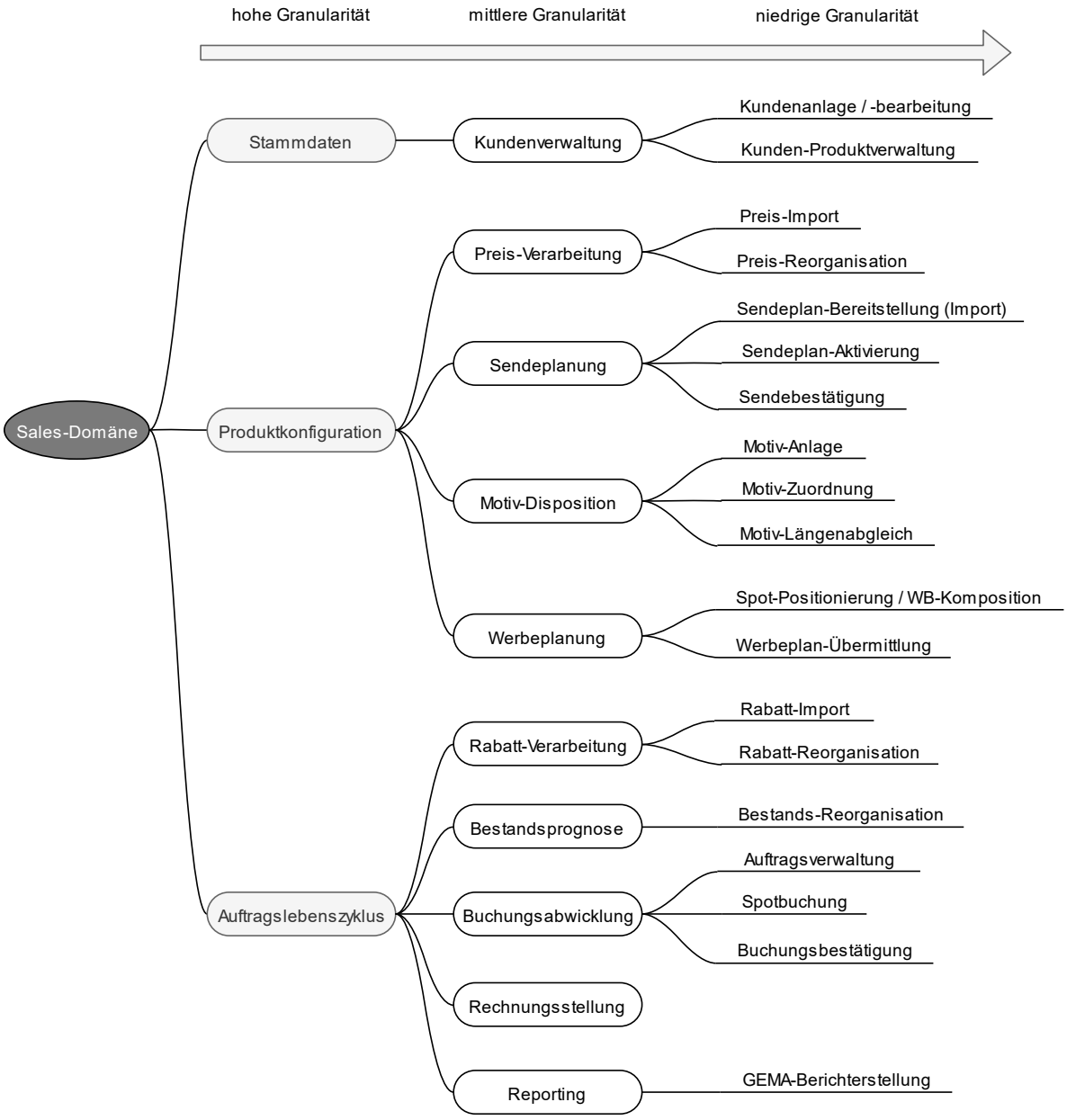

Abbildung 33: Identifizierte Kontexte und Geschäftsfähigkeiten

Die Identifizierung der Kontextgrenzen ist von großer Bedeutung für den domänengetriebenen Entwurf der Microservice-Architektur (vgl. Kapitel 4.3.3) und den gesamten Transformationsprozess (vgl. Kapitel 4.2.2). Dieser Schritt sollte daher mit besonderer Sorgfalt durchgeführt werden und in mehreren Iterationen erfolgen. Die dargestellten Workshop-Ergebnisse basieren lediglich auf einer einzigen Iteration. Es handelt sich daher um einen ersten Entwurf für mögliche Kontextgrenzen. Sowohl die Ergebnis-Modelle (vgl. Abbildung A-4 bis Abbildung A-7) als auch die abgeleiteten Kontextgrenzen müssten im Rahmen einer realen Umsetzung durch weitere Analysen überprüft und verfeinert werden.

Auf Basis der Workshop-Erfahrungen konnte festgestellt werden, dass die Identifizierung von Kontextgrenzen zu den größten Herausforderungen im domänengetriebenen Entwurf zählt. Diese Erkenntnis wird dadurch verstärkt, dass die Identifizierung auf Basis von frühen Workshop-Ergebnissen erfolgte. Zumindest auf Ebene mittlerer Granularität sind die kontextuellen Grenzen deutlicher erkennbar. Auf Ebene niedriger Granularität ist die kontextuelle Trennschärfe dahingegen gering. Es muss folglich in Frage gestellt werden, ob die im Workshop identifizierten Kontextgrenzen valide sind. Beispielsweise sollte hinterfragt werden, ob es sich bei der Spotbuchung und Buchungsbestätigung um zwei separate Kontexte handelt. Weiterhin erwecken einige Ergebnisse niedriger Granularität den Eindruck, dass es sich um einzelne Funktionalitäten der monolithischen Anwendung handelt und keine abgegrenzten Kontexte. Diese Erfahrungen bekräftigen die Strategie einen Monolithen zunächst in grobgranulare Microservices zu schneiden (vgl. Kapitel 4.4.2). Jeder Microservice würde demnach einem Bounded Context mittlerer Granularität entsprechen. Wenn sich die Kontextgrenzen über die Zeit und mit zunehmender Erfahrung etabliert haben, können die Services bei Bedarf weiter zerlegt werden. So kann das Risiko minimiert werden, dass aufgrund von falsch gewählten Servicegrenzen hoher nachträglicher Korrekturaufwand entsteht.

In Kapitel 4.4.2 wurde beschrieben, dass ein Bounded Context die Obergrenze für die Größe eines Microservice darstellt. Im Gegensatz dazu kann ein Microservice nicht kleiner als ein Aggregat sein, das Teil der Implementierung innerhalb eines Bounded Context ist. Dahingehend konnte im Workshop festgestellt werden, dass die Orientierung an den Kontexten hoher Abstraktionsstufe (Auftragslebenszyklus und Produktkonfiguration) nur zu kleineren Monolithen führen würde, da die Kontexte groß und komplex sind. Die Wahl der Servicegröße wird daher auch durch die Komplexität der Bounded Contexts beeinflusst. Die Kundenverwaltung ist z. B. ein einfacher Kontext, der keine wesentliche Fachlogik im System abbildet. Er ist in erster Linie durch unkomplizierte Operationen wie Anlegen, Ändern und Löschen von Kunden oder Kundenprodukten geprägt. Es ist valide diese Standardfunktionen durch einen einzelnen Microservice umzusetzen, solange die an den Microservice geknüpften Qualitätsanforderungen erfüllt sind (vgl. Kapitel 4.4.4). Im Kontrast dazu steht beispielsweise der Kontext der Buchungsabwicklung. Er beinhaltet eine große Menge an komplexer Fachlogik und kann z. B. in Hinsicht auf die Aspekte der Verständlichkeit und Wartbarkeit von einer Aufteilung in mehrere Microservices profitieren. In diesem Zusammenhang können weitere Faktoren für die Wahl von Servicegrößen in Betracht gezogen werden (vgl. Kapitel 4.4.2).

5.2.4 Erstellen einer Kontextlandkarte

In diesem Kapitel wird das Konzept der *Kontextlandkarten (Context Maps)* anhand der Werbezeitvermarktungsplattform verdeutlicht. Die in Kapitel 5.2.3 identifizierten Kontexte sind in dem Monolithen über ein einziges großes Legacy-Domänenmodell miteinander verwoben (vgl. Kapitel 4.2.2; Abbildung 20). Die Methodik des Context-Mappings kann verwendet werden, um die Beziehungen und Abhängigkeiten der einzelnen Kontexte darzustellen und die Aufteilung in mehrere kleine Domänenmodelle entlang der identifizierten Kontextgrenzen zu unterstützen.
In Kapitel 5.2.3 wurde festgestellt, dass die identifizierten Kontextgrenzen niedriger Granularität noch zu unklar und unausgereift sind. Daher findet das Context-Mapping auf Grundlage der Bounded Contexts mittlerer Abstraktionsstufe statt. Da die Ergebnisse lediglich frühe Grobentwürfe möglicher Bounded Contexts repräsentieren, wird das Konzept der Kontextlandkarten nur angedeutet. Die Ausführungen beschränken sich dabei auf vereinfachte Abhängigkeitsbeziehungen und gehen nicht auf die spezifischen Integrationsmuster (vgl. Kapitel 2.4.2) des Domain-Driven Designs ein. Die Ergebnisse sind demnach als erste Ausbaustufe einer Kontextlandkarte zu verstehen. In Abbildung 34 ist die Kontextlandkarte für die zwei grobgranularen Kontexte der Produktkonfiguration und des Auftragslebenszyklus dargestellt.

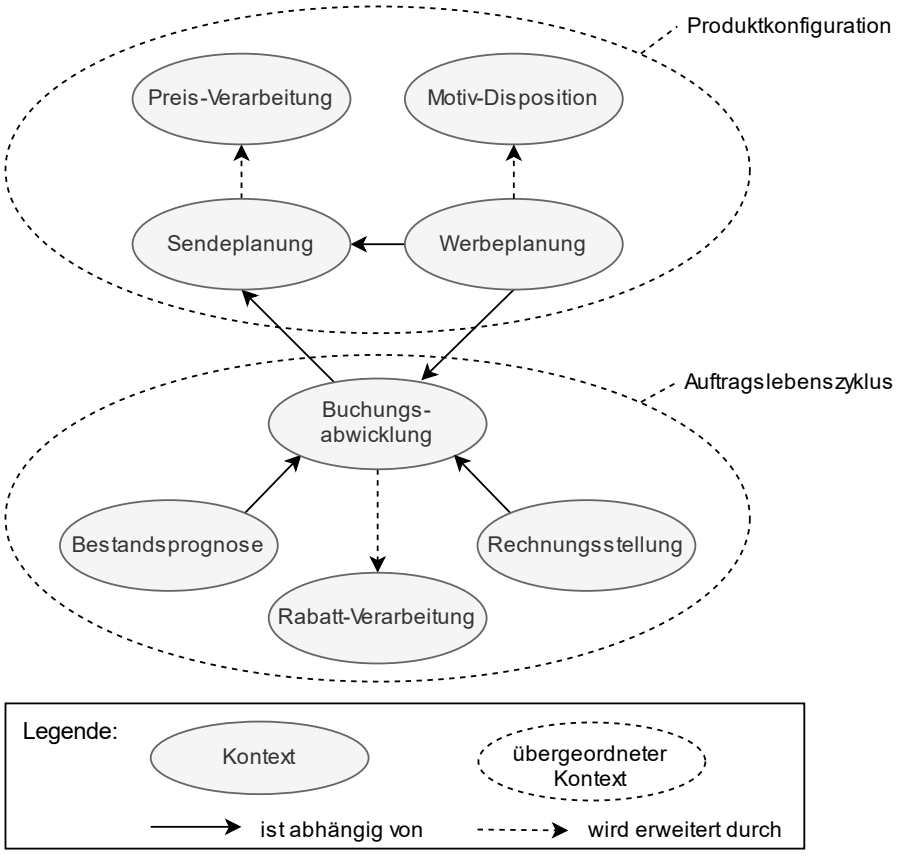

Abbildung 34: Kontextlandkarte der identifizierten Kontexte

Anhand der Kontextlandkarte ist erkennbar, dass der Kontext bzw. die Geschäftsfähigkeit der Buchungsabwicklung eine zentrale Rolle im Umfeld der monolithischen Anwendung einnimmt. Im Kontext des Auftragslebenszyklus muss für die Bestandsprognose nach jeder Spotbuchung eine Bestandsreorganisation vorgenommen werden. Weiterhin werden die Buchungsinformationen zur Rechnungstellung benötigt. Die Buchungsabwicklung dagegen ist auf die Erweiterung durch Rabattinformationen angewiesen. Sie benötigt für jede Spotbuchung ebenfalls Werbeblock-Informationen aus der Sendeplanung. Im Sendeplanungs-Kontext werden die Werbeblöcke außerdem mit Preisinformationen angereichert. Die Werbeplanung wiederum wird auf Grundlage des Sendeplans und der Buchungsdaten aus der Buchungsabwicklung durchgeführt. Die Spots werden für die Werbeplanung außerdem mit Motivdaten angereichert.

Kontextlandkarten bilden implizit auch die Auswirkungen des *Gesetzes von Conway* ab (vgl. Kapitel 2.3.1). Durch Anwendung des Inversen Conway Manövers können die Organisationsstrukturen entsprechend den identifizierten Kontexten optimiert werden. Eine Kontextlandkarte spiegelt demnach die Beziehungen zwischen verschiedenen Teams wieder, die jeweils für unterschiedliche Bounded Contexts verantwortlich sind.

6. Evaluierung

In diesem Kapitel erfolgt eine Bewertung des in Kapitel 4 vorgeschlagenen Ordnungsrahmens. Dazu werden die Teilartefakte einzeln als auch in sinngemäßer Kombination diskutiert. Weiterhin wird die Validität der Ergebnisse in Bezug auf die in Kapitel 5 durchgeführte Demonstration des Ordnungsrahmens untersucht und wesentliche Schlussfolgerungen für die Praxis gezogen.

