Karlsruher Institut für Technologie

Iterative Migration monolithischer Softwaresysteme in eine domänengetriebene Microservice-Architektur

zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Benjamin Hippchen

aus Sankt Wendel

Tag der mündlichen Prüfung:	22.02.2021
Erster Gutachter:	Prof. Dr. Sebastian Abeck
Zweiter Gutachter:	Prof. Dr. Jürgen Beyerer

Ehrenwörtliche Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Karlsruhe, den 21. Dezember 2020

_____

Benjamin Hippchen

Danksagung

Zunächst möchte ich gerne die ersten Abschnitte meiner Dissertation nutzen, um mich bei all denjenigen zu bedanken, die mich während der langen Zeit der Promotion begleitet und unterstützt haben. Ohne meine Familie, Freunde und Kollegen wäre ein erfolgreicher Abschluss der Promotion nicht möglich gewesen.

An erster Stelle möchte ich mich bei Herrn Prof. Dr. Sebastian Abeck bedanken, der mir das Vorhaben einer Promotion ermöglichte. Er brachte mir ein hohes Maß an Vertrauen entgegen und war mir stets ein außerordentlich guter Doktorvater, der mit hilfreichen Anregungen und konstruktiver Kritik meine Forschungstätigkeit unterstützte. Für das entgegengebrachte Vertrauen und alle Bemühungen möchte ich mich herzlich bei Herrn Prof. Dr. Sebastian Abeck bedanken. Ein weiterer Dank gilt Herrn Prof. Dr. Jürgen Beyerer, der sich bereit erklärte, das Ko-Referat meiner Dissertation zu übernehmen.

Weiterhin möchte ich mich bei meinen Kollegen Michael Schneider, Stefan Throner, Eric Braun und Jannik Sidler für zahlreiche Diskussionen und ein angenehmes Arbeitsklima in und außerhalb der Forschungsgruppe bedanken. Auch das gemeinsame Verfassen von Veröffentlichungen hat mir viel Freude bereitet. Ich möchte an dieser Stelle festhalten, dass ich die vier nicht nur als Kollegen, sondern auch als Freunde schätze.

Ebenfalls möchte ich mich bei der Evana AG bedanken, die mir als Industriepartner die Möglichkeit zur Validierung meiner Forschungsbeiträge gegeben hat. Ohne das Zutun der Evana AG wäre eine qualitative Validierung nicht möglich gewesen.

Eine große Stütze während der Durchführung der Promotion waren meine Freunde, die stets an meiner Seite waren und mich motiviert haben. Insbesondere möchte ich mich da bei Sven Thewes, Dennis Schlehuber, Benjamin Rupp, Alexander Warken, Anke Streßer, Lukas Reckel, Patrick und Rubina Kapahnke und Roland Steinegger bedanken. Besonders hervorheben möchte ich Sarah und Christian Schöler, die mühevoll meine Dissertation Korrektur gelesen haben.

Eine weitere wichtige Person in meinem Leben stellt Dr. Pascal Giessler dar, der stets als sehr guter Freund und Mentor mich in meinem wissenschaftlichen Vorhaben bestärkt und vorangetrieben hat. Für sein Engagement möchte ich mich bedanken.

Nicht zu vergessen ist meine Familie, die mir durch ihre Unterstützung den Freiraum geschaffen haben, neben der Ausübung meines Berufs die Promotion erfolgreich abzuschließen. Gerade meinen Schwiegereltern Brigitte und Harald Kuhn habe ich sehr viel zu verdanken. Weiterhin möchte ich mich

bei meinen Eltern Markus und Susanne Hippchen und meinen Geschwistern Stefan Hippchen und Emilie Hippchen für ihre Unterstützung bedanken. Meine unendliche Dankbarkeit gilt meiner Ehefrau Laura Hippchen. Über den gesamten Zeitraum der Promotion nahm sie sich zurück und stellte meine Interessen über ihre. Zudem stand sie mir in guten wie in schlechten Zeiten zur Seite und fand stets die richtigen Worte, um mich wieder auf den richtigen Weg zu bringen. Im letzten Jahr meiner Promotion schenkte sie mir zudem einen gesunden Sohn. Trotz dieser neuen und herausfordernden Aufgabe schaffte sie es, mir den Freiraum für das Verfassen der Dissertation zu schaffen. Darüber hinaus las sie meine Dissertation mehrfach Korrektur und sorgte damit für eine hohe sprachliche Qualität. Keine Worte beschreiben meine Dankbarkeit und Hochachtung gegenüber meiner Ehefrau.

Zum Schluss möchte ich gerne noch zwei wichtigen Menschen in meinem Leben diese Dissertation widmen. Mein liebster Sohn Lasse, ich bin überglücklich dich seit dem Jahr 2020 in meinem Leben haben zu dürfen. Tag für Tag hast du mir gezeigt, was wichtig im Leben ist und mir unendliche Freude bereitet. Selbst an schlechten Tagen konntest du mir ein Lachen entlocken. Ich bin dankbar, einen tollen Sohn wie dich haben zu dürfen. Des Weiteren widme ich die Dissertation meinem Patenonkel Michael Hippchen, der immer eine wichtige Bezugsperson für mich darstellte. Leider weiß ich zum Zeitpunkt des Verfassens dieser Widmung, dass du den langen und harten Kampf gegen deine Krankheit verloren hast. Trotz deines eigenen Kampfes hast du jederzeit ein Ohr für mich gehabt und warst für mich da. Es bereitet mir große Schmerzen zu wissen, dass du deinem Wunsch, der Teilnahme an meiner Promotionsprüfung, nicht mehr nachkommen kannst. Dennoch weiß ich, dass du immer bei mir sein wirst und doch ist die einzige Möglichkeit, mich noch bei dir zu bedanken das Verfassen dieser Widmung. Du warst ein wunderbarer Mensch, den ich mein Leben lang vermissen werde. Danke, dass ich dich in meinem Leben haben durfte.

Inhaltsverzeichnis

1 Einleitung

Mit diesem Kapitel und den nachfolgenden Abschnitten wird die hier vorliegende Forschungsarbeit grundlegend betrachtet. Zunächst erfolgt mit Abschnitt 1.1 die Betrachtung der Motivation zur Durchführung der Forschungsarbeit. In Abschnitt 1.2 wird die Forschungsarbeit in bestehende Themenbereiche der Informatik eingeordnet. Zur Veranschaulichung der Problemstellungen und Forschungsbeiträge wird in Abschnitt 1.3 ein beispielhaftes Szenario vorgestellt. Die in dem Szenario aufgezeigten Problemstellungen werden in Abschnitt 1.4 detailliert beschrieben. Darauf aufbauend thematisiert der Abschnitt 1.5 die Zielsetzung der Forschungsarbeit und die Forschungsbeiträge, die dem Stand der Forschung hinzugefügt werden sollen. Um den Kontext der Forschungsarbeit zu fassen, befasst sich Abschnitt 1.6 mit Prämissen, welche an die Ergebnisse gestellt werden. Abschließend für dieses Kapitel zeigt Abschnitt 1.7 den allgemeinen Aufbau der Forschungsarbeit.

1.1 Motivation

In Zeiten der Digitalisierung ist der Erfolg von Organisationen maßgeblich von deren digitaler Reife abhängig. Je weiter die Organisation in der digitalen Evolution vorangeschritten ist, desto besser kann sie sich auf dem Markt behaupten. Ein wichtiger Bestandteil dieses evolutionären Prozesses ist das Überdenken bestehender Geschäftszweige, da mit der Digitalisierung Organisationen zwangsläufig globaler agieren müssen, ohne den eigenen Standort zu verändern oder auszuweiten.

Der Einsatz von Softwaresystemen spielt bei der Digitalisierung eine zentrale Rolle. Hinter einem Softwaresystem verbirgt sich neben der Software auch die IT-Infrastruktur mit ihren physischen und mittlerweile verstärkt virtuellen Komponenten. Unabdingbar für die Digitalisierung ist dabei, sondern auch die bestehenden Softwaresysteme modernisiert werden. Wichtig ist die Berücksichtigung der Kundenbedürfnisse, denn nur dann kann der Erfolg der geplanten Digitalisierung gewährleistet werden.

Angetrieben wird die Digitalisierung in Deutschland von der immer stärker wachsenden Szene an Tech-Startups. Diese Form der Organisation zielt darauf ab, bestehende Branchen einer digitalen Revolution zu unterziehen. Hier werden alle Dienstleistungen werden den Kunden durch Software in Form von modernen Web-Anwendungen zur Verfügung gestellt. Dabei entscheidet die Umsetzung der Software über den Erfolg der gesamten Organisation. Erfüllt die Software nicht die Anforderungen der Endkunden, so ist sie nicht zielführend und generiert keinen Mehrwert für das Unternehmen

[Be12]. Gerade die Anforderungen an das Softwaresystem können sich im Rahmen der Digitalisierung schnell verändern. In diesem Punkt haben die Tech-Startups einen Vorteil gegenüber den traditionellen Organisationen. Ein wesentliches Merkmal eines solchen Startups ist die agile organisatorische Struktur, die einen schnellen Wandel auf neue Anforderungen zulässt. Zudem sind finanzielle Mittel oft kein Problem, da Investoren Unternehmensanteile zu hohen Summen erwerben. So können Einnahmeverluste durch die Anpassung an neue Anforderungen einfach überbrückt werden. Dagegen bringen diese Beteiligungen jedoch auch Nachteile mit sich, da die Investoren an einem schnellen Wachstum des Unternehmenswerts interessiert sind. Entsprechend werden Ziele in einem engen zeitlichen Rahmen gesetzt. Dies hat zur Folge, dass zur Erreichung der Ziele Kompensationen notwendig sind.

Meist kommen diese Kompensationen während der Softwareentwicklung zum Tragen, so dass technische Schulden innerhalb der Software entstehen [KN+12]. Diese Schulden beeinflussen die Qualität und rufen damit Defizite in der Architektur und Anforderungsumsetzung auf [Ch05]. In Bezug auf die Softwareentwicklung geht gerade durch architekturelle Probleme die Agilität verloren, die durch die Digitalisierung gefordert wird. Das Reagieren auf neue oder veränderte Anforderungen wird mit Voranschreiten des Softwaresystems zur Herausforderung. Es ist daher naheliegend, den technischen Schulden auf Ebene der Architektur entgegenzuwirken. Hierfür etabliert haben sich Cloud-basierte Architekturen [BH+16]. Ein hervorstechender Architekturstil der Cloud-basierten Architekturen ist die *Microservice-Architektur*, die auch gerade im Kontext der Digitalisierung an Beliebtheit gewinnt [GG+16].

Die auswirkungsvollsten technischen Schulden eines Softwaresystems sind die falsche Umsetzung von Anforderungen und die Umsetzung von falschen Anforderungen [Gl02]. Beide Probleme lassen sich auf das fehlende Wissen und Verständnis des Entwicklungsteams über die Geschäftsabläufe der beauftragenden Organisation zurückführen [Ev04]. Davon betroffen sind alle Phasen eines klassischen Softwareentwicklungsprozesses, die laut Bruegge und Dutoit die Analyse, der Entwurf und die Implementierung sind. Geschäftsabläufe und die zu ihrer Durchführung benötigten Geschäftsobjekte beschreiben die Struktur und Geschäftsfähigkeiten der Organisationen, die auch unter dem Begriff der Kundendomäne (engl. customer domain)–oder auch Geschäftsdomäne–zusammengefasst werden. Dabei unterscheiden sich Organisationen in der Anzahl und im Aufbau der Geschäftsprozesse, auch wenn diese Organisationen dieselbe Branche bedienen. Zudem begrenzt eine Geschäftsdomäne die Gültigkeit der darin befindlichen Geschäftsobjekte [Ve13]. So bezeichnet ein *Objekt* im Immobilienwesen eine beliebige Art eines Gebäudes. Dagegen wird unter *Objekt* in der Softwareentwicklung die Instanziierung einer Klasse verstanden. Aus diesem Grund ist es wichtig, die Interpretation des Geschäftsobjekts durch eine Geschäftsdomäne einzuschränken.

Eine Geschäftsdomäne kann hinreichend komplex sein. Gerade komplexe Geschäftsdomänen müssen für eine erfolgreiche Softwareentwicklung bewältigt und verstanden werden [Ve13]. Eine Möglichkeit, dies zu gewährleisten, stellt der Softwareentwicklungsansatz des *domänengetriebenen Entwurfs*

(engl. Domain-Driven Design, DDD) dar. Im Kontext dieses Ansatzes werden alle Geschäftsprozesse, Geschäftsobjekte oder sonstigen Informationen innerhalb der Geschäftsdomäne als *Domänenwissen* oder auch als Domänenkonzepte bezeichnet. Quellen, die für Domänenwissen herangezogen werden, sind im Kontext von DDD als Domänenexperten bekannt [Ev04]. Neben Mitarbeitern der Organisation sind auch sonstige Medien wie beispielsweise Dokumentationen als Domänenexperten anzusehen.

Das Ziel von DDD ist es, das Domänenwissen direkt von den Domänenexperten zu extrahieren, in ein Domänenmodell zu überführen und dieses als Grundlage zum Entwurf des Softwaresystems heranzuziehen. Eine wichtige Vorarbeit für DDD ist jedoch die Betrachtung der Anforderungen an das Softwaresystem. Festzuhalten ist dabei, dass DDD nicht für die Anforderungserhebung selbst genutzt wird. Innerhalb des Domänenmodells befindet sich nur das Wissen der Domäne, welches aus den Anforderungen hervorgeht. Zur Erhebung muss auf andere Ansätze wie die *verhaltensgetriebene Entwicklung* (engl. Behavior-Driven Development, BDD) zurückgegriffen werden [Sm14]. BDD setzt an den Anforderungen des Softwaresystems an und dient als Informationsquelle für die Domänenmodellierung mit DDD.

Der in den letzten Jahren steigende Trend, das eigene Softwaresystem mit dem Architekturstil der Microservice-Architektur zu gestalten, wird häufig in Verbindung mit DDD gesehen [Ne15]. Denn zentrale Konzepte von DDD harmonieren mit den grundlegenden Charakteristiken einer Microservice-Architektur. Zu diesen Prinzipien gehören eine hohe Autonomie, einfache Web-Schnittstellen, Datenhoheit und Flexibilität bei der Wartung von den Softwarebausteinen, den Microservices [FL14]. Bei dem Entwurf und der Implementierung eines Softwaresystems hilft eine Microservice-Architektur zudem dabei, das Einführen von technischen Schulden, wie sie bereits angesprochen wurden, zu vermeiden. So wird beispielsweise das Prinzip *Separation of Concerns* forciert, was bei einem monolithischen Architekturstil nicht gegeben ist. Problematisch ist, dass der monolithische Architekturstil meist zu Beginn einer Softwareentwicklung gewählt wird [Fo15] und durch die Anhäufung der technischen Schulden ein Zwang zur Modernisierung besteht. Die Modernisierung erfolgt meist im Rahmen einer Migration in eine Microservice-Architektur [Ot15, Iz16, Hm19].

Das Durchführen einer solchen Migration ist jedoch kein triviales Vorhaben. Es bedingt einen hohen zeitlichen und finanziellen Aufwand. Zudem stellen sich verschiedene Herausforderungen, die es aufzulösen gilt [KM+17]. Eine wissenschaftliche Betrachtung der Migration ist aus diesen Gründen und aufgrund der hohen Relevanz für die Industrie wichtig. Der Forschungsgegenstand der iterativen Migration von monolithischen Softwaresystemen in eine Microservices-Architektur ist bereits durch verschiedene Arbeiten betrachtet worden. Ein wesentlicher Schritt bei der Bereitstellung eines ganzheitlichen *Migrationsrahmenwerks*, nach der Definition von [TL+17], ist nicht abschließend betrachtet worden. Bestehende Arbeiten thematisieren nur Teilbereiche der iterativen Migration, sodass keine *Systematik* erkennbar ist. Mit der hier vorliegenden Forschungsarbeit werden Beiträge zur Bereitstellung eines ganzheitlichen Migrationsrahmenwerks für eine iterative Migration monolithischer Softwaresysteme in eine Microservice-Architektur getätigt. Fokussiert werden Aspekte wie

strukturelle Grundlagen einer iterativen Migration, Migrationsaktivitäten zur Bereitstellung einer ganzheitlichen Systematik, formale Softwareartefakte für die Nachvollziehbarkeit der iterativen Migration, Ableiten eines domänengetriebenen Entwurfs für die Microservice-Architektur, Reorganisation des Entwicklungsteams und Inbetriebnahme der Microservices.

1.2 Einordnung der Arbeit

Alle Forschungsbeiträge der hier vorliegenden Arbeit lassen sich in den Bereich der Softwaretechnik (engl. software engineering) einordnen. Unter dem Begriff der Softwaretechnik werden mehrere Wissensgebiete zusammengefasst, die über das Projekt *Engineering Body of Knowledge* der Organisationen IEEE und ACM definiert werden [BF+14]. Aus der Softwaretechnik soll ein ingenieurmäßiges Verhalten während der Softwareentwicklung resultieren, um ein hohes Maß an Qualität des Softwaresystems zu gewährleisten [Pr05]. Die Wissensgebiete (1) Anforderungsanalyse (engl. software requirements), (2) Softwareentwurf (engl. software design) und (3) Entwicklungswerkzeuge und -methoden (engl. software engineering tools and methods) treffen auf diese Arbeit zu.

Die drei oben genannten Wissensgebiete werden im Querschnitt durch den im Vordergrund stehenden Entwurf eines Migrationsrahmenwerks betrachtet. Bei der Modernisierung eines Monolithen ist die Wiederherstellung der Anforderungsspezifikation mit den funktionalen Anforderungen relevant. Nur so kann das migrierte Microservice-basierte Softwaresystem den Monolithen letztendlich ersetzen. Für die Wiederherstellung der Anforderungsspezifikation wird auf eine Anforderungsanalyse Bezug genommen. Mit der Anforderungsanalyse werden funktionale und nicht-funktionale Anforderungen des Auftraggebers ermittelt [KO+06, CN+12]. Die Aufgaben beziehen sich auf die Anforderungserhebung (engl. requirements elicitation), Anforderungsanalyse (engl. requirements analysis), Anforderungsspezifikation (engl. requirements specification) und Anforderungsbewertung (engl. requirements validation).

Nach Wiederherstellung der Anforderungsspezifikation gilt es, den Entwurf der Microservice-Architektur zu erstellen. Zentral für den Softwareentwurf ist die Softwarearchitektur. Wie Entwurfsentscheidungen der Softwarearchitektur dargestellt werden, ist von hoher Relevanz, denn das Wissen muss an die Interessenvertreter vermittelt werden. Die primär in dieser Forschungsarbeit betrachteten Softwarearchitekturen sind zum einen der monolithische Architekturstil und die Microservice-Architektur. Die Microservice-Architektur beschreibt in dem Kontext der Softwaresystem-Struktur die Unterteilung des Softwaresystems in Softwarebausteine [Ne15]. Primäre Charakteristiken der Microservices, die Softwarebausteine der Microservice-Architektur, sind Abgeschlossenheit, Unabhängigkeit und leichtgewichtige Kommunikation.

Weiterhin erfolgt in verschiedenen Migrationsaktivitäten des Migrationsrahmenwerks eine Wiederherstellung der Architekturbeschreibung, die auch die Anforderungsspezifikation inkludiert. Hierfür

werden formale Modelle eingeführt. Der Einsatz von formalen Modellen im Rahmen dieser Arbeit dient nicht zum Einsatz der modellgetriebenen Entwicklung (engl. Model-Driven Development, MDD). Charakteristisch für MDD ist das Platzieren des gesamten Wissens in Modellen, um automatisch weitere Artefakte wie bspw. Quellcode oder Textdateien zu generieren [Se03]. Zwar verfolgt die hier angestrebte Methodik den Einsatz von Modellen zum Vermitteln von Wissen, aber sie setzt im Gegensatz zu MDD nicht auf die automatisierte Generierung der Artefakte.

1.3 Betrachtetes Szenario

Zur besseren Darstellung der in dieser Forschungsarbeit behandelten Problemstellungen und verfolgten wissenschaftlichen Fragestellungen wird in dem nachfolgenden Abschnitt ein Migrationsszenario beschrieben. Das Szenario soll beispielhaft die vorliegenden Rahmenbedingungen einer Organisation und die typischen Migrationsaktivitäten einer iterativen Migration darstellen. Abbildung 1.1 zeigt den Aufbau dieses Szenarios. Zunächst werden die vorliegenden Rahmenbedingungen als Ausgangssituation beschrieben. Anschließend erfolgt die Darstellung der Durchführung der iterativen Migration.

1.3.1 Ausgangssituation

Ausgegangen wird bei dem Szenario von einer Organisation, welche Gewinn durch die Bereitstellung einer Software-as-a-Service-Lösung (SaaS) erzielt. Die Organisation selbst verfügt über ein Entwicklungsteam, das für die Entwicklung und den Betrieb des Softwaresystems verantwortlich ist. Das Softwaresystem wird Kunden einer spezifischen Branche zur Verfügung gestellt. Aus der Spezialisierung auf eine Branche resultiert eine Organisation mit einem branchenorientierten Domänenwissen. Hierbei bezeichnet das Domänenwissen ein umfangreiches Verständnis über die in der Branche notwendigen Geschäftsfähigkeiten und Geschäftsprozesse. Das Entwicklungsteam ist im klassischen Sinne aufgebaut und besteht aus Frontend-, Backend- und Operations-Team. Organisiert ist das Entwicklungsteam jedoch als ein großes Entwicklungsteam; man spricht auch hier von einer monolithischen Teamstruktur [HS17].

Das entwickelte Softwaresystem basiert auf Kundenbedürfnissen der Organisation. Aus den Kundenbedürfnissen werden die funktionalen Anforderungen abgeleitet, die sich auf das Domänenwissen beziehen. Funktionale Anforderungen beschreiben dabei das gewünschte Verhalten des Softwaresystems [Gl07, CN+12]. Die Anforderungen werden von dem Entwicklungsteam entsprechend in einem Softwaresystem implementiert. Geschuldet der Zeit und Erfahrung des Teams erfolgt initial nur eine kurze Entwurfsphase, die zudem durch wenige Dokumentationsartefakte beschrieben ist. Insbesondere werden nicht-funktionale Anforderungen an das Softwaresystem vernachlässigt. Letztendlich entsteht

ein Softwaresystem, welches eine logische und physische monolithische Softwarearchitektur aufweist. Bei der Softwareentwicklung wurde auf das Paradigma der objektorientierten Programmierung (OOP) gesetzt, sodass sich Pakete und Klassen im Quellcode wiederfinden. Aus diesem Grund kann auch der Monolith als unzureichend modularisiert bezeichnet werden (vgl. Abschnitt 2.2.3).

Durch das immer weiter wachsende monolithische Softwaresystem wird die Effektivität und Effizienz des Entwicklungsteams beeinträchtigt. Anpassungen und Weiterentwicklungen sind mit einem hohen zeitlichen und finanziellen Aufwand verbunden. Um der Beeinträchtigung durch die monolithische Architektur entgegenzuwirken, wird eine Migration des Softwaresystems beschlossen und fortan als *Migrationsvorhaben* angesehen. Das Migrationsvorhaben wird ebenfalls durch die Organisation (im Speziellen die Führungsebene) unterstützt.

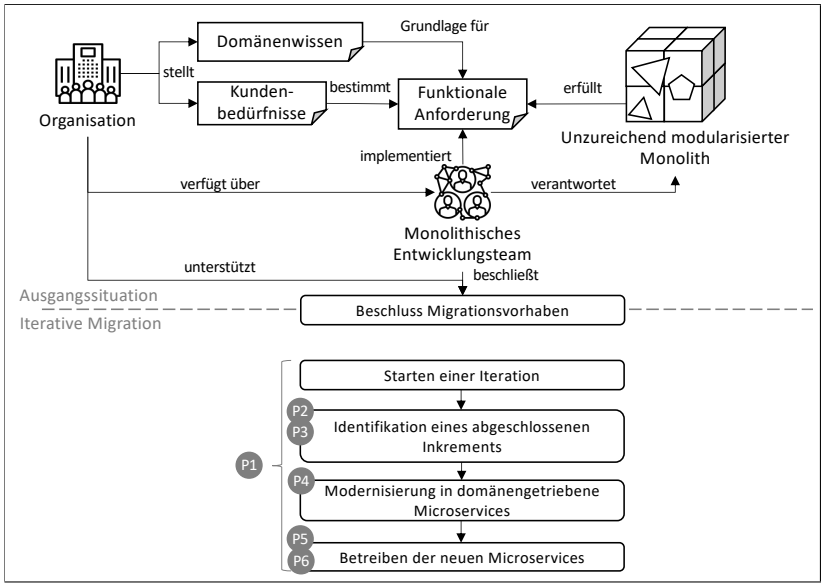

Abbildung 1.1: Aufbau des betrachteten Szenarios

1.3.2 Durchführung des Migrationsvorhabens

Für eine erfolgreiche Transition von der Ausgangssituation zur wirklichen iterativen Migration ist ein Beschluss des Entwicklungsteams und die Unterstützung der Organisation notwendig. Ist eine der beiden Voraussetzungen zu Beginn nicht gegeben, kann zwar ein Migrationsvorhaben durchgeführt werden, aber eine Erreichung des Ziels ist unwahrscheinlich. Denn eine iterative Migration bindet sowohl finanzielle als auch personelle Ressourcen, welche für die Wartung des bestehenden Monolithen wegfallen. Der prozessuale Ablauf der in dem Szenario dargestellten iterativen Migration richtet sich nach den in bestehenden Arbeiten vorzufindenden Abläufen [BH+16, HS17, KH18, BH+18, Ne19, HR20].

Zur Durchführung der iterativen Migration startet das monolithische Entwicklungsteam zunächst eine Iteration. Eine Iteration beruht, neben dem allgemeinen Beschluss des Migrationsvorhaben, dabei auf einem gewissen *Auslöser*. Bei der Modernisierung des Softwaresystems müssen folgende Facetten jenes Softwaresystems betrachtet werden [BB+13]: (1) technische Aspekte; (2) geschäftliche Aspekte. Die technischen Aspekte geben Aufschluss über die Softwarearchitektur des Monolithen. Unter den geschäftlichen Aspekten versteht man die funktionalen Anforderungen des Softwaresystems.

Beide Aspekte werden innerhalb einer Iteration für ein abgeschlossenes Inkrement detailliert betrachtet. Ein Inkrement bezeichnet dabei einen *Ausschnitt des Monolithen*. Damit das Inkrement zu einer Microservice-Architektur modernisiert werden kann, müssen die durch das Inkrement betrachteten Softwarebausteine die von den Microservices geforderte Autonomie ermöglichen. Hierfür muss das Entwicklungsteam kohäsive Softwarebausteine wie beispielsweise Klassen bestimmen. Die Architekturbeschreibung gewinnt an dieser Stelle an besonderer Bedeutung, da sie als Quelle für alle notwendigen Informationen dient. Für Softwaresysteme, die unter Zeitdruck entstanden sind, ist eine Vernachlässigung der Dokumentation jedoch typisch [MB+15]; dies gilt auch für das in dem Szenario betrachtete Softwaresystem. Existiert keine aussagekräftige Architekturbeschreibung, muss diese zunächst wiederhergestellt werden. Eine Wiederherstellung der Architekturbeschreibung kann anhand des *Design Recoverys* erfolgen [CC90].

Wurde ein Inkrement durch das Entwicklungsteam identifiziert, kann die Modernisierung in Microservices erfolgen. Für die Modernisierung wird ein domänengetriebener Entwurf der Microservice-Architektur verfolgt, um später eine Wiederverwendbarkeit und hohe Wartbarkeit zu gewährleisten [Ne15, NM+16]. Die Geschäftsdomäne entscheidet über die Kohäsion des Domänenwissens, welches in den Microservices implementiert wird. Durch die Beantwortung der Frage zur Kohäsion des Domänenwissens ist das Entwicklungsteam zudem in der Lage, den Umfang eines Microservices zu bestimmen.

Nach einer erfolgten Implementierung der Microservices müssen diese in Betrieb genommen werden. Dabei wird das Entwicklungsteam durch den neuen Architekturstil mit einer neuen Komplexität konfrontiert. Während der Monolith, der auch nach der Iteration weiterhin verfügbar sein muss, aus wenigen eigenständigen Softwarebausteinen besteht, entstehen durch die Microservice-Architektur deutlich mehr eigenständige Softwarebausteine. Der neuen Komplexität muss durch geeignete Konzepte entgegengewirkt werden, sodass der Betrieb beherrschbar bleibt.

1.4 Problemstellungen

Nachfolgend werden Problemstellungen anhand der im Szenario durchgeführten iterativen Migration identifiziert und näher beleuchtet. Sie bilden die Grundlage für die Festlegung der Zielsetzung und der darauf aufbauenden Forschungsbeiträge der vorliegenden Forschungsarbeit. Die jeweiligen

Problemstellungen wurden in der Abbildung 1.1 an den auftretenden Stellen der iterativen Migration gekennzeichnet.

1.4.1 P1 - Fehlende Systematik und Nachvollziehbarkeit

Diese Problemstellung betrifft die iterative Migration ganzheitlich. In dem betrachteten Szenario durchläuft das Entwicklungsteam offensichtliche, durch bestehende Arbeiten identifizierte Migrationsaktivitäten, die bei einem iterativen Vorgehen notwendig sind. Die dargestellten Migrationsaktivitäten befinden sich jedoch auf einem hohen Abstraktionsniveau, sodass für die eigentliche Durchführung einer solchen Aktivität mehrere Subaktivitäten notwendig sind. Welche Subaktivitäten notwendig sind, ist für das Entwicklungsteam nicht eindeutig definiert. Folglich ist ein Experimentieren notwendig, was wiederum finanzielle und zeitliche Ressourcen bindet. Aus diesem Grund bezieht sich die erste Problemstellung auf die fehlende *Systematik* und *Nachvollziehbarkeit* bei der iterativen Migration eines monolithischen Softwaresystems in eine Microservice-Architektur. Eine Systematik deckt dabei alle notwendigen Migrationsaktivitäten auf einem angemessenen Abstraktionsniveau ab, damit das Entwicklungsteam eine klare Führung durch eine Iteration erhält. Zu klären ist dabei, welche Migrationsaktivitäten notwendig sind, damit eine Iteration gestartet, ein Inkrement identifiziert und modernisiert wird und damit die daraus resultierenden Microservices betrieben werden. Zur Gewährleistung der Nachvollziehbarkeit sind formale Softwareartefakte einzuführen und mit den Migrationsaktivitäten zu verknüpfen. So können nachvollziehbar Entscheidungen durch das Entwicklungsteam getroffen werden.

1.4.2 P2 - Festlegen des Umfangs eines Inkrements

Das iterative Vorgehen sieht die Definition eines Inkrements pro durchlaufener Iteration vor. Ein Inkrement, so Kruchten [Kr04], beschreibt ein in sich abgeschlossenes *Release* eines Softwaresystems, welches Benutzern des Softwaresystems zur Verfügung gestellt wird. Die Abgeschlossenheit eines Inkrements im Kontext der iterativen Migration definiert sich durch die geforderte Autonomie der Microservices. Damit müssen innerhalb einer Iteration die Bestandteile des Monolithen betrachtet werden, die zu autonomen Microservices führen. An dieser Stelle wird die zweite Problemstellung gesehen. Das Festlegen des Umfangs eines Inkrements, das ausschließlich autonome Microservices umfasst, bedingt ein Verständnis über kohäsiv zueinanderstehende Softwarebausteine des Monolithen. Hierzu muss zudem eine Methodik gegeben sein, die das Beantworten der Frage der Kohäsion zwischen Softwarebausteinen nachvollziehbar gestaltet wird.

1.4.3 P3 - Veraltete und fehlende Architekturbeschreibung

Ein wichtiger Faktor für das Festlegen des Umfangs eines Inkrements ist der Ist-Zustand des monolithischen Softwaresystems. Der Ist-Zustand zeigt die bestehenden Softwarebausteine auf, die letztendlich durch Iteration modernisiert werden sollen. Eine Quelle für den Ist-Zustand ist die Architekturbeschreibung. Als dritte Problemstellung wird jedoch eine veraltete und fehlende Architekturbeschreibung gesehen. Gerade bei monolithischen Softwaresystemen ist der Trend zu erkennen, dass die Pflege der Architekturbeschreibung vernachlässigt wird, sodass nur noch der Quellcode des Monolithen und die Softwareentwickler als Quelle für den Ist-Zustand herangezogen werden können [RH06, FL+18]. Auf diese Gegebenheit muss bei dem Migrationsvorhaben reagiert werden, sodass der Ist-Zustand einer Beschreibung entnommen werden kann. Die beschreibenden Modelle der Architekturbeschreibung sollten auf Basis einer formalen Modellierungssprache erstellt werden, damit ein einheitliches Verständnis bei den Softwareentwicklern gewährleistet werden kann [RN+13].

1.4.4 P4 - Funktionsumfang eines Microservices bestimmen

Eine Microservice-Architektur stellt Funktionalitäten eines Softwaresystems durch mehrere Softwarebausteine zur Verfügung, die eine einzelne Aufgabe autonom erledigen können [FL14]. Bei dem Entwurf der Microservices stellen sich verschiedene Herausforderungen [EC+16]. Dabei ist gerade die Bestimmung des Funktionsumfangs eine zentrale Herausforderung. Ein zu groß gewählter Funktionsumfang birgt erneut die Gefahr–wie bei einem Monolithen–die Wartbarkeit durch zu hohe Komplexität zu reduzieren. Dagegen führen zu kleine Microservices durch die große Anzahl an eigenständigen Softwarebausteinen erneut zu einer hohen Komplexität im Betrieb. Auch ist das Erfüllen von nicht-funktionalen Anforderungen wie die Performance erschwert. Der effektive und effiziente Einsatz einer Microservice-Architektur ist damit an einen angemessenen Entwurf geknüpft. Ein etablierter Ansatz für den Entwurf einer Microservice-Architektur ist DDD, der anhand der Geschäftsdomäne sogenannte *Bounded Contexts* ableitet [Ne15]. Ein Bounded Context kapselt kohäsive Funktionalität, die eine Autonomie eines Microservices ermöglicht. Der Einsatz von DDD ist allerdings nur oberflächlich definiert und benötigt Richtlinien für den systematischen Einsatz durch das Entwicklungsteam [HG+17]. Weiterführend muss festgehalten werden, dass der Entwurf der Microservice-Architektur nach DDD nicht die nicht-funktionalen Anforderungen an das Softwaresystem berücksichtigt. Aus diesem Grund muss eine nachgelagerte Überarbeitung durch genau diese nicht-funktionalen Anforderungen erfolgen.

1.4.5 P5 - Festlegen von Verantwortlichkeiten für die Microservices

Die Autonomie der Microservices dient dem Zweck einer losen gekoppelten Wartung. Änderungen oder Neuerungen können unabhängig von den anderen Microservices durch das Entwicklungsteam umgesetzt werden. Gleichermaßen gilt dies für die Bereitstellung des Microservices für den Benutzer. Durch die lose Kopplung können neue Releases ohne Beeinträchtigung anderer Microservices bereitgestellt werden. Diese Charakteristik wird durch Prinzipien wie die der *Service-Teams* maßgeblich unterstützt [Ne15]. Ein Service-Team ist definiert als ein kleines, autonomes Entwicklungsteams, welches die Verantwortung für den gesamten Lebenszyklus eines einzelnen Microservices trägt. Folglich muss das monolithisch geprägte Entwicklungsteam bei der Migration an den neuen Architekturstil angepasst werden. Damit das Service-Team den gesamten Lebenszyklus abdecken kann, müssen verschiedene Fähigkeiten der Softwareentwickler vorgesehen werden. Als Problemstellung wird bei dieser Reorganisation die Auswahl der Softwareentwickler aus dem bestehenden Entwicklungsteam gesehen, sodass alle Fähigkeiten vertreten sind und keine Interessenskonflikte bei der Weiterentwicklung auftreten.

1.4.6 P6 - Höhere Komplexität des Betriebs

Im Gegensatz zu einem monolithischen Softwaresystem besteht ein Microservice-basiertes Softwaresystem aus einer Vielzahl autonomer Softwarebausteine, die eigenständig betrieben werden müssen. Die hohe Anzahl an Softwarebausteinen führt zu einer neuen Komplexität des Betriebs für das Entwicklungsteam, sodass geeignete Konzepte zur Unterstützung notwendig sind. Dies ist die Problemstellung im Kontext des betrachteten Szenarios. Zum Erreichen der gewünschten Unterstützung haben sich die *Development & Operations-Prinzipien* (DevOps) etabliert [BH+16]. Die DevOps-Prinzipien zielen auf eine hohe Automatisierung für den Betrieb der Microservices ab [BW+15]. Das Etablieren der Prinzipien beruht auf dem Einsatz von Werkzeugen. Während dem Entwurf und der Implementierung der Microservices müssen diese Werkzeuge berücksichtigt werden.

1.5 Zielsetzung und Forschungsbeiträge

Aus den identifizierten Problemstellungen werden die Zielsetzung und Forschungsbeiträge dieser Arbeit abgeleitet. Ein Forschungsbeitrag kann dabei mehrere Problemstellungen behandeln. Zunächst erfolgt die Betrachtung der Zielsetzung. Anschließend werden aufbauend auf dieser Zielsetzung die eigentlichen Forschungsbeiträge dargestellt.

1.5.1 Zielsetzung

Das Ziel dieser Forschungsarbeit ist es, den Entwurf eines ganzheitlichen Migrationsrahmenwerks für eine iterative Migration monolithischer Softwaresysteme in eine domänengetriebene Microservice-Architektur voranzutreiben.

Es gilt für das Migrationsrahmenwerk, den sequentiellen Ablauf von Migrationsaktivitäten zu definieren. Dem ausführenden Entwicklungsteam soll dabei eine gewisse Flexibilität bei der Durchführung einer einzelnen Migrationsaktivität eingeräumt werden, sodass eigene entwicklungsnahe Bedürfnisse und Rahmenbedingungen eingebracht werden können. Um eine Nachvollziehbarkeit der Systematik zu gewährleisten, müssen formale Softwareartefakte definiert werden, welche bei der Durchführung von Migrationsaktivitäten als Informationsquelle dienen oder das daraus resultierende Ergebnis festhalten. Auch bezüglich der Nachvollziehbarkeit soll der Organisation Flexibilität ermöglicht werden. Zur Gewährleistung dieser Flexibilität soll zunächst ein generalisierter Entwurf des Migrationsrahmenwerks erfolgen. Der gewünschten Flexibilität wird jedoch weiterhin das übergeordnete Ziel der Modernisierung in eine domänengetriebene Microservice-Architektur zugrunde gelegt. Damit jedoch nicht nur im Kontext dieser Forschungsarbeit ein generalisiertes Migrationsrahmenwerk entworfen wird, werden konkrete Ausprägungen der generalisierten Migrationsaktivitäten ausführlich diskutiert. Aufgrund der zeitlichen und finanziellen Belastung einer Migration für die Organisation gilt es, zur Minimierung dieser Belastung, klare Verantwortlichkeiten bei den Migrationsaktivitäten zu schaffen. Hierfür sollen für die Migrationsaktivitäten Rollen festgelegt werden, die zu Beginn einer iterativen Migration festzulegen sind.

1.5.2 B1 - Entwurf eines Migrationsrahmenwerks

Der erste Beitrag dieser Forschungsarbeit thematisiert den Entwurf des Migrationsrahmenwerks und greift im Querschnitt alle identifizierten Problemstellungen aus Abschnitt 1.4 auf. Festgelegt werden Migrationsaktivitäten, formale Softwareartefakte und Rollen, die für eine iterative Migration notwendig sind.

Als erster Schritt, bevor der eigentliche Entwurf erfolgt, wird eine Betrachtung grundlegender Konzepte einer iterativen Migration durchgeführt. Insbesondere wird festgelegt, wie Iterationen einer iterativen Migration aussehen und wie diese gestartet werden. Durch diese Betrachtung werden *Definitionen* und *Bestandteile* einer iterativen Migration abgeleitet, die im Migrationsrahmenwerk verankert werden müssen. Ausgehend von den Definitionen, wird sich ein verbessertes Verständnis des Migrationsrahmenwerks während des Entwurfs und später bei der Verwendung versprochen.

Damit die in der Zielsetzung geforderte Flexibilität des Migrationsrahmenwerks als erreicht gilt, erfolgt ein *generalisierter Entwurf*, der auf die Verwendung konkreter Softwareentwicklungsansätze und

konkreter Softwareartefakte in den Migrationsaktivitäten verzichtet. Das anwendende Entwicklungsteam ist damit nicht an Softwareentwicklungsansätze gebunden, die durch diese Forschungsarbeit vorgegeben werden. Ein wichtiger Bestandteil des Migrationsrahmenwerks ist die *Systematik*. Die Systematik besteht aus Migrationsaktivitäten, die sequentiell durchlaufen werden, und bildet die iterative Migration ganzheitlich ab. Das Entwicklungsteam wird so durch die Systematik bereits vor Beginn einer Iteration unterstützt.

1.5.3 B2 - Bestimmen des Umfangs eines Inkrements

Eine geringe Komplexität und hohe Beherrschbarkeit eines Inkrements ist entscheidend für den Erfolg der iterativen Migration. Der zweite Forschungsbeitrag befasst sich daher mit dem Bestimmen des Umfangs dieses Inkrements, der maßgeblich die Komplexität und Beherrschbarkeit bestimmt. Folglich fokussiert dieser Forschungsbeitrag die Problemstellungen P2 und P3.

Zur Bestimmung des Umfanges werden Migrationsaktivitäten der generalisierten Systematik (vgl. Abschnitt 1.5.2) mittels Softwareentwicklungsansätzen und formalen Softwareartefakten konkretisiert. Im Zuge dieser Konkretisierung werden den generalisierten Migrationsaktivitäten Subaktivitäten hinzugefügt, um den systematischen und nachvollziehbaren Gedanken des Migrationsrahmenwerks weiterzuführen. Die Subaktivitäten werden im weiteren Verlauf der Forschungsarbeit als *konkrete Migrationsaktivitäten* bezeichnet, um eine Abgrenzung zu den abstrakten Migrationsaktivitäten zu erreichen.

Der Umfang eines Inkrements wird definiert durch die Softwarebausteine, die im bestehenden monolithischen Softwaresystem vorhanden sind und letztendlich in eine Microservice-Architektur überführt werden sollen. Für das Bestimmen des Umfangs werden die klassischen Softwareentwicklungsphasen Analyse und Entwurf durchlaufen, was dem Umsetzen eines *Design Recoverys* entspricht [CC90]. In der Analysephase werden bestehende Anforderungen erhoben, um verhaltensspezifische Aspekte in dem Inkrement nachvollziehen zu können. Im Entwurf werden die Software- und Hardwarebausteine des Monolithen betrachtet, um die strukturellen Aspekte zu verstehen. Das Ergebnis des Design Recoverys ist eine aktuelle Architekturbeschreibung. Die mit diesem Forschungsbeitrag angestrebte Methodik beschränkt den Umfang der Architekturbeschreibung immer auf das betrachtete Inkrement. So wird ein überflüssiger Mehraufwand für die Dokumentation nicht relevanter Softwarebausteine des Monolithen vermieden.

1.5.4 B3 - Domänengetriebener Entwurf der Microservice-Architektur

Ein übergeordnetes Ziel der iterativen Migration durch das Migrationsrahmenwerk ist der Entwurf einer domänengetriebenen Microservice-Architektur. Der zu erstellende Entwurf ist dabei im Kontext

einer Iteration zu sehen und betrifft nur die Softwarebausteine des Inkrements. Wie auch bei dem Bestimmen des Umfangs des Inkrements, werden für den Entwurf konkrete Softwareentwicklungsansätze und formale Softwareartefakte eingeführt, um systematisch und nachvollziehbar an den Entwurf der Microservice-Architektur zu gelangen.

Für den Entwurf der Microservice-Architektur wird auf den Softwareentwicklungsansatz DDD [Ev04] zurückgegriffen, der gerade im Kontext von Microservices ein hohes Ansehen genießt [Ne15]. Der Entwurf der Microservice-Architektur leitet sich aus der *strategischen Modellierung* von DDD ab. Das zentrale Konzept der strategischen Modellierung ist der *Bounded Context*, der kohäsive Geschäftslogik kapselt und damit einen Kandidaten für einen Microservice darstellt. Eine Schwachstelle von DDD ist jedoch die fehlende Systematik zur Unterstützung bei der strategischen Modellierung [HG+17]. Weiterführend wird für den Entwurf der domänengetriebenen Microservice-Architektur eine nachgelagerte Überarbeitung des Entwurfs anhand nicht-funktionaler Anforderungen durchgeführt, um bestimmten Qualitätsmerkmalen gerecht zu werden. Bedingt durch das Ignorieren technisch getriebener Anforderungen durch DDD ist eine solche Überarbeitung notwendig.

1.5.5 B4 - Vorbereitungen zur Inbetriebnahme der Microservices

Der vierte Forschungsbeitrag sieht konkrete Vorbereitungen für die Inbetriebnahme der im Inkrement geschaffenen Microservices vor. Hierfür werden erneut konkrete Migrationsaktivitäten vorgestellt, die aufbauend auf der generellen Systematik des Migrationsrahmenwerks entworfen werden. Fokussiert werden die Problemstellungen P5 und P6. Zu diesen gehören die Anpassung des Entwicklungsteams auf den neuen Architekturstil, indem für jeden Microservice ein festes Service-Team vorgesehen wird. Jedem Service-Team werden ausgehend von Erfahrung und Fähigkeiten Entwickler zugewiesen. So wird die Wartung der neuen Microservices sichergestellt und die für die Microservice-Architektur zentrale Charakteristik der Autonomie aufgegriffen. Ebenfalls gehört zu den Vorbereitungen das Berücksichtigen bestimmter Cloud-nativer Faktoren [Wi17, GB+17], wie die Containerisierung der Microservices, um ein einheitliches und reproduzierbares Veröffentlichen der Microservices zu ermöglichen.

1.5.6 Demonstration der Tragfähigkeit

Die Demonstration der Tragfähigkeit der Forschungsbeiträge beruht auf der praktischen Anwendung an einem monolithischen Softwaresystem. Anhand des Softwaresystems wird gezeigt, dass das Migrationsrahmenwerk und die dazugehörigen konkreten Migrationsaktivitäten aus den Forschungsbeiträgen das monolithische Softwaresystem iterativ modernisiert.

Als Kooperationspartner für diese Forschungsarbeit fungiert die Evana AG, ein Softwarehaus, welches in der Domäne der Immobilienwirtschaft agiert. Die SaaS *Evana360* der Evana AG wird für die Validierung des iterativen Migrationsprozesses herangezogen. Um die Geschäftsgeheimnisse der Evana AG während der Demonstration wahren zu können, werden die geschäftskritischen Inhalte abstrahiert und im Kontext eines *fiktiven Dokumentenmanagementsystems* (DMS) dargestellt. Die Durchführung einer Iteration und die Bestimmung eines Inkrements wird anhand eines in der Evana AG aufgetretenen Szenarios veranschaulicht.

Die Evana360-Lösung besteht aus zwei Monolithen (Abbildung 1.2): (1) Benutzeroberfläche und (2) Geschäftslogik. Für die Demonstration wird nur die Geschäftslogik fokussiert, da dort zum einen der höchste Geschäftswert der Evana AG zu finden und zum anderen eine für die Migration notwendige Komplexität gegeben ist. In dem fiktiven DMS wird somit auch nur die Geschäftslogik der Evana360-Lösung abgebildet.

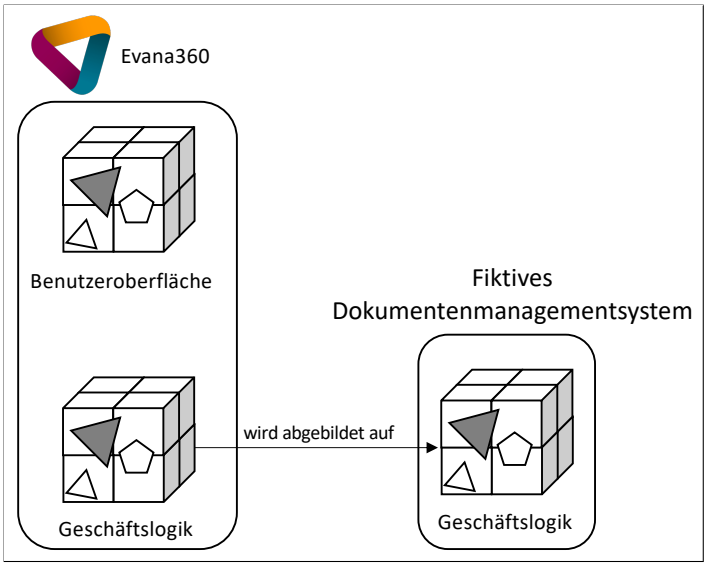

Abbildung 1.2: Umfang des fiktiven Dokumentenmanagementsystems anhand der Evana360-Lösung

Die Benutzer des fiktiven DMS sind *Asset-Manager*, die als Mitarbeiter von großen Immobilienunternehmen *Fonds* und *Gebäude*, auch *Assets* genannt, verwalten. Im täglichen Umgang mit den Gebäuden fallen eine Vielzahl von unterschiedlichen Dokumenten an, die für den Werterhalt des Gebäudes unabdingbar sind. Das geschäftliche Interesse des Asset-Managers, diese Dokumente zu verwalten ist daher sehr hoch. Ein Dokument entspricht einer digitalen Datei, was bei der fortlaufenden Betrachtung des fiktiven DMS vor allem bei den technischen Aspekten von Relevanz ist. Durch die hohe Anzahl an Dokumenten ist eine manuelle Verwaltung nur sehr schwer möglich und mit hohem Aufwand verbunden. An dieser Stelle kommt das fiktive DMS zum Tragen, das dem Asset-Manager als intelligente Ablage die Verwaltung der Dokumente automatisiert.

Die domänenspezifischen Zusammenhänge und der Funktionsumfang, den das fiktive DMS zur

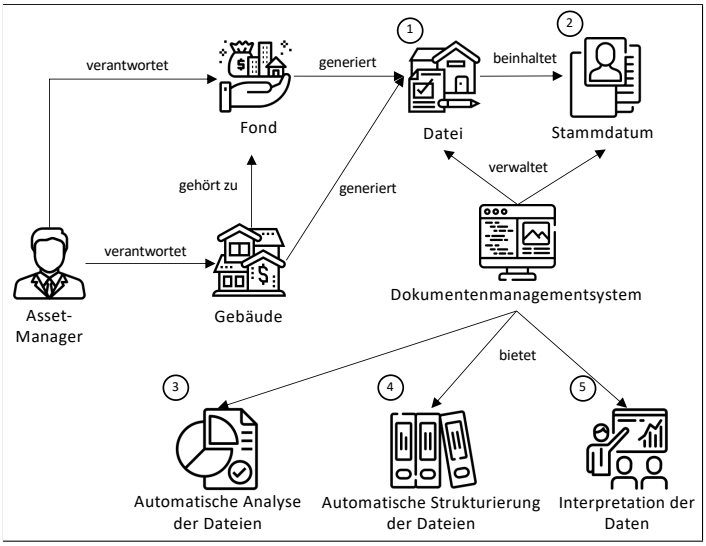

Abbildung 1.3: Zusammenhang der Domäne und darauf aufbauender Funktionsumfang des fiktiven Dokumentenmanagementsystems

Verfügung stellt, wird in Abbildung 1.3 veranschaulicht. Standardmäßig können Dokumente und Stammdaten innerhalb des DMS verwaltet werden. Dokumente (1) können hochgeladen, gelesen, bearbeitet und gelöscht werden. Das Bearbeiten betrifft dabei nicht die Inhalte der Dokumente. Die Stammdaten (2) können angelegt, bearbeitet, gelöscht und Dokumenten zugewiesen werden. Beispiele für Stammdaten sind die einzelnen Gebäude oder Fonds des Asset-Managers. Durch die enge Bindung des DMS an die Immobilienbranche wird die Stammdatenverwaltung in eine Fond- und Gebäudeverwaltung spezialisiert. Herausstellungsmerkmale des DMS sind die automatische Analyse (3) und Strukturierung (4) der Dokumente, die für das Bestimmen der Art des Dokuments, das Extrahieren von Inhalten und die Dokumente in Ordnerstrukturen abzulegen verantwortlich sind. Die extrahierten Inhalte können interpretiert (5) und in grafische Auswertungen überführt werden, um schnell einen Überblick über die Fonds oder Gebäude zu bekommen. Die beschriebene Funktionalität des Monolithen beschränkt sich auf die für die Geschäftsdomäne der Evana AG relevanten Funktionalitäten. Technische Funktionalität und Querschnittsfunktionalität wie Benutzerverwaltung oder Rechteverwaltung werden aufgrund ihrer geringen Bedeutung für die Domäne nicht betrachtet.

Die zugrunde liegende Softwarearchitektur des fiktiven DMS entspricht einem unzureichend modularisierten Monolithen (vgl. Abschnitt 2.2.3). Somit gilt die Annahme, dass während der Entwicklung des Softwaresystems das *Single Responsibility Principle* (SRP) und das *Separation of Concern* vernachlässigt wurde und kohäsive Funktionalität in verschiedenen Softwarebausteinen wiederzufinden ist. Verfolgt wurde bei der Implementierung das Paradigma der objektorientierten Programmierung, sodass Pakete, Klassen, Attribute und Methoden im Quellcode vorzufinden sind. Weiterhin muss festgehalten werden, dass keine Dokumentation über den Ist-Zustand des Monolithen existiert. Der Quellcode und das Wissen des Entwicklungsteams sind damit die Informationsquellen.

1.6 Prämissen der Arbeit

Nachfolgend werden Einschränkungen und Annahmen für die Bearbeitung der Forschungsbeiträge getroffen, welche die Problemdomäne dieser Arbeit einschränken und präzisieren. Die aufgestellten Prämissen dienen auch dazu die Komplexität und Varianz von Migrationsvorhaben einzuschränken und damit den Anwendungsbereich der getätigten Forschungsbeiträge festzulegen.

1.6.1 Prämisse 1 - Objektorientiertes monolithisches Softwaresystem

Bei der Entwicklung eines Softwaresystems kann auf eine Vielzahl von verschiedenen Paradigmen und Architekturstilen zurückgegriffen werden. Das Softwaresystem, welches im Rahmen dieser Forschungsarbeit einer Migration unterzogen wird, unterliegt dabei einigen Einschränkungen. Als Architekturstil wird die *monolithische Architektur* [DG+17] festgelegt. Bei der Implementierung der Mikroarchitektur des Monolithen wurde auf das Paradigma der *objektorientierten Programmierung* [Bo06] zurückgegriffen. So finden sich in der Mikroarchitektur *Pakete*, *Klassen*, *Methoden* und *Attribute* wieder. Weiterführend wird festgehalten, dass das monolithische Softwaresystem auch nur *unzureichend modularisiert* ist. Bei einem modularisierten Monolithen ist die Migration in eine Microservice-Architektur trivial. Die Datenhaltung des unzureichend modularisierten Monolithen ist ebenfalls monolithisch geprägt und entspricht damit einer *geteilten Datenbank* [Ne19].