6.1 Diskussion der Ergebnisse

Im Folgenden wird auf die in Kapitel 1.2 definierten sekundären Forschungsfragen eingegangen. Die zusammenfassende Beantwortung der primären Forschungsfrage erfolgt abschließend in Kapitel 7.1.

(1) Vorbedingungen und Anforderungen des Microservice-Architekturstils

Das Ziel der ersten Forschungsfrage war die Feststellung der grundlegenden Vorbedingungen und Anforderungen, die mit der Einführung des Microservice-Architekturstils einhergehen. Zur Beantwortung der Frage war eine Untersuchung der Einflussfaktoren und Schnittbereiche des Microservice-Architekturstils notwendig. Im theoretischen Grundlagenteil der Arbeit wurden dazu die wesentlichen Zusammenhänge herausgearbeitet (vgl. Kapitel 2.5). Um einen Mehrwert hinsichtlich der Anwendung dieses Wissens zu generieren, wurde das Hilfsmittel der Situationsanalyse entwickelt. Es soll Anspruchsgruppen ermöglichen, eine kritische Einschätzung der Situation im Unternehmen bzw. Projekt vorzunehmen und sieht dafür die Bewertung des Bedarfs sowie des Reifegrades vor.

Die bereitgestellten Informationen zur Durchführung der Situationsanalyse sind als richtungsweisende Hilfestellung zu verstehen. Sie stellen einen allgemeinen Rahmen zur Situationsbewertung dar und beinhalten keine konkreten Handlungsanweisungen zur Ermittlung des Bedarfs oder des Reifegrades. Die Analyse des Reifegrades wird zudem durch die Anwendung eines Fragenkataloges unterstützt. Dieser beinhaltet verallgemeinerte Fragen, die eine initiale Einschätzung des Reifegrades ermöglichen. Die Nutzung des Kataloges stellt daher keine explizite Vorgehensweise zur Bewertung des Reifegrades dar und liefert keine messbaren Ergebnisse.
Die weiterführende Beurteilung des Reifegrades, die über die Anwendung von Fragenkatalogen oder Checklisten hinausgeht, kann durch die Nutzung von Reifegradmodellen erfolgen. Mit dem Beitrag von Behara und Khandrika[303] existiert ein wissenschaftlicher Ansatz zur Bewertung des Reifegrades von Microservice-Anwendungen. Das von ihnen vorgeschlagene Reifegradmodell basiert auf einer Sammlung verschiedener Reifegrad-Parameter, sechs Reifegrad-Niveaus und einer *Maturity Score Card*, die zusammen eine systematische und messbare Einordnung ermöglichen. Auch für die erweiterte Bewertung des Reifegrades von Unternehmen bzw. Projekten wäre ein systematisches Reifgradmodell von Vorteil. Im Rahmen der Literaturrecherchen dieser Arbeit konnte jedoch kein derartiges Modell ausfindig gemacht werden.

[303] Behara und Khandrika 2017.

Für die Herleitung des Artefaktes der Situationsanalyse wurden einzig die Vorbedingungen und Anforderungen in Bezug auf den Microservice-Architekturstil betrachtet. Spezielle Vorbedingungen und Anforderungen, die sich auf die Durchführung von Migrationen beziehen, wurden nicht untersucht. Durch weiterführende Analysen können in diesem speziellen Kontext möglicherweise weitere Einflussbereiche und Abhängigkeiten identifiziert werden.

(2) Prozess zur domänengetriebenen Transformation der fachlichen Architektur

Zur Beantwortung der zweiten Forschungsfrage konnte ein Prozessmodell zur domänengetriebenen Transformation der fachlichen Architektur hergeleitet werden (vgl. Kapitel 4.2; Abbildung 20), das auf den grundlegenden Konzepten des Hufeisen-Modells basiert. Die Vorarbeit von Razavian und Lago[304] hatte dabei wesentlichen Einfluss auf den Entwurf des Prozessmodells. Sie ist zwar im Kontext serviceorientierter Architekturen entstanden, aufgrund der Abstraktheit des Modells und dem Aspekt der Service-Basiertheit waren die wesentlichen Ergebnisse jedoch auf Microservice-Architekturen übertragbar. Auch die Erfahrungen und Erkenntnisse von Di Francesco et al.[305] haben maßgeblich zum Entwurf des Prozessmodells und zur Beschreibung der wesentlichen Aktivitäten und Herausforderungen in den Prozessphasen beigetragen. Die Auswertung der relevanten Theorien und Konzepte resultierte in einem Prozessmodell, das alle servicebezogenen Aktivitäten enthält (vgl. Kapitel 4.2; Abbildung 21), die von grundlegender Bedeutung für den domänengetriebenen Transformationsprozess sind.
Das in Abbildung 20 vorgeschlagene Prozessmodell unterscheidet sich insofern von den Ergebnissen von Razavian und Lago, dass die Aktivitäten auf Ebene der Geschäftsarchitektur getrennt von der Anwendungs- und Datenarchitektur betrachtet werden. Sie können unabhängig durchgeführt werden und resultieren in geschäftlichen Rahmenbedingungen für den Entwurf der Microservice-Architektur. Weiterhin wird im Kontext des Prozessmodells ein Bezug zu den servicebezogenen Aktivitäten (vgl. Abbildung 21) und den enthaltenen Modell-Artefakten (vgl. Abbildung 22) hergestellt. Das vorgestellte Prozessmodell sieht auf Ebene der Geschäftsarchitektur zudem die Transformation in mehrere serviceorientierte Domänenmodelle vor. Dieser Aspekt ist von grundlegender Bedeutung für die servicebasierte Transformation monolithischer Anwendungen und wurde im Modell von Razavian und Lago nicht berücksichtigt.

Anhand der Betrachtung und Abgrenzung alternativer wissenschaftlicher Ansätze (vgl. Kapitel 4.2.3) kann geschlussfolgert werden, dass das in Abbildung 20 vorgestellte Prozessmodell hinreichend abstrakt ist, um als generisches Modell für die Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils verwendet werden zu können. Die Auswertung verdeutlicht zudem, dass diese Arbeiten vordergründig die technischen Aspekte von Microservice-Architekturen betrachten. Zur Synthese eines fachlich orientierten Prozessmodells konnten diese Ansätze daher nicht beitragen. Weiterhin bestätigt sich in der Diskussion der Arbeiten, dass die domänenorientierte Ausrichtung des Prozessmodells von hoher Relevanz ist.

[304] Razavian und Lago 2010.
[305] Di Francesco et al. 2018.

(3) Domänengetriebener Entwurf mit Domain-Driven Design

Zur Beantwortung der dritten Forschungsfrage galt es festzustellen, inwiefern der domänengetriebene Entwurf der fachlichen Architektur durch die Anwendung der Muster und Prinzipien des Domain-Driven Design unterstützt werden kann. Das domänengetriebene Vorgehen nach DDD ist jedoch weder in Form eines expliziten Prozesses definiert noch ausführlich in Verbindung mit dem Microservice-Architekturstil beschrieben. In Kapitel 4.3 wurde daher ein Prozessmodell hergeleitet, das die wesentlichen Elemente und Tätigkeiten des domänengetriebenen Entwurfs der fachlichen Architektur vereint und einen Bezug zum Microservice-Architekturstil herstellt.

Im Rahmen der Literaturanalyse wurde eine signifikante Forschungslücke identifiziert (vgl. Kapitel 3.2). Es konnten keine wissenschaftlichen Beiträge identifiziert werden, die einen auf Domain-Driven Design basierenden Ansatz zur Dekomposition monolithischer Anwendungen in Microservices verfolgen. Daher kann angenommen werden, dass DDD zum Großteil in der industriellen Praxis verwurzelt ist und diskutiert wird. Dies scheint insbesondere im Kontext des noch verhältnismäßig neuen Paradigmas der Microservice-Architekturen zuzutreffen, das ebenfalls zu großen Anteilen von der Praxis getrieben wird.[306] Auf Grundlage der Ergebnisse der Literaturanalyse konnte die Entwicklung des Prozessmodells folglich nicht hinreichend unterstützt werden. Zur Herleitung des Prozessmodells wurde daher vermehrt auf Standardwerke wie Evans[307] oder Millet und Tune[308] zurückgegriffen.

Für die Erarbeitung des Prozessmodells war es zunächst notwendig, das allgemeine Vorgehen im Sinne der Kerngedanken von Domain-Driven Design abzubilden (vgl. Kapitel 4.3.1). Aufgrund der identifizierten Probleme und Limitierungen von DDD wurde das Konzept der Geschäftsfähigkeiten zusätzlich mit in die Betrachtungen einbezogen (vgl. Kapitel 4.3.2). Da sich Microservices ohnehin an Geschäftsfähigkeiten orientieren sollten (vgl. Kapitel 2.2.3), konnten die DDD-spezifischen Konzepte der Bounded Contexts und Subdomänen gezielt erweitert werden. Die Relation zwischen diesen Konzepten konnte im Rahmen der Untersuchungen jedoch nicht durch unterschiedliche Quellen oder wissenschaftliche Beiträge untermauert werden. Es handelt sich daher vorwiegend um eine Annahme, die den gedanklichen Prozess des domänengetriebenen Entwurfs stützt. Auf dieser Basis und unter Einbeziehung der theoretischen Grundlagen des Domain-Driven Designs (vgl. Kapitel 2.4) konnte ein Prozessmodell zum domänengetriebenen Entwurf der fachlichen Architektur abgeleitet werden.

Das Prozessmodell betrachtet ausschließlich die strategischen Elemente des Domain-Driven Designs (vgl. Kapitel 2.4.2), da diese von zentraler Bedeutung für die fachliche Aufteilung von monolithischen Anwendungen sind. Das Modell beschränkt sich damit auf die Dekomposition der fachlichen Architektur und legt den Schwerpunkt bewusst auf den kritischen Bereich der Identifizierung von Kontext- bzw. Servicegrenzen. Die logische Grenze des Prozessmodells ist die Implementierungsebene. Sämtliche Aspekte, die die Imple-

306 Vgl. Di Francesco et al. 2017, S. 24.
307 Evans 2003.
308 Millett und Tune 2015.

mentierung einzelner Microservices betreffen, finden im Modell daher keine Berücksichtigung. Das strategische Design auf Ebene der Makroarchitektur und das taktische Design auf Ebene der Mikroarchitektur können im Kontext von Microservices als zwei unabhängige Bereiche betrachtet werden. Das strategische Design legt die fachlichen Grenzen für jeden Microservice fest und die Muster des taktischen Designs können angewendet werden, um jede abgegrenzte Fachlichkeit auf die Implementierungsebene zu überführen. Für eine DDD-spezifische Implementierung eines Microservice sollten demnach die Muster und Prinzipien des taktischen Designs zur Anwendung kommen. Der Schwerpunkt sollte dabei auf der korrekten Umsetzung der Fachlichkeit liegen, die in den Domänenmodellen abgebildet ist. Im Sinne des *Model-Driven Developments* kann dies automatisiert und unter Sicherstellung der Konformität von Quellcode und Domänenmodellen unterstützt werden. Im Umfeld von Domain-Driven Design ist dies jedoch mit gewissen Herausforderungen verbunden, wie z. B. in Rademacher et al.[309] beschrieben wird.

(4a) Prinzipien, Praktiken und Strategien im Transformationsprozess

Die Beantwortung der vierten Forschungsfrage ging mit der gezielten Aggregation und Strukturierung von Prinzipien, Praktiken und Strategien einher, die im Prozess der domänengetriebenen Transformation der fachlichen Architektur (vgl. Kapitel 4.2.2) von Bedeutung sind. Als Ergebnis konnten vier Bereiche identifiziert werden (vgl. Kapitel 4.4.1 bis 4.4.4), die zu einem erweiterten Verständnis des Transformationsprozesses beitragen. Die identifizierten Prinzipien, Praktiken und Strategien können folglich als konzeptionelle Erweiterung des Prozessmodells angesehen werden. Sie kennzeichnen unterstützende, prozessübergreifende Aspekte, die im Rahmen des Prozessmodells von Bedeutung sind, sich aber nicht für eine Abbildung auf der Modellebene eigneten.

Hinsichtlich der *Vorgehensmuster und Strategien* kann festgehalten werden, dass in der Mehrheit der Anwendungsfälle das Vorgehen der schrittweisen Transformation der Architektur gewählt werden sollte. Darauf aufbauend kann das Strangler Application Pattern besonders hervorgehoben werden. Durch inkrementelles Herauslösen von Fachlichkeit aus dem Monolithen ist das Risiko der Architektur-Transformation vermindert. Gleichzeitig bietet der Ansatz ein hohes Maß an Flexibilität, da während der Transformation auch neue Funktionalität umgesetzt und an den Monolithen angebunden werden kann. Ein wesentliches Ziel dabei ist die Minimierung von Abhängigkeiten der Microservices zum Monolithen. Weiterhin gilt es die vorgestellten Strategien und Praktiken zu berücksichtigen, die die grundlegende und wiederkehrende Entscheidung über die Extraktion oder Neuentwicklung von Geschäftsfähigkeiten unterstützen können.

Die *Dekomposition und Bestimmung der Größe einzelner Microservices* wird im domänengetriebenen Transformationsprozess nicht alleinig durch die Prinzipien und Muster des Domain-Driven Designs beeinflusst. Es handelt sich um einen erfahrungsbasierten Annäherungsprozess, bei dem die tatsächliche Servicegröße von unterschiedlichen Einflussfaktoren bestimmt werden kann. In Kapitel 4.4.2 wird daher eine Übersicht über die wesentlichen Einflussfaktoren gegeben. Neben dem Konzept der Bounded Contexts als obere Grenze für die Servicegröße sind vor allem die Faktoren der Ersetzbarkeit, der verteilten

[309] Rademacher et al. 2018.

Kommunikation und das Single Responsibility Principle hervorzuheben. Zusammenfassend machen die gesammelten Praktiken und Erfahrungen deutlich, dass keine feste Vorgabe für die ideale Größe eines Microservice existiert. Weiterhin kann argumentiert werden, dass auch der Reifegrad aus den Bereichen der Organisation, der Prozesse und der Technologien großen Einfluss auf die realisierbaren Servicegrößen besitzt (vgl. Kapitel 4.1.2).

Die identifizierten Microservices sollten in der Prozessphase des Forward Engineerings in einer entsprechend priorisierten Reihenfolge umgesetzt werden. Zur Unterstützung der *Priorisierung der zu entwickelnden Microservices* wurden daher verschiedene Strategien und Praktiken zusammengetragen. Die Strategie der kombinierten Berücksichtigung von Geschäftswert und Änderungshäufigkeit ist dabei von besonderem Stellenwert. Um eine einseitige Sichtweise zu verhindern, wurden die Priorisierungsaspekte sowohl aus geschäftlicher Perspektive als auch aus Sicht der Anwendung betrachtet.

Die Umsetzung von *Qualitätseigenschaften* wird maßgeblich durch die Softwarearchitektur beeinflusst. Daher sollten sich sämtliche Architekturentscheidungen auf die im Vorfeld definierten Qualitätseigenschaften konzentrieren (vgl. Kapitel 4.4.4). Dieser Zusammenhang spiegelt sich auch im gesamten Transformationsprozess der fachlichen Architektur wieder, in dem das Treffen von Architekturentscheidungen eine fundamentale und allgegenwärtige Aufgabe darstellt. Da alle Architekturstile die Qualität einer Software beeinflussen, erscheint diese Praktik von gewisser Trivialität. Das Ziel der Realisierung bestimmter Qualitätseigenschaften sollte jedoch aktiv in alle Architekturentscheidungen einbezogen werden. Die Auswahl des Architekturstils ist allein nicht zielführend, weil dadurch lediglich der grobe Rahmen für die Architektur definiert wird. Diese Praktik ist daher von großer Relevanz und Bedeutung für alle Architekturentscheidungen, die im gesamten Transformationsprozess getroffen werden.

(4b) Betrachtungen im Kontext agiler Softwareentwicklung

In Hinsicht auf die vierte Forschungsfrage hat sich im Rahmen der Untersuchungen ein weiteres unterstützendes Teilartefakt ergeben (vgl. Kapitel 4.5). Da sich der Microservice-Architekturstil und die Prinzipien und Praktiken der agilen Softwareentwicklung gegenseitig beeinflussen, wurden diese Zusammenhänge im Kontext der Tätigkeiten des Transformationsprozesses (vgl. Kapitel 4.2) als auch des domänengetriebenen Entwurfs (vgl. Kapitel 4.3) näher untersucht.

In Kapitel 4.5 konnte untermauert werden, dass beide Prozessmodelle als dynamische und nicht-lineare Prozesse zu verstehen sind. Weiterhin konnte gezeigt werden, dass sich sowohl die Tätigkeiten der Architektur-Transformation als auch des domänengetriebenen Entwurfs mit dem agilen Entwicklungsansatz vereinen lassen. Die Prozessmodelle sind auf das wiederholte Sammeln von Erfahrungen und den fortwährenden Wissensaufbau angewiesen. Sie können im Umfeld agiler Softwareentwicklung daher von dem schrittweisen und iterativen Vorgehen profitieren. Die Anwendung von agilen Prinzipien und Praktiken kann folglich die notwendige Flexibilität für diese erfahrungsbasierten Prozesse bereitstellen.

6.2 Validität der Ergebnisse

Zur Überprüfung des konzeptionellen Ordnungsrahmens (vgl. Kapitel 4) soll in diesem Kapitel eine Bewertung der Validität der Ergebnisse stattfinden. Die Validierung erfolgt in Bezug auf die Demonstration des Ordnungsrahmens (vgl. Kapitel 5) und hat das primäre Ziel die Teilartefakte auf dessen Anwendbarkeit zu untersuchen. Außerdem wird auf den Nutzen und die Genauigkeit der erreichten Ergebnisse eingegangen.

Die vollständige Überprüfung der *Anwendbarkeit* konnte im Zuge der Demonstration des Ordnungsrahmens nicht sichergestellt werden. Für eine praktische Anwendung im zeitlichen Rahmen der Arbeit eigneten sich ausschließlich die Teilartefakte der Situationsanalyse (vgl. Kapitel 4.1 und 5.1) und des Prozessmodells zum domänengetriebenen Entwurf (vgl. Kapitel 4.3.3 und 5.2). Dies kann auf die unterschiedlichen Eigenschaften und Zielstellungen der Teilartefakte zurückgeführt werden. Eine aussagekräftige Überprüfung des Prozessmodells zur Architektur-Transformation (vgl. Kapitel 4.2.2) kann durch die konsequente Anwendung der Prozessaktivitäten im Rahmen einer umfassenden Fallstudie erfolgen. Da sich Migrationsprojekte meist über einen großen Zeitraum erstrecken, war eine Überprüfung des Prozessmodells im zeitlichen Rahmen der Arbeit nicht realisierbar. Auch die unterstützenden Teilartefakte des Ordnungsrahmens (vgl. Kapitel 4.4 und 4.4.4) eigneten sich nicht für eine Demonstration. Sie besitzen informativen Charakter und haben das Ziel die Primärartefakte (vgl. Kapitel 4.2 und 4.3) zu ergänzen und zu unterstützen. Sie wurden im Rahmen der Arbeit daher nicht untersucht.

Die Bewertung des *Nutzens* und der *Genauigkeit* kann infolge der eingeschränkten Anwendbarkeit der Teilartefakte einzig in Bezug auf die Demonstrationsergebnisse der angewendeten Artefakte erfolgen. Daher können nur die Teilartefakte der Situationsanalyse (vgl. Kapitel 4.1 und 5.1) und des domänengetriebenen Entwurfs (vgl. Kapitel 4.3 und 5.2) evaluiert werden.

Die Durchführung der *Situationsanalyse* ermöglichte im Sinne des Teilartefaktes eine erste Einschätzung des Bedarfs und des Reifegrades der Organisation.
Die *Beurteilung des Bedarfs* konnte sowohl auf Basis der allgemeinen Motivationsfaktoren als auch der Einordung der Situation in Bezug auf das Verhältnis von Produktivität und Komplexität erfolgen. Die grundlegende Anwendbarkeit des Artefaktes ist damit gegeben. Da die Einschätzung in Abstimmung mit unterschiedlichen Anspruchsgruppen erfolgte, ist die Genauigkeit der Ergebnisse hoch einzustufen. Der Nutzen der Ergebnisse kann ebenfalls hoch eingestuft werden, da es sich um die erste Bedarfsanalyse im Projektkontext handelte. Der tatsächliche Nutzen ist jedoch stark von dem bereits vorhandenen Wissen und den Erfahrungen der Projektbeteiligten abhängig. Es können somit keine Rückschlüsse auf den allgemeinen Nutzen gezogen werden.
Zur *Beurteilung des Reifegrades* wurde der Fragenkatalog in Bezug auf die sechs Reifegradfaktoren beantwortet. Die Ergebnisse dienten als initiale Betrachtung des Reifegrades und resultierten in einer positiven Bewertung. Einzig im Bereich der DevOps-Kultur konnte aufgrund der organisatorischen Rahmenbedingungen der Untersuchung keine eindeutige Einschätzung des Reifegrades erfolgen. Es ist daher von einem mittelmäßigen bis hohen Anwendbarkeitsgrad auszugehen. Analog zur Bewertung des Bedarfs erfolgte

die Einschätzung in Abstimmung mit unterschiedlichen Anspruchsgruppen. Zudem handelte es sich um die ersten Erfahrungen der Organisation zur Bewertung des Reifegrades. Die Genauigkeit sowie der Nutzen der Ergebnisse sind daher hoch einzustufen. Abhängig vom Vorwissen und den Erfahrungen der Projektbeteiligten kann der tatsächliche Nutzen in anderen Projekten jedoch niedriger ausfallen.

Die Anwendung des *Prozessmodells zum domänengetriebenen Entwurf* ermöglichte die Komplexität der Problemdomäne zu strukturieren und erste Kontextgrenzen potenzieller Microservices zu identifizieren. Da das Prozessmodell auf den Mustern des strategischen Designs und deren grundlegenden Zusammenhängen aufbaut, kann von einer allgemeinen Anwendbarkeit im Sinne von Domain-Driven Design ausgegangen werden. Das bedeutet zugleich, dass sich die Anwendung des Modells vorwiegend für komplexe Problemdomänen und im Kontext objektorientierter Software eignet.
Da das Prozessmodell keine spezifischen Vorgaben zur Analyse der Domäne beinhaltet, werden die Genauigkeit und der Nutzen der Ergebnisse stark von den gewählten Techniken zur Domänenanalyse und dem Analyseumfang beeinflusst. Im Rahmen der Anwendung des Prozessmodells wurde einzig die Workshop-basierte Analysetechnik des Event Stormings angewendet. Weiterhin wurde der Workshop nur einmalig und mit einer geringen Anzahl an Teilnehmern sowie Anspruchsgruppen durchgeführt. Da der domänengetriebene Entwurf ein langwieriger Prozess ist, der von der Zusammenarbeit, den Erfahrungen und der Experimentierfreudigkeit aller Anforderungsgruppen lebt, ist die Genauigkeit der Ergebnisse nur niedrig bis mittelmäßig einzustufen. Zudem hätte die Anwendung alternativer Analysetechniken zu abweichenden Ergebnissen führen können.

Infolgedessen waren auch die entstandenen Domänenmodelle, die identifizierten Kontextgrenzen und die resultierende Kontextlandkarte noch unausgereift. Die Durchführung weiterer Workshops mit einer ausgedehnten Teilnehmerzahl und die Anwendung alternativer Analysetechniken würden entsprechend zu genaueren Ergebnissen führen. Gleichzeitig müsste vermehrt auf die zentralen Domänenobjekte (Aggregate) eingegangen werden, die in unterschiedlichen Geschäftsbereichen vorkommen und an verschiedenen Geschäftsprozessen beteiligt sind. Dies gilt ebenso für die verwendete Terminologie, die Konzepte sowie die Organisations- und Teamstrukturen, die sich hinter den abgebildeten Geschäftsbereichen und -prozessen verbergen (vgl. Kapitel 4.3.3). Darüber hinaus spielt auch das Feedback aus der Implementierung eine bedeutende Rolle für das Etablieren von Kontextgrenzen. In jedem Fall sollte der Schwerpunkt auf dem Erreichen einer hohen Modularität und hohen Unabhängigkeit der Systembestandteile liegen.
Das angewandte Vorgehen ermöglichte die systematische Identifizierung erster Kontextgrenzen, die eine grobe Richtung erkennen lassen und auf einer fundierten domänengetriebenen Vorgehensweise beruhen. Der Nutzen dieser frühen Ergebnisse ist in Bezug auf das betrachtete Unternehmen bzw. Projekt hoch einzuschätzen. Folglich kann davon ausgegangen werden, dass auch der allgemeine Nutzen des Prozessmodells gegeben ist.

Zur Untersuchung der externen Validität bzw. Allgemeingültigkeit sollten die Teilartefakte in mehreren unabhängigen Fallstudien überprüft werden. Dies war im zeitlichen Rahmen dieser Arbeit nicht realisierbar. Weiterführend kann die Validität der Ergebnisse auch in Bezug auf verschiedene Kombinationen der Teilartefakte untersucht werden. Beispielsweise kann die kombinierte Untersuchung des *Prozessmodells zur Architektur-*

Transformation (vgl. Kapitel 4.2.2) und des *Prozessmodells zum domänengetriebenen Entwurf* (vgl. Kapitel 4.3.3) Einsichten darüber liefern, wie sich die Tätigkeiten des domänengetriebenen Entwurfs auf den gesamten Transformationsprozess auswirken.

6.3 Praktische Implikationen

Auf Grundlage der Erkenntnisse der Ergebnisdiskussion (vgl. Kapitel 6.1) und der Validität der Ergebnisse (vgl. Kapitel 6.2) werden an dieser Stelle einige Schlussfolgerungen für die Praxis gezogen.

Das konkrete Vorgehen zur Transformation der fachlichen Architektur monolithischer Anwendungen wird in der Praxis von einer Vielzahl unterschiedlicher Faktoren bestimmt. Dazu zählen vor allem die vorherrschenden Rahmenbedingungen und die Zielstellungen des Unternehmens. Die Einbeziehung des Ordnungsrahmens kann Architekten, Entwicklern und Projektmanagern folglich als wesentliche Orientierungshilfe im Migrationsprozess dienen. Der Ordnungsrahmen kann dazu beitragen, dass Vorbedingungen sowie bedeutende Zusammenhänge berücksichtigt und das Vorgehen zur Migration grundlegend unterstützt wird. Dadurch kann die Komplexität von Migrationsprojekten strukturiert und handhabbar gemacht werden. Das gilt vor allem, wenn im Unternehmen oder Projekt wenig Erfahrung und Vorwissen vorhanden ist. Den Ergebnissen der Arbeit kann jedoch entnommen werden, dass Migrationen auch unter Zuhilfenahme des Ordnungsrahmens ein komplexes Vorhaben bleiben. Die Komplexität lässt sich nicht reduzieren. Sie kann lediglich strukturiert und besser sichtbar gemacht werden, was durch den Einsatz des Microservice-Architekturstils und Domain-Driven Design unterstützt werden kann.