1.6.2 Prämisse 2 - Trennung der Präsentationslogik von der Geschäftslogik

Der monolithische Architekturstil erlaubt es, bei dem Entwurf des Softwaresystems die Schichten der Schichtenarchitektur von DDD [Ev04] auf mehrere Subsysteme zu verteilen. Der in dieser Forschungsarbeit betrachtete Monolith besteht aus zwei voneinander abhängigen Softwaresystemen: (1) *Benutzeroberfläche* und (2) *Geschäftslogik*. Die Aufteilung in diese beiden Subsysteme entspricht laut Newman [Ne19] der Mehrzahl der monolithischen Softwaresysteme in der Industrie. Daher bietet es sich an, sich auf die *2-Schichten* Monolithen zu beschränken. In dem Subsystem der Benutzeroberfläche findet sich die *Präsentationslogik* wieder. Dagegen umfasst die Geschäftslogik die restlichen Schichten der *Anwendungs-*, *Geschäfts-* und *Infrastrukturlogik*. Das Subsystem der Geschäftslogik stellt damit auch den eigentlichen Geschäftswert für die Organisation dar und aufgrund dieses hohen Stellenwerts wird bei der Migration nur das Subsystem der Geschäftslogik betrachtet.

1.6.3 Prämisse 3 - Existenz betriebsrelevanter Werkzeuge

Während der Migration werden verschiedene Ansätze und Konzepte umgesetzt, die den Einsatz bestimmter betriebsrelevanter Werkzeuge bedingen. So ist beispielsweise für das Verteilen von Client-

Anfragen auf den Monolithen und die entstehenden Microservices ein *API-Gateway* notwendig. Auch bei dem Betrieb der neuen Microservices werden Werkzeuge wie *Kubernetes* [Lin-Kub] als *Container-Orchestrierung* oder *Jenkins* [Jen-Jen] als *Continuous Integration/Continuous Delivery-Pipeline* (CI/CD-Pipeline) benötigt, um die Komplexität einer verteilten Softwarearchitektur wie die der Microservice-Architektur beherrschen zu können. Zwar verwendet der in dieser Forschungsarbeit vorgestellter Migrationsansatz solche Werkzeuge, aber das Einführen und Betreiben dieser Werkzeuge liegt außerhalb des Geltungsbereichs. Folglich wird im weiteren Verlauf die Existenz der notwendigen Werkzeuge angenommen.

1.6.4 Prämisse 4 - Beschlossenes Migrationsvorhaben

Die Durchführung einer Migration innerhalb einer Organisation bindet in einem hohen Maße zeitliche und finanzielle Ressourcen. Das Bereitstellen dieser Ressourcen erfolgt durch Managementpositionen wie den Vorstand einer Organisation. Damit entsteht für das Entwicklungsteam, welches eine Migration anstrebt, eine Abhängigkeit zu anderen Interessenvertretern innerhalb der Organisation, die das Migrationsvorhaben ebenfalls unterstützen müssen. Das Politikum eines Beschlusses für ein Migrationsvorhaben wird nicht als Teil dieser Forschungsarbeit gesehen. Es wird angenommen, dass das Migrationsvorhaben durch die Verantwortlichen der Organisation bestimmt wurde und die Ressourcen verlässlich zur Verfügung gestellt werden. Dazu zählen auch Interessenvertreter, die das Entwicklungsteam fachlich begleiten müssen.

1.6.5 Prämisse 5 - Ausschließen der Implementierung

Innerhalb einer iterativen Migration durchläuft das Migrationsteam die klassischen Phasen einer Softwareentwicklung: (1) *Analyse*, (2) *Entwurf* und (3) *Implementierung*. Den Fokus legt diese Forschungsarbeit auf die Phasen der Analyse und des Entwurfs, denn dort werden die größten Herausforderungen einer Migration gesehen. Auf die Betrachtung der Implementierung wird verzichtet. Zum einen beruht die Implementierung auf den Ergebnissen der beiden Phasen Analyse und Entwurf. Bei einer geeigneten Darstellung der Ergebnisse aus diesen Phasen in Softwareartefakten ist die Implementierung problemlos möglich und damit auch keine forschungsrelevante Herausforderung. Zum anderen besteht bei der Implementierung durch die Technologieabhängigkeit wie beispielsweise die gewählte Programmiersprache ein großer Lösungsraum, der nicht ganzheitlich abgedeckt werden kann und gleichzeitig auch nicht eingeschränkt werden soll.

1.7 Aufbau der Arbeit

Nachfolgend wird der Aufbau dieser Arbeit beschrieben, welcher in Abbildung 1.4 zusammengefasst wird. In **Kapitel 1** wird einleitend die Motivation für die Forschungsarbeit und die Einordnung der Forschungsbeiträge in den Forschungsgegenstand der iterativen Migration monolithischer Softwaresysteme in eine Microservice-Architektur dargestellt. Zudem werden wissenschaftliche Problemstellungen identifiziert, auf denen die Zielsetzung und Forschungsbeiträge dieser Arbeit beruhen. Das **Kapitel 2** führt für die Bearbeitung der Forschungsbeiträge und die zur Gewährleistung eines einheitlichen Verständnisses notwendigen Grundlagen ein. Betrachtet werden Migrationen, Softwarearchitekturen und für die Migration relevanten Softwareentwicklungsansätze. Das Identifizieren des aktuellen Stands der Forschung wird in **Kapitel 3** durchgeführt. Anhand eines Anforderungskatalogs werden die bestehenden Arbeiten bewertet. Aus der Bewertung wird ein Handlungsbedarf abgeleitet, der die wissenschaftliche Motivation für die Bearbeitung der darauf folgenden Forschungsbeiträge darstellt.

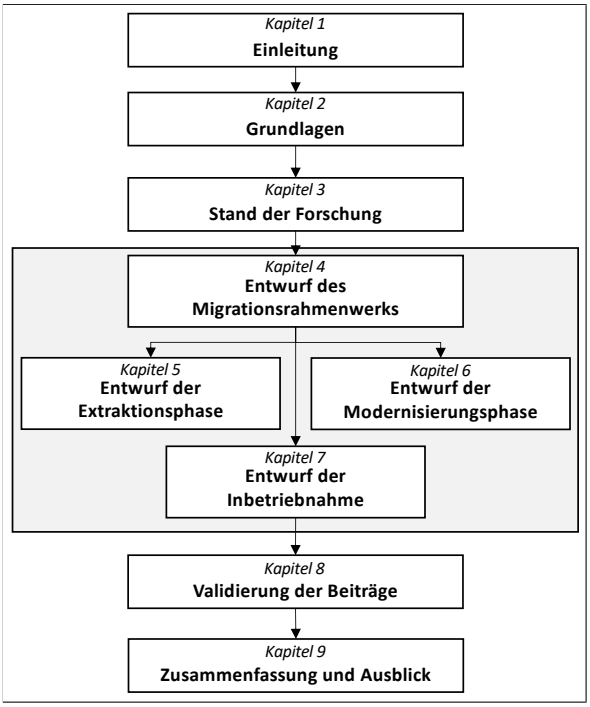

Abbildung 1.4: Aufbau dieser Forschungsarbeit

Das **Kapitel 4** beinhaltet den ersten Forschungsbeitrag dieser Arbeit, welcher den Entwurf eines Migrationsrahmenwerks fokussiert. Zunächst werden grundlegende Begriffe und Bestandteile einer iterativen Migration anhand bestehender Arbeiten abgeleitet und in einheitliche Definitionen überführt. Aufbauend auf den Definitionen wird ein generelles Migrationsrahmenwerk entworfen, das zunächst ohne den Einsatz konkreter Softwareentwicklungsansätze eine iterative Migration anhand

Migrationsaktivitäten und Modellierungsartefakten beschreibt. Die Migrationsaktivitäten und Modellierungsartefakte werden dabei in sogenannte Migrationsphasen gegliedert. Abschließend erfolgt eine Betrachtung des Parallelbetriebs des Monolithen und der in der Iteration geschaffenen Microservices. Der Forschungsbeitrag in **Kapitel 5** diskutiert, ausgehend von dem generellen Migrationsrahmenwerk, die Extraktionsphase anhand konkreter Softwareentwicklungsansätze. Zunächst werden konkrete Migrationsaktivitäten festgelegt, zu denen anschließend formale Modellierungsartefakte bestimmt werden. Für einige Modellierungsartefakte werden Unified Modeling Language-Profile (UML) vorgestellt, die eine Formalisierung ermöglichen. Auch in **Kapitel 6** wird ausgehend von dem generellen Migrationsrahmenwerk eine konkrete Migrationsphase als Forschungsbeitrag entworfen. Die Modernisierungsphase befasst sich mit dem Entwurf der domänengetriebenen Microservice-Architektur. Der Weg zu dem Entwurf wird anhand konkreter Migrationsaktivitäten und formaler Modellierungsartefakte als Systematik abgebildet. Der letzte und im Vergleich zu den anderen Forschungsbeiträgen kleiner Forschungsbeitrag wird in **Kapitel 7** vorgestellt. Dort wird die letzte Migrationsphase des generellen Migrationsrahmenwerks, die Inbetriebnahmephase, anhand konkreter Migrationsaktivitäten und Modellierungsartefakte entworfen. Vorgestellt wird eine Systematik, welche Vorbereitungen zur Inbetriebnahme der in der Iteration geschaffenen Microservices trifft. Die dort angewandten Methodiken richten sich nach den Prinzipien eines Cloud-native Softwaresystems und DevOps.

In **Kapitel 8** werden die geleisteten Forschungsbeiträge in verschiedenen Kontexten validiert, um die Eignung und Anwendbarkeit feststellen zu können. Die Validierung erfolgt in einer industriellen Kooperation mit der Evana AG. Abschließend für diese Forschungsarbeit wird in **Kapitel 9** eine Zusammenfassung der gesamten Arbeit und ein Ausblick für weitere Beiträge am Migrationsrahmenwerk vorgenommen.

2 Grundlagen

Dieses Kapitel umfasst die Grundlagen für das Verständnis über die iterative Migration. Ziel ist es, ein einheitliches Verständnis über Konzepte zu schaffen, die für den Entwurf und die Durchführung eines Migrationsrahmenwerks benötigt werden.

Zunächst wird in Abschnitt 2.1 allgemein auf die Migration von Softwaresystemen eingegangen. In Abschnitt 2.2 werden die Begriffe der Software- und Systemarchitektur diskutiert, sodass Definitionen vorliegen und eine Abgrenzung der beiden Architekturen erfolgen kann. Ausgehend von der Definitionen der Architekturbegriffe werden die Architekturstile der Monolithen und Microservices detailliert betrachtet. Aufgrund der zentralen Rolle des Softwareentwicklungsansatzes der verhaltensgetriebenen Entwicklung innerhalb dieser Forschungsarbeit werden Grundlagen zu diesem Ansatz in Abschnitt 2.3 näher betrachtet. Abschließend wird in Abschnitt 2.4 der domänengetriebene Entwurf vorgestellt, der ebenfalls eine zentrale Rolle für diese Forschungsarbeit einnimmt.

2.1 Migration von Softwaresystemen

Die Entwicklung von Microservices in bestehenden Softwareprojekten beruht auf der Überführung der Softwarebausteine des monolithischen Softwaresystems hinzu in die Microservice-Architektur. Eine Möglichkeit, diese Überführung zu gestalten, ist die Migration des Softwaresystems.

Ausgehend von der Definition von Bisbal et al. [BL+99] ermöglicht die *Migration*, ein Softwaresystem in eine neue Zielumgebung zu überführen, um eine einfache Wartbarkeit und Adaption neuer Anforderungen für das Softwaresystem zu gewährleisten. Bisbal et al. ergänzen zudem, dass bestehende Funktionalität und Daten erhalten bleiben können, ohne diese neu entwickeln zu müssen. Aus der Migration erfolgt eine Erhöhung der Nutzungsdauer des Softwaresystems [AG+05], denn nur Softwaresysteme, die sich kontinuierlich ändern, werden verwendet [LB85]. Nach Gimnich und Winter [GW05] betrifft eine solche Migration nur die technischen Aspekte eines Softwaresystems, die sich in vier Kategorien einteilen lassen:

Hardwareumgebung Umfasst den Wechsel der zugrunde liegenden Hardware.

Laufzeitumgebung Bezieht sich auf den Wechsel der zugrunde liegenden Softwaresysteme, auf die das betroffene Softwaresystem aufbaut. Ein Beispiel dafür sind Betriebssysteme.

Softwarearchitektur Bezeichnet die Änderung der zugrunde liegenden Softwarearchitektur des Softwaresystems. Betroffen sind sowohl die logische als auch die physische Architektur (vgl. Abschnitt 2.2.2).

Softwareentwicklungsumgebung Umfasst einen technologischen Wechsel, wie die Einführung neuer Programmiersprachen.

Der technische Aspekt der Softwarearchitektur kann dabei noch durch die feingranulareren Aspekte der Schnittstellen, Daten und Programmlogik erweitert werden [SH+04]. Die Programmlogik bezeichnet die in der *Geschäftslogik* (vgl. Abschnitt 2.2.3) befindlichen Abläufe. Ackermann et al. [AG+05] ordnen die Migration in den Bereich der *Softwaretechnik* (engl. software engineering) ein und fassen sie als *Engineering-Prozess* auf. Allgemein wird unter dem Software-Engineering ein systematischer und nachvollziehbarer Ansatz verstanden, der die Anforderungen, Entwicklung, Wartung und den Betrieb von Softwaresystemen im Sinn hat [ANSI-729].

Für den weiteren Verlauf dieser Forschungsarbeit wird für die Migration eine kombinierte Definition verwendet:

> Eine *Migration* ist ein Engineering-Prozess zur systematischen und nachvollziehbaren Überführung eines Softwaresystems in eine neue Zielumgebung, wobei diese Zielumgebung sowohl technische als auch organisatorische Aspekte des Softwaresystems beschreibt.

2.1.1 Bezug zu monolithischen Softwaresystemen

In der Literatur sind die von Migrationen betroffenen Softwaresysteme häufig *Legacy-Systeme*. Nach Brodie und Stonebraker [BS95] ist ein Legacy-System

> any information system that significantly resists modification and evolution,

das gleichzeitig einen hohen Einfluss auf die Geschäftsausübung der Organisation hat. Ist ein Softwaresystem mehr als fünf Jahre alt, wird davon ausgegangen, dass die typischen Eigenschaften eines Legacy-Systems wie eine *veraltete Dokumentation* oder die fehlende Berücksichtigung des *Single Responsiblitiy Principle* (SRP), auf das Softwaresystem zutreffen [Sn84]. Ein monolithisches Softwaresystem muss nicht gleichzeitig ein Legacy-System sein, ebenso wie ein Legacy-System kein monolithisches Softwaresystem sein muss. Bennett und Rajlich [BR00] gehen jedoch davon aus, dass ein Legacy-System überwiegend über eine monolithische Architektur verfügt. Laut Feathers [Fe05] bestehen zwischen Monolithen und Legacy-Systemen Verknüpfungen, die besonders die Eigenschaften und Problemstellungen betreffen. Schlussfolgern lässt sich daraus, dass die in der Literatur betrachteten Ansätze der Migration von Legacy-Systemen ebenfalls im Kontext der monolithischen

Softwaresysteme Anwendung finden. Die bestehenden Erkenntnisse werden somit für die Beiträge dieser Forschungsarbeit herangezogen.

2.1.2 Softwarewartung als Auslöser

Ein Softwaresystem ist einem stetigen Wandel unterzogen, was Lehman und Belady [LB85] als *Law of Continuous Change* bezeichnen. Kontinuierlich werden an das bestehende Softwaresystem neue Anforderungen gestellt, die sowohl funktionaler als auch nicht-funktionaler Natur sind. Für die Implementierung der neuen Anforderungen muss das Entwicklungsteam den Quellcode anpassen [Me12]. Aus den Anpassungen folgt eine stetige Zunahme an Komplexität des Softwaresystems, das Lehman [Le80] als *Law of Increasing Complexity* bezeichnet. Die Zunahme an Komplexität ist somit ein natürlicher Prozess des Softwaresystems. Die Komplexität des Softwaresystems nimmt mit steigender Anzahl der Softwarebausteine und der Zeilen an Quellcode zu [NM+16]. Das Verhalten wird auch als *Lebenszyklus eines Softwaresystems* bezeichnet, den Bennett und Rajlich [BR99] im Kontext der *Softwarewartung* (engl. software maintenance) über ein *Stufenmodell* (engl. staged model) abbilden. Unter Softwarewartung wird ein Prozess zur Modifikation eines Softwaresystems oder einzelner Komponenten des Softwaresystems verstanden, mit dem Ziel, nach der Bereitstellung des Softwaresystems Fehler zu korrigieren oder Verbesserungen von funktionalen oder nicht-funktionalen Anforderungen zu implementieren [IEEE-14764].

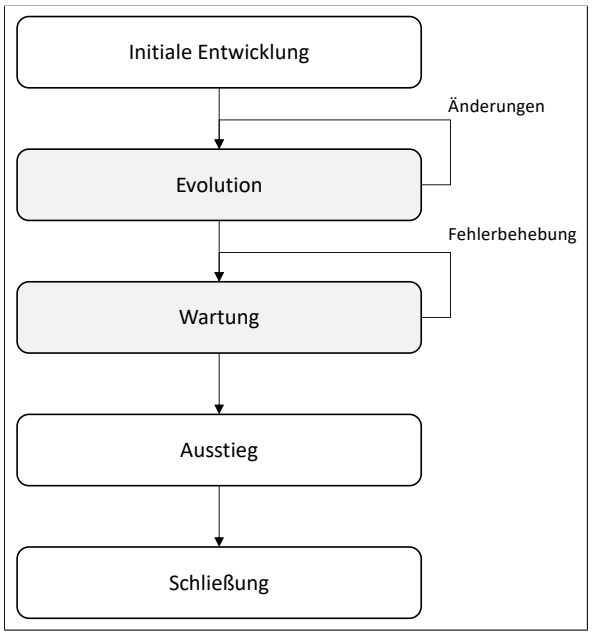

Abbildung 2.1: Phasen des Software-Lebenszyklus nach [BR99]

Das Stufenmodell aus Abbildung 2.1 hebt besonders die Phasen der Evolution und Wartung hervor. Die Evolutionsphase, auch als *Softwareevolution* (engl. software evolution) bezeichnet, tritt

nach der initialen Entwicklung des Softwaresystems ein und befasst sich mit der Umsetzung neuer Anforderungen, die wie das Law of Continuous Change besagt, kontinuierlich eingehen. Die Implementierungstätigkeiten in der Softwareevolution beruhen auf zwei in der initialen Entwicklung geschaffenen Grundlagen [BR99]: (1) Softwarearchitektur, (2) Wissen des Entwicklungsteams. Die Softwarearchitektur wird für die gesamte Evolution des Softwaresystems gleich bleiben. Das Wissen des Entwicklungsteams bezieht sich auf das Verständnis über die Interna des Softwaresystems. Ausgehend von dem Law of Increasing Complexity sehen Bennett und Rajlich [BR00] als Gefahren für die Softwareevolution den Verfall der Softwarearchitektur durch eine *Architekturerosion* oder den Verlust von Wissen im Entwicklerteam. Beide Gefahren bedingen sich gegenseitig. Ist eine Architekturerosion eingetreten, steht der Implementierung der neuen Anforderung ein zu geringer *Kosten-Nutzen-Faktor* (engl. cost-benefit) gegenüber. An dieser Stelle findet die Transition der Evolutionsphase in die Wartungsphase statt. In der Wartungsphase werden nur Fehlerbehebungen durch das Entwicklungsteam durchgeführt, auch wenn das Softwaresystem noch einen hohen Nutzen für die Organisation hat.

Der Auslöser von Migrationen monolithischer Softwaresysteme kann somit auf die Softwarewartung und den damit einhergehenden kontinuierlichen Wandel zurückgeführt werden [GW05, AG+05]. Die Ineffizienzen, die durch Architekturerosionen und Wissensverluste im Entwicklungsteam entstehen, müssen durch die Überführung des Softwaresystems in eine neue Umgebung aufgelöst werden.

2.1.3 Methoden, Vorgehensweisen und Strategien

Die Überführung des Softwaresystems in eine neue Zielumgebung wird auch als *Software-Reengineering-Prozess* verstanden. Folgt man der Definition von Chikofsky und Cross [CC90], ist das *Reengineering*

> the examination [understanding] and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.

In Verbindung mit der Definition der Migration stellt das Reengineering ein Mittel zur Durchführung der Migration des Softwaresystems in eine neue Zielumgebung dar und bringt gleichzeitig ein systematisches und nachvollziehbares Vorgehen mit sich. Das Reengineering wirkt auf unterschiedliche Phasen des Software-Lebenszyklus ein. Abbildung 2.2 stellt einen vereinfachten Lebenszyklus dar und zeigt gleichzeitig die für das Reengineering essentiellen Phasen: (1) *Implementierung*, (2) *Entwurf*, (3) *Anforderungen*. Während des Durchlaufens der unterschiedlichen Phasen sind jeweils unterschiedliche Aspekte des Softwaresystems betroffen und werden durch das Reengineering auf die neue Zielumgebung angepasst. Die Implementierung bezieht sich auf den Quellcode, der Entwurf befasst sich mit der Softwarearchitektur und die Anforderungen mit den funktionalen und nicht-funktionalen Anforderungen an das Softwaresystem.

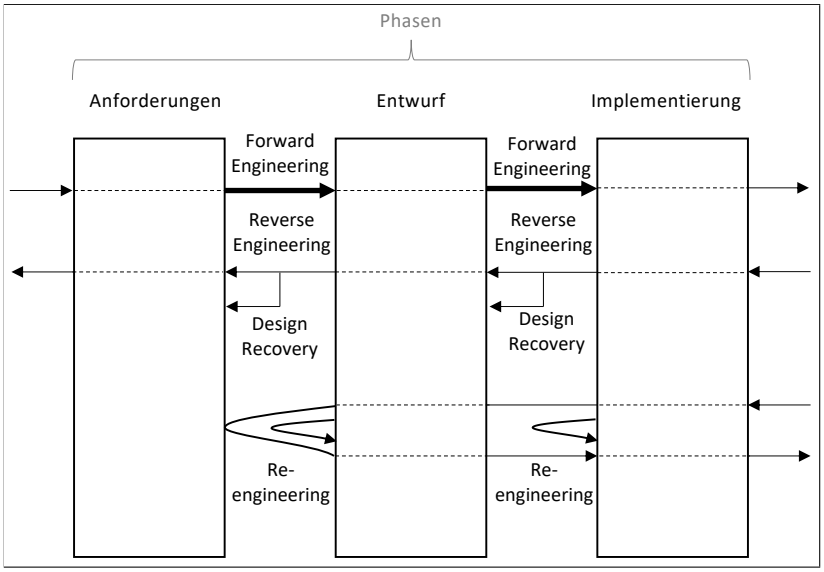

Abbildung 2.2: Einordnung des Reengineerings, Reverse Engineerings und Forward Engineerings in einen vereinfachten Lebenszyklus eines Softwaresystems nach [CC90]

Das Reengineering baut auf den zwei grundlegenden Softwareentwicklungansätzen des *Reverse Engineering* und *Forward Engineering* auf [CC90]. Das Ziel des Reverse Engineering ist es, den Ist-Zustand an verschiedenen Stellen des Software-Lebenszykluses in Form einer Architekturbeschreibung festzuhalten, um das Wissen innerhalb des Softwaresystems abzubilden. Dieser Schritt ist meist notwendig, da monolithische Softwaresysteme nur über eine veraltete oder gar keine Architekturbeschreibung verfügen [LS+03, RH06, FL+18]. Das Reverse Engineering setzt für die Wiederherstellung der Architekturbeschreibung auf das *Design Recovery*, mit dem Ziel, das Wissen sowohl aus dem Quellcode, aber auch von Interessenvertretern und dem Entwicklungsteam zu extrahieren [Bi89]. Bowman und Holt [BH98] beziehen auch *Conway's Law* [Co68] in das Design Recovery mit ein und schließen aus den Kommunikationswegen der Organisation auf die Makroarchitektur (vgl. Abschnitt 2.2.1) des Softwaresystems. Durch das Einbeziehen der Interessenvertreter und des Entwicklungsteams wird das *Domänenwissen* (vgl. Abschnitt 2.4) in die Architekturbeschreibung aufgenommen. Wurden alle notwendigen Informationen über das Softwaresystem in der Architekturbeschreibung festgehalten, können diese durch das Forward Engineering an neue Anforderungen angepasst und entsprechend umgesetzt werden. Das Forward Engineering entspricht dabei dem klassischen Vorgehen der Softwareentwicklung mit seiner Spezifikation der Anforderungen, Entwerfen der Softwarearchitektur und der Implementierung der geschaffenen Modelle [BM97].

Reussner und Hasselbring [RH06] stellen fest, dass ein Migrationsprozess auch noch durch seine *Vorgehensweise* und die damit verbundenen *Strategien* klassifiziert wird. Ein Migrationsprozess umfasst *Methodiken* und folgt einer bestimmten *Vorgehensweise*, die durch Strategien (auch Migrationsstrategien) verfeinert wird (Abbildung 2.3). In der Literatur finden sich drei konkrete Vorgehensweisen

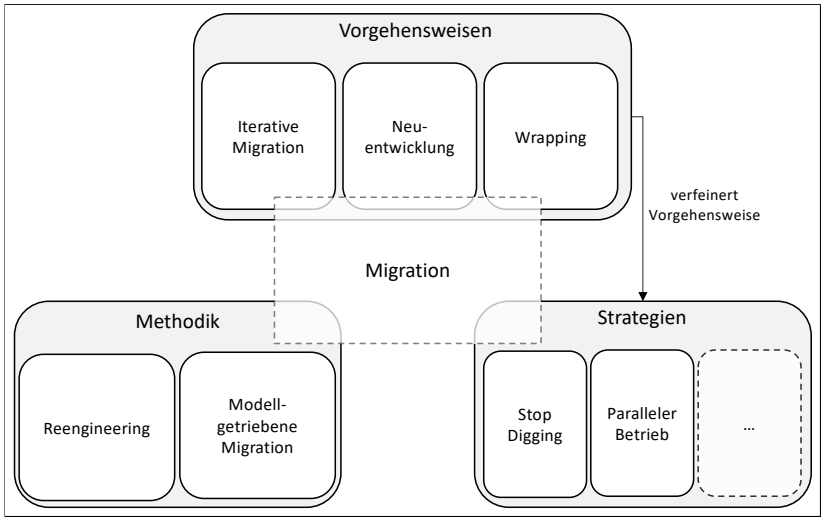

Abbildung 2.3: Taxonomie eines Migrationsprozesses nach [RH06]

für einen Migrationsprozess wieder [BL+99, Fo04, RH06, Ri16, Ne19]: (1) *Iterative Migration*, (2) *Neuentwicklung* (engl. redevelopment) und (3) *Wrapping*. Fowler [Fo04] stellte mit seinem Muster *StranglerFigApplication* eine iterative Vorgehensweise für die Migration eines Softwaresystems vor. Diese Vorgehensweise greift bereits bei der Implementierung neuer Anforderungen ein und besagt, dass keine Anforderungen mehr in dem monolithischen Softwaresystem implementiert werden. Stattdessen werden diese in neue Services, bevorzugt in Microservices, implementiert. Bei dieser Implementierung wird gleichzeitig weitere Funktionalität mit in den neuen Microservice übernommen. Für die iterative Migration stellen Richardson [Ri16] und Newman [Ne19] verschiedene Strategien vor. Beispielhaft werden diese in Abbildung 2.3 aufgezeigt.

Bei der Neuentwicklung eines Softwaresystem werden keine bestehenden Softwarebausteine oder sonstigen Artefakte wiederverwendet, damit eine Entwicklung sozusagen auf der grünen Wiese begonnen werden kann. Die Neuentwicklung wird häufig auch als *Big Bang* referenziert, da am Ende der Migration der gesamte Monolith durch ein neues Softwaresystem ersetzt wird.

Das Wrapping strebt die Entwicklung von sogenanntem *Glue Code* an, der den Zugriff auf das bestehende Softwaresystem definiert [Ri16]. Der Glue Code fördert die Interoperabilität des Softwaresystems, doch Defizite in der logischen oder auch physischen Architektur können damit nicht aufgelöst werden, denn die Softwarebausteine verbleiben innerhalb des Monolithen. Für diese Forschungsarbeit wird als Vorgehensweise des Migrationsprozesses die iterative Migration gewählt. Aufgrund dieser Einschränkung werden die Strategien der Neuentwicklung und des Wrappings nicht weiter thematisiert.

Wie eine Migration konkret bei der iterativen Vorgehensweise durchgeführt wird, definiert ein *Migrationsrahmenwerk* (engl. migration framework) [TL+17, AF+18]. Ein solches Rahmenwerk besteht

aus *Migrationsaktivitäten*, *Softwareartefakten* und *Rollen*, die in Verbindung ein systematisches und nachvollziehbares Vorgehen bereitstellen. Ausgehend von dieser Definition nimmt das Migrationsrahmenwerk eine zentrale Position in der vorliegenden Forschungsarbeit ein.

2.1.4 Modellgetriebene Migration

Die in Abbildung 2.3 dargestellte Taxonomie eines Migrationsprozesses beinhaltet neben dem Reengineering auch die Methodik der *modellgetriebenen Migration* (engl. model-driven migration). Der Gedanke der modellgetriebenen Migration basiert auf der *modellgetriebenen Entwicklung* (engl. Model-Driven Development, MDD), die durch Modelle und Abstraktionen die Komplexität der Softwareentwicklung verringert [SK03]. Die Verwendung von Modellen ist jedoch nicht für die modellgetriebene Migration einzigartig, denn das Reengineering setzt mit dem Reverse Engineering ebenfalls auf die Wiederherstellung von Modellen für die Architekturbeschreibung. Die Besonderheit der modellgetriebenen Migration ist an dieser Stelle die Absicht, über Automatisierungen die Modelle zur Generierung von Softwarebausteinen und Quellcode zu verwenden. Hierfür werden bestimmte Anforderungen an die Modelle gestellt. Ein *Modell* ist nach der Definition von Stachowiak [St73] ein vereinfachtes Abbild eines Objektes, das in der realen Welt vorzufinden ist. Die Eigenschaften werden als (1) *Abbildung*, (2) *Verkürzung* und (3) *Pragmatismus* zusammengefasst. Jedes Modell wird in einer bestimmten Modellierungssprache beschrieben, die aus einer Semantik und Syntax besteht [PM06]. Bei Modellierungssprachen unterscheidet man zwischen informellen, semi-formellen und formellen Sprachen [ML+12]. Eine Modellierungssprache ist die *Unified Modeling Language* (UML) [BR+05], die aufbauend auf der *Meta Object Facility* (MOF) die Syntax und Semantik festlegt. Die MOF verleiht der UML einen gewissen Formalisierungsgrad, der programmatisch interpretiert werden kann. Erst durch diese Formalisierung der Modellierungssprache ist die modellgetriebene Migration in der Lage, automatisiert Softwareartefakte zu generieren.

Eine konkrete Ausprägung einer modellgetriebenen Migration ist die *Architecture-Driven Modernization* (ADM), die von der Object Management Group (OMG) vorgestellt wurde [OMG20b]. Die OMG reagierte auf das steigende Interesse an der Migration von bestehenden Softwaresystemen und auf die Kritik an der *Model-Driven Architecture* (MDA) [OMG20], die nur für die Entwicklung eines neuen Softwaresystems geeignet ist [RH06]. Der Grundgedanke und das prinzipielle Vorgehen der ADM beruht auf der MDA, die über formelle Modellierungssprachen eine automatisierte Modelltransformation in Softwareartefakte anstrebt [SK03]. Der Anteil an manueller Softwareentwicklung soll auf ein Minimum reduziert werden. Diese formelle Modellierungssprache fasste die OMG im *Knowledge Discovery Meta-Model* (KDM) zusammen [PG+11]. Mit der ADM und KDM verfolgt die OMG eine Standardisierung der modellgetriebenen Migration. Favre [Fa05] nennt im Kontext eines modellgetrieben Reverse Engineering-Ansatzes die ADM und bezieht damit das Reengineering in die modellgetriebene Migration mit ein.

2.2 Softwarearchitektur

Ein zentraler Aspekt bei der Migration von monolithischen Softwaresystemen in eine Microservice-Architektur ist die grundlegende Betrachtung der *Softwarearchitektur* und der damit verbundenen Konzepte. Nachfolgend wird somit der Begriff der Softwarearchitektur allgemein diskutiert. Zum Zeitpunkt dieser Forschungsarbeit liegt keine von der Literatur anerkannte einheitliche Definition für den Begriff der Softwarearchitektur vor. Um dennoch eine Definition für den Entwurf des iterativen Migrationsprozesses zu gewährleisten, werden existierende Definitionen aufgelistet und eine dieser Definitionen als für diese Arbeit maßgebend festgelegt.

Der Ausgangspunkt der Softwarearchitektur-Diskussion sind Softwaresysteme. Folgt man der Definition von Reekie und McAdam [RM06], besteht ein Softwaresystem aus einer Menge an *Bausteinen* (engl. building blocks) und Prinzipien. Aus den Prinzipien folgt eine Vereinheitlichung der Bausteine. Jeder Baustein stellt dabei eigene Funktionalität bereit, welche die Absicht des Bausteins definiert. Durch *Beziehungen* zwischen den Bausteinen wird das Softwaresystem zu einem Ganzen zusammengeführt und die Absichten der einzelnen Bausteine zu einem einheitlichen Ziel zusammengefasst. Bausteine werden in *Software-* und *Hardwarebausteine* untergliedert. Während die Softwarebausteine Quellcode referenzieren, stellen die Hardwarebausteine die Umgebung dar, in denen die Softwarebausteine ausgeführt werden. Paulish und Bass [PB01] fassen die Entscheidungen, aus welchen Bausteinen das Softwaresystem besteht, wie diese miteinander verbunden sind und welche vereinheitlichenden Prinzipien verwendet werden, als Softwarearchitektur zusammen. Eine ähnliche Sicht haben Taylor et al. [TM+09]:

> A software system's architecture is the set of principal design decisions made about the system.

Ergänzend fügt Clemens et al. [CG+03] hinzu, dass die Softwarearchitektur auf einem hohen Abstraktionsniveau die Bausteine des Softwaresystems beschreibt. Zusammenfassend beschreiben Rozanski und Woods [RW12] die Softwarearchitektur als

> a structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

Die Autoren bauen auf der weit verbreiteten IEEE-42010 [IEEE-42010] auf, die besagt, dass die Architektur eines Systems eine Menge an fundamentalen Eigenschaften des Systems innerhalb einer Umgebung ist. Das System selbst besteht aus Elementen, Beziehungen und Prinzipien. Aufgrund der detaillierten Ausgestaltung der Softwarearchitektur wird im restlichen Verlauf dieser Forschungsarbeit auf die Definition von Rozanski und Woods zurückgegriffen.

2.2.1 Bezug zum Softwaresystem

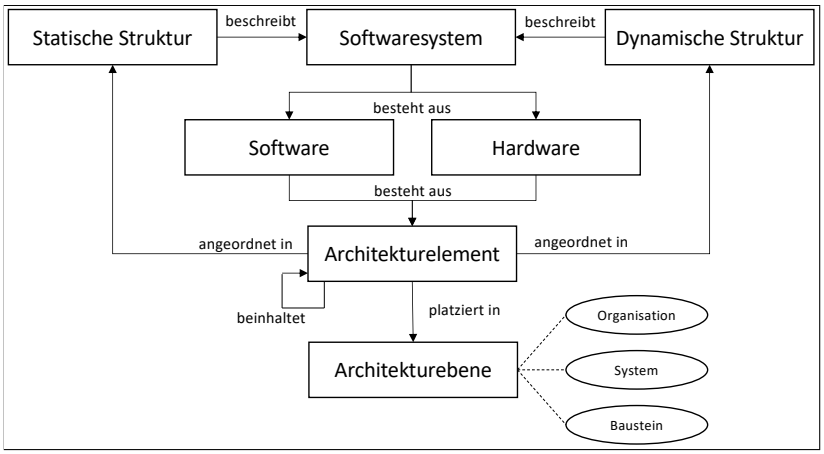

Abbildung 2.4: Metamodel eines Softwaresystems nach den Elementen von Rozanski und Woods [RW12]

Rozanski und Woods [RW12] liefern eine umfangreiche Definition eines Systems beziehungsweise Softwaresystems (Abbildung 2.4). Ein System besteht aus drei übergeordneten Bestandteilen: (1) *Software*, (2) *Hardware* und (3) *Daten*. Alle drei Bestandteile bedingen sich gegenseitig, um die Ausführbarkeit des Systems zu gewährleisten. Wird von einem *Softwaresystem* gesprochen, sind die einzelnen Bestandteile der Software als auch der Hardware inkludiert. Um das Zusammenspiel von Software und Hardware zu verstehen, muss die Softwarearchitektur betrachtet werden. Die Software und auch die Hardware bestehen wiederum aus einer Vielzahl an einzelnen Bestandteilen. Die Struktur der Software wird durch in sich abgeschlossene *Quellcode-Einheiten* (engl. self-contained code unit) beschrieben wie beispielsweise Subsysteme, Module, Komponenten, Klassen oder Funktionen. Dagegen besteht die Hardware aus physischen Komponenten wie Server, Netzwerkschnittstellen oder auch der CPU. Der Begriff des *Architekturelements* (kurz *Element*) wird allgemein für die Bezeichnung von jeglichen Bestandteilen des Softwaresystems verwendet [BC+03]. Wichtige Eigenschaften der Elemente sind eine festgelegte Verantwortlichkeit und eine klare Abgrenzung der Inhalte und Schnittstellen, um auf die Interna zugreifen zu können. Komponenten oder Klassen haben bereits im Kontext verschiedener Ansätze wie der komponentenbasierten Entwicklung eine umfassendere Bedeutung, daher bietet es sich an, den Begriff des Elements im Verlauf dieser Forschungsarbeit zu verwenden. Elemente können zueinander in Beziehung gesetzt werden, was eine bestimmte Anordnung zur Folge hat. Die Anordnung kann bedeuten, dass ein Element ein weiteres Element beinhaltet. Dadurch entsteht eine Hierarchie der Elemente, was sich auf Architekturebenen abbilden lässt. Vogel et al. [VA+11] führt drei *Architekturebenen* ein (Abbildung 2.5):

Organisation Auf dieser Ebene finden sich Elemente wieder, die unmittelbar mit der Struktur der Organisation in Verbindung stehen. Für das Softwaresystem sind diese insofern relevant, dass

diese die organisatorische Umgebung wie beispielsweise die Domäne des Softwaresystems darstellen.

System Die Systemebene kann als multidimensionale Ebene angesehen werden, in der mit unterschiedlichen Abstraktionsniveaus das Softwaresystem betrachtet wird. Begonnen wird mit der Darstellung des dokumentierten Softwaresystems neben weiteren Softwaresystemen der Organisation. Bei abnehmendem Abstraktionsniveau wird das Softwaresystems nach und nach in seine mikroskopischen Bestandteile zerlegt. So kann ein Softwaresystem aus weiteren Subsystemen bestehen.

Bausteine Als Bausteine werden die Elemente aufgefasst, die ein Softwaresystem beziehungsweise ein Subsystem beschreiben. Das können beispielsweise Module und Softwarekomponenten sein, die wiederum durch Klassen implementiert werden. An dieser Stelle erfolgt eine Trennung der Softwarearchitektur in eine Makro- und Mikroarchitektur [BM+98].

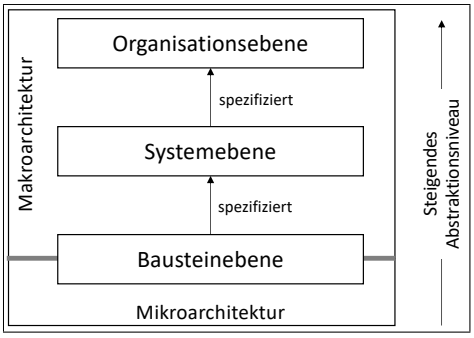

Abbildung 2.5: Zuordnung der Softwarearchitekturebenen nach [VA+11] in die Makro- und Mikroarchitektur

Aus struktureller Sicht kann das Softwaresystem in zwei unterschiedliche Strukturen unterteilt werden. Die Menge der Elemente und die Beziehungen zwischen ihnen werden als *statische Struktur* bezeichnet und beschreiben das Softwaresystem zur Entwurfszeit (engl. design-time). Neben der statischen existiert eine *dynamische Struktur*, die das Softwaresystem zur Laufzeit (engl. runtime) beschreibt. Die Elemente der dynamischen Struktur sind beispielsweise Nachrichten, die zwischen zwei Elementen ausgetauscht werden. Eine Betrachtung der Struktur ist gerade im Umfeld nachrichtengetriebener Architekturen besonders wichtig, da der gesamte Ablauf des Softwaresystems auf dem Zusammenspiel der Nachrichten basiert.

2.2.2 Entwurf der Softwarearchitektur

Architekturelemente sind die Bausteine einer Softwarearchitektur. Verbindungen zwischen den Architekturelementen können bewusst oder unbewusst entstehen, aber jedes Softwaresystem besitzt

eine Softwarearchitektur [RW12]. Eine bewusst gewählte statische Struktur erfolgt im Rahmen eines Architekturentwurfs. Laut Fairbanks [Fa10] ist es kritisch, für Qualitätsmerkmale in Form der nichtfunktionalen Anforderungen [WS+06] des Softwaresystems, einen geeigneten Architekturentwurf zu erstellen. Fairbanks sieht für den Entwurf der Softwarearchitektur folgende Einflussfaktoren:

- Die Softwarearchitektur tritt sowohl in der statischen als auch in der dynamischen Struktur des Softwaresystems auf.
- Die Softwarearchitektur kann die Qualitätseigenschaften des Softwaresystems sowohl unterstützen als auch hemmen, wenn falsche architekturelle Entscheidungen getroffen werden.
- Auch die funktionalen Anforderungen können durch die Softwarearchitektur unterstützt oder gehemmt werden.
- Einige Softwarearchitekturen fördern die Wiederverwendbarkeit der Architekturelemente. Die Wiederverwendbarkeit der Elemente ist gerade im Rahmen von Cloud-nativen Softwaresystemen eine essentielle Anforderung [BH+16].

Für die Erstellung eines Entwurfs gilt es jedoch, den Aufwand gegenüber dem Nutzen abzuwägen. So bietet es sich für Softwaresysteme an, in komplexen Domänen mehr Aufwand in den Entwurf zu investieren, während bei der prototypischen Umsetzung einer einfachen Funktionalität der Entwurfsaufwand minimal sein sollte. Fairbanks [Fa10] fasst dies als "*just enough software architecture*" zusammen. Der Entwurf der Softwarearchitektur besteht aus einer Menge an *architekturellen Entwurfsentscheidungen* (engl. architectural design decisions), die entweder während der initialen Entwurfsphase oder im Rahmen der *Softwarearchitekturevolution* (engl. software architecture evolution) getroffen werden [JB05]. Unter der Softwarearchitekturevolution versteht man die kontinuierliche Anpassung der Softwarearchitektur durch Anforderungen oder Änderungen am Softwaresystem [Ga00]. Ein Entwurf bleibt somit nicht über den gesamten Lebenszyklus des Softwaresystems gleich.

Der Entwurf eines Softwaresystems wird in einer Architekturbeschreibung festgehalten [RW12]. Das Erstellen der Architekturbeschreibung ist ein weiterer Teilbereich der Softwarearchitektur, den Kruchten et al. [KO+06] als *Architekturrepräsentation* bezeichnen. Eine Architekturbeschreibung besteht dabei aus einer Menge an *Artefakten*, die zur Kommunikation mit Interessenvertretern genutzt werden [CG+03]. Booch [Bo06] sieht vor allem einen großen Vorteil bei der Nutzung der Architekturbeschreibung für Diskussionen, die außerhalb der Softwareentwicklung stattfinden.

An den Entwurf der Softwarearchitektur werden von unterschiedlichen Interessenvertretern unterschiedliche Anforderungen gestellt, denen dieser auch genügen muss. Entscheidend für die Architekturbeschreibung ist, dass diese unterschiedlichen Anforderungen festgehalten werden. Nur so kann die Architekturbeschreibung auch in Diskussionen mit Interessenvertretern einen Mehrwert bieten. Der

Standard IEEE 1471 [IEEE-1471] führt für die Repräsentation der Anforderungen durch die Interessenvertreter das Konzept der *Blickwinkel* (engl. viewpoints) ein. Mit dem *4+1 Sichtenmodell* (siehe Abbildung 2.6) definiert Kruchten [Kr95] aufbauend auf dem IEEE 1471 folgende Blickwinkel:

Logische Sicht In der logischen Sicht werden die Architekturelemente der statischen Struktur des Softwaresystems dargestellt. Die Elemente werden von Kruchten als Abstraktionen der Domäne beschrieben, was Objekten der realen Welt entspricht. Durch den hohen Detaillierungsgrad betrachtet diese Sicht die Elemente auf der Architekturebene der Bausteine. Es kann an dieser Stelle von der Mikroarchitektur gesprochen werden.

Prozesssicht Die Prozesssicht beschreibt das Softwaresystem während der Laufzeit. Konkret wird auf laufende Prozesse eingegangen und wie diese über Nachrichten miteinander kommunizieren. Somit wird die dynamische Struktur des Softwaresystems abgebildet.

Entwicklungssicht Kruchten beschreibt die Entwicklungssicht als Grundlage für die Diskussion mit dem Softwareentwicklungsteam. Die dargestellten Elemente sind abstrahierter als die Elemente der logischen Sicht, was bei Subsystemen, Modulen oder auch Komponenten der Fall ist. Diese Sicht beschreibt somit ebenfalls die statische Struktur des Softwaresystems, befindet sich dabei aber auf den Abstraktionsebenen der Systeme und Bausteine. Es wird eine Sicht auf die Makroarchitektur eingenommen.

Physische Sicht Während die zuvor beschriebenen Sichten die Software des Softwaresystems betrachten, nimmt die physische Sicht die Ausführungsumgebung und damit primär die Hardware in den Fokus. Die Elemente dieser Sicht sind aber sowohl softwarebezogene als auch hardwarebezogene Elemente, die benötigt werden, um die Ausführungsarchitektur (engl. execution architecture) darstellen zu können [RM06]. Auch diese Sicht befindet sich sowohl auf der System- als auch auf der Bausteinebene und bezieht sich auf die Makroarchitektur.

Szenarien Unter den Szenarien werden die wichtigsten Anforderungen verstanden, die in den 4 primären Sichten abgebildet werden. Die Szenarien sind eine Art fünfte Sicht auf die Softwarearchitektur.

Um im Laufe der Forschungsarbeit einfacher auf die Inhalte der Sichten referenzieren zu können, wird eine Gruppierung definiert. Zu *logische Architektur* werden die logische Sicht, die Entwicklungs- und die Prozesssicht zusammengefasst. Die physische Sicht wird analog als *physische Architektur* bezeichnet. Ausgehend von dieser Gruppierung kann die Begrifflichkeit des Architekturelements verfeinert werden. So finden sich in der logischen Architektur Softwarebausteine wieder, während die physische zusätzlich zu den Software- auch Hardwarebausteine beinhaltet. In diesem Kontext definert Vogel et al. [VA+11] diesen Begriff des Bausteins. Wird im weitere Verlauf der Forschungsarbeit der Begriff des Bausteins verwendet, referenziert dieser einen Softwarebaustein. Hardwarebausteine werden immer explizit erwähnt.

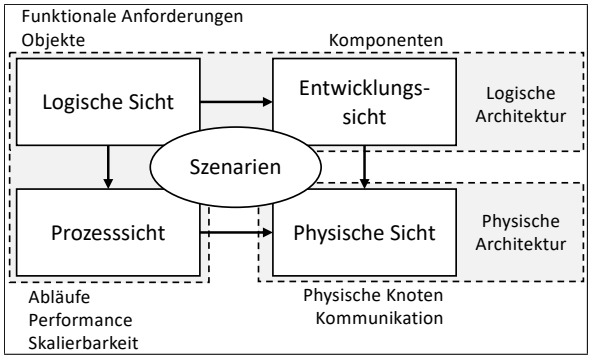

Abbildung 2.6: Blickwinkel auf die Softwarearchitektur nach Kruchtens 4 + 1 Sichtenmodell [Kr04]

Die Repräsentation der Architekturartefakte in der Architekturbeschreibung kann auf verschiedenen Wegen erfolgen. Grundlegend werden jedoch Modellierungssprachen verwendet, die in semi-formellen oder formellen Ausprägungen existieren. Die bekannteste semi-formelle Modellierungssprache ist die UML [BR+05], die eine Vielzahl an unterschiedlichen Grammatiken zur Modellierung zur Verfügung stellt. Die UML bietet für sämtliche Blickwinkel der Architekturbeschreibung eine angemessene Repräsentation. Eine weitere, aber weniger in der Literatur betrachtete semi-formelle Sprache ist *NoUML* [Br13], die gerade im Kontext des C4-Sichtenmodells [Br20] angewandt wird. Aufgrund der geringen Akzeptanz in der Literatur wird jedoch das C4-Sichtenmodell nicht für diese Forschungsarbeit herangezogen.

2.2.3 Unterstützung durch Architekturstile

Ein Softwaresystem verfolgt das Ziel, bestimmte Problemstellungen durch die Bereitstellung von Funktionalitäten zu lösen. Abhängig von der gewählten Softwarearchitektur ist das Softwaresystem besser oder schlechter in der Lage, das gestellte Problem zu lösen [Fa10]. Um einfacher die richtigen architekturellen Entwurfsentscheidungen zu treffen, wurden die sogenannten *Architekturstile* eingeführt. Taylor et al. [TM+09] definieren einen Architekturstil als Sammlung an architekturellen Entwurfsentscheidungen, die für eine bestimmte Problemstellung anwendbar ist, den Spielraum der Entwurfsentscheidungen einschränkt und dem Softwaresystem bestimmte Qualitätseigenschaften verleiht. Ergänzend kann die Definition von Reussner und Hasselbring [RH06] herangezogen werden, die Architekturstile als prinzipielle Lösungsstrukturen sehen, die für ein Architekturelement einer Softwarearchitektur durchgängig und mit weitgehendem Verzicht auf Ausnahmen angewandt werden. So kann ein Softwaresystem verschiedene Architekturstile verwenden, wobei eine Vermischung unterschiedlicher Architekturstile auf derselben Architekturebene vermieden werden sollte [RB+16].

Die Idee der Architekturstile beruht auf der Tatsache, dass Problemstellungen während der Soft-

wareentwicklung und vor allem in dem Entwurf der Softwarearchitektur wiederkehrend sind. Ein Architekturstil wird somit gezielt für eine Problemstellung herangezogen. Die Betrachtung der Problemstellung findet jedoch auf einem *hohen Abstraktionsniveau* statt und gibt lediglich Aufschluss darüber, was die fundamentalen statischen und dynamischen Strukturen des Softwaresystems sind und welche Eigenschaften dieses dadurch besitzt [VA+11]. Aus der abstrakten Beschreibung folgt ein hoher Grad an Freiheit für die Umsetzung des Architekturstils, was zugleich Vorteil und Nachteil darstellt. Vorgegeben sind nur die grundlegenden Architekturelemente, die Topologie in der die Elemente angeordnet werden und die Verbindungen zwischen den Elementen, welche die Kommunikation beschreiben [SG96].

Sehr ähnlich und nur schwer von den Architekturstilen zu trennen sind *Architekturmuster*. Eine eindeutige Definition der Architekturmuster in der Literatur ist nicht zu finden. Taylor et al. [TM+09] definiert ein Architekturmuster als Menge von architekturellen Entwurfsentscheidungen, die auf eine wiederkehrende Problemstellung angewandt werden kann. Ein Architekturmuster ist geringfügig parametrisierbar und kann damit auf unterschiedliche Kontexte der Softwareentwicklung angewandt werden. Vogel et al. [VA+11] greift die Anwendbarkeit in unterschiedlichen Kontexten auf und definiert, dass ein Architekturmuster allgemein formuliert werden muss, jedoch gleichzeitig so spezifisch, dass der Anwender einen klaren Leitfaden zur Umsetzung bekommt. Folglich gibt ein Architekturmuster die Implementierung des Musters vor, während bei einem Architekturstil keine Vorgaben zur Implementierung gegeben werden.

Monolithische Architektur

Ein in der Praxis auch gerade durch seine Simplizität häufig verwendeter Architekturstil ist die monolithische Architektur. Gerade in den ersten Entwicklungszyklen bringt diese Architektur viele Vorteile mit sich, die sich jedoch bei ansteigendem Funktionalitätsumfang Stück für Stück in Nachteile verwandeln. Fowler [Fo15] definierte aufbauend auf diesem Gedanken das Muster *MonolithFirst*.

Nach Dragoni et al. [DG+17] bestehen monolithische Softwaresysteme aus Softwarebausteinen, die sich nicht unabhängig voneinander ausführen lassen. Dabei spielt es keine Rolle, auf welcher Architekturebene sich die Elemente befinden. Ergänzend fügt Richardson [Ri19] hinzu, dass sich die Softwarebausteine in nur eine ausführbare Einheit zusammenfügen. Damit ist eine Unterteilung auf Systemarchitekturebene möglich, sodass ein Subsystem zwar ausführbar ist, aber dennoch in Abhängigkeit zu den anderen Subsystemen steht. Ein Subsystem besitzt dabei eine eigene *Quellcodebasis* [Ne15]. Der Vorteil durch die eigene (große) Quellcodebasis ist, dass innerhalb des Subsystems auf die gesamte Funktionalität zugegriffen werden kann. Der Austausch von Informationen beruht bei ausführbaren Softwarebausteinen auf *lokalen Schnittstellen* wie Random-Access Memory-Speicher (RAM), Datenbanken (meist im selben Datenbankschema) oder einfachen Dateien [LX+13]. Insbesondere für

das Testen des Softwaresystems in der lokalen Entwicklungsumgebung der Softwareentwickler hat dies einen positiven Einfluss [PM+19].

Meist verfolgen Organisationen bei der Entwicklung von Monolithen einen *Schichtenarchitekturstil* und trennen damit auf der Systemarchitekturebene die Softwaresysteme typischerweise in drei Subsysteme [FL14]: (1) Benutzeroberfläche für die Client-seitige Ausführung, (2) Geschäftslogik für die Server-seitige Ausführung und (3) Datenhaltung für die Persistierung der Server-seitigen Daten. Aus der hohen Kopplung entsteht ein Gefüge zwischen den Softwarebausteinen, sodass die Benutzeroberfläche (1) auf Schnittstellen der Geschäftslogik zugreift, um die Funktionalitäten dem Benutzer zur Verfügung zu stellen. Die Geschäftslogik (2) beinhaltet die gesamte Funktionalität, weshalb die Benutzeroberfläche alleine keinen Mehrwert bietet. Zur Persistierung greift die Geschäftslogik auf Schnittstellen der Datenhaltung (3) zu, die meist in Form von Bibliotheken im Quellcode verwendet werden.

Newman [Ne19] kategorisiert den monolithischen Architekturstil weiterführend und beschreibt den möglichen Einsatz von drei unterschiedlichen Stilen:

Einzelner Prozess-Monolith Ein *einzelner Prozess-Monolith* (engl. single-process monolith) stellt die überwiegende Mehrheit der Monolithen dar. Zur Ausführungszeit läuft der gesamte Quellcode innerhalb eines einzelnen Prozesses ab, daher auch diese Bezeichnung. Ein Prozess ist dabei nicht gleichzusetzen mit einem Prozess auf Ebene des Betriebssystems, denn Newman ergänzt:

> In reality, these single-process systems can be simple distributed systems in their own right, as they nearly always end up reading data from or storing data into a database.