Generell sollte im Unternehmens- bzw. Projektkontext das Ziel verfolgt werden, sich in allen Reifegrad-Bereichen zu entwickeln, um geeignete Vorbedingungen für den Einsatz des Microservice-Architekturstils zu schaffen (vgl. Kapitel 4.1.2). Der Bereich der agilen Softwareentwicklung spielt dabei eine besondere Rolle, da er sich sowohl auf den Prozess der Architektur-Transformation als auch den Prozess des domänengetriebenen Entwurfs positiv auswirkt (vgl. Kapitel 4.5).

In Hinsicht auf die Problemstellung (vgl. Kapitel 1.2) hat sich in den Ausführungen der Arbeit bestätigt, dass die Identifizierung von Servicegrenzen und das Festlegen der Größe von Microservices bedeutende Herausforderungen im Prozess der Architektur-Transformation und im domänengetriebenen Entwurf sind. Die Erfahrungen, die beim domänengetriebenen Entwurf der Werbezeitvermarktungsplattform gesammelt werden konnten, verdeutlichen dies (vgl. Kapitel 5.2). Sie bekräftigen die Strategie, einen Monolithen zunächst in grobgranulare Microservices zu schneiden und dadurch das Risiko für das Auftreten falsch gewählter Servicegrenzen zu senken (vgl. Kapitel 4.4.2). Durch die Orientierung an gröberen Kontextgrenzen können nachträgliche und oftmals aufwendige Korrekturen vermindert werden. Dieses Vorgehen ist zumindest in Erwägung zu ziehen, wenn Unsicherheit hinsichtlich der identifizierten Servicegrenzen besteht.

7. Schlussbetrachtung

7.1 Zusammenfassung

Das grundlegende Ziel der Arbeit war die Erarbeitung eines konzeptionellen Ordnungsrahmens, der die domänengetriebene Transformation der fachlichen Architektur monolithischer Anwendungen auf Basis des Microservice-Architekturstils unterstützen kann. Der Ordnungsrahmen sollte sowohl einen Gesamtblick auf den Transformationsprozess ermöglichen als auch einen detaillierten Blick auf den Bereich des domänengetriebenen Entwurfs der fachlichen Architektur vermitteln. Die wesentlichen Aspekte der Architektur-Transformation, fachlichen Dekomposition und Service-Identifikation sollten dabei in Form von Prozessmodellen strukturiert und verallgemeinert werden.

Die Erarbeitung der Lösung erfolgte auf der Grundlage einer systematischen Analyse von wissenschaftlicher Literatur. Sie diente der Erhebung des Forschungsstandes sowie der Identifizierung von relevanten Theorien und Konzepten zur Herleitung des Ordnungsrahmens. Die Ergebnisse bauen folglich zum großen Teil auf wissenschaftlichen Erkenntnissen und Erfahrungswerten aus der industriellen Praxis auf.

Der vorgestellte Ordnungsrahmen setzt sich aus fünf Teilartefakten zusammen, die konzeptionell sowie inhaltlich aneinander ausgerichtet sind. Den Kern des Ordnungsrahmens bilden das *Prozessmodell zur Architektur-Transformation* und das *Prozessmodell zum domänengetriebenen Entwurf*. Sie ermöglichen sowohl einen ganzheitlichen Blick auf den Migrationsprozess als auch einen detaillierten Blick auf den bedeutenden Teilbereich des fachlichen Schnittes auf Basis der Muster und Prinzipien des Domain-Driven Designs. Zur konzeptionellen Unterstützung der Primärartefakte wurden relevante Prinzipien, Praktiken und Strategien untersucht, die in zwei weiteren Teilartefakten zusammengefasst werden konnten. Sie beschreiben relevante Vorgehensmuster sowie wesentliche Aspekte der Dekomposition, Größe und Priorisierung von Microservices. Weiterhin wird auf die Rolle der Qualitätseigenschaften sowie die Prinzipien der agilen Softwareentwicklung im Kontext der Primärartefakte eingegangen. Der konzeptionelle Ordnungsrahmen wird folglich durch *Strategien und Praktiken im Transformationsprozess* als auch die *Betrachtungen im Kontext agiler Softwareentwicklung* erweitert. Darüber hinaus konnten die grundlegenden Vorbedingungen und Anforderungen des Microservice-Architekturstils abgeleitet werden und in Form einer *Situationsanalyse* in den Ordnungsrahmen einfließen.

Das *Prozessmodell zur Architektur-Transformation* basiert im Kern auf dem Konzept des Hufeisen-Modells und betrachtet die Anwendungs- und Datenarchitektur sowie die Geschäftsarchitektur. Es stellt einen von der Domäne getriebenen Transformationsprozess dar und vereint die wesentlichen fachlich motivierten Aktivitäten in drei Prozessphasen. Das Modell verkörpert einen langwierigen Transformationsprozess, der umfassende Auswirkungen auf die Softwarearchitektur hat und auf Ebene der Geschäftsarchitektur mit dem domänengetriebenen Entwurf der fachlichen Architektur verknüpft ist.
Das *Prozessmodell zum domänengetriebenen Entwurf* basiert auf dem allgemeinen Vorgehen und den Prinzipien des Domain-Driven Designs. Darauf aufbauend wurde ein Ansatz vorgeschlagen, der die Muster des strategischen Designs mit dem Konzept der Geschäfts-

fähigkeiten verbindet. Das resultierende Prozessmodell zum domänengetriebenen Entwurf ermöglicht damit die fachliche Dekomposition einer Domäne in serviceorientierte Domänenmodelle. Diese können als Ausgangsbasis für den fachlichen Schnitt eines Monolithen und die Komposition einer Microservice-Architektur dienen.

Die Überprüfung des Ordnungsrahmens erfolgte praxisnah an dem repräsentativen Beispiel einer monolithischen Anwendung zur Werbezeitvermarktung. Im Rahmen der Arbeit war die Anwendung und Überprüfung der Teilartefakte der Situationsanalyse und des domänengetriebenen Entwurfs realisierbar. Die Durchführung der Situationsanalyse führte zu dem Erkenntnisgewinn, dass das betrachtete Unternehmen die grundlegenden Vorbedingungen und Anforderungen für die Einführung des Microservice-Architekturstils weitestgehend erfüllt. Weiterhin konnten erste potenzielle Microservices durch Anwendung des Prozessmodells zum domänengetriebenen Entwurf sowie auf Basis der Ergebnisse eines Event Storming Workshops identifiziert werden. Die Genauigkeit der Ergebnisse war aufgrund des Analyseumfangs noch nicht ausreichend für eine fundierte fachliche Aufteilung der monolithischen Anwendung. Durch das Vorgehen konnte jedoch eine grobe Richtung zur Dekomposition des monolithischen Systems und eine Grundlage für anknüpfende Analysen erarbeitet werden. Bereits die Anwendung von zwei Teilartefakten konnte einen Nutzen für das betrachtete Unternehmen generieren. Der generelle Nutzen des konzeptionellen Ordnungsrahmens ist jedoch von dem Vorwissen und den Erfahrungen der Projektbeteiligten in Bezug auf die Transformation der fachlichen Architektur sowie deren Schnittbereiche abhängig. Gemäß der Evaluierungsergebnisse der Arbeit ist grundsätzlich von einer breiten Anwendbarkeit des Ordnungsrahmens auszugehen. Einzig die Muster und Prinzipien des Domain-Driven Designs sind vorwiegend auf komplexe Domänenlogik und Objektorientierung ausgelegt. Der Einsatz des Prozessmodells zum domänengetriebenen Entwurf eignet sich daher primär für komplexe Problemdomänen und objektorientierte Software.

Zusammenfassend konnten die Ausführungen dieser Arbeit zeigen, dass der konzeptionelle Ordnungsrahmen dazu beitragen kann das Vorgehen und die Komplexität von domänengetriebenen Migrationen monolithischer Anwendungen auf Basis des Microservice-Architekturstils zu strukturieren und handhabbar zu machen. Der vorgeschlagene Ordnungsrahmen vereint dafür wesentliche und zusammenhängende Aspekte von domänengetriebenen Transformationen der fachlichen Architektur auf konzeptioneller Ebene. Auf dem Weg zu einer Microservice-Architektur kann der Ordnungsrahmen folglich als Hilfsmittel für die Transformation der fachlichen Architektur dienen. Die Gesamtheit der sich gegenseitig unterstützenden Teilartefakte ermöglicht dabei ein planerisches Grundgerüst für Migrationsprojekte zu definieren. Die Berücksichtigung des Ordnungsrahmens kann daher insbesondere in frühen Erfahrungsstadien nützlich sein.

Letztlich hat sich in den Ausführungen der Arbeit bestätigt, dass die fachliche Dekomposition von Monolithen die größte Herausforderung im Prozess der Architektur-Transformation darstellt. Die Erfahrungen, die beim domänengetriebenen Entwurf der monolithischen Werbezeitvermarktungsplattform gesammelt wurden, untermauern dies in Bezug auf die Identifizierung von Servicegrenzen und das Festlegen geeigneter Servicegrößen.

7.2 Ausblick

Zur Weiterentwicklung des Ordnungsrahmens können mehrere Anknüpfungspunkte in Erwägung gezogen werden, die sich auf die Primärartefakte des Ordnungsrahmens beziehen und durch deren eingeschränkte Geltungsbereiche motiviert sind.

Das Hufeisen-Modell als Grundlage des Prozessmodells zur Architektur-Transformation sieht neben fachlichen Anpassungen auch Änderungen an der technischen Architektur vor. Diese wurden aufgrund der fachlichen Ausrichtung des Ordnungsrahmens ausgeblendet. Das Prozessmodell kann demnach durch verschiedene technische Aspekte erweitert werden. Eine Erweiterung kann z. B. durch das Ableiten technischer Anforderungen, die mit der Entwicklung und dem Betrieb von Microservices einhergehen, erfolgen.

Das Prozessmodell zum domänengetriebenen Entwurf beschränkt sich auf das Finden von Servicegrenzen auf Ebene der Makroarchitektur. In weiterführenden Betrachtungen kann eine Erweiterung und Überführung des Prozessmodells auf die Mikroarchitektur, also die Implementierungsebene, erfolgen. Der zentrale Anknüpfungspunkt ist dabei das Konzept der Bounded Contexts als fachliche Grenzen einzelner Microservices. Zur Erweiterung des Ordnungsrahmens können innerhalb dieser Abgrenzungen die Muster des taktischen Designs angewendet werden. Durch sie können Domain-Driven Design konforme Domänenmodelle erstellt und die Domänenlogik auf die Ebene der Code- und Datenmodelle abgebildet werden. Der Ansatz des Model-Driven Developments kann dies automatisiert und unter Sicherstellung der Konformität von Quellcode und Domänenmodellen unterstützten. In Hinsicht auf das Prozessmodell zur Architektur-Transformation würden die Muster und Prinzipien des Domain-Driven Designs somit auch auf der Ebene der Anwendungs- und Datenarchitektur angewendet werden.

Abschließend soll eine Handlungsempfehlung in Bezug auf das betrachtete Unternehmen und eine grundsätzliche Empfehlung gegeben werden.
Zur Erhöhung der Genauigkeit und des Nutzens der Ergebnisse sollten im Kontext der Werbezeitvermarktungsplattform weitere Domänenanalysen durchgeführt werden. Die Ergebnisse des Event Stormings können durch aufbauende Workshops verbessert und durch die Anwendung alternativer Analysetechniken untermauert werden.
Im Umfeld von Microservice-Architekturen kann die umzusetzende Fachlichkeit schnell von der Vielzahl der technischen Möglichkeiten zur Realisierung der Software überschattet werden. Die Möglichkeiten zur korrekten Umsetzung der Fachlichkeit sind dahingegen begrenzt. Zudem können technologische Fehlentscheidungen häufig unkomplizierter korrigiert werden als falsch abgegrenzte und umgesetzte Fachlogik. Dies trifft insbesondere für Microservices zu, die durch die Vorteile der Technologiefreiheit und leichten Ersetzbarkeit gekennzeichnet sind. Der Leitgedanke *Fachlichkeit vor Technik*, der dieser Arbeit zugrunde liegt, sollte demnach auch für jedes Migrationsprojekt gelten. In Bezug auf das einleitende Zitat dieser Arbeit wird abschließend behauptet:

> *Durch eine geeignete fachliche Aufteilung können Monolithen in modulare Microservice-Architekturen transformiert werden. Sie ermöglicht den Teams unabhängig und produktiv zu arbeiten und steht daher in enger Verbindung mit der Realisierung der positiven Eigenschaften guter Softwarearchitekturen.*

Literaturverzeichnis

Ariff, Ali; Steffens, Andreas (2018): Aspects of Software Complexity. In: RWTH Aachen University (Hg.): *Continuous Software Engineering & Full-scale Software Engineering - Seminar Winter Term 2017/2018.* Faculty of Mathematics, Computer Science, and Natural Sciences - Research Group Software Construction, S. 25–30.

Atchison, Lee (2016): Architecting for Scale. High Availability for Your Growing Applications. First Edition. Sebastopol, CA: O'Reilly Media.

Avram, Maricela-Georgiana (2014): Advantages and Challenges of Adopting Cloud Computing from an Enterprise Perspective. In: *Procedia Technology* 12, S. 529–534. DOI: 10.1016/j.protcy.2013.12.525.

Balalaie, Armin; Heydarnoori, Abbas; Jamshidi, Pooyan (2016a): Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. In: *IEEE Software* 33 (3), S. 42–52. DOI: 10.1109/MS.2016.64.

Balalaie, Armin; Heydarnoori, Abbas; Jamshidi, Pooyan (2016b): Migrating to Cloud-Native Architectures Using Microservices: An Experience Report. In: Antonio Celesti und Philipp Leitner (Hg.): *Advances in service-oriented and cloud computing. Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015: revised selected papers* (Communications in Computer and Information Science, 567). Switzerland: Springer, S. 201–215.

Balalaie, Armin; Heydarnoori, Abbas; Jamshidi, Pooyan; Tamburri, Damian A.; Lynn, Theo (2018): Microservices Migration Patterns. In: *Software: Practice and Experience* 48 (11), S. 2019–2042. DOI: 10.1002/spe.2608.