Somit orientiert sich der Begriff des Prozesses eher an Abläufen der Geschäftslogik inklusive der Benutzeroberfläche und Datenhaltung. Eine verfeinerte Variante dieses Monolithen ist der *modulare Monolith*, der das Softwaresystem in einzelne Module unterteilt. Die Module stellen dann die Softwarebausteine dar. Durch die Trennung ist das Arbeiten am Softwaresystem in größeren Teams einfacher. Dennoch sind die Softwarebausteine nicht unabhängig voneinander ausführbar. Für den modularen Monolithen werden für den restlichen Verlauf der Forschungsarbeit zwei Darstellungsweisen gewählt, die der Abbildung 2.7 zu entnehmen sind. Unterschieden wird zwischen einem gut modularisierten und *schlecht modularisierten Monolithen*. In der Zusammensetzung der Bausteine und im Quellcode des schlecht modularisierten Monolithen wurde das von Martin [Ma09] aufgestellte *SRP* nur unzureichend eingesetzt.

Verteilter Monolith Ein Softwaresystem kann aus vielen einzelnen ausführbaren Softwarebausteinen bestehen und dennoch eine monolithische Architektur aufweisen. Newman bezeichnet dies als *verteilten Monolithen*, denn die Bausteine des Softwaresystems sind zwar unabhängig ausführbar, aber um die Funktionalität zu erbringen, müssen sie trotzdem immer zusammen

bereitgestellt werden. Ein solches Softwaresystem vereint die Nachteile einer *serviceorientierten Architektur* (engl. Service-Oriented Architecture, SOA) und einer monolithischen Architektur, ohne die Vorteile beider einzubringen. Der Grund für die Entstehung verteilter Monolithen liegt in der Vernachlässigung grundlegender Konzepte wie Kohäsion verteilter Architekturen.

Black-Box Softwaresysteme Softwaresysteme, die eingekauft werden, um mit dem eigenen Softwaresystem zusammenzuarbeiten, werden von Newman als *Black-Box Softwaresysteme* von Drittanbietern (engl. third-party black-box software systems) bezeichnet. Der große Nachteil an solchen Softwaresystemen ist der fehlende Einfluss auf den Quellcode, denn dieser liegt in der Verantwortung des Drittanbieters.

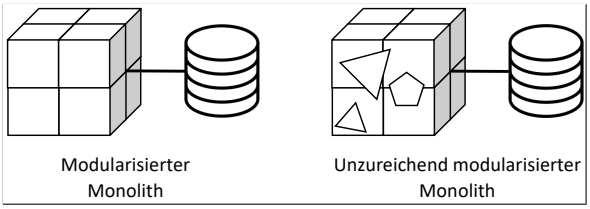

Abbildung 2.7: Ausprägungen eines modularisierten Monolithen aufbauend auf [Ne19]

Die Datenhaltung ist bei monolithischen Softwaresystemen eine zentrale Anlaufstelle, um Informationen auszutauschen (Abbildung 2.7). In der Regel besitzt ein Monolith eine Datenbank, in der Daten von allen Softwarebausteinen abgelegt werden. Meist finden sich auch *Dinge* in demselben Datenbankschema wieder, obwohl diese keine Kohäsion aufweisen [Ne15]. Ein Ding beschreibt dabei eine Abstraktion eines Objektes der realen Welt, was im Rahmen des domänengetriebenen Entwurfs (engl. Domain-Driven Design, DDD) als *Domänenobjekt* (vgl. Abschnitt 2.4) bezeichnet wird. Im restlichen Verlauf dieser Forschungsarbeit wird der Begriff des Domänenobjekts verwendet, auch wenn kein direkter Bezug zu dem Softwareentwicklungsansatz DDD besteht.

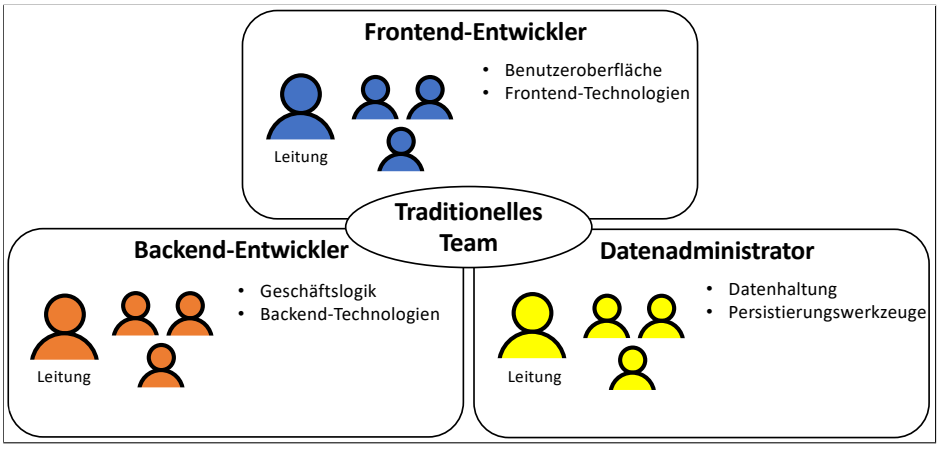

Abbildung 2.8: Traditionelle Aufteilung des Entwicklungsteams nach [Ne19]

Neben dem Softwaresystem betrifft der Architekturstil auch die Struktur des Softwareentwicklungsteams. Conway [Co68] stellte das gleichnamige Gesetz *Conway's Law* auf:

> Any organization that designs a system ... will inevitably produce a design whose structure is a copy of the organization's communication structure

Traditionell ist ein Entwicklungsteam nach den Kompetenzen Frontend-, Backend- und Datenhaltung-Technologien aufgeteilt (Abbildung 2.8) [BH+16, Ne19]. Somit entspricht die Verteilung des Monolithen in der physischen Architektur der Struktur des Entwicklungsteams.

Microservice-Architektur

Um den steigenden Anforderungen an das Softwaresystem und auch an das Entwicklungsteam gerecht zu werden sind verteilte Architekturen notwendig, die gleichzeitig effektiv in der Cloud angewandt werden können. An großem Interesse hat in den vergangenen Jahren die *Microservice-Architektur* gewonnen, ein Architekturstil, der Zimmermann [Zi17] zufolge eine Ausprägung der SOA darstellt.

Eine erste Definition dieses Architekturstils haben Fowler und Lewis [FL14] veröffentlicht:

> The microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.

Aus dieser Definition kann abgeleitet werden, dass ein Softwaresystem aus einer Vielzahl von autonomen Softwarebausteinen besteht, die auf den Makroarchitekturebenen der Systeme und Bausteine anzusiedeln sind. Diese Softwarebausteine werden als *Microservices* bezeichnet [DG+17]. Im Gegensatz zu dem monolithischen Architekturstil bedingt die Microservice-Architektur eine strikte Trennung der Bausteine sowohl in der logischen als auch in der physischen Architektur. Denn während ein Monolith die Trennung auf Ebene von Modulen oder Klassen anstrebt, trennt die Microservice-Architektur die Funktionalität in autonome Subsysteme auf [BP19]. Daraus folgt, dass für die Erbringung der gesamten Funktionalitäten des Softwaresystems eine Komposition der Microservices durchgeführt werden muss [DF+16]. Newman [Ne15] stellt hierfür zwei Arten der Komposition vor: (1) *Orchestrierung* und (2) *Choreographie*. Als weitere Stärke der Microservice-Architektur benennen Dragoni et al. [DL+17] die Skalierbarkeit, die im Gegensatz zur monolithischen Architektur das Skalieren der einzelnen Microservices erlaubt.

Zentral für den Einsatz einer Microservice-Architektur ist eine Definition der grundlegenden Microservices, denen Bruce und Pereira [BP19] vier grundlegende Prinzipien zusprechen: (1) *Autonomie*, (2) *Resilienz*, (3) *Transparenz* und (4) *Automation*. Giessler [Gi18] definiert ergänzend, dass ein Microservice ein autonomer Softwarebaustein ist, der genau einen festgelegten (3) minimalen Ausschnitt der Geschäftslogik mit hoher Kohäsion (1) abbildet und über wohldefinierte Web-Schnittstellen (2)

zur Interaktion bereitstellt. Die Kapselung der Geschäftslogik in kohäsive Ausschnitte entspricht der Idee des SRP von Martin [Ma09]:

> Gather together those things that change for the same reasons. Separate those things that change for different reasons.

Auch in diesem Kontext referenziert ein *Ding* auf ein Domänenobjekt. Aus der losen Kopplung der Microservices folgt die vollständig automatisierbare (4) Bereitstellbarkeit. Weitere Eigenschaften der Microservices sind die Eignung für das automatisierte Testen und für das anschließend automatisierte Veröffentlichen (engl. deployment) [DG+17] und eine effiziente und effektive Weiterentwicklung und Wartung durch eine zwangsläufig kleine Codebasis [Ne15].

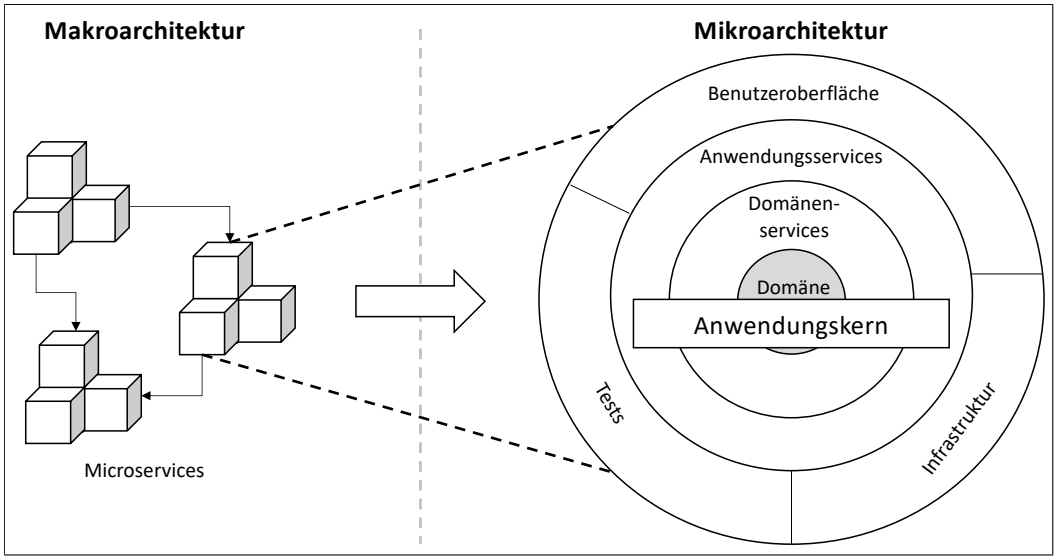

Abbildung 2.9: Mikro- und Makroarchitektur der logischen Architektur des Softwaresystems

Ebenfalls wird der Gedanke, innerhalb eines Microservices kohäsive Geschäftslogik abzubilden, von der Mikroarchitektur des Softwarebausteins getragen. Die Mikroarchitektur beruht zunächst auf dem Architekturmuster der *Schichtenarchitektur* (engl. layered architecture), die durch Evans [Ev04] geprägt wurde (Abbildung 2.9). Das Architekturmuster trennt einen Microservice in die vier Schichten: (1) *Präsentation*, (2) *Anwendung*, (3) *Domäne* und (4) *Infrastruktur*. Vernon [Ve13] sieht jedoch bei der Implementierung der Schichtenarchitektur Probleme, weshalb er für die Mikroarchitektur eine von Cockburn [Co05] vorgestellte *hexagonale Architektur* (engl. hexagonal architecture) vorsieht. Eine weitere Alternative zur Schichtenarchitektur stellt die von Palermo [Pa08] vorgestellte *Zwiebelarchitektur* (engl. onion architecture) dar, die auch im Kontext der Microservices Anwendung findet. Abbildung 2.9 zeigt die Schichten der Zwiebelarchitektur. Die beiden Architekturmuster sind stark miteinander verwandt. Für die in der Forschungsarbeit entworfenen Microservices wird die Zwiebelarchitektur als Architekturmuster der Mikroarchitektur festgelegt.

Entgegen eines Monolithen verfolgt die Microservice-Architektur den Ansatz, dass kleine, ebenfalls *autonome Entwicklungsteams* gebildet werden, welche die Verantwortung für den gesamten Lebenszyklus des Microservices tragen [Ne19]. Übertragen auf *Conway's Law* [Co68] bedeutet dies, dass der Architekturstil einen Einfluss auf die organisatorischen Strukturen hat und nicht umgekehrt. Die Frage nach der Größe des sogenannten *Service-Teams* kann nicht eindeutig beantwortet werden, es existieren jedoch gewisse Ansätze wie der von Amazon geprägte Ansatz der *Two Pizza-Teams* [Wi17]. Zur sinnvollen Verteilung der Fähigkeiten und Erfahrungen innerhalb des Service-Teams kann das Prinzip der *funktionsübergreifenden Teams* (engl. cross-functional teams) aus der *Development & Operations-Bewegung* (DevOps) herangezogen werden. Bass et al. [BW+15] nennen Vorteile der kleinen Service-Teams und geben Empfehlungen zur Strukturierung (Abbildung 2.10).

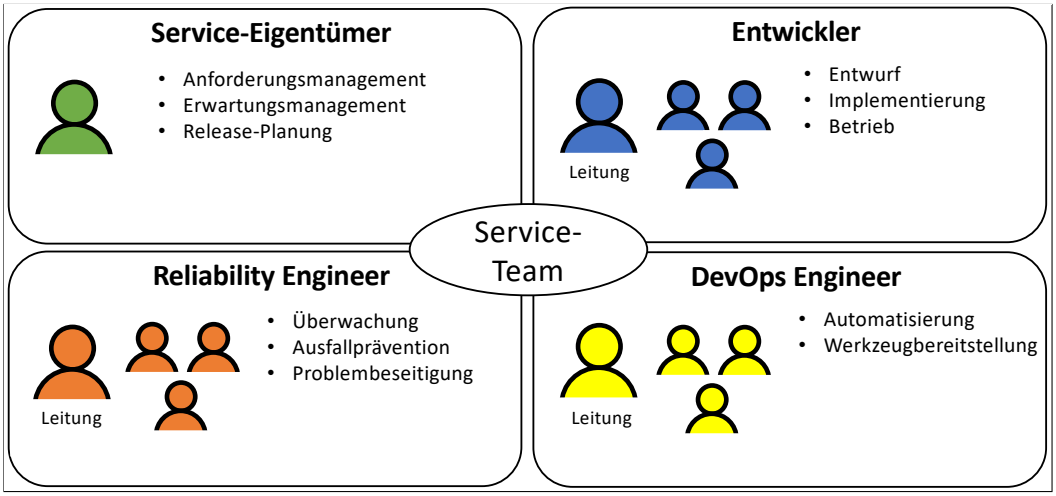

Abbildung 2.10: Rollenverteilung des Service-Teams nach [BW+15]

2.3 Verhaltensgetriebene Entwicklung

Bei der iterativen Migration eines Softwaresystems ist das Verhalten der Software bei der Betrachtung zweier Aspekte relevant: (1) Identifikation von Softwarebausteinen und (2) Ist- und Soll-Zustand des Softwaresystems. Ausgehend von ihrem Verhalten können Softwarebausteine im Monolithen identifiziert werden. Zudem ist über das Verhalten eine *Verifizierung* des Ist- und Soll-Zustandes nach der Migration möglich, denn das Softwaresystem soll den Benutzern weiterhin dieselben Funktionalitäten zur Verfügung stellen. Im Kontext dieser Forschungsarbeit wird für die Betrachtung des Verhaltens der Softwareentwicklungsansatz der *verhaltensgetriebenen Entwicklung* (engl. Behavior-Driven Development, BDD) verwendet.

Ursprünglich wurde BDD von North [No06] vorgestellt, der aufbauend auf dem Softwareentwicklungsansatz der *testgetriebenen Entwicklung* (engl. Test-Driven Development, TDD) Prinzipien für

das *Ende-zu-Ende-Testen* von Softwaresystemen aufgestellt hat. Weitere wichtige Beiträge wurden auch von den Arbeiten von Keogh [Ke11] und Smart [Sm14] getätigt.

Mit BDD wird das Ziel verfolgt, die unterschiedlichen Sichten der Interessenvertreter und Softwareentwickler bei der Erhebung des Verhaltens in Form von Anforderungen zu vereinen. Primär geht es um das kollaborative Erheben des angestrebten Verhaltens, um ein einheitliches Verständnis unter allen Projektmitgliedern sicherzustellen. Dabei wird auf ein einheitliches, nicht-technisches und durch die Geschäftsdomäne beeinflusstes Vokabular gesetzt. Das erhobene Verhalten wird in der nachgelagerten Implementierungsphase des Softwaresystems zum automatisierten Testen herangezogen. So wird das Verhalten in der Implementierung des Softwaresystems verankert.

2.3.1 Grundsätzliches Vorgehen

BDD ist ein agiler Softwareentwicklungsansatz [No09]. Von dem agilen Vorgehen geht ein hoher Grad an Kollaboration mit dem Benutzer beziehungsweise den Interessenvertretern aus. So ist das Ziel, dass das Entwicklungsteam gemeinsam mit den Interessenvertretern das Verhalten des Softwaresystems spezifiziert. Die Spezifikation erfolgt in Form von sogenannten Gherkin Features, die mit der natürlichen Sprache und vordefinierten Schlüsselwörtern formuliert werden [Sm14]. Ist das Verhalten definiert, gilt es, die spezifizierten Gherkin Features als automatisierte Tests im Quellcode zu verankern. Durch die Verankerung des spezifizierten Verhaltens im Quellcode wird bei der Entwicklung sichergestellt, dass das Softwaresystem das gewünschte Verhalten zeigt. Die Verknüpfung von spezifiziertem Verhalten und Quellcode wird als *lebendige Dokumentation* (engl. living documentation) bezeichnet [Ad11].

Überwiegend stimmen die Prinzipien und Vorgehensweisen von BDD mit denen von TDD miteinander überein. Die Besonderheit von BDD ist jedoch der Blickwinkel auf das erhobene Verhalten. Während TDD in sich abgeschlossenes Verhalten auf Ebene von Softwarebausteinen wie Klassen oder Methoden betrachtet, greift BDD die *Benutzersicht* auf [Ke11]. Sämtliches Verhalten, das der Benutzer im Softwaresystem benötigt, wird durch BDD in *Gherkin Features* überführt. Der Unterschied zu TDD ist damit, dass keine technische Sicht auf das Verhalten eingenommen wird, sondern eine Sicht ausgehend von dem Geschäftswert, den es für die Benutzer hat.

Ein weiterer Vorteil, neben der Zentralisierung des Geschäftswerts, ist die durch die Benutzersicht geführte Erhebung des Verhaltens. Durch die Simulation der Benutzersicht ist ein *roter Faden* bei der Erhebung des Verhaltens zu erkennen. Das Benutzerverhalten wird so, North [No09] zufolge, von *außen nach innen* (engl. outside-in) erhoben. Dies dient auch dem Verständnis der erhobenen Gherkin Features, denn anfänglich wird ein hohes Abstraktionsniveau für das betrachtete Verhalten eingenommen. Kontinuierlich wird dann das Abstraktionsniveau für jedes weitere Gherkin Feature reduziert.

Bei der Erhebung des Verhaltens und der Spezifikation der Gherkin Features sieht das agile Vorgehen einen kollaborativen und iterativen Prozess vor [Sm14]. Interessenvertreter und Softwareentwickler spezifizieren gemeinsam Gherkin Features. Dabei wird, wie bereits durch *outside-in* beschrieben, mit dem offensichtlichsten Benutzerverhalten begonnen. Anschließend wird dieses implementiert, bis neues Verhalten erneut gemeinsam spezifiziert werden muss. Das Verhalten innerhalb eines Gherkin Features wird in Form von *konkreten Beispielen* beschrieben, weshalb gerade die Kollaboration mit den Interessenvertretern notwendig ist, da nur diese reale Beispiele einbringen können. Wichtig bei der Kollaboration ist weiterführend auch die Verwendung einer einheitlichen Sprache, der *ubiquitären Sprache* (vgl. Abschnitt 2.4.1).

Ein Gherkin Feature ist zunächst eine textuelle Beschreibung des Verhaltens. Als automatisierter Test entspricht das Gherkin Feature entweder der Form eines Integrations- oder der eines Unit-Tests [No09]. Dadurch entsteht die bereits angesprochene *living documentation*. Der Vorteil der *living documentation* ist die Wechselwirkung bei Änderungen an dem Verhalten oder dem Quellcode. Unabhängig davon, ob Verhalten oder der Quellcode geändert wird, durch die automatisierten Tests wird eine Korrektheit erzwungen. Dies bedingt jedoch eine hohe Disziplin des Entwicklungsteams beim Testen in Form der Implementierung von neuem Verhalten.

2.3.2 Abbildung von funktionalen Anforderungen

Das durch BDD betrachtete Verhalten wird im Kontext des Requirements Engineering auf funktionale Anforderungen abgebildet. Dabei beschreibt eine funktionale Anforderung, wie das Verhalten im Softwaresystem implementiert werden muss. Eine implementierte funktionale Anforderung wird dann als Funktionalität des Softwaresystems bezeichnet.

Der Aufbau der Gherkin Features richtet sich zunächst nach dem Aufbau einer *User Story* der objektorientierten Analyse [No07]. Weiterführend werden jedoch für das Gherkin Feature noch *Kontextinformationen* spezifiziert, welche die Ausgangssituation des Benutzers beschreiben. An dieser Stelle wird die Benutzersicht verstärkt eingebracht.

Ein beispielhaftes Gherkin Feature kann dem Listing 2.1 entnommen werden. Am Anfang wird die funktionale Anforderung als User Story festgehalten. Anschließend werden für die funktionale Anforderung der Geschäftsdomäne entsprechende Beispiele in Form von *Szenarien* beschrieben. Ein Szenario kann gleichzeitig als Akzeptanzkriterium verstanden werden, welches aus sogenannten *Schrittdefinitionen* (engl. step definitions) besteht. Eine Schrittdefinition wird zudem später in der Implementierung als Test realisiert. Zur Beschreibung eines Szenarios werden die Schlüsselwörter "*given*", "*when*" und "*then*" vorangestellt. Mit "*given*" werden Vorbedingungen des Szenarios festgelegt. Eine Benutzerinteraktion mit dem Softwaresystem wird durch "*when*" gekennzeichnet. Das aus der Benutzerinteraktion resultierende Ergebnis wird durch "*then*" beschrieben.

```
Feature: Uploading a file
               as an asset manager
               I want to upload one or more files to a fund or building
               So that I have access to it at any time

        Background:
            Given I am in the context of a specific fund or building
              And I am on the page for managing documents

        Scenario: Upload a single file
            Given I have a single file
             When I drag the single file to the page
             Then the upload of the file is started
              And it is assigned to the fund or building

        Scenario: Upload multiple files
            Given I have several files
             When I drag the files together onto the page
             Then the upload of the files is started
              And they are assigned to the fund or building
```

Quelltext 2.1: Gherkin Feature zur Versionierung und Historisierung der Dateien im Dokumentenmanagementsystem

2.4 Domänengetriebener Entwurf

Ein weiterer grundlegender Softwareentwicklungsansatz ist *DDD*, der besonders im Kontext des Entwurfs einer Microservice-Architektur an großer Bedeutung gewonnen hat. Um bei der Migration eines monolithischen Softwaresystems eine Ausrichtung der Microservice-Architektur an der Geschäftsdomäne zu erreichen wird DDD zentral in der Migration verankert. Nachfolgend werden die Konzepte, Praktiken und Prinzipien von DDD, die bei der Migration zum Tragen kommen, detailliert vorgestellt.

Der Softwareentwicklungsansatz findet zeitlich vor der Microservice-Architektur seinen Ursprung und wurde von Evans [Ev04] vorgestellt. Auf Basis der von Evans gelegten Grundlage zu DDD wurden weitere bedeutende Arbeiten [Ve13, MT15, Tu20] verfasst. Diese werden im weiteren Verlauf ebenfalls für die Definition von DDD herangezogen.

Das Ziel von DDD ist es, während der Entwurfsphase eines Softwaresystems den Fokus auf die *Geschäftsdomäne* der Organisation zu legen und explizit eine Repräsentation der Geschäftsdomäne in

einem *Domänenmodell* festzuhalten. Dieses Domänenmodell wird mit der Implementierung des Softwaresystems verknüpft. So wird sichergestellt, dass das Softwaresystem durch das Berücksichtigen der Geschäftsdomäne den Anforderungen der jeweiligen Organisation entspricht. Eine Organisation bezeichnet dabei einen Zusammenschluss von Interessenvertretern, die Anforderungen an ein Softwaresystem stellen, um Geschäftsabläufe durch dieses Softwaresystem zu unterstützen. Die Organisation kann dabei das Softwaresystem selbst oder durch einen Dritten entwickeln lassen.

2.4.1 Grundsätzliches Vorgehen

Das grundsätzliche Vorgehen definiert Evans [Ev04] als Exploration der Geschäftsdomäne anhand einer *Domänenmodellierung*. Abbildung 2.11 fasst dieses Vorgehen zusammen. Die Domänenmodellierung erfolgt kollaborativ mit sogenannten *Domänenexperten* und den Softwareentwicklern, welche die technische Verantwortung für das Softwaresystem haben. Das Ergebnis der Domänenmodellierung wird schließlich in einem sogenannten Domänenmodell festgehalten. Das explorative und kollaborative Modellieren wird als *Knowledge Crunching* bezeichnet.

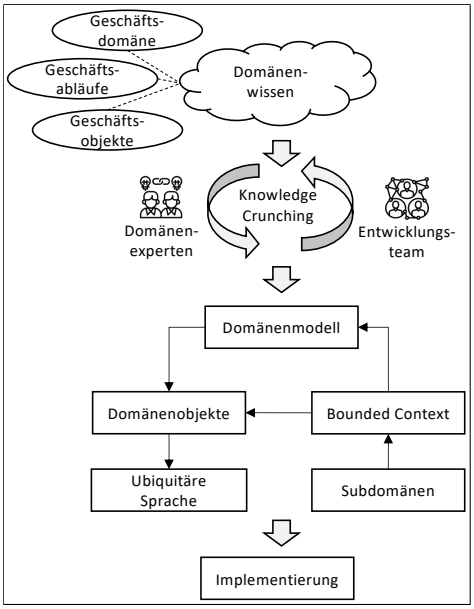

Abbildung 2.11: Grundsätzliche Vorgehensweise des domänengetriebenen Entwurfs

Nach der Definition von Evans [Ev04] ist das Domänenmodell zunächst als Container für eine Sammlung von Informationen zu sehen. Als Information zählt alles, was den Domänenexperten und Softwareentwicklern hilft, die Geschäftsdomäne zu verstehen. Folglich ist das Domänenmodell eine Sammlung an formalen und informellen Modellen, die das Wissen der Geschäftsdomäne repräsentieren. Es werden nicht alle Aspekte der Geschäftsdomäne in das Domänenmodell aufgenommen. Relevant sind die Informationen, die unmittelbar mit der Entwicklung des Softwaresystems und den

Tabelle 2.1: Stereotypen der Domänenobjekte nach [Ev04, Ve13]

Stereotyp	Beschreibung
Entität	Eine *Entität* (engl. entity) ist ein Domänenobjekt, welches nicht über die Attribute beziehungsweise Werte der Attribute identifiziert wird. Stattdessen besitzt die Entität eine explizite Identität, über die das Domänenobjekt eindeutig identifiziert werden kann. Es handelt sich nur dann um dieselbe Entität, wenn die Identität übereinstimmt.
Werteobjekt	Das *Werteobjekt* (engl. value object) besitzt keine explizite Identität. Ein Werteobjekt ist dann identisch mit einem anderen Werteobjekt, wenn die Attribute dieselben Werte besitzen.
Aggregat	*Aggregate* beschreiben Domänenobjekte, die neben eigenen Attributen auch durch Beziehungen zu anderen Domänenobjekten ein Ganzes ergeben. Über die Beziehungen zu den anderen Domänenobjekten forciert das Aggregat Invarianten, welche im Kontext der Geschäftsdomäne bestehen.
Repository	Ein *Repository* ist ein Domänenobjekt-spezifischer Softwarebaustein, der zum Zugriff auf die Infrastrukturschicht wie beispielsweise der Datenhaltung dient. Über das Repository werden Instanzen des betreffenden Domänenobjekts erstellt.
Domänen-Service	Geschäftslogik, die nicht eindeutig einem Domänenobjekt zugeordnet werden kann, kann in einem *Domänen-Service* (engl. domain service) der Domänenschicht hinzugefügt werden.
Domänenereignis	Ein *Domänenereignis* (engl. domain event) stellt explizit eine Zustandsänderung eines Domänenobjektes dar. Zustandsänderungen werden durch die Anwendungs- oder Geschäftslogik hervorgerufen.

damit unterstützten Geschäftsabläufen in Verbindung stehen und somit auch für die Erfüllung der Bedürfnisse der Interessenvertreter notwendig sind. So entsteht ein geeigneter Ausschnitt bei der Betrachtung der Geschäftsdomäne, sodass eine Beherrschbarkeit der Komplexität möglich ist.