Barcia, Roland; Brown, Kyle; Chen, Gang; Daya, Shahir; Martins, Marcelo; Osowski, Richard; Verwaayen, James (2017): Microservices Decision Guides. URL: https://www.ibm.com/cloud/garage/files/Microservices-Decision-Guides-FINAL-1.pdf, zuletzt geprüft am: 15.10.2018.

Bass, Len; Clements, Paul; Kazman, Rick (2013): Software Architecture in Practice. Third Edition. Upper Saddle River, NJ: Addison-Wesley.

Bass, Len; Weber, Ingo; Zhu, Liming (2015): DevOps. A Software Architect's Perspective. eBook (Safari Tech Books Online). Upper Saddle River, NJ: Addison-Wesley. URL: https://www.safaribooksonline.com/library/view/devops-a-software/9780134049885/.

Behara, Gopala Krishna; Khandrika, Tirumala (2017): Microservices Maturity Model. In: *International Journal of Engineering and Computer Science* 6 (11), S. 22861–22864. DOI: 10.18535/ijecs/v6i11.07.

Bonér, Jonas (2017): Reactive Microsystems. The Evolution of Microservices at Scale. First Edition. Sebastopol, CA: O'Reilly Media.

Bucchiarone, Antonio; Dragoni, Nicola; Dustdar, Schahram; Larsen, Stephan T.; Mazzara, Manuel (2018): From Monolithic to Microservices. An Experience Report from the Banking Domain. In: *IEEE Software* 35 (3), S. 50–55. DOI: 10.1109/MS.2018.2141026.

Carneiro, Cloves; Schmelmer, Tim (2016): Microservices From Day One. Build robust and scalable software from the start. Berkeley, CA: Apress.

Cerny, Tomas; Donahoo, Michael J.; Trnka, Michal (2017): Contextual Understanding of Microservice Architecture: Current and Future Directions. In: *ACM SIGAPP Applied Computing Review* 17 (4), S. 29–45.

Conway, Melvin E. (1968): How Do Committees Invent? In: *Datamation* 14 (4), S. 28–31.

Cooper, Harris M. (1988): Organizing Knowledge Syntheses: A Taxonomy of Literature Reviews. In: *Knowledge in Society* 1 (1), S. 104–126. DOI: 10.1007/BF03177550.

Davis, Jennifer; Daniels, Katherine (2016): Effective DevOps. Building a Culture of Collaboration, Affinity, and Tooling at Scale. First Edition. Sebastopol, CA: O'Reilly Media.

Daya, Shahir (2015): Microservices from Theory to Practice. Creating Applications in IBM Bluemix Using the Microservices Approach. First Edition (IBM redbooks).

Dehghani, Zhamak (2018): How to break a Monolith into Microservices. URL: https://martinfowler.com/articles/break-monolith-into-microservices.html, zuletzt geprüft am: 27.11.2018.

Di Francesco, Paolo; Lago, Patricia; Malavolta, Ivano (2018): Migrating Towards Microservice Architectures: An Industrial Survey. In: *2018 IEEE International Conference on Software Architecture (ICSA).* Seattle, WA, 4/30/2018 - 5/4/2018. IEEE, S. 29–38.

Di Francesco, Paolo; Malavolta, Ivano; Lago, Patricia (2017): Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In: *2017 IEEE International Conference on Software Architecture (ICSA).* Gothenburg, Sweden, 03.04.2017 - 07.04.2017. IEEE, S. 21–30.

Dowalil, Herbert (2018): Grundlagen des modularen Softwareentwurfs. Der Bau langlebiger Mikro- und Makro-Architekturen wie Microservices und SOA 2.0. eBook. München: Hanser.

Dragoni, Nicola; Lanese, Ivan; Larsen, Stephan Thordal; Mazzara, Manuel; Mustafin, Ruslan; Safina, Larisa (2018): Microservices: How To Make Your Application Scale. In: Alexander K. Petrenko und Andrei Voronkov (Hg.): *Perspectives of System Informatics. 11th International Andrei P. Ershov Informatics Conference, PSI 2017, Moscow, Russia, June 27-29, 2017, Revised Selected Papers, Bd.* 10742. Cham: Springer International Publishing, S. 95–104.

Duden online, Suchbegriff: „Monolith“ (2018). URL: https://www.duden.de/rechtschreibung/Monolith, zuletzt geprüft am: 06.09.2018.

DZone Inc. (2018): Microservices. Breaking Down the Monolith (1). URL: https://dzone.com/guides/microservices-breaking-down-the-monolith, zuletzt geprüft am: 21.09.2018.

Erder, Murat; Pureur, Pierre (2016): Continuous Architecture. Sustainable Architecture in an Agile and Cloud-Centric World. Waltham, MA: Morgan Kaufmann.

Esposito, Dino (2016): Modern Web Development. Understanding domains, technologies, and user experience. eBook. Redmond, WA: Microsoft Press.

Evans, Eric (2003): Domain-Driven Design. Tackling Complexity in the Heart of Software. Upper Saddle River, NJ: Addison-Wesley.

Evans, Eric (2015): Domain-Driven Design. Definitions and Pattern Summaries. URL: https://domainlanguage.com/ddd/reference/, zuletzt geprüft am: 18.10.2018.

Evans, Eric; Wolff, Eberhard (2015): Eric Evans on Domain-Driven Design at 10 Years. In: *InfoQ eMag: Architectures You Always Wondered About* (31). URL: https://www.infoq.com/minibooks/emag-microservice-architecture, zuletzt geprüft am: 07.01.2019.

Familiar, Bob (2015): Microservices, IoT, and Azure. Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions. New York, NY: Apress.

Fan, Chen-Yuan; Ma, Shang-Pin (2017): Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. In: *2017 IEEE International Conference on AI & Mobile Services (AIMS).* Honolulu, HI, USA, 25.06.2017 - 30.06.2017. IEEE, S. 109–112.

Fehling, Christoph; Leymann, Frank; Retter, Ralph; Schupeck, Walter; Arbitter, Peter (2014): Cloud Computing Patterns. Fundamentals to Design, Build, and Manage Cloud Applications. Wien: Springer.

Fowler, Martin (2004): StranglerApplication. URL: https://www.martinfowler.com/bliki/StranglerApplication.html, zuletzt geprüft am: 10.10.2018.

Fowler, Martin (2015a): MonolithFirst. URL: https://martinfowler.com/bliki/MonolithFirst.html, zuletzt geprüft am: 17.09.2018.

Fowler, Martin (2015b): Microservice Premium. URL: https://martinfowler.com/bliki/MicroservicePremium.html, zuletzt geprüft am: 06.09.2018.

Fowler, Martin; Lewis, James (2015): Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr? In: *OBJEKTspektrum* (01), S. 14–20.

Fowler, Martin; Rice, David (2002): Patterns of Enterprise Application Architecture. 17. Print. Boston, MA: Addison-Wesley.

Fowler, Susan J. (2017): Production-Ready Microservices. Building Standardized Systems Across an Engineering Organization. First Edition. Sebastopol, CA: O'Reilly Media.

Frey, Frank J.; Hentrich, Carsten; Zdun, Uwe (2015): Capability-based Service Identification in Service-Oriented Legacy Modernization. In: Christian Kohls (Hg.): *Proceedings of the 18th European Conference on Pattern Languages of Program.* Irsee, Germany, 7/10/2013 - 7/14/2013. New York, NY: ACM, S. 1–12.

Fritzsch, Jonas; Bogner, Justus; Zimmermann, Alfred; Wagner, Stefan (2019): From Monolith to Microservices: A Classification of Refactoring Approaches. In: Jean-Michel Bruel, Manuel Mazzara und Bertrand Meyer (Hg.): *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment. DEVOPS 2018* (Lecture Notes in Computer Science, 11350). Cham: Springer International Publishing, S. 128–141.

Global Microservices Trends. A Survey Of Development Professionals (2018). URL: https://go.lightstep.com/global-microservices-trends-report-2018, zuletzt geprüft am: 05.01.2019.

Goll, Joachim; Dausmann, Manfred (2013): Architektur- und Entwurfsmuster der Softwaretechnik. Mit lauffähigen Beispielen in Java. Wiesbaden: Springer Vieweg.

Gonzalez, David (2016): Developing Microservices with Node.js. Learn to develop efficient and scalable microservices for server-side programming in Node.js using this hands-on guide. Birmingham: Packt Publishing.

Gouigoux, Jean-Philippe; Tamzalit, Dalila (2017): From Monolith to Microservices. Lessons Learned on an Industrial Migration to a Web Oriented Architecture. In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW).* Gothenburg, Sweden, 05.04.2017 - 07.04.2017. IEEE, S. 62–65.

Gruhn, Volker (2018): Die Organisation der Zukunft: Microservices. In: *Wirtschaftsinformatik & Management* 10 (1), S. 52–56.

Hasselbring, Wilhelm (2018): Software Architecture: Past, Present, Future. In: Volker Gruhn und Rüdiger Striemer (Hg.): *The Essence of Software Engineering*. Cham: Springer International Publishing, S. 169–184.

Herden, Sebastian; Gómez, Jorge Marx; Rautenstrauch, Claus; Zwanziger, André (2006): Software-Architekturen für das E-Business. Enterprise-Application-Integration mit verteilten Systemen. Berlin, Heidelberg: Springer.

Hevner, Alan R.; March, Salvatore T.; Park, Jinsoo; Ram, Sudha (2004): Design Science in Information Systems Research. In: *MIS Quarterly* 28 (1), S. 75–105.

Hippchen, Benjamin; Giessler, Pascal; Steinegger, Roland H.; Schneider, Michael; Abeck, Sebastian (2017): Designing Microservice-Based Applications by Using a Domain-Driven Design Approach. In: *International Journal On Advances in Software* 10 (3-4), S. 432–445.

ISO/IEC 9126-1:2001. Software engineering - Product quality - Part 1: Quality model. URL: https://www.iso.org/standard/22749.html, zuletzt geprüft am: 20.12.2018.

ISO/IEC/IEEE 42010:2011. Systems and software engineering - Architecture description. URL: https://www.iso.org/standard/50508.html, zuletzt geprüft am: 12.12.2018.

Joselyne, Munezero Immaculee; Kanagwa, Benjamin; Balikuddembe, Joseph (2017): A framework to Modernize SME Application in Emerging Economies. Microservice Architecture Pattern Approach. *International Conference on Microservices 2017*. University of Southern Denmark. Odense, Denmark, 25.10.2017.

Kakivaya, Gopal; Modi, Vipul; Mohsin, Mansoor; Kong, Ray; Ahuja, Anmol; Platon, Oana et al. (2018): Service fabric: A Distributed Platform for Building Microservices in the Cloud. In: Pascal Felber, Y. Charlie Hu und Rui Oliveira (Hg.): *Proceedings of the Thirteenth EuroSys Conference on - EuroSys '18.* Porto, Portugal, 23.04.2018 - 26.04.2018. New York, NY: ACM, Article No. 33.

Kalske, Miika; Mäkitalo, Niko; Mikkonen, Tommi (2018): Challenges When Moving from Monolith to Microservice Architecture. In: Irene Garrigós und Manuel Wimmer (Hg.): *Current Trends in Web Engineering* (Lecture Notes in Computer Science, 10544). Cham: Springer International Publishing, S. 32–47.

Kazman, Rick; Woods, Steven G.; Carriere, S. Jeromy (1998): Requirements for integrating software architecture and reengineering models: CORUM II. In: *Fifth Working Conference on Reverse Engineering.* Honolulu, HI, USA, 12-14 Oct. 1998. Los Alamitos, CA: IEEE Computer Society Press, S. 154–163.

Khusidman, Vitaly; Ulrich, William (2007): Architecture-Driven Modernization: Transforming the Enterprise. URL: https://www.omg.org/adm/ADMRoadmapv6_VK.pdf, zuletzt geprüft am: 29.09.2018.

Knoche, Holger; Hasselbring, Wilhelm (2018): Using Microservices for Legacy Software Modernization. In: *IEEE Software* 35 (3), S. 44–49. DOI: 10.1109/MS.2018.2141035.

Larrucea, Xabier; Santamaria, Izaskun; Colomo-Palacios, Ricardo; Ebert, Christof (2018): Microservices. In: *IEEE Software* 35 (3), S. 96–100. DOI: 10.1109/MS.2018.2141030.

Lewis, James; Fowler, Martin (2014): Microservices. a definition of this new architectural term. URL: https://martinfowler.com/articles/microservices.html, zuletzt geprüft am: 15.09.2018.

Lilienthal, Carola (2017): Langlebige Softwarearchitekturen. Technische Schulden analysieren, begrenzen und abbauen. eBook. 2. Aufl. Heidelberg: dpunkt.verlag.

Martin, Robert C. (2017): Clean Architecture. A Craftsman's Guide to Software Structure and Design. eBook. First Edition. Boston, MA: Prentice Hall. URL: https://www.oreilly.com/library/view/clean-architecture-a/9780134494272/.

Mell, Peter; Grance, Timothy (2011): The NIST definition of cloud computing. Recommendations of the National Institute of Standards and Technology. NIST. Gaithersburg, MD. URL: https://csrc.nist.gov/publications/detail/sp/800-145/final.

Microsoft Corporation (2009): Microsoft Application Architecture Guide. Patterns & Practices. Second Edition. Redmond, WA: Microsoft Press.

Millett, Scott; Tune, Nick (2015): Patterns, Principles, and Practices of Domain-Driven Design. Indianapolis, IN: John Wiley & Sons.

Monson-Haefel, Richard (2009): 97 Things Every Software Architect Should Know. Sebastopol, CA: O'Reilly Media.

Mosleh, Mohsen; Dalili, Kia; Heydari, Babak (2018): Distributed or Monolithic? A Computational Architecture Decision Framework. In: *IEEE Systems Journal* 12 (1), S. 125–136. DOI: 10.1109/JSYST.2016.2594290.

Nadareishvili, Irakli; Mitra, Ronnie; McLarty, Matt; Amundsen, Mike (2016): Microservice Architecture. Aligning Principles, Practices, and Culture. First Edition. Sebastopol, CA: O'Reilly Media.

Newman, Sam (2015): Building Microservices. Designing Fine-Grained Systems. First Edition. Sebastopol, CA: O'Reilly Media.

Oelmann, Guido (2018): Modularisierung mit Java 9. Grundlagen und Techniken für langlebige Softwarearchitekturen. eBook. 1. Aufl. Heidelberg: dpunkt.verlag.

O'Hanlon, Charlene (2006): A conversation with Werner Vogels. Learning from the Amazon technology platform. In: *ACM Queue* 4 (4), S. 14–22.

Oquendo, Flavio; Leite, Jair; Batista, Thaís (2016): Software Architecture in Action. Designing and Executing Architectural Models with SysADL grounded on the OMG SysML Standard. Cham: Springer International Publishing.

Pahl, Claus; Jamshidi, Pooyan (2016): Microservices: A Systematic Mapping Study. In: Jorge Cardoso, Donald Ferguson, Víctor Méndez Muñoz und Markus Helfert (Hg.): *CLOSER 2016. Proceedings of the 6th International Conference on Cloud Computing and Services Science.* Rome, Italy, 4/23/2016 - 4/25/2016. Setúbal, Portugal: SCITEPRESS - Science and Technology Publications, S. 137–146.

Pahl, Claus; Jamshidi, Pooyan; Zimmermann, Olaf (2018): Architectural Principles for Cloud Software. In: *ACM Transactions on Internet Technology* 18 (2), Article No. 17. DOI: 10.1145/3104028.

Pautasso, Cesare; Zimmermann, Olaf; Amundsen, Mike; Lewis, James; Josuttis, Nicolai (2017a): Microservices in Practice, Part 1. Reality Check and Service Design. In: *IEEE Software* 34 (1), S. 91–98. DOI: 10.1109/MS.2017.24.

Pautasso, Cesare; Zimmermann, Olaf; Amundsen, Mike; Lewis, James; Josuttis, Nicolai (2017b): Microservices in Practice, Part 2. Service Integration and Sustainability. In: *IEEE Software* 34 (2), S. 97–104. DOI: 10.1109/MS.2017.56.

Peffers, Ken; Tuunanen, Tuure; Rothenberger, Marcus A.; Chatterjee, Samir (2007): A Design Science Research Methodology for Information Systems Research. In: *Journal of Management Information Systems* 24 (3), S. 45–77. DOI: 10.2753/MIS0742-1222240302.

Pérez-Castillo, Ricardo; Rodríguez de Guzmán, Ignacio García; Piattini, Mario (2011): Architecture-Driven Modernization. In: Srikanta Patnaik, Ali H. Dogru und Veli Biçer (Hg.): *Modern Software Engineering Concepts and Practices* (Advances in Computer and Electrical Engineering). IGI Global, S. 75–103.

Project Management Institute (2008): A guide to the Project Management Body of Knowledge. (PMBOK guide). Fourth Edition. Newtown Square, PA: PMI.

Rademacher, Florian; Sorgalla, Jonas; Sachweh, Sabine (2018): Challenges of Domain-Driven Microservice Design. A Model-Driven Perspective. In: *IEEE Software* 35 (3), S. 36–43. DOI: 10.1109/MS.2018.2141028.

Razavian, Maryam; Lago, Patricia (2010): Towards a Conceptual Framework for Legacy to SOA Migration. In: Asit Dan, Frédéric Gittler und Farouk Toumani (Hg.): *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*. ServiceWave 2009, ICSOC 2009 (Lecture Notes in Computer Science, 6275). Berlin: Springer, S. 445–455.

Richards, Mark (2016): Microservices AntiPatterns and Pitfalls. First Edition. Sebastopol, CA: O'Reilly Media.

Richardson, Chris (2018): Microservice Patterns. Shelter Island, NY: Manning.

Salah, Tasneem; Jamal Zemerly, M.; Yeun, Chan Yeob; Al-Qutayri, Mahmoud; Al-Hammadi, Yousof (2016): The Evolution of Distributed Systems Towards Microservices Architecture. In: *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST).* Barcelona, Spain, 12/5/2016 - 12/7/2016. Piscataway, NJ: IEEE, S. 318–325.

Santis, Sandro de; Florez, Luis; Nguyen, Duy V.; Rosa Eduardo (2016): Evolve the Monolith to Microservices with Java and Node (IBM redbooks). Poughkeepsie, NY: IBM International Technical Support Organization.

Schwartz, Alexander (2017): Microservices. Mehr als nur ein Hype? In: *Informatik-Spektrum* 40 (6), S. 590–594. DOI: 10.1007/s00287-017-1078-6.

Sharma, Sourabh; RV, Rajesh; Gonzalez, David (2016): Microservices: Building Scalable Software. Discover how to easily build and implement scalable microservices from scratch. Birmingham, UK: Packt Publishing.

Starke, Gernot (2018): Effektive Softwarearchitekturen. Ein praktischer Leitfaden. 8. Aufl. München: Hanser.

Stine, Matt (2015): Migrating to Cloud-Native Application Architectures. First Edition. Sebastopol, CA: O'Reilly Media.

Taibi, Davide; Lenarduzzi, Valentina (2018): On the Definition of Microservice Bad Smells. In: *IEEE Software* 35 (3), S. 56–62. DOI: 10.1109/MS.2018.2141031.

Taibi, Davide; Lenarduzzi, Valentina; Pahl, Claus (2017): Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation. In: *IEEE Cloud Computing* 4 (5), S. 22–32. DOI: 10.1109/MCC.2017.4250931.

Takai, Daniel (2017): Architektur für Websysteme. Serviceorientierte Architektur, Microservices, domänengetriebener Entwurf. München: Hanser.

ThoughtWorks Inc. (2015): Inverse Conway Maneuver. ThoughWorks Technology Radar. URL: https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver, zuletzt geprüft am: 20.09.2018.

Tiemeyer, Ernst (2014): Handbuch IT-Projektmanagement. Vorgehensmodelle, Managementinstrumente, Good Practices. 2. Aufl. München: Hanser.

Toth, Stefan (2014): Vorgehensmuster für Softwarearchitektur. Kombinierbare Praktiken in Zeiten von Agile und Lean. München: Hanser.

Turowski, Klaus (2003): Fachkomponenten. Komponentenbasierte betriebliche Anwendungssysteme (Magdeburger Schriften zur Wirtschaftsinformatik). Aachen: Shaker.

Uludağ, Ömer; Hauder, Matheus; Kleehaus, Martin; Schimpfle, Christina; Matthes, Florian (2018): Supporting Large-Scale Agile Development with Domain-Driven Design. In: Juan Garbajosa, Xiaofeng Wang und Ademar Aguiar (Hg.): *Agile Processes in Software Engineering and Extreme Programming* (Lecture Notes in Business Information Processing, 314). Cham: Springer International Publishing, S. 232–247.

Venugopal, M.V.L.N. (2017): Containerized Microservices architecture. In: *International Journal of Engineering and Computer Science* 6 (11), S. 23199–23208. DOI: 10.18535/ijecs/v6i11.20.

Vernon, Vaughn (2013): Implementing Domain-Driven Design. eBook. Upper Saddle River, NJ: Addision-Wesley.

Vernon, Vaughn; Lilienthal, Carola; Schwentner, Henning (2017): Domain-Driven Design kompakt. 1. Aufl. Heidelberg: dpunkt.verlag.

Vigenschow, Uwe (2015): Der APM-Guide. APM - Agiles Projektmanagement. 1. Aufl. Heidelberg: dpunkt.verlag. URL: http://www.dpunkt.de/buecher/5392/der-apm-guide.html, zuletzt geprüft am: 22.09.2018.

Vom Brocke, Jan; Simons, Alexander; Niehaves, Bjoern; Niehaves, Bjorn; Riemer, Kai; Plattfaut, Ralf; Cleven, Anne (2009): Reconstructing the Giant: On the Importance of Rigour in Documenting the Literature Search Process. In: Susan Newell, Edgar Whitley, Nancy Pouloudi, Jonathan Wareham und Lars Mathiassen (Hg.): *Information systems in a globalising world. Challenges, ethics and practices; ECIS 2009, 17th European Conference on Information Systems, 8 - 10 June 2009*. Verona, Italy, S. 2206–2217.

Wagner, Christian (2014): Model-Driven Software Migration: A Methodology. Reengineering, Recovery and Modernization of Legacy Systems. Wiesbaden: Springer Vieweg.

Wasukar, Amit; Deshmukh, Ritesh; Newaskar, Anuja (2013): A Review on Architecture Driven Modernization - ADM. In: *International Journal of Scientific & Engineering Research* 4 (12), S. 140–146.

Webster, Jane; Watson, Richard T. (2002): Analyzing the past to prepare for the future: writing a literature review. In: *MIS Quarterly* 26 (2), S. xiii–xxiii.

Wilder, Bill (2012): Cloud Architecture Patterns. First Edition. Sebastopol, CA: O'Reilly.

Wolff, Eberhard (2015): Continuous Delivery. Der pragmatische Einstieg. eBook. 1. Aufl. Heidelberg: dpunkt.verlag.

Wolff, Eberhard (2016): Microservices. Grundlagen flexibler Softwarearchitekturen. e-Book. 1. Aufl. Heidelberg: dpunkt.verlag.

Workplace Solutions GmbH: Domain Storytelling. URL: http://www.domainstorytelling.org/, zuletzt geprüft am: 14.11.2018.

Zhu, Liming; Bass, Len; Champlin-Scharff, George (2016): DevOps and Its Practices. In: *IEEE Software* 33 (3), S. 32–34. DOI: 10.1109/MS.2016.81.

Zimmermann, Olaf (2017): Microservices Tenets. Agile Approach to Service Development and Deployment. In: *Computer Science - Research and Development* 32 (3-4), S. 301–310. DOI: 10.1007/s00450-016-0337-0.

Zörner, Stefan (2012): Softwarearchitekturen dokumentieren und kommunizieren. Entwürfe, Entscheidungen und Lösungen nachvollziehbar und wirkungsvoll festhalten. München: Hanser.

A. Anhang

A.1 Abbildungen

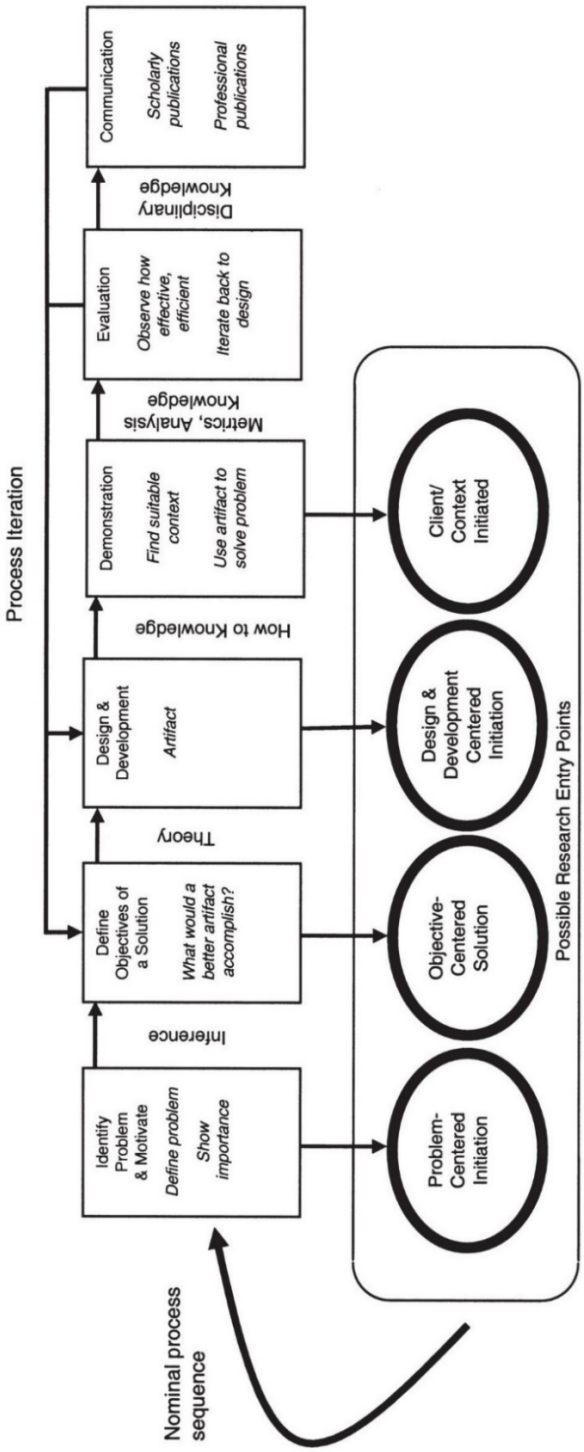

Abbildung A-1: Das DSRM-Prozessmodell [310]

[310] Peffers et al. 2007, S. 54.

Abbildung A-2: Übersicht der Muster des Domain-Driven Designs [311]

[311] Evans 2015, vii.