Die relevanten Informationen innerhalb des Domänenmodells werden als *Domänenwissen* (engl. domain knowledge) bezeichnet. Das Domänenwissen wird durch die sogenannten *Domänenobjekte* (engl. domain objects) und die Beziehungen zwischen diesen repräsentiert. Ein Domänenobjekt ist dabei zunächst ein Modell eines in der realen Welt auftretenden Geschäftsobjekts. Geschäftsobjekte treten dabei in allen Aspekten der Organisation auf, aber als Hauptquelle werden Geschäftsabläufe gesehen, die gerade durch das Softwaresystem unterstützt werden sollen. DDD führt *unterschiedliche Stereotypen* ein, um einen Mehrwert für die Implementierung der Domänenobjekte zu schaffen. Die Stereotypen werden in Tabelle 2.1 erläutert.

Die Orientierung des Softwaresystems an der Geschäftsdomäne erfolgt durch die Verknüpfung der Implementierung mit dem Domänenmodell. Zunächst führt DDD dafür den Architekturstil der *Schichtenarchitektur* (engl. layered architecture) ein, welche die *Domänenlogik* von der restlichen Implementierung abgrenzt (Abbildung 2.12). Das Domänenmodell beschreibt dabei nur die Inhalte der *Domänenlogikschicht* (engl. domain logic layer). Vorgesehen ist eine genaue Abbildung des Domänenmodells in dieser Domänenlogikschicht, sodass die Struktur der Geschäftsdomäne identisch in der Implementierung vorzufinden ist. Durch die Strukturierung der Informationen anhand von

Domänenobjekten bietet es sich bei der Implementierung des Softwaresystems an, dem Paradigma der objektorientierten Programmierung zu folgen. Der grundsätzliche Gedanke der Klassen entspricht dem der Domänenobjekte.

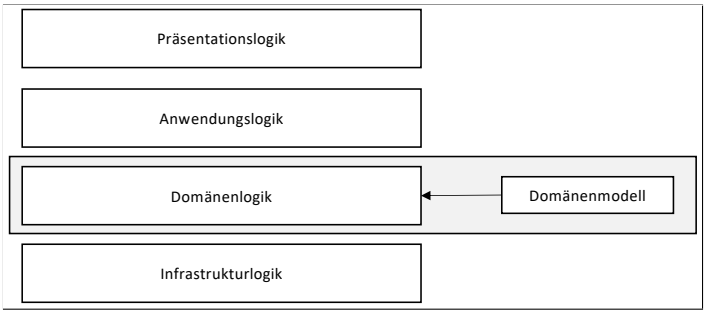

Abbildung 2.12: Architekturstil der Schichtenarchitektur nach dem domänengetriebenen Entwurf [Ev04]

Ein weiterer Effekt des Domänenmodells, neben der Verankerung in der Implementierung, ist die Festlegung einer *ubiquitären Sprache* (engl. ubiquitous language). Die ubiquitäre Sprache fungiert als Vertrag zwischen allen Projektmitgliedern, der die Bedeutung und den Gebrauch von Begrifflichkeiten regelt. So wird eine gemeinsame Sprache erreicht.

Die Geschäftsdomäne wird jedoch nicht nur auf einer mikroskopischen, sondern auch auf einer makroskopischen Ebene betrachtet. Während bei der mikroskopischen Betrachtung die Domänenobjekte und deren Beziehungen untereinander von Relevanz sind, wird bei einer makroskopischen Sicht die *Domänenstruktur* beleuchtet. Unter der Domänenstruktur wird die hierarchische Zusammensetzung der Geschäftsdomäne in eine oder mehrere *Subdomänen* (engl. subdomains) zusammengefasst. Durch den hierarchischen Ansatz können Subdomänen wiederum durch eine oder mehrere Subdomänen untergliedert werden. Ein weiteres Element der Domänenstruktur ist ein *Bounded Context*, der die Gültigkeit eines Domänenmodells und der ubiquitären Sprache explizit kennzeichnet. Gleichzeitig ist der Bounded Context ein autonomer und abgeschlossener Bestandteil der Geschäftsdomäne. Aus diesem Grund misst Newman [Ne15] dem Konzept der Bounded Contexts für den Entwurf einer Microservice-Architektur eine hohe Bedeutung zu und definiert eine Abbildung zwischen Bounded Context und Kandidaten für einen Microservice.

2.4.2 Geschäftsdomäne

Die Abbildung 2.13 zeigt die Bestandteile einer Geschäftsdomäne zum Zwecke einer Definition. Evans [Ev14] definiert eine *Geschäftsdomäne*, auch nur Domäne bezeichnet, als

> A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.

Ausgehend von dieser Definition besteht eine Domäne aus (1) *Wissen*, (2) *inneren Einflüssen* und (3) *Abläufen*, die als in sich abgeschlossen anzusehen sind. Ein Softwaresystem kann im Kontext einer Domäne angesiedelt werden, sodass die Bestandteile der Domäne grundlegend für die Funktionsweise des Softwaresystems sind.

Vernon [Ve13] beschreibt eine Domäne als

> ... what an organization does and the world it does it in... each kind of organization has its own unique realm of know-how and way of doing things... Domain can refer to both the entire domain of the business, as well as just one core or supporting area of it.

Aus dieser Definition geht eine Abhängigkeit der Domäne von der Organisation hervor, die letztendlich innerhalb der Domäne das Wissen, die Einflüsse und die Abläufe beeinflusst. Dadurch ist die Domäne und das darin enthaltende Wissen *organisationsspezifisch*, auch wenn zwei Organisationen im selben Wirtschaftssegment agieren. Weiterhin ist hervorzuheben, dass eine Domäne in weitere Teilbereiche untergliedert werden kann. Diese Teilbereiche werden als *Subdomänen* bezeichnet.

Eine weitere Definition für eine Domäne liefert Bjørner [Bj06], die ihm zufolge ein Zielgegenstand eines Softwaresystems ist. Bjørner führt daher auch als Synonym für die Domäne den Begriff der Anwendungsdomäne ein. Zum Zeitpunkt der Softwareentwicklung liefert die Domäne eine Menge an allgemeingültigen *Anforderungen*, *Terminologien* und *Funktionalitäten*, die im Softwaresystem berücksichtigt werden müssen. Nur dann kann das richtige Softwaresystem entwickelt werden.

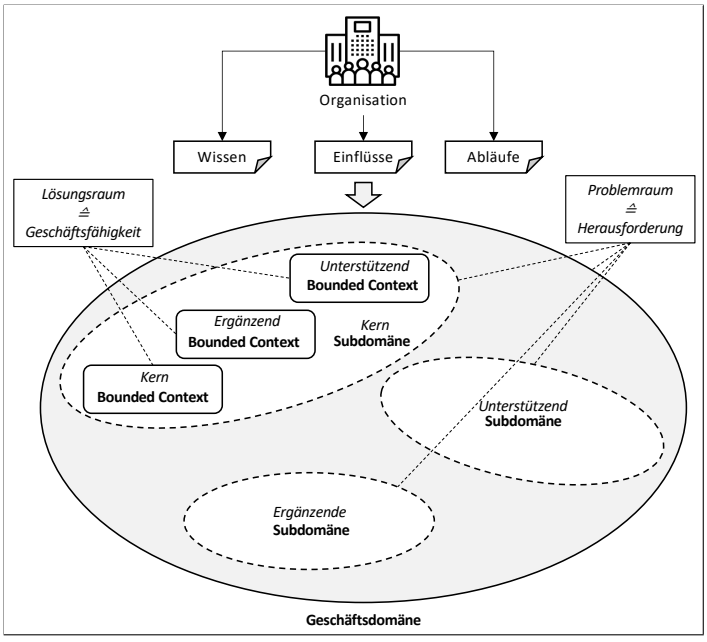

Abbildung 2.13: Grundlegender Aufbau einer Domäne und Zusammenhang zu Geschäftsfähigkeiten

Ausgehend von diesen bestehenden Definitionen leitet sich die Definition einer Geschäftsdomäne für diese Forschungsarbeit wie folgt ab:

> Eine *Geschäftsdomäne* ist ein in sich abgeschlossener Bereich, der organisationsspezifisches Wissen, Einflüsse und Abläufe beinhaltet, die dogmatisch bei der Entwicklung eines Softwaresystems berücksichtigt werden müssen, um das richtige Softwaresystem entwickeln zu können.

2.4.3 Geschäftsfähigkeiten

Das Wissen, die Einflüsse und Abläufe einer Geschäftsdomäne zielen darauf ab, einen bestimmten wirtschaftlichen Mehrwert für die Organisation oder deren Kunden zu erzielen. Die Fähigkeit, einen solchen Mehrwert zu erzielen, wird als *Geschäftsfähigkeit* (engl. business capability) bezeichnet [Le95]. Eine Geschäftsfähigkeit ist dabei als fraktal zu betrachten, denn sie kann aus mehreren *Subgeschäftsfähigkeiten* bestehen, die zusammengesetzt den eigentlichen Mehrwert erzielen [Tu18]. Abbildung 2.13 zeigt die Einordnung der Geschäftsfähigkeiten in das Gesamtbild der Geschäftsdomäne.

Die Verbindung zwischen einer Geschäftsfähigkeit und der Domäne schafft Vernon [Ve13] durch die Einführung der Begriffe *Problemraum* (engl. problem space) und *Lösungsraum* (engl. solution space). Der Problemraum beschreibt Herausforderungen, die bei einer Lösung eine wirtschaftliche Relevanz besitzen. Dagegen beschreibt der Lösungsraum konkrete Lösungsansätze zur Beherrschung der Herausforderungen des Problemraums. Im Kontext von DDD wird der Lösungsraum durch die Bounded Contexts und die darin befindlichen Domänenmodelle beschrieben.

Eine wichtige Abgrenzung zwischen Typen von Geschäftsfähigkeiten führte Leonard-Barton [Le95] ein. Unterschieden wird zwischen *Kernfähigkeiten* (engl. core capabilities), *ergänzenden Fähigkeiten* (engl. supplemental capabilities) und *unterstützenden Fähigkeiten* (engl. enabling capabilities). Eine Kernfähigkeit stellt einen zentralen Geschäftswert für eine Organisation dar, deren Problemraum schwer zu beherrschen und gleichzeitig schwer nachzuahmen ist. Ergänzende Fähigkeiten liefern ebenfalls einen Geschäftswert, der jedoch nur als ergänzend anzusehen ist und gleichzeitig einen einfachen Problemraum besitzt. Die unterstützenden Fähigkeiten stellen keinen Geschäftswert dar, sondern sind notwendig für die Erbringung der anderen Geschäftsfähigkeiten.

Eine solche Klassifikation in “Kern”, “ergänzend” und “unterstützend” findet sich ebenfalls bei der Betrachtung der Geschäftsdomäne und der Subdomänen. Als Muster von DDD führt Vernon [Ve13] analog zu dieser Klassifikation der Geschäftsfähigkeiten eine Klassifikation der Subdomänen ein. Es folgt aus der Klassifikation der Subdomänen und Geschäftsfähigkeiten eine implizite Klassifikation des Lösungsraums. Weiterführend kann die Klassifikation auf das Domänenmodell und die damit

verbundenen Domänenobjekte angewendet werden, sodass die Bedeutung eines Domänenobjekts für den Geschäftswert einer Organisation bestimmt werden kann.

2.4.4 Modellierung der Domäne

Der domänengetriebene Entwurf führt bei der Betrachtung der Geschäftsdomäne zwei unterschiedliche Sichten ein, die auf unterschiedlichen Abstraktionsniveaus die Struktur der Geschäftsdomäne betrachten [Ev04]: (1) *strategische Sicht* und (2) *taktische Sicht*. Beide Sichten werden durch eine Modellierung in das Domänenmodell überführt.

Die strategische Sicht verfolgt das Ziel, dem Entwicklungsteam einen Überblick über die gesamte Geschäftsdomäne zu verschaffen. Zur Beschreibung der Geschäftsdomäne wird ein hohes Abstraktionsniveau eingenommen, um zunächst nur die strukturgebenden Aspekte zu berücksichtigen. In Abbildung 2.13 wird eine solche strategische Sicht allgemein auf eine Geschäftsdomäne eingenommen. Somit werden für die Geschäftsdomäne alle Subdomänen und Bounded Contexts dargestellt. Weiterführend werden zwischen den Bounded Contexts *Beziehungen* eingeführt, um notwendige Verknüpfungen zur Lösung eines Problems explizit darstellen zu können. Hierfür führt Vernon [Ve13] verschiedene Typen von Beziehungen ein. Gleichzeitig werden auch Lösungen durch die expliziten Grenzen der Bounded Contexts voneinander abgegrenzt.

Die Aktivität der Modellierung aus strategischer Sicht wird als *strategische Modellierung* bezeichnet. Eine solche strategische Modellierung ist ebenfalls von dem Knowledge Crunching abhängig, was dazu führt, dass die strategische Sicht kollaborativ zwischen Domänenexperten und Entwicklungsteam modelliert wird. Zum Festhalten der strategischen Sicht, die modelliert auch als *strategischer Entwurf* bezeichnet wird, führt DDD die sogenannte *Context Map* ein. Die Context Map ist das einzige Modell des Domänenmodells, welches explizit von Evans [Ev04] eingeführt wird. Die Modellierungselemente der Context Map entsprechen den Bestandteilen der Domänenstruktur. Eine formale Modellierungssprache für die Context Map wird jedoch nicht eingeführt.

Der strategische Entwurf nimmt im Kontext der Microservice-Architekturen eine wichtige Rolle ein. Newman [Ne15] zufolge stellen die modellierten Bounded Contexts Kandidaten für Microservices dar, da ein Bounded Context in sich abgeschlossenes Domänenwissen kapselt. Weiterführend können die Beziehungen zwischen den Bounded Context als Abhängigkeiten zwischen Microservices und den verantwortlichen Service-Teams betrachtet werden. Aus den Beziehungen kann zudem die Richtung des Informationsflusses zwischen zwei Microservices bestimmt werden, die Vernon [Ve13] als *Upstream*- und *Downstream*-Partei bezeichnet.

Das in den Bounded Context gekapselte Domänenwissen wird durch die taktische Sicht beschrieben, die ein wesentlich geringeres Abstraktionsniveau einnimmt und bereits Bezug auf die Implementierung des Domänenwissens nimmt. Gleichzeitig bedeutet dies, aufgrund der expliziten Kapselung von

Domänenwissen eines Bounded Contexts, dass in jedem Bounded Context eine eigene taktische Sicht auf das Domänenwissen eingenommen wird. Die taktische Sicht vermittelt durch die Domänenobjekte die Funktionsweise der durch den Bounded Context beschriebenen Lösung.

Die Funktionalität der taktischen Sicht wird durch Domänenobjekte beschrieben. Ein Domänenobjekt beherbergt strukturelle Informationen und Verhalten von Geschäftsobjekten. Über Beziehungen zwischen den Domänenobjekten entsteht ein zusammenhängendes Gebilde, das letztendlich die Gesamtfunktionalität des Bounded Context darstellt.

Die taktische Sicht wird durch eine *taktische Modellierung* in einen *taktischen Entwurf* überführt. Im Gegensatz zu der strategischen Modellierung führt DDD kein explizites Modell für die taktische Sicht ein. Bestehende Arbeiten wie Evans [Ev04] und Vernon [Ve13] modellieren den taktischen Entwurf in einem UML Klassendiagramm. Jedoch hält Evans fest, dass jegliche Modellierungsformen für den taktischen Entwurf möglich sind, solange das Domänenwissen transportiert wird.

Um das Domänenwissen im Softwaresystem beziehungsweise in dem Microservice zu verankern muss der taktische Entwurf im Quellcode abgebildet werden. Hierfür bietet sich das Paradigma der objektorientierten Programmierung an, das es erlaubt, die Domänenobjekte in Form von Klassen direkt abzubilden.

3 Stand der Forschung

Das Durchführen der Migration eines monolithischen Softwaresystems ist ein in der Praxis relevantes Thema. Untermauert wird dies durch die hohe Anzahl an Arbeiten über bereits durchgeführte Migrationen. Einige relevante Arbeiten werden im Rahmen dieses Kapitels detailliert betrachtet. Zunächst werden die für eine iterative Migration grundlegenden Anforderungen definiert. Diese werden in einem Anforderungskatalog festgehalten und zur Bewertung der betrachteten Arbeiten herangezogen. Aus der Untersuchung bestehender Arbeiten kann letztendlich der Handlungsbedarf für die eigenen Forschungsbeiträge abgeleitet werden.

3.1 Anforderungen an die Migration von monolithischen Softwaresystemen in eine Microservice-Architektur

Die Anforderungen wurden anhand von Literaturrecherchen im Kontext des Forschungsgegenstands identifiziert. Es lassen sich sieben Anforderungen (A1 bis A7) benennen, welche sich den Beiträgen dieser Arbeit zuordnen lassen (Abbildung 3.1).

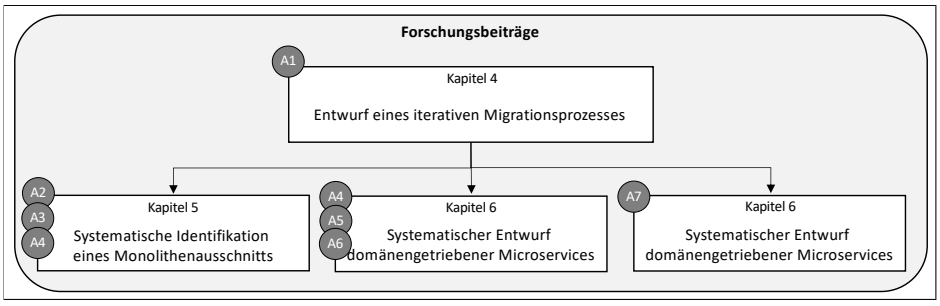

Abbildung 3.1: Zuordnung der Anforderungen zu den Forschungsbeiträgen der vorliegenden Arbeit

A1 - Systematischer und nachvollziehbarer Migrationsprozess

Eine Migration ist eine komplexe und zeitintensive Aufgabe, die durch ein *Migrationsrahmenwerk* (engl. migration framework) möglichst unterstützt werden sollte [TL+17]. Inhaltlich definiert ein Rahmenwerk Migrationsaktivitäten, ausführende Rollen, Softwareartefakte und sonstige Richtlinien.

Die erste Anforderung an einen iterativen Migrationsprozess für die Migration monolithischer Softwaresysteme in eine Microservice-Architektur ist es, der Definition eines Migrationsrahmenwerks zu entsprechen, damit eine systematische und nachvollziehbare Migration gewährleistet werden kann. Einflüsse auf die Migration wie finanzielle, zeitliche und personelle Situationen, sollten im Rahmenwerk berücksichtigt werden [KH18]. Die Systematik, die in Form eines Prozesses das Migrationsteam durch die Migration führt, ergibt sich aus den Migrationsaktivitäten. Nachvollziehbar wird die Migration, laut Schwarz [Sc12], durch den Einsatz formaler Softwareartefakte. Diese ermöglichen, Beziehungen zwischen zwei Entitäten des Softwaresystems oder der Entwicklung zu verfolgen. Die Nachvollziehbarkeit ist sehr eng mit der Systematik verbunden, denn eine systematische Ausführung von Schritten bedingt das nachvollziehbare Treffen von Entscheidungen in den Migrationsaktivitäten.

A2 - Wiederherstellung der Architekturbeschreibung

Nach der Modernisierung eines monolithischen Softwaresystems im Rahmen einer Migration muss die Funktionstüchtigkeit, wie sie vor der Migration vorgelegen hat, gewährleistet sein. Um jedoch die Funktionstüchtigkeit sicherstellen zu können, muss diese dem Migrationsteam bekannt sein. Aus diesem Grund wird an eine iterative Migration die Anforderung zur Wiederherstellung der Architekturbeschreibung gestellt. An dieser Stelle wird von einer Wiederherstellung gesprochen, denn monolithische Softwaresysteme zeichnen sich meist, wie Reussner und Hasselbring [RH06] feststellen, durch das Fehlen einer aktuellen Architekturbeschreibung aus. Um eine Allgemeingültigkeit des Migrationsrahmenwerks zu erreichen, muss daher jenes Fehlen angenommen werden, sodass eine Wiederherstellung der Architekturbeschreibung obligatorisch wird. Der Aufwand zur Wiederherstellung der Architekturbeschreibung sollte dabei angemessen sein, denn nach der Migration ist diese Architekturbeschreibung obsolet.

Die Architekturbeschreibung sollte möglichst auf formale Modellierungssprachen zurückgreifen, um das Verständnis und die Nachvollziehbarkeit zu erhöhen [RN+13]. Ein Beispiel hierfür ist die *Unified Modeling Language* (UML) [BD09], die für unterschiedliche Aspekte des Softwaresystems eine formale Modellierungssprache anbietet. Als Quellen für die Modelle der Architekturbeschreibung sollten, neben dem Quellcode des Monolithen [RH06], auch verantwortliche Softwareentwickler und Testbeschreibungen hinzugezogen werden [FL+18].

A3 - Identifikation kohäsiver Softwarebausteine

Die Grundlage eines iterativen Vorgehens ist das Bestimmen von Inkrementen, die innerhalb einer Iteration umgesetzt werden [Kr04, Bo06]. Bei der iterativen Migration eines monolithischen Softwaresystems umfasst eine Iteration einen Ausschnitt des Monolithen. Damit bei der Modernisierung die

Möglichkeit besteht, aus dem Ausschnitt des Monolithen Microservices ableiten zu können, müssen die Softwarebausteine innerhalb des Ausschnittes eine hohe Kohäsion aufweisen und gleichzeitig eine lose Kopplung zu den im Monolithen verbleibenden Softwarebausteinen besitzen. Die Bedeutung dieser Anforderung begründet sich durch die notwendige Autonomie der angestrebten Microservices [FL14]. Die Autonomie lässt sich durch die Kenngrößen Kopplung und Kohäsion bestimmen [NM+16].

A4 - Etablieren eines Domänenverständnisses

Den Hauptgrund für das Scheitern von Softwareentwicklungsprojekten sieht Evans [Ev04] bei einem fehlerhaften Verständnis der Softwareentwickler und der Interessenvertreter von der Geschäftsdomäne der Organisation, die das Softwaresystem verwendet oder verantwortet. Aus dem fehlenden Verständnis folgt eine ineffiziente und ineffektive Implementierung der Anforderungen mit einer Anhäufung von Softwareänderungen, welche den Aufwand zur Wartung des Softwaresystems erhöht. Um bei der iterativen Migration eine effiziente und effektive Implementierung zu gewährleisten und den Wartungsaufwand der entworfenen Microservices zu reduzieren, wird die Anforderung an das Migrationsrahmenwerk gestellt, in den Migrationsaktivitäten ein *Domänenverständnis* bei den Projektmitgliedern zu etablieren. Weiterhin hilft das Verständnis über die Geschäftsdomäne, Entscheidungen über die Kohäsion von Softwarebausteinen zu treffen (vgl. Abschnitt 3.1). Ein geeigneter Ansatz zum Etablieren des Domänenverständnisses ist der *domänengetriebene Entwurf* (engl. Domain-Driven Design, DDD) [Ve13]. DDD wird häufig durch die damit einhergehenden Konzepte in Verbindung mit Microservices und der Modernisierung von monolithischen Softwaresystemen gebracht [Ne15, NM+16, BP19].

A5 - Domänengetriebener Entwurf der Microservice-Architektur

Ein zentrales Prinzip beim Verwenden einer Microservice-Architektur ist die Wiederverwendung von Funktionalitäten, was durch die autonomen Microservices begünstigt werden soll [Ne15]. Ob ein Microservice autonom beziehungsweise wiederverwendbar ist, hängt von dem gewählten Entwurf der Microservice-Architektur ab. Zum Entwurf eines Microservice-basierten Softwaresystems bestehen unterschiedliche Herangehensweisen. Um eine hohe Wiederverwendbarkeit von geschäftslogikbezogener Funktionalität zu erreichen, sollte sich der gewählte Entwurf an der Geschäftsdomäne orientieren [HG+17]. Diese Orientierung des Entwurfs wird als weitere Anforderung an eine iterative Migration gestellt. Auch hier ist der Einsatz von DDD eine Möglichkeit zur Realisierung der Domänenorientierung.

A6 - Reorganisation des monolithischen Entwicklungsteams

Ein wichtiger Treiber für die Einführung einer Microservice-Architektur ist die punktuelle Skalierbarkeit der autonomen Softwarebausteine [TL+17]. Doch ebenso wichtig wie die Skalierbarkeit der Softwarebausteine eines Softwaresystems ist die Skalierbarkeit des Entwicklungsteams, welches das Softwaresystem betreut [HS17]. Eine weitere Anforderung an die iterative Migration ist damit, die Reorganisation des monolithisch geprägten Entwicklungsteams und dieses somit auf die neuen Bedürfnisse der Microservice-Architektur anzupassen. So müssen vertikale, entlang der Geschäftsdomäne–analog zu den Microservices–geschnittene Entwicklungsteams etabliert werden. Ein solches Entwicklungsteam trägt dabei die Verantwortung für einen einzelnen Microservice [Ne15]. Das Ausrichten der organisatorischen Struktur des Entwicklungsteams an dem Entwurf des Softwaresystems entspricht einem umgekehrten *Conway's Law* [Co68]. Durch die kleinen Entwicklungsteams lassen sich zudem die Development und Operations-Prinzipien (DevOps) besser etablieren [BW+15].

A7 - Berücksichtigung des komplexen Betriebs

Ein monolithisches Softwaresystem besteht aus wenigen Softwarebausteinen, die bei der Bereitstellung des Softwaresystems durch das Entwicklungsteam betreut werden müssen. Die während der iterativen Migration etablierte Microservice-Architektur führt dazu, dass die Anzahl an Softwarebausteinen jedoch deutlich zu nimmt und eine gesteigerte Komplexität in den Betrieb eingeführt wird [BH+16]. Um dieser Entwicklung entgegenzuwirken wird als eine weitere Anforderung an die iterative Migration die Berücksichtigung der neuen Komplexität festgelegt. Allen voran sollen während einer Iteration für die Microservices die DevOps-Prinzipien etabliert werden. So soll durch einen gesteigerten Automatisierungsgrad das Entwicklungsteam beim Betrieb entlastet werden [BW+15]. Um einfacher die DevOps-Prinzipien ermöglichen zu können haben sich Konzepte wie Cloud-native [GB+17] oder 12factors [Wi17] etabliert.

Anforderungskatalog

Die in Abschnitt 3.1 aufgestellten Anforderungen an eine iterative Migration werden in Tabelle 3.1 zusammenfassend dargestellt. Dieser Katalog wird zur Bewertung bestehender Arbeiten herangezogen.

Tabelle 3.1: Anforderungskatalog einer iterativen Migration zur Bewertung bestehender Arbeiten

Index	Anforderung
A1	Systematischer und nachvollziehbarer Migrationsprozess
A2	Wiederherstellung der Architekturbeschreibung
A3	Identifikation kohäsiver Softwarebausteine
A4	Etablieren eines Domänenverständnisses
A5	Domänengetriebener Entwurf der Microservice-Architektur
A6	Reorganisation des monolithischen Entwicklungsteams
A7	Berücksichtigung des komplexen Betriebs

3.2 Bewertung bestehender Arbeiten

Bedingt durch den breit gefassten Forschungsgegenstand von Softwaresystemmigrationen wurden Arbeiten mit Bezug auf die Forschungsbeiträge dieser Arbeit (vgl. Abschnitt 1.5) gewählt. So werden Arbeiten wie [BH+18], [Ne19] und [HR20] betrachtet, die aufbauend auf dem *Situational Method Engineering* ein Migrationsmuster-getriebenes Migrationsvorgehen präsentieren. Konträr zu der manuellen und nicht werkzeugunterstützten Migration aus dieser Forschungsarbeit werden mit [GK+16] und [MC+17] semi-automatisierte Migrationen vorgestellt. Einen systematischen Migrationsprozess, der vergleichbar ist mit dem in dieser Forschungsarbeit angestrebten Migrationsprozess, stellen die Arbeiten [KH18] und [BH+16] vor. Die Betrachtung einer bestehenden Arbeit inkludiert eine Bewertung anhand des Anforderungskatalogs aus Abschnitt 3.1.

3.2.1 Balalaie et al. - Microservices Migration Patterns

In der bestehenden Arbeit von Balalaie et al. [BH+18] wird die iterative Migration eines monolithischen Softwaresystems in eine Microservice-Architektur anhand des *Situational Method Engineerings* betrachtet. Eine ausführliche Beschreibung des Situational Method Engineerings liefert [HB+94]. Unter den Gesichtspunkten des Situational Method Engineerings stellen die Autoren *Migrationsmuster* (engl. migration patterns) vor, die in Abhängigkeit einer vorherrschenden Situation komponiert werden, um eine Migration durchzuführen. Zur Auswahl der Migrationsmuster stellen die Autoren *Architekturfaktoren* vor, die organisatorische Aspekte der Migration und nicht-funktionale Anforderungen des Softwaresystems betreffen. Die Beschreibung der Migrationsmuster erfolgt auf Basis eines Metamodells, welches die Autoren spezifisch auf die Bedürfnisse der Migrationsmuster zugeschnitten haben.

Durch die Verankerung des Situational Method Engineerings bei der Gestaltung des Migrationsansatzes verfolgen die Autoren das Ziel, abhängig von der vorherrschenden Situation der Organisation, den Ablauf der Migration auf die Bedürfnisse anzupassen. Im Gegensatz zu den starren, prozessorientierten Migrationsansätzen soll laut den Autoren ein breiter Anwendungsbereich des Migrationsansatzes

erreicht werden. Denn gerade die vorherrschende Situation unterscheidet sich von Organisation zu Organisation. Die vorherrschende Situation selbst beschreibt eine Ausgangssituation und eine Zielsituation, die nach einer Migration erreicht sein soll. Die Ausgangssituation umfasst den Ist-Zustand des Monolithen und die Gegebenheiten der Organisation wie beispielsweise den Entschluss zur Migration. Eine Zielsituation beschreibt die Zielarchitektur des Softwaresystems, welche bestimmte nicht-funktionale Anforderungen–auch Architekturfaktoren–erfüllt.

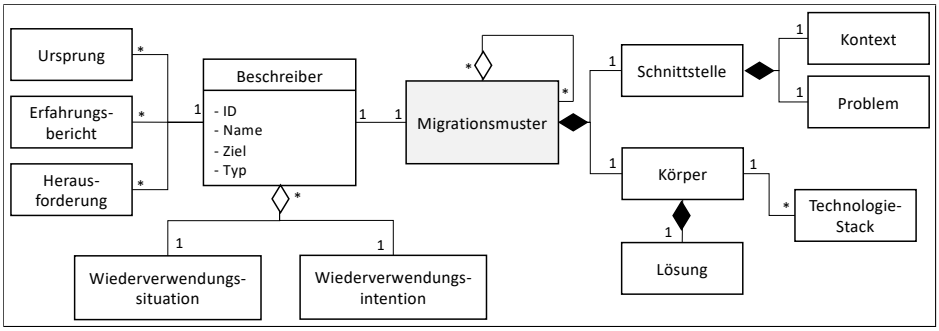

Abbildung 3.2: Metamodell der Migrationsmuster aus [BH+18]

Ein Migrationsmuster beschreibt, ebenso wie Architekturmuster oder Entwurfsmuster, eine wiederkehrende Problematik und bietet dazu einen Lösungsvorschlag an. Der detaillierte Aufbau eines Migrationsmusters wird durch ein Metamodell, welches in Abbildung 3.2 dargestellt wird, bestimmt. Ein Migrationsmuster ist zunächst an einen *Beschreiber* geknüpft, der für die Wiederauffindbarkeit in einem *Musterregister* (engl. pattern registry) zuständig ist. Weiterführend ist der Beschreiber mit einer *Wiederverwendungssituation*, die einem Architekturfaktor entspricht, und einer *Intention* der Wiederverwendung verknüpft. Neben dem Beschreiber besteht das Migrationsmuster aus einem *Körper*, der die konkrete Lösung und zusätzlich einen möglichen Technologie-Stack definiert. Die *Schnittstelle* eines Migrationsmusters gibt Aufschluss über die Situation, in der das Migrationsmuster eingesetzt werden kann. Das *Problem* einer Schnittstelle stellt die Intention des Musters in Form von Fragestellungen dar. Der *Kontext* beschreibt eine vorherrschende Situation.

In Tabelle 3.2 werden die Migrationsmuster von Balalaie et al. zusammenfassend dargestellt. Für jedes der Migrationsmuster wird die Intention durch einen beschreibenden Text erläutert. Aus Gründen der Übersichtlichkeit werden nicht alle Bestandteile der Migrationsmuster in die Tabelle aufgenommen.

Ein prozessuales Vorgehen bei der Migration ist basierend auf dem beschriebenen Ansatz durch die Komposition der Migrationsmuster möglich. Sogenannte *Migrationsingenieure* (engl. migration engineers) betrachten die vorherrschende Situation und leiten daraus die notwendigen Migrationsmuster ab. Durch die Verknüpfung der Migrationsmuster mit den Architekturfaktoren ist ein Migrationsingenieur in der Lage, die relevanten Migrationsmuster zu identifizieren. Für die Aufstellung der Sequenz, in der die Migrationsmuster ausgeführt werden, geben die Autoren Empfehlungen. Zunächst sollen die Migrationsmuster ausgeführt werden, die keinerlei Vorbedingungen haben beziehungsweise nicht

Tabelle 3.2: Migrationsmuster der Autoren Balalaie et al. [BH+18]

Kürzel	Name	Beschreibung
MP_1	Continuous Integration ermöglichen	Einführen von Prinzipien für die Etablierung einer Continuous Integration-Philosophie des Entwicklungsteams.
MP_2	Ist-Architektur wiederherstellen	Wiederherstellen der Architekturbeschreibung des monolithischen Softwaresystems.
MP_3	Aufspalten des Monolithen	Anhand der Geschäftslogik Softwarebausteine des monolithischen Softwaresystems gruppieren und aus dem Monolithen extrahieren.
MP_4	Aufspalten des Monolithen anhand Datenhoheit	Ähnlich wie MP_3 werden die Softwarebausteine des Monolithen gruppiert, doch dies erfolgt anhand der Datenhoheit in der Persistierung.
MP_5	Quellcode-Abhängigkeiten in Service-Aufrufe ändern	Alle lokalen Schnittstellen zwischen Softwarebausteinen werden zu Web-Schnittstellen migriert.
MP_6	Einführen einer Service-Registry	Zur Auffindbarkeit (Service Discovery) der Microservices wird eine Service-Registry eingeführt.
MP_7	Einführen eines Service-Registry-Clients	Ergänzend zu MP_6 wird ein Client eingeführt, der eine Instanz des Microservices bei der Service Registry registriert.
MP_8	Einführen eines internen Load Balancers	Ein interner Load Balancer befindet sich in einem Microservice und speichert die verfügbaren Instanzen relevanter Microservices.
MP_9	Einführen eines externen Load Balancers	Der externe Load Balancer dient als Ansprechpartner für Dienste zum Erhalten der Adressen laufender Microservice-Instanzen.
MP_{10}	Einführen eines Circuit Breakers	Ein Circuit Breaker wird als Teil des Microservices eingeführt, der auf den Zustand des Microservices Acht nimmt und gegebenenfalls Anfragen an den Microservice bei zu hoher Auslastung ablehnt.
MP_{11}	Einführen eines Konfigurationsservers	Zur Auslieferung der Laufzeitkonfiguration für Instanzen der Microservices wird ein dedizierter Server eingeführt.
MP_{12}	Einführen eines Edge-Servers	Damit Web-Schnittstellen der Microservices durch einen einheitlichen Eintrittspunkt veröffentlicht werden, wird ein Edge-Server eingeführt.
MP_{13}	Containerisierung der Microservices	Für ein vereinfachtes und reproduzierbares Veröffentlichen der Microservices werden die Microservices containerisiert.
MP_{14}	Bereitstellung in eine Container-Orchestrierung	Zur Ausführung der containerisierten Microservices werden die Images der Microservices in eine Container-Orchestrierung veröffentlicht.
MP_{15}	Überwachen des Softwaresystems	Zum Betrieb des Softwaresystems werden kontinuierlich Metriken gesammelt, um Aussagen über den Zustand des Softwaresystems treffen zu können.

auf Vorarbeiten beruhen. Anschließend sollen diejenigen Muster ausgeführt werden, welche nicht die Benutzer des Softwaresystems beeinflussen. Erst dann soll die Einführung von Migrationsmuster stattfinden, welche offensichtlich den Benutzer betreffen.

Bewertung

Die Autoren Balalaie et al. stellen einen flexiblen Migrationsansatz vor, dessen grundsätzliche Vorgehensweise auf dem Situational Method Engineering begründet ist. Ein Migrationsteam verknüpft Migrationsmuster in sequentieller Form–auch als Meilensteinplan bezeichnet–, um einen prozessualen Ablauf der Migration des Monolithen zu erreichen. Im Hinblick auf die Anforderung eines systematischen und nachvollziehbaren Migrationsprozesses (A1) zur Unterstützung des Migrationsteams kann festgehalten werden, dass diese Anforderung nur teilweise erfüllt ist. Zwar stellt die Roadmap eine Systematik dar, doch der Weg hin zu dieser Roadmap bleibt ohne jegliche Führung des vorgestellten Ansatzes; dies entspricht jedoch auch der Philosophie des Situational Method Engineerings. Zudem werden keine Softwareartefakte für eine Nachvollziehbarkeit diskutiert. Durch das Migrationsmuster "MP_2 *Ist-Architektur wiederherstellen*" ist die Anforderung der Wiederherstellung der Architekturbeschreibung (A2) erfüllt. Lediglich die Frage bezüglich des betriebenen Aufwandes für die Wiederherstellung ist offen. Die Anforderung der Identifikation kohäsiver Softwarebausteine (A3) wird wiederum nur als teilweise erfüllt angesehen. Die Migrationsmuster "MP_3 *Aufspalten des Monolithen*" und "MP_4 *Aufspalten des Monolithen anhand Datenhoheit*" beabsichtigen genau eine solche Kohäsionsbestimmung, doch eine detaillierte Beschreibung zur Ausführung dieser Migrationsmuster bleibt aus. Gerade die Identifikation der kohäsiven Softwarebausteine ist ein zentraler Aspekt der Migration. Das Etablieren eines Domänenverständnisses (A4) bei den Entwicklern ist in keinem der Migrationsmuster vorgesehen, weshalb die Anforderung als nicht erfüllt angesehen wird. Das bereits erwähnte Migrationsmuster MP_3 zielt auf eine Dekomposition des Monolithen anhand DDD ab, weshalb die Anforderung, einen domänengetriebenen Entwurf der Microservice-Architektur (A5) zu verfolgen, erfüllt ist. Eine Reorganisation des monolithischen Entwicklungsteams (A6) ist in keinem der vorliegenden Migrationsmuster zu finden, weshalb die betreffende Anforderung ebenfalls nicht erfüllt ist. Umfangreich betrachten die Autoren die Komplexität des Betriebs der Microservices. Verschiedene Migrationsmuster (MP_1, MP_6, MP_7, MP_8, MP_9, MP_{11}, MP_{12}, MP_{13}, MP_{14} und MP_{15}) betrachten ausschließlich den Betrieb betreffende Aspekte der Migration. Somit kann die Anforderung der Berücksichtigung des komplexen Betriebs (A7) als vollständig erfüllt angesehen werden.

3.2.2 Henry und Ridene - Migrating to Microservices

Eine weitere Arbeit, die sich in das *Situational Method Engineering* einordnen lässt, wird von Henry und Ridene [HR20] vorgestellt. Die Autoren führen, motiviert durch in der Industrie durchgeführte

Tabelle 3.3: Migrationsmuster und Einordnung in die Gruppen der Autoren Henry und Ridene [HR20]

Name	Beschreibung	Gruppierung
Warm-Up und Skalierung	Zunächst wird ein einfacher, trivialer Microservice abgeleitet. Anhand dieses Microservices werden Grundlagen für den Betrieb geschaffen.	Betrieb
Daten sobald wie möglich freigeben	Auflösen der Datenabhängigkeiten zwischen Softwarebausteinen des Monolithen.	Datenhaltung
Schnelles Ausgliedern für einen Quick Win	Softwarebausteine, die keine Symbol- oder Datenabhängigkeiten zu anderen Softwarebausteinen haben, können schnell und einfach aus dem Monolithen ausgegliedert werden.	Anwendungs- und Geschäftslogik
Vertikal graben und Schreiboperationen isolieren	Vertikales Schneiden des Monolithen anhand kohäsiver Anwendungslogik.	Anwendungs- und Geschäftslogik
Domänengrenzen sind kein Datenzugriff	Es werden Bounded Contexts als Kandidaten für Microservices identifiziert.	Anwendungs- und Geschäftslogik
Mit dem größten Geschäftswert starten	Zunächst die wichtigste Anwendungs- und Geschäftslogik in Microservices modernisieren.	Anwendungs- und Geschäftslogik
Minimierung der Abhängigkeiten zum Monolithen	Bereits modernisierte Microservices dürfen nur so wenig wie möglich Abhängigkeiten zu dem Monolithen aufweisen. Daher müssen diese maximal minimiert werden.	Kopplung

Umfragen, *Migrationsmuster* ein, die zur Erstellung eines sogenannten *Migrationsmeilensteinplans* (engl. migration roadmap) dienen. Die Industrieumfragen wurden nicht von den Autoren selbst durchgeführt. Der vorgestellte Migrationsansatz verfolgt das Ziel, iterativ und inkrementell den vorliegenden Monolithen zu zerschneiden. Entsprechend dem Situational Method Engineering ist das Vorgehen während der Migration durch die bedarfsgetriebene Auswahl der einzelnen Migrationsmuster flexibel. Dennoch stellen die Autoren einen *Referenzmeilensteinplan* vor, welche eine beispielhafte Verkettung von Migrationsmustern zur Migration beschreibt.

Die vorgestellten Migrationsmuster zielen darauf ab, eine flexible und agile Microservice-Architektur zu entwerfen, um einer Änderungspolitik an den durch das Softwaresystem erbrachten Geschäftsfähigkeiten entgegenzuwirken. Welche der Migrationsmuster für das Migrationsvorhaben verwendet werden, hängt zudem von den Geschäftsfähigkeiten ab, die das Softwaresystem erbringen muss. Jedes der Migrationsmuster bringt neben den Vorteilen auch Nachteile mit sich, welche bei der Ausgestaltung des Migrationsvorhabens gegeneinander abgewägt werden müssen, damit die Geschäftsfähigkeiten auch erbracht werden können. Die von Henry und Ridene vorgestellten Migrationsmuster betreffen unterschiedliche architekturelle Aspekte des Softwaresystems. Die Muster lassen sich wie folgt gruppieren: Datenhaltung, Anwendungs- und Geschäftslogik, Betrieb und Kopplung. Eine Einordnung in diese Gruppen und eine Kurzbeschreibung der vorgestellten Migrationsmuster kann der Tabelle 3.3 entnommen werden.

Dem vorgestellten Referenzmeilensteinplan werden durch die Autoren Prinzipien zugrunde gelegt: (1) Keine perfekte Microservice-Architektur entwerfen, (2) kurze Zyklen für schnelle Ergebnisse und (3)

Änderungen am Migrationsvorhaben annehmen. Besonders das Berücksichtigen von (3) Änderungen am Migrationsvorhaben wie beispielsweise eine Verkürzung des Migrationszeitraums, wird zentral im Referenzmeilensteinplan vorgesehen. Daher wird auch das Prinzip der (2) kurzen Zyklen verfolgt, denn es sollen eher schnell Ergebnisse erzielt werden als kontinuierlich einen perfekten Entwurf zu erarbeiten. Die Autoren sind der Auffassung, dass das Anstreben einer (1) perfekten Microservice-Architektur der Fehler der meisten Migrationsprojekte ist. Die Autoren teilen die Migration in drei Phasen ein: (1) Vorbereitungen zum Betrieb, (2) Skalierung und Erweiterung und (3) Optimierung und Anbieten von Schnittstellen. In jeder Phasen kommen verschiedene Migrationsmuster zum Tragen. Wie der Monolith letztendlich in die vertikalen Bestandteile aufgeteilt wird, deckt der Referenzmeilensteinplan nicht ab.

Bewertung

Der von Henry und Ridene vorgestellte Ansatz findet sich ebenso wie [BH+18] im Kontext des Situational Method Engineerings wieder. Die Autoren führen durch die Industrie motivierte Migrationsmuster ein, die einem Migrationsteam die Möglichkeit geben, diese in sequentieller Form zu verketten. Auch bei diesem Ansatz wird von einem Meilensteinplan gesprochen, der zwar selbst ein systematisches Vorgehen beschreibt, aber der Weg hin zu diesem Meilensteinplan ist nicht geführt. Im Gegensatz zu [BH+18] stellen die Autoren jedoch einen Referenzmeilensteinplan vor, der dem Migrationsteam ansatzweise als Leitlinie dient. Trotz des Referenzmeilensteinplans gilt die Anforderung, einen systematischen und nachvollziehbaren Migrationsprozess (A1) einzuführen, als nur teilweise erfüllt. Die Anforderung, eine Architekturbeschreibung wiederherzustellen (A2), wird nicht erfüllt, denn keines der vorgestellten Migrationsmuster geht auf den Ist-Zustand des monolithischen Softwaresystems ein. Dieser wird mehr oder weniger als bereits vorhanden angenommen. Zwei der vorgestellten Migrationsmuster befassen sich mit der Kohäsion von Softwarebausteinen innerhalb des Monolithen. Zum einen auf der Ebene der Datenhaltung und zum anderen auf der Ebene der Anwendungslogik. Gerade bei der Anwendungslogik wird auf werkzeuggetriebene statische Quellcodeanalyse gesetzt, sodass ausgehend von dem Monolithen Microservice-Kandidaten abgeleitet werden. Die Bewertung der Kohäsion erfolgt jedoch anhand der Quellcodeaufrufe, was nicht unbedingt der Orientierung an der Domäne entspricht. Dennoch kann die Anforderung der Identifikation der kohäsiven Softwarebausteine (A3) als erfüllt angesehen werden, denn durch das Ableiten der Microservice-Kandidaten werden dem Migrationsteam kohäsive Softwarebausteine präsentiert. Keines der vorgestellten Migrationsmuster zielt darauf ab, dem Entwicklungsteam Informationen über die Geschäftsdomäne aufzubereiten, weshalb die hierfür vorgesehene Anforderung (A4) als nicht erfüllt gilt. Eines der vorgestellten Migrationsmuster sieht das Schneiden des monolithischen Softwaresystems anhand vertikaler Gruppen vor. Die Gruppierung erfolgt unter Einbezug von DDD. Zwar bleiben bei der Anwendung von DDD klare Leitlinien offen, doch durch die Realisierung der Ableitung der Microservice-Kandidaten in Form eines Werkzeuges kann die betreffende Anforderung

(A5) als erfüllt angesehen werden. Die Anforderung, das monolithische Entwicklungsteam für die Microservice-Architektur zu reorganisieren (A6), wird als nicht erfüllt angesehen. Für die Reorganisation wird kein Migrationsmuster vorgestellt. Abschließend kann festgehalten werden, dass der komplexe Betrieb und damit die Anforderung der Berücksichtigung des komplexen Betriebs (A7) ausführlich durch Migrationsmuster betrachtet wird; die Anforderung gilt somit als erfüllt.

3.2.3 Newman - Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith

Mit der Arbeit von Newman [Ne19] existiert ein weiterer Beitrag, der die iterative Migration monolithischer Softwaresysteme in eine Microservice-Architektur mittels des *Situational Method Engineerings* betrachtet und hierfür geeignete *Migrationsmuster* einführt. Bei dem Entwurf der Migrationsmuster fokussiert Newman die Modernisierung des Monolithen und die Restrukturierung der Datenhaltung. Die Anwendung der Migrationsmuster wird ausführlich anhand Beispielen demonstriert; eine durchgängige, industrienahe Fallstudie ist nicht gegeben. Zusätzlich zu den Migrationsmustern geht Newman auf grundsätzliche Aspekte wie Rahmenbedingungen und Zielsetzungen der iterativen Migration ein.

Die Verwendung der iterativen Vorgehensweise wird über die Beherrschbarkeit der Migration, die Minimierung von Fehlern während der Migration und die Maximierung des Lernfortschritts bei der Entwicklung von Microserivces des Entwicklungsteams motiviert. Weiterführend hebt Newman die Bedeutung der Organisation für eine erfolgreiche Migration hervor. Ohne die Unterstützung der Organisation können für die Migration keine *Vision* und keine *Strategie* festgelegt werden, denn über personelle, finanzielle und zeitliche Ressourcen entscheidet letztendlich die Organisation. Als ebenso wichtigen Aspekt bei der Migration sieht Newman die Anpassung der Struktur des Entwicklungsteams an den neuen Architekturstil der Microservices. Wird die organisatorische Struktur des Entwicklungsteams nicht auf den neuen Stil angepasst, können die Vorteile der Microservice-Architektur nicht vollends genutzt werden. Wie das Entwicklungsteam umstrukturiert wird, liegt jedoch nicht im Fokus der Arbeit.

In [Ne19] wird eine explizite Kategorisierung der Migrationsmuster in die Kategorien *Softwarebausteine* und *Datenhaltung* verfolgt. So wird verdeutlicht, dass die iterative Migration verschiedene Aspekte eines Softwaresystems betrifft. Eine erfolgreiche iterative Migration ist an die Verwendung der Migrationsmuster aus beiden Kategorien verbunden, auch wenn diese prinzipiell unabhängig voneinander angewendet werden können. Prozessuale Vorgaben zur Verknüpfung der Migrationsmuster werden nicht vorgestellt, sodass ein systematisches Vorgehen durch das Entwicklungsteam aufgestellt werden muss.

Die Kategorie der Softwarebausteine befasst sich mit dem Quellcode des monolithischen Softwaresystems. Berücksichtigt werden die Softwarebausteine aller Schichten der Schichtenarchitektur aus

Tabelle 3.4: Migrationsmuster zur Migration der Softwarebausteine des Monolithen von Newman [Ne19]

Name	**Beschreibung**	**Zugriff Quell-code?**
StranglerFig-Application	Dieses Migrationsmuster wurde ursprünglich von Fowler [Fo04] vorgestellt. Inkrementell wird bestehende oder neue Funktionalität des monolithischen Softwaresystems ausgegliedert und in Form von Microservices bereitgestellt. Durch das inkrementelle Vorgehen existieren der Monolith und neue Microservices parallel.	Ja
Komposition der Benutzeroberfläche	Die Benutzeroberfläche (engl. User Interface, UI) eines Softwaresystems wird für den Zugriff meist über das *Hypertext Transfer Protocol* (HTTP) auf die Funktionalität im Backend verwendet. Das Migrationsmuster sieht vor, einzelne Bestandteile des UIs notwendige Zugriffe auf migrierte Funktionalität der Microservices durchführen zu lassen. So kann nach und nach auch die Benutzeroberfläche auf die Microservices angepasst werden.	Ja
Branch by Abstraction	Quellcodeanpassungen an dem Monolithen können während einer Migration Konflikte auslösen, sobald parallel an dem Monolithen weiterentwickelt wird. Zur Auflösung führt das Migrationsmuster Abstraktionen für Funktionalitäten ein, auf die andere Softwarebausteine referenzieren. Es bestehen damit keine direkten Abhängigkeiten mehr zu der konkreten Implementierung der Funktionalität. Folglich kann die konkrete Implementierung ohne Beeinträchtigung der Abhängigkeiten ausgetauscht werden, was eine einfache parallele Bearbeitung der Funktionalität während der Migration erlaubt.	Ja
Paralleler Betrieb	In Microservices migrierte Funktionalität, die parallel zu dem Monolithen betrieben werden kann, wird zunächst gemeinsam mit der alten Funktionalität in dem Monolithen bereitgestellt. Anfragen an die Funktionalität werden sowohl mit der neuen als auch der alten Funktionalität bearbeitet. Das Ergebnis des Monolithen wird dem Client zur Verfügung gestellt. Das Ergebnis eines Microservices wird persistiert und mit dem Ergebnis des Monolithen verglichen. Damit kann der Microservice mit Szenarien des Produktivbetriebs getestet werden.	Nein
Decorating Collaborator	Eingeführt wird ein Proxy, der als *Decorator* Anfragen an den Monolithen weiterleitet und das Ergebnis entgegennimmt. Ausgehend von dem erhaltenen Ergebnis entscheidet der Proxy, ob zusätzliche Funktionalität von Microservices angefragt wird. So kann neue Funktionalität den Clients zur Verfügung gestellt werden, ohne dass der Monolith angepasst werden muss.	Nein
Change Data Capture	Anstatt Anfragen der Clients zum Monolithen abzufangen, werden Änderungen von Datensätzen in der Datenhaltung registriert und als Auslöser für Funktionalität in Microservices verwendet.	Nein

DDD [Ev04]. Besonders hervorzuheben ist, dass auch die Präsentationsschicht betrachtet wird. Über die Anwendbarkeit der Migrationsmuster entscheiden, so Newman, zwei Faktoren: (1) Zugriff auf den Quellcode des Monolithen und (2) Absicht zur Anpassung des Monolithen. Dabei bedingt die (2) Absicht zur Anpassung des Monolithen den (1) Zugriff des Entwicklungsteams auf den Quellcode. Ist dies ausgeschlossen, muss die Strategie des Entwicklungsteams entsprechend gewählt werden. Hierfür werden ebenfalls Migrationsmuster vorgestellt. Die Migrationsmuster werden in Tabelle 3.4 zusammengefasst.