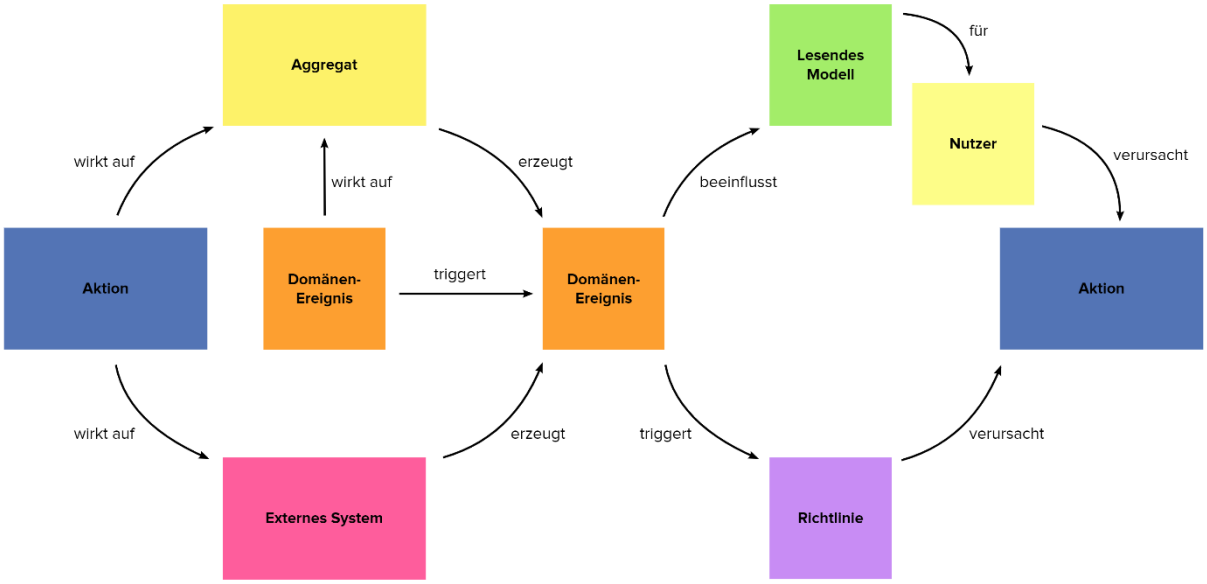

Abbildung A-3: Legende des Event Storming Workshops

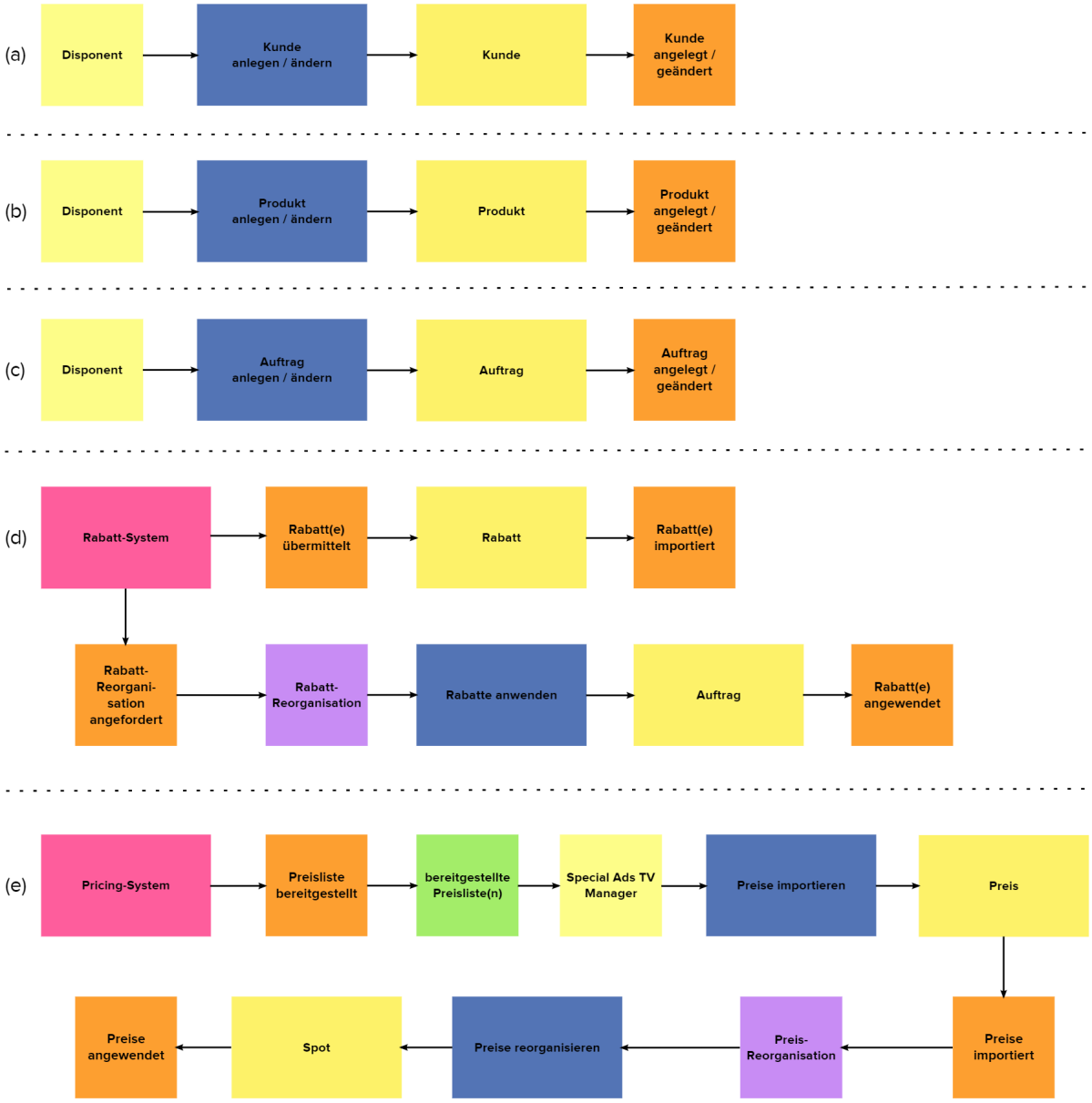

Abbildung A-4: Ergebnisse des Event Storming Workshops (Teil 1)

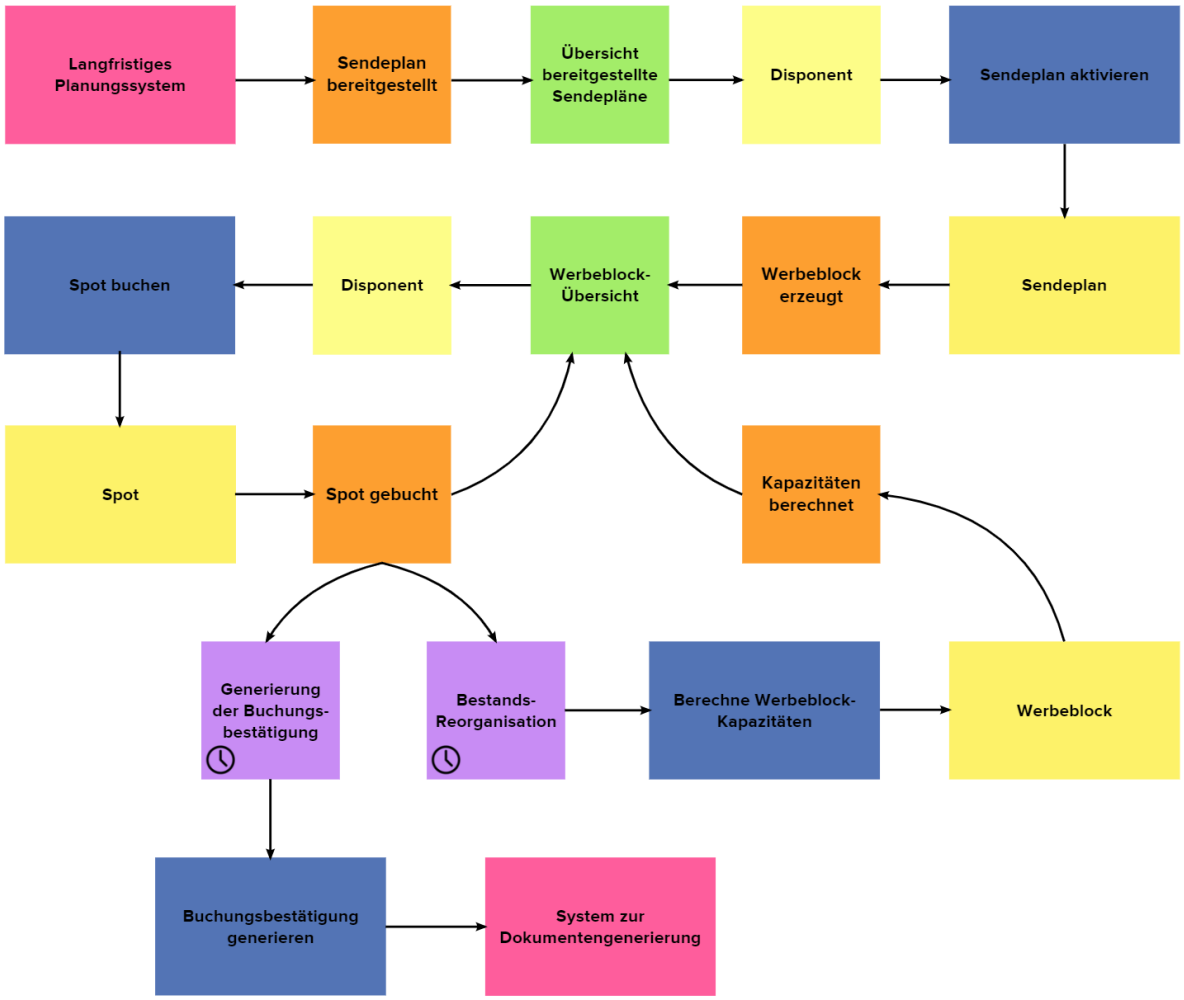

Abbildung A-5: Ergebnisse des Event Storming Workshops (Teil 2)

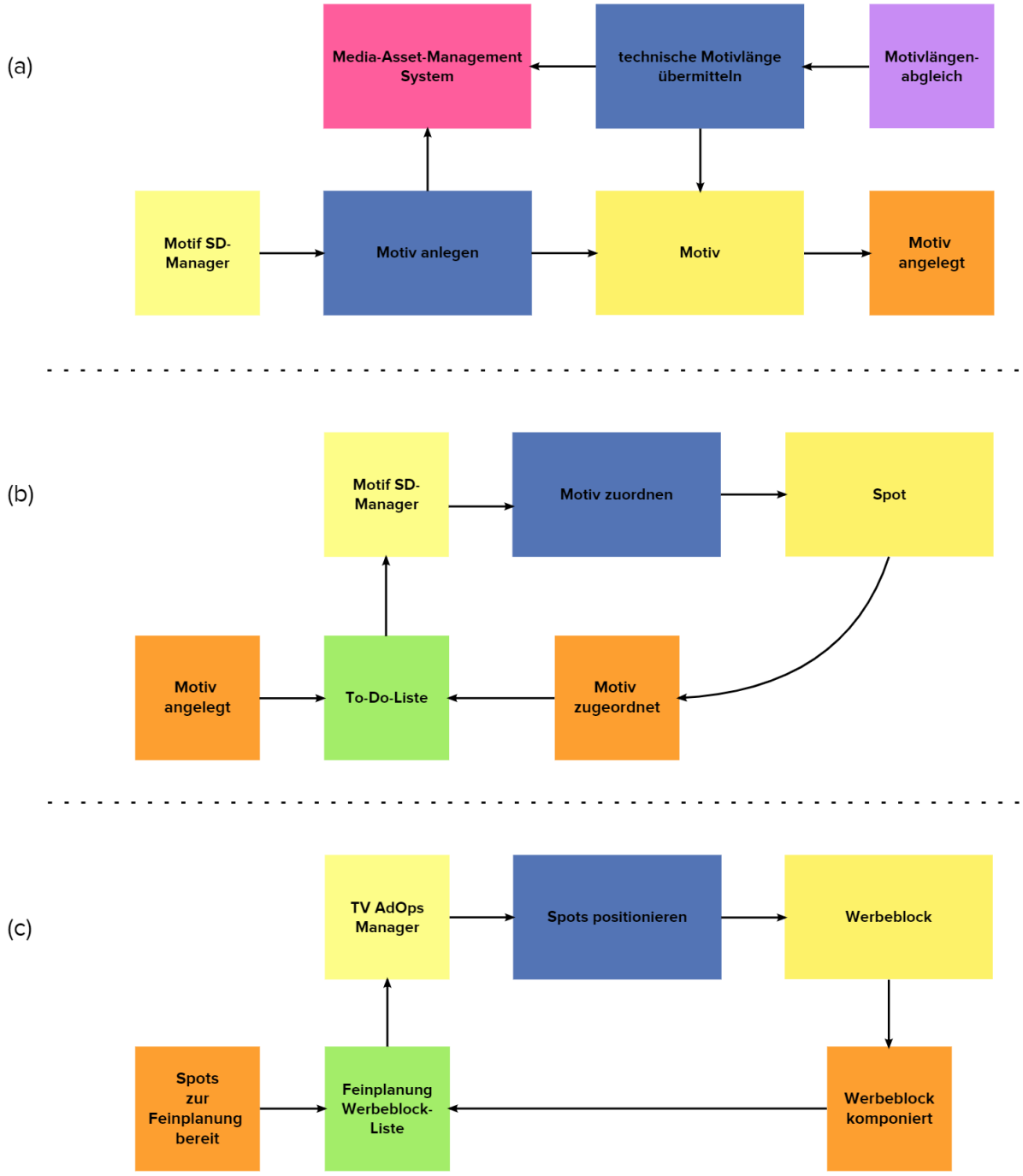

Abbildung A-6: Ergebnisse des Event Storming Workshops (Teil 3)

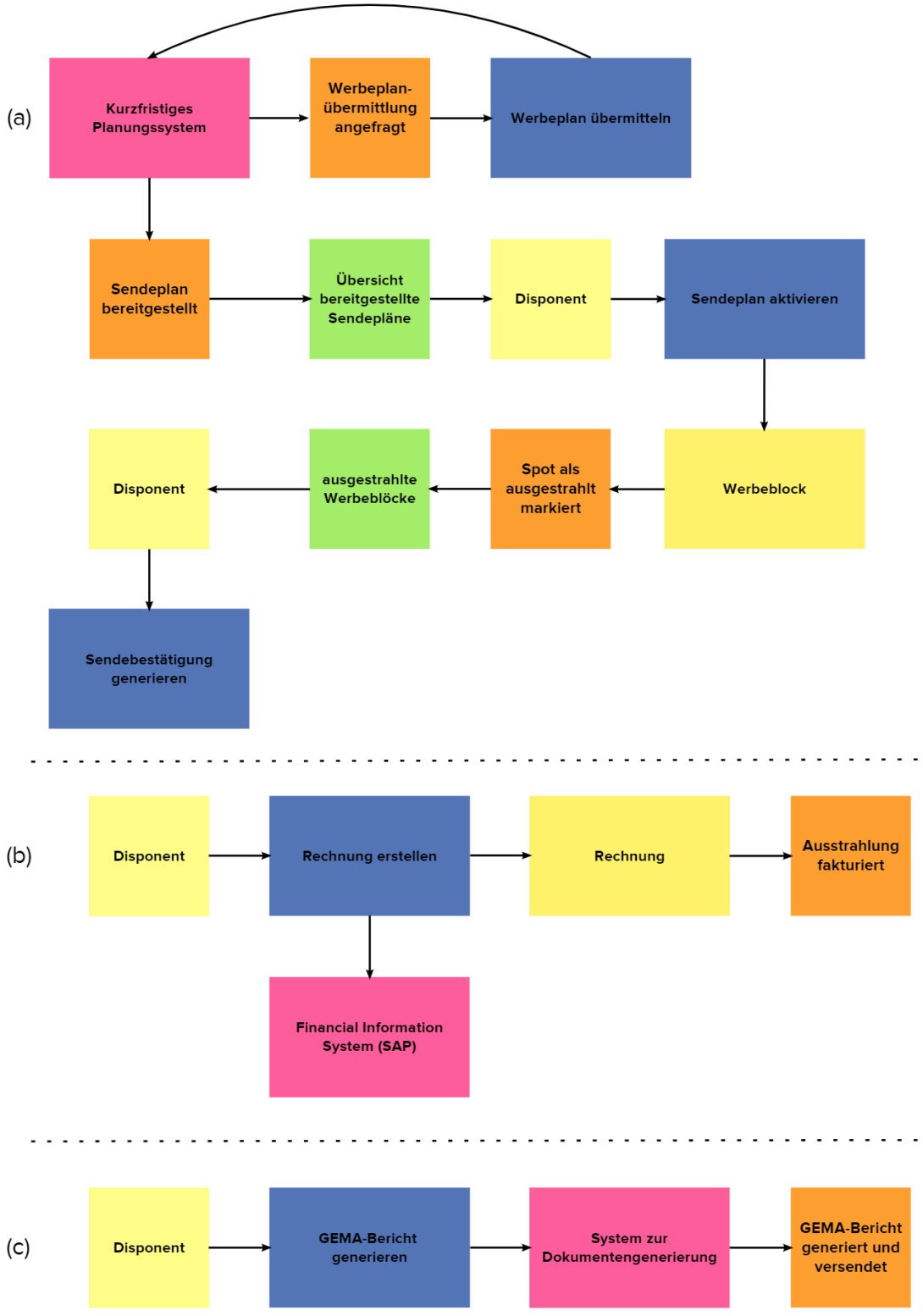

Abbildung A-7: Ergebnisse des Event Storming Workshops (Teil 4)

A.2 Tabellen

Nr.	DSRM-Richtlinie	DSRM-Richtlinienbeschreibung
I	Design als Artefakt	Das Ergebnis muss ein realisierbares Artefakt in Form eines Konstruktes, Modells, einer Methode oder Instanziierung sein.
II	Problemrelevanz	Das Ziel ist die Entwicklung von technologiebasierten Lösungen für bedeutende und relevante Geschäftsprobleme.
III	Design Evaluierung	Der Nutzen, die Qualität und die Wirksamkeit eines Design-Artefakts müssen durch gut ausgeführte Bewertungsmethoden genau demonstriert werden.
IV	Forschungsbeitrag	Effektive Design-Science Research muss klare und nachprüfbare Beiträge in den Bereichen Design-Artefakt, Design-Grundlagen und / oder Design-Methoden liefern.
V	Forschungsgenauigkeit	Design-Science Research beruht auf der Anwendung von strengen Methoden bei der Konstruktion und Bewertung des Design-Artefakts.
VI	Design als Suchprozess	Die Suche nach einem effektiven Artefakt erfordert die Verwendung verfügbarer Mittel, um gewünschte Ziele zu erreichen, während die Gesetze in der Problemumgebung erfüllt werden.
VII	Kommunikation der Forschungsergebnisse	Design-Science Research muss sowohl technologieorientierten als auch managementorientierten Zielgruppen effektiv präsentiert werden.

Tabelle A-1: DSRM-Richtlinien [312]

[312] Übersetzt aus: Hevner et al. 2004, S. 83.

Kategorie	Monolithisches System	Microservice-basiertes System
(1) Komplexität	- Zentralisiert - In den monolithischen Strukturen	- Verteilt - Auf Ebene der Komposition einzelner Microservices
(2) Quellcode	- Eine gemeinsame, große Codebasis für die gesamte Anwendung	- Jeder Microservice besitzt seine eigene, kleine Codebasis
(3) Verständnis Wartbarkeit	- Schwer verständlich - Meist schlecht wartbar	- Aufgrund der Größe einzelner Microservices besser verständlich und wartbar
(4) Deployment	- Komplex und selten - Mit hohem Risiko verbunden - Nur als „Ganzes“ bereitstellbar - Erfordert Wartungsfenster und Ausfallzeiten für gesamte Anwendung	- Einfach und häufig - Risikoarm - Jeder Microservice kann unabhängig bereitgestellt werden - Verursacht nur minimale oder keine Ausfallzeiten für einzelne Microservices
(5) Technologie und Sprache	- Meist nur in einer Programmiersprache implementiert - Stark an verwendete Technologie gebunden	- Freie Technologie- und Sprachwahl für jeden Microservice - Technologien können leichter ausgetauscht werden
(6) Skalierbarkeit	- Die gesamte Anwendung muss skaliert werden	- Services mit Skalierungsbedarf können einzeln skaliert werden
(7) Kommunikation	- Über direkte Methodenaufrufe	- Über wohldefinierte Schnittstellen
(8) Datenhaltung	- Meist eine große, zentrale Datenbank	- Dezentralisierte Datenhaltung - Jeder Microservice besitzt seine eigene Datenbank
(9) Cloud-Kompatibilität	- Aufgrund der Größe und Struktur nur bedingt kompatibel	- Ansatz profitiert stark von Cloud-Technologien
(10) Organisation	- Wenige große Teams - Teile der Anwendung überlagern sich - Hoher Aufwand für Kommunikation und Koordination	- Viele kleine Microservice-Teams - Keine/wenig überlappende Bereiche der Anwendung - Minimierter Aufwand für Kommunikation und Koordination

Tabelle A-2: Abgrenzung von Monolithen und Microservices

#	Autor (Jahr)	Titel	Inhaltlicher Schwerpunkt	Relevanz
1	Di Francesco et al. 2018	Migrating Towards Microservice Architectures: An Industrial Survey	Aktivitäten und Herausforderungen bei einer Migration	✓
2	Balalaie et al. 2018	Microservices migration patterns	In Migrationsprojekten anwendbare Muster und Prinzipien	✓
3	Joselyne et al. 2017	A framework to Modernize SME Application in Emerging Economies: Microservice Architecture Pattern Approach	Aggregation fachlicher und technischer Aktivitäten in einem Rahmenwerk	✓
4	Taibi et al. 2017	Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation	Gründe für Migrationen; Praxiserfahrungen zum Migrationsprozess	✓
5	Razavian et al. 2010	Towards a Conceptual Framework for Legacy to SOA Migration	Konzeptionelles Rahmenwerk für Migrationen im SOA-Kontext	✓
6	Kalske et al. 2018	Challenges When Moving from Monolith to Microservice Architecture	Technische und organisatorische Herausforderungen	–
7	Knoche et al. 2018	Using Microservices for Legacy Software Modernization	Serviceorientierte Plattform-Migration von Cobol in Java	–
8	Bucchiarone et al. 2018	From Monolithic to Microservices: An Experience Report from the Banking Domain	Technische Architektur; Erfahrungen im Migrationsprozess	–
9	Gouigoux et al. 2017	From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture	Erkenntnisse hinsichtlich Granularität, Deployment und Orchestrierung	–
10	Fan et al. 2017	Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report	Migrationsprozess basierend auf einem vorgeschlagenen Entwicklungsprozess	–
11	Zirkelbach et al. 2018	On the Modernization of ExplorViz towards a Microservice Architecture	Verwendete Protokolle und Technologien; technische Architektur	×
12	Furda et al. 2018	Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency	Handhabung drei typischer datenspezifischer Herausforderungen bei Migrationen	×
13	Sabiri et al. 2017	A Legacy Application Meta-model for Modernization	Cloud-Migration, technische Architektur, Metamodell	×
14	Krämer et al. 2017	From Monolithic to Modern Software Development	Migration durch Nutzung von Containern, Microservices und Cloud Computing	×
15	Dragoni et al. 2017	Microservices: Migration of a Mission Critical System	Neuentwicklung eines Monolithen unter dem Aspekt der verbesserten Skalierbarkeit	×

16	Balalaie et al. 2016	Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture	Transformation der technischen Architektur	×
17	Balalaie et al. 2016	Migrating to Cloud-Native Architectures Using Microservices: An Experience Report.	Technische Architektur; Neuentwicklung	×
Legende: ✓ relevant; – bedingt relevant; × nicht relevant				

Tabelle A-3: Rechercheergebnisse (RF1)

#	Autor (Jahr)	Titel	Inhaltlicher Schwerpunkt	Relevanz
(1) Domäne / Geschäft				
1	Steinegger et al. 2017	Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications	Fallstudie zur Anwendung von DDD für den Entwurf einer Microservice-basierten Anwendung	–
2	Hippchen et al. 2017	Designing Microservice-Based Applications by Using a Domain-Driven Design Approach	Erweiterung von [1]	–
3	Frey et al. 2015	Capability-based Service Identification in Service-Oriented Legacy Modernization	Business Capabilities; Enterprise-Architektur; SOA	–
4	Rademacher et al. 2018	Towards a UML Profile for Domain-Driven Design of Microservice Architectures	DDD basiertes UML-Profil zur Validierung von Domänen-modellen und Implementierung von Microservices	×
5	Diepenbrock et al. 2017	An Ontology-based Approach for Domain-driven Design of Microservice Architectures	Ontologie-basiertes Metamodell für die semantische Modellierung von Microservices mit DDD	×
(2) Quellcode / Daten				
5	Götz et al. 2018	Challenges of production microservices	Identifikation durch Analyse von Datenstrukturen und Verhalten in Bezug auf Geschäftsprozesse	×
6	Chen et al. 2017	From Monolith to Microservices: A Dataflow-Driven Approach	Algorithmus zur Identifizierung von Microservices auf Basis von Datenflussdiagrammen	×
7	Mazlami et al. 2017	Extraction of Microservices from Monolithic Software Architectures	Graphen-basiertes Clustering basierend auf Metadaten der Versionskontrolle	×
8	Escobar et al. 2016	Towards the understanding and evolution of monolithic applications as microservices	Code-Analyse mit Java Annotationen	×

9	Levcovitz et al. 2016	Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems	Dekomposition auf Basis eines Abhängigkeitsgraphen von Modulen und Datenbanktabellen	×
10	Procaccianti et. al. 2016	Towards a MicroServices Architecture for Clouds	Daten- und Prozess-getriebener Algorithmus	×
(3) Schnittstellen (APIs)				
11	Baresi et. al. 2017	Microservices Identification Through Interface Analysis	Clustering von API-Spezifikationen	×
(4) Sonstige				
12	Amiri 2018	Object-aware Identification of Microservices	Dekomposition durch Clustering auf Basis der Abhängigkeiten von Prozessaktivitäten und Datenobjekten	×
13	Mustafa et. al. 2017	GranMicro: A Black-Box Based Approach for Optimizing Microservices Based App's	Analyse von Web Access Logs	×
14	Hassan et. al. 2017	Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity	Dynamische Anpassung von Servicegrenzen zur Laufzeit durch Nutzung einer Metasprache	×
15	Klock et. al. 2017	Workload-based Clustering of Coherent Feature Sets in Microservice Architectures	Algorithmus für optimiertes Deployment und Granularität basierend auf Workload-Modell	×
16	Zdun et al. 2017	Ensuring and Assessing Architecture Conformance to Microservice Decomposition Patterns	Metriken zur Messung der Konformität von Microservices und gängigen Dekompositions-Mustern	×
17	Kecskemeti et al. 2017	Towards a Methodology to Form Microservices from Monolithic Ones	Dekomposition monolithischer SOA-Services in Microservices durch einen VM-Abbild-Syntheseansatz	×
18	Kecskemeti et al. 2016	The ENTICE approach to decompose monolithic services into microservices	Dekomposition monolithischer SOA-Services in Microservices durch einen VM-Abbild-Syntheseansatz	×
19	Ahmadvand et. al. 2016	Requirements Reconciliation for Scalable and Secure Microservice (De)composition	Dekomposition auf Basis von Anforderungen an Skalierbarkeit und Sicherheit	×
20	Gysel et. al. 2016	Service Cutter: A systematic approach to service decomposition	Graphen-basierter Algorithmus auf Basis eines Kataloges verschiedener Kopplungs-Kriterien	×
Legende: ✓ relevant; – bedingt relevant; × nicht relevant				

Tabelle A-4: Rechercheergebnisse (RF2)