Eine fundamentale Herausforderung bei der iterativen Migration des Monolithen in die Microservice-Architektur sieht Newman in der Datenhaltung. Die geteilte Datenhaltung beziehungsweise geteilte Datenbank des Monolithen muss auf die Microservice-Architektur angepasst werden, sodass das

Tabelle 3.5: Migrationsmuster zur Migration der Datenhaltung von Newman [Ne19]

Name	Beschreibung
Geteilte Datenbank	Das Migrationsmuster beschreibt den Ist-Zustand des monolithischen Softwaresystems. Die gesamte Datenhaltung des Monolithen liegt innerhalb eines Datenbankschemata.
Datenbank-View	Die Datenhaltung eines Microservices wird über eine Datenbank-View gewährleistet. Dabei repliziert und transformiert eine *Datenbank-View* die Daten des Monolithen. Das Datenbankschemata bleibt unberührt, wodurch die Ursprungsdaten weiterhin in der Datenhaltung des Monolithen angesiedelt sind.
Datenbank-Wrapping-Service	Über das Migrationsmuster wird für einen bestimmten Ausschnitt des Datenbankschemata ein eigenständiger *Wrapping-Service* bereitgestellt. Der Zugriff und die Manipulation der Daten erfolgt nur noch über den Wrapping-Service.
DaaS-API	Eine dedizierte Datenbank wird als Schnittstelle (engl. Application Programming Interface, API) für Microservices bereitgestellt. Die Microservices haben auf die *Database-as-a-Service* (DaaS) nur lesenden Zugriff. Über Sychnronisationsverfahren werden die Daten anderer Datenbanken in die DaaS übertragen.
Synchronisierung von Daten im Softwaresystem	Bei der Übertragung eines Teils des Datenbankschemata in die Datenhaltung eines Microservices müssen die Datenbestände zwischen den Datenbanken synchronisiert werden. Die Synchronisation kann von dem Monolithen und dem Microservice übernommen werden, indem alle Schreib- oder Änderungsoperationen beider Softwarebausteine auf beiden Datenbanken ausgeführt werden.
Tracer Writer	Ist eine Synchronisation von Daten zwischen der Datenhaltung des Monolithen und der Datenhaltung eines Microservices notwendig, kann der Monolith Schreib- und Änderungsoperationen auf beiden Datenbanken ausführen. Der Zugriff auf die Datenhaltung des Microservices erfolgt jedoch nur über die Web-API des Microservices.
Aufteilen der Datenbanktabelle	Benötigen der Monolith und ein Microservice Daten innerhalb derselben Datenbanktabelle, wird diese Datenbanktabelle entsprechend der benötigten Tabellenspalten in zwei oder mehr Tabellen aufgeteilt. Dies ist insbesondere dann möglich, wenn der Monolith und der Microservice nicht dieselben Tabellenspalten verwenden. Liegt ein Konflikt vor, wird einem der Bausteine die Hoheit über die Spalte übertragen und der Zugriff auf die Daten erfolgt nur über entsprechende Web-APIs.
Fremdschlüsselbeziehung in den Quellcode verschieben	Dieses Migrationsmuster verfolgt das Ziel, Fremdschlüsselbeziehungen zwischen Datenbanktabellen nicht mehr über Datenbank-Joins aufzulösen. Stattdessen werden die Daten innerhalb des Monolithen oder eines Microservices zusammengesetzt. Hierfür werden die Daten hinter dem Fremdschlüssel über dedizierte Web-APIs bei dem entsprechenden Softwarebaustein angefragt.

Prinzip des *Information Hidings* durch die Microservices verwirklicht werden kann. Zur Bewältigung dieser fundamentalen Herausforderung führt Newman mehrere Migrationsmuster ein, die zusammenfassend in der Tabelle 3.5 festgehalten werden. Ausgehend von der Anzahl der Migrationsmuster ist zu erkennen, dass der Betrachtung der Datenhaltung in dieser Arbeit große Bedeutung zugemessen wird.

Bewertung

Der von Newman vorgestellte Migrationsansatz basiert auf der Anwendung von Migrationsmustern. Bei der Anwendung der Migrationsmuster unterstützt der Autor mit umfangreichen Beispielen, die

überwiegend auf fiktiven Szenarien basieren. Das Vorgehen des Entwicklungsteams bei den einzelnen Migrationsmustern ist damit ausführlich beschrieben. Dagegen bleibt jedoch eine Verkettung der Muster zu einem systematischen Ablauf aus. Folglich muss das Entwicklungsteam das gesamte Vorgehen definieren. Zudem werden keine formalen Softwareartefakte für die Modellierung der Ergebnisse genannt. Auch wenn das Vorgehen der Migrationsmuster ausführlich beschrieben ist, gilt aufgrund der identifizierten Schwachstellen die Anforderung eines systematischen und nachvollziehbaren Migrationsprozesses (A1) als nicht erfüllt. Die ganzheitliche Unterstützung des Entwicklungsteams bei der iterativen Migration steht im Vordergrund. Die Anforderung der Wiederherstellung der Architekturbeschreibung (A2) ist nicht erfüllt, denn keine der Migrationsmuster zielt auf Erstellung von Softwareartefakten ab. Einige der Migrationsmuster betrachten Abhängigkeiten zwischen Softwarebausteinen und Elementen in der Datenhaltung, doch die Identifikation der Abhängigkeit wird nicht thematisiert, weshalb auch keine kohäsiven Softwarebausteine identifiziert werden. Die Identifikation ist jedoch für die Bestimmung des Inkrements einer Iteration notwendig. Daher ist auch die Anforderung der Identifikation kohäsiver Softwarebausteine (A3) nicht erfüllt. Nicht explizit als Migrationsmuster vorgesehen ist das Etablieren eines Domänenverständnisses (A4) bei dem Entwicklungsteam. Dennoch betont Newman die Wichtigkeit eines etablierten Domänenverständnisses und verankert den Softwareentwicklungsansatz DDD in einigen der Migrationsmuster. So kann abgeleitet werden, dass das Entwicklungsteam durch die allgemeine Berücksichtigung der Geschäftsdomäne sich mit dieser auch beschäftigen muss. Die betreffende Anforderung gilt als teilweise erfüllt. Weiterführend kann festgehalten werden, dass die Anforderung des domänengetriebenen Entwurfs der Microservice-Architektur (A5) durch die Verwendung von DDD und den damit verbundenen Bounded Contexts als erfüllt gilt. Die von den Migrationsmustern angestrebten Microservices weisen eine hohe Orientierung an der Geschäftsdomäne auf. Die grundlegenden Aspekte, die Newman in der bestehenden Arbeit betrachtet, thematisieren auch die Anpassung des Entwicklungsteams auf die Microservice-Architektur. Konkrete Anweisungen zur Reorganisation des monolithischen Entwicklungsteams (A6) sind jedoch nicht gegeben, weshalb die entsprechende Anforderung nur als teilweise erfüllt angesehen wird. Die letzte Anforderung der Berücksichtigung des komplexen Betriebs (A7) gilt als nicht erfüllt, da der Betrieb mit keinem der vorgestellten Migrationsmuster unterstützt wird.

3.2.4 Balalaie et al. - Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture

[BH+16] thematisiert die Erfahrung eines Entwicklungsteams mit der inkrementellen Migration eines monolithischen Softwaresystems in eine Microservice-Architektur. Bei diesem Softwaresystem handelt es sich um eine *Mobile-Backend-as-a-Service*-Lösung (MBaaS), die für den Zugriff von mehreren externen Kunden entworfen wurde. Die Autoren zeigen in ihrer Arbeit die einzelnen ausgeführten Migrationsschritte auf und heben die enge Verbindung zwischen der neuen Microservice-Architektur und dem DevOps-Paradigma hervor. Abgeleitet von ihren eigenen Erfahrungen mit der

Migration werden zusätzlich *Migrationsmuster* vorgestellt, um die Durchführung einer Migration zu unterstützen. Diese Migrationsmuster wurden in der Arbeit [BH+18] separat betrachtet.

Das monolithische Softwaresystem trägt den Namen *Backtory* und verfolgt das Ziel, Entwicklern von mobilen Softwaresystemen Backend-Services zur Verfügung zu stellen. Entwickler, die keine Erfahrung mit der Entwicklung von serverseitigen Softwaresystemen haben, können mittels Backtory dennoch auf Logiken wie der Persistierung von Daten in externen Speichern zurückgreifen. Der Monolith wurde in Java geschrieben und mittels der üblichen Werkzeuge wie *Maven* und *Jetty* verwaltet beziehungsweise bereitgestellt (engl. deployed). Eine Bereitstellung für den Produktionsbetrieb mussten die Entwickler manuell durchführen. Eine neue Anforderung stellt die Motivation zur Migration des bestehenden Monolithen dar. Backtory sollte um eine Chat-Funktionalität erweitert werden, welche den Mobile-Entwicklern bei Abruf (engl. on-demand) zur Verfügung stehen muss. Dabei ist gerade die on-demand-Fähigkeit die Hauptproblematik, da das Softwaresystem bei einem initialen Aufruf des Chats die notwendigen Provisionierungen des Backends wie beispielsweise die Initialisierung des Datenbankschemas vornehmen muss. Die notwendigen Abläufe sind, zu dem Zeitpunkt vor der Migration, noch manuell zu tätigen.

Bei der Migration in [BH+16] wird inkrementell unter der Verwendung von DDD und zentralen DevOps-Paradigmen der Monolith in eine Microservice-Architektur überführt. Weiterhin werden die von dem Monolithen eingesetzten Technologien durch Modernere ersetzt. Im Kontext von DDD findet das Muster der *Bounded Contexts* Anwendung. Damit die Bereitstellung des Softwaresystems zukünftig automatisiert ablaufen kann, werden Cloud-nahe Technologien eingesetzt. Die Autoren trennen den inkrementellen Migrationsprozess in zwei Aufgabenbereiche: (1) architekturelles Refactoring und (2) Querschnittsänderungen.

Der inkrementelle Ansatz kommt bei dem architekturellen Refactoring zum Tragen, bei dem entweder Softwarebausteine hinzugefügt oder Teile des Monolithen verschoben werden. Als ersten Schritt, noch vor der eigentlichen Migration des Monolithen, führen die Autoren *Continuous Integration* (CI) als weiteres Entwicklungsparadigma ein, um letztendlich *Continuous Delivery* (CD) zu ermöglichen. Weiterhin wird zur Gewährleistung von CI das *konsumentengetriebene Testen* (engl. consumer-driven testing) eingeführt. Nach den vorbereitenden Maßnahmen wird die Migration des Monolithen durchgeführt. Dabei wird ein zentraler Softwarebaustein des Monolithen als eigener Service gefasst und mit einer *Representational State Transfer-Schnittstelle* (REST) ausgestattet, damit Änderungen an dem Softwarebaustein nicht die abhängigen Softwarebausteine beeinflussen. Um dem Bedarf einer unabhängigen Bereitstellung des neu geschaffenen Microservice gerecht zu werden, wird im nächsten Schritt eine weitere Grundlage für CD geschaffen. Es wird ein zentraler Konfigurationsserver eingeführt, welcher für die Bereitstellung der notwendigen Konfiguration während des Betriebs verantwortlich ist. Dabei wird für den Betrieb der Microservices auf eine Containerisierung wie *Docker* gesetzt. Abschließend wird für das Refactoring die Einführung von essenziellen Komponenten

für den Betrieb der Microservices vorgesehen. Zu diesen gehören: (1) *Service Discovery*, (2) *Load Balancing* and (3) *Circuit Breaking*.

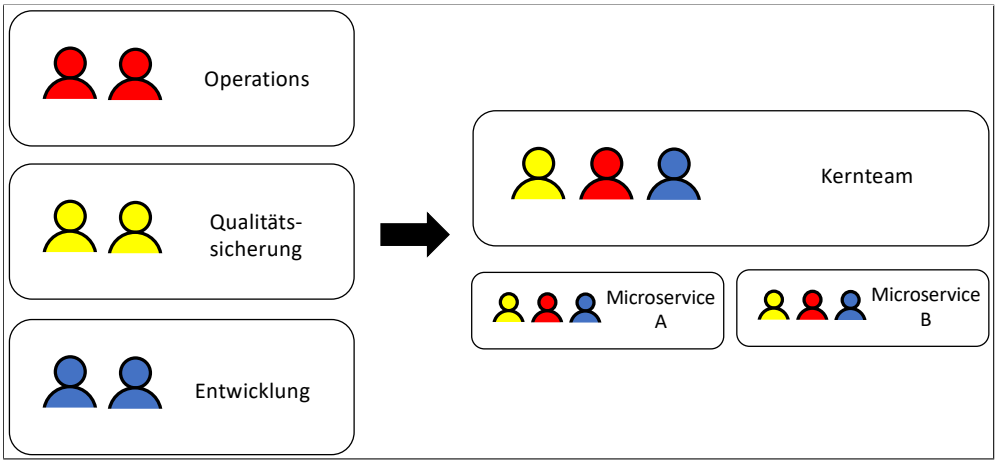

Abbildung 3.3: Migration des Entwicklungsteams zum effizienteren Betrieb der Microservices

Unter den Änderungen für Querschnittsthemen verstehen die Autoren die Einführung einer allgemeinen Überwachung (engl. monitoring) und die Veränderung der Teamstruktur im Sinne eines effizienteren Betriebs der Microservices. Jeder Microservice offenbart Laufzeitinformationen, die zentral in einem Monitoring-Ökosystem ausgewertet werden. Für die Umstrukturierung des Entwicklungsteams wird eine flache Hierarchie eingeführt (siehe Abbildung 3.3). Das sonst funktional aufgeteilte Entwicklungsteam (Operations, Qualitätskontrolle, Softwareentwicklung (Front- und Backend)) wird entlang der DevOps-Paradigmen in kleinere und autarke Teams aufgeteilt. In einem Team befinden sich alle Kompetenzen, die für den Lebenszyklus des Microservices notwendig sind. Übergeordnet zu jedem Microservice-Team besteht ein Kernteam, welches beratend den einzelnen Teams zur Verfügung steht.

Abschließend stellen die Autoren Migrationsmuster vor, die auf Basis von den ihnen getätigten Migrationsschritten abgeleitet wurden. Die Migrationsmuster sind stark auf die Einführung von DevOps-Paradigmen ausgerichtet und bestehen daher aus einem Musternamen und der Auswirkung des Musters auf die DevOps-Philosophie.

Bewertung

Balalaie et al. führen einen iterativen Migrationsprozess ein, der zwei Aspekte einer Monolithenmigration betrachtet. Zum einen wird ein architekturelles Refactoring durchgeführt, welches neue Softwarebausteine oder bestehende Teile aus dem Monolithen als eigenständige Softwarebausteine einführt. Zum anderen werden Querschnittsänderungen durchgeführt, welche insbesondere den

Betrieb der neuen Microservices betreffen. Die erste Anforderung, einen systematischen und nachvollziehbaren Migrationsprozess (A1) einzuführen, kann für die betrachtete Arbeit als nicht erfüllt angesehen werden. Zwar erfolgt die Aufteilung von Migrationsaktivitäten in die zwei Aspekte der Migration, doch eine ausreichende Unterstützung für das Migrationsteam ist dadurch nicht gegeben; auch fehlen formale Softwareartefakte. Der Migrationsprozess der Autoren sieht eine Wiederherstellung der Architekturbeschreibung (A2) vor, weshalb die entsprechende Anforderung als erfüllt angesehen wird. Ein durch diese bestehende Arbeit nicht betrachteter Aspekt der Migration ist die Identifikation kohäsiver Softwarebausteine (A3). Damit ist diese Anforderung nicht erfüllt. Ebenfalls nicht explizit im Migrationsprozess vorgesehen ist die Etablierung eines Domänenverständnisses in dem Entwicklungsteam. Zwar wird auf DDD verwiesen, doch die Befähigung des Entwicklungsteams steht nicht im Vordergrund. Dagegen wird DDD in Bezug auf den domänengetriebenen Entwurf der Microservice-Architektur verwendet. Aus diesem Grund gilt die Anforderung der Etablierung eines Domänenverständnisses (A4) als nicht erfüllt, dafür jedoch die Anforderung des domänengetriebenen Entwurfs der Microservice-Architektur (A5). Als Teil des Migrationsprozesses wird die Reorganisation des monolithischen Entwicklungsteams (A6) in kleinere Service-Teams gesehen, weshalb die entsprechende Anforderung als erfüllt gilt. Die Stärke des Migrationsprozesses von Balalaie et al. liegt insbesondere bei der Berücksichtigung des komplexen Betriebs (A7). Der Aspekt der Querschnittsänderungen befasst sich überwiegend mit der Etablierung grundlegender DevOps-Prinzipien. Aus diesem Grund gilt diese Anforderung als erfüllt.

3.2.5 Gysel et al. - Service Cutter: A Systematic Approach to Service Decomposition

Einen weiteren Ansatz zur Migration eines monolithischen Softwaresystems in eine Microservice-Architektur stellen Gysel et al. [GK+16] vor. Der Ansatz der Autoren umfasst ein semi-automatisiertes Ableiten von Microservice-Kandidaten auf Basis eines *gewichteten* und *ungerichteten Graphen*. Der Graph wird auf Basis von gewichteten *Kopplungskriterien* (engl. weighted coupling criteria) und *Softwaresystemartefakten* (engl. software system artifacts) konstruiert. Ein konstruierter Graph wird anhand vorgegebener Graphalgorithem in die Microservice-Kandidaten aufgeteilt. Verwendet werden die Algorithmen *Girvan-Newman* [NG04] und *Epidemic Label Propagation* (ELP) [RA+07]. Die Anwendung der Graphalgorithmen erfolgt durch den von den Autoren entwickelten *Service Cutter*, ein auf Java basierendes Werkzeug. Abschließend führen die Autoren einen Migrationsprozess ein, der die Anwendung des vorgestellten Ansatzes unterstützt.

Die Abbildung 3.4 zeigt das sequentielle Vorgehen bei der Anwendung des Service Cutter-Ansatzes. Es erfolgt eine klare Trennung zwischen manuell durch einen Softwarearchitekten ausgeführten Prozessschritten und den automatisierten Prozessschritten, die durch das Service Cutter-Werkzeug ausgeführt werden. Zunächst erstellt der Softwarearchitekt die formalen Softwaresystemartefakte,

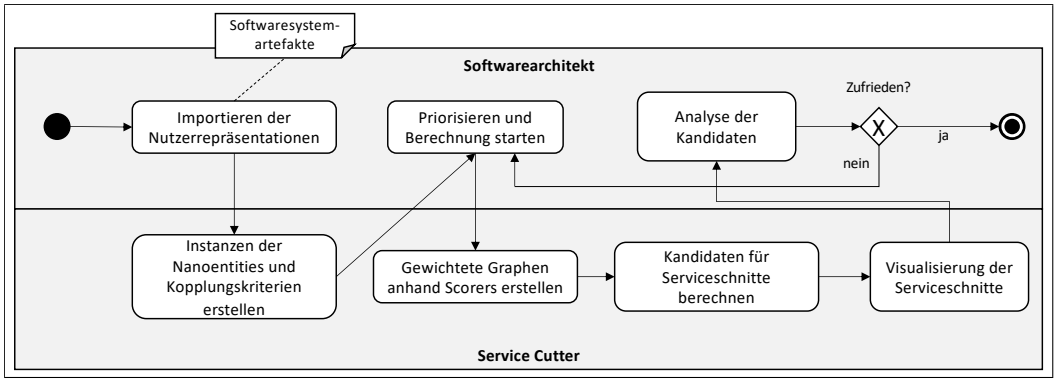

Abbildung 3.4: Prozessuales Vorgehen des Service Cutter-Ansatzes [GK+16]

welche Informationen über bestimmte *Nanoentitäten* des Softwaresystems verfügen. Nanoentitäten stellen entweder (1) Daten, (2) Operationen oder (3) Artefakte dar. Ein Softwaresystemartefakt modelliert somit den strukturellen und verhaltensspezifischen Ist-Zustand des Monolithen. Im Kontext der Arbeit werden die Softwaresystemartefakte auch als *Benutzerrepräsentationen* bezeichnet. Ausgehend von den Benutzerrepräsentationen werden automatisiert Instanzen von Nanoentitäten und Kopplungskriterien erstellt. Die maschinelle Interpretierbarkeit der Benutzerrepräsentationen ist durch die Verwendung von JSON als Modellierungssprache möglich. Eine Visualisierung durch Modellierungssprachen wie die UML [BD09] ist nicht möglich. Die Benutzerrepräsentationen müssen durch den Softwarearchitekten priorisiert werden, um anschließend die Berechnung des gewichteten Graphen starten zu können. Aufbauend auf dem gewichteten Graphen werden Kandidaten für Microservices berechnet. Ein Microservice-Kandidat besteht somit aus Nanoentitäten; eine Nanoentität kann dabei nur einem Kandidaten zugeordnet sein. Das visualisierte Ergebnis wird abschließend durch den Softwarearchitekten bewertet und gegebenenfalls durch Anpassung der Prioritäten in einem weiteren Berechnungslauf des Service Cutters überarbeitet.

Bei der Berechnung der Microservice-Kandidaten sehen Gysel et al. neben den bekannten Gruppierungsstrategien von DDD [Ne15] weitere durch Interessenvertreter eingebrachte architekturelle Anforderungen als relevant an. Aufbauend auf Literaturanalysen, Retrospektiven von vergangenen Projekten und Expertenworkshops wurde ein *Kopplungskriterienkatalog* (engl. coupling criteria catalog) abgeleitet, der die signifikantesten architekturellen Anforderungen als Kopplungskriterien festhält (Tabelle 3.6). Ein Kopplungskriterium beschreibt dabei die gewichtete Beziehung zwischen Nanoentitäten, wobei eine hohe Gewichtung für eine hohe Kopplung zwischen den Nanoentitäten steht. Architekturelle Anforderungen referenzieren auf nicht-funktionale Anforderungen, die während der Analyse und des Entwurfs des Softwaresystems berücksichtigt werden. Der eingeführte Kopplungskriterienkatalog kann somit während der Analyse und Entwurfsphase den Softwarearchitekten beim Treffen von Entwurfsentscheidungen unterstützen. Weiterhin dient der Katalog als ubiquitäre Sprache bei der Dekomposition des Monolithen in Services.

Tabelle 3.6: Kopplungskriterienkatalog

Kopplungskriterium	Gruppierung
Semantische Ähnlichkeit	Kohäsion
Identitäts- und Lebenszyklusgemeinsamkeit	
Geteilter Eigentümer	
Latenz	
Sicherheitskontext	
Strukturelle Volatilität	Kompatibilität
Konsistenzkritizität	
Speicherähnlichkeit	
Inhaltliche Volatilität	
Verfügbarkeitskritizität	
Sicherheitskritizität	
Konsistenzeinschränkung	Einschränkungen
Sicherheitseinschränkung	
Vordefinierte Service-Einschränkung	
Wandlungsfähigkeit	Kommunikation
Tauglichkeit des Netzwerkverkehrs	

Bewertung

Der von Gysel et al. vorgestellte Migrationsansatz sieht eine semi-automatisierte Migration vor, wobei lediglich der Entwurf der Microservice-Architektur durch das Werkzeug Service Cutter automatisiert wird. Die Verwendung des Service Cutters beruht auf einer klaren Prozessbeschreibung. Die Eingabeparameter des Service Cutters stellen formale Softwareartefakte dar, die von den Autoren selbst definiert wurden. Die Systematik der Autoren sieht jedoch nicht alle notwendigen Aktivitäten einer Migration vor, so ist beispielsweise die Wiederherstellung der Architekturbeschreibung nicht abgebildet. In Anbetracht des gegebenen Prozesses und der formalen Softwareartefakte kann die Anforderung, einen systematischen und nachvollziehbaren Migrationsprozess (A1) einzuführen als teilweise erfüllt angesehen werden. Durch die eingeführten Softwareartefakte und das Vorsehen dieser als Eingabeparameter ist die Wiederherstellung der Architekturbeschreibung (A2) zwar nicht explizit im Migrationsprozess berücksichtigt, aber obligatorisch für den Migrationsansatz. Daher wird diese Anforderung als erfüllt angesehen. Die Anforderung der Identifikation kohäsiver Softwarebausteine (A3) ist durch das Werkzeug Service Cutter sogar automatisiert vorgesehen, weswegen diese Anforderung erfüllt ist. Weniger von Relevanz ist das Etablieren eines Domänenverständnisses (A4), zumal der Migrationsansatz nur durch einen Softwarearchitekten durchgeführt wird. Das Entwicklungsteam oder sonstige Interessenvertreter sind nicht involviert. Die betreffende Anforderung ist damit nicht erfüllt. Der aus dem Service Cutter resultierende Entwurf der Microservice-Architektur basiert auf der Priorisierung von Kopplungskriterien durch den Softwarearchitekten, was dazu führt, dass nicht zwangsläufig ein domänengetriebener Entwurf der Microservice-Architektur (A5) entsteht. Die Orientierung an der Domäne wird damit nicht forciert, ist aber prinzipiell möglich. Die Anforderung wird daher als teilweise erfüllt angesehen. Das Reorganisieren des monolithischen Entwicklungsteams

(A6) und die Berücksichtigung des komplexen Betriebs (A7) der neuen Microservices werden von den Autoren nicht betrachtet, weswegen beide Anforderungen als nicht erfüllt gelten.

3.2.6 Knoche und Hasselbring - Using Microservices for Legacy Software-Modernization

Die Autoren Knoche und Hasselbring berichten in [KH18] von ihren Erfahrungen während der Migration eines Legacy-Softwaresystems. Das Softwaresystem fußt auf einer monolithischen Softwarearchitektur und soll aufgrund der Komplexität des Softwaresystems und der Ineffizienzen bei der Softwareentwicklung in eine Microservice-Architektur überführt werden. Über einen aus fünf Schritten bestehenden Prozess soll Schritt für Schritt eine Migration erfolgen. Für die einzelnen Prozessschritte wird neben den darin auszuführenden Tätigkeiten auch auf Best Practices und häufige Fehler hingewiesen. Der Fokus des Migrationsprozesses liegt auf der Dekomposition des monolithischen Softwaresystems.

Betrachtet wird ein monolithisches Legacy-Softwaresystem in der Versicherungsdomäne. Das Softwaresystem verfolgt den Zweck, die Kunden der Versicherungsorganisation zu verwalten, daher wird es als *Customer Application* bezeichnet. Im klassischen Sinne werden in dem Softwaresystem alle Informationen zu ihren Kunden abgelegt. Das seit 1970 bestehende Softwaresystem wurde bereits mehrfach modernisiert, ohne die darunterliegende Softwarearchitektur zu erneuern. Zum Stand des Migrationsvorhabens wurde das Softwaresystem mit den Programmiersprachen Cobol und Java Swing realisiert. Die Kundendaten des Monolithen werden ebenfalls in anderen Softwaresystemen der Organisation benötigt, weshalb das Softwaresystem auch eine zentrale Rolle in der Organisation spielt. Die Persistierung der Daten erfolgt über eine mit anderen Softwaresystemen geteilte Datenbank. Der Zugriff auf die Daten erfolgt über unterschiedliche Wege: (1) vordefinierte Schnittstellen; (2) Aufruf von Modulen (Quellcode); (3) Zugriff auf das Datenbankschema. Die Verwendung von Schnittstellen stellt dabei den kleineren Teil der Zugriffe dar.

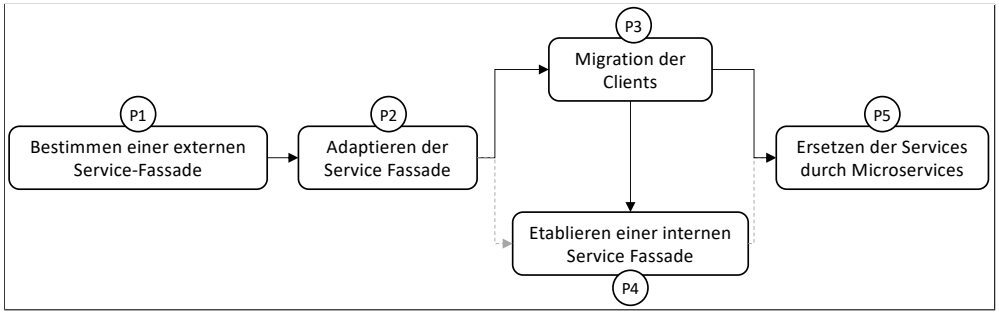

Abbildung 3.5: Schrittweise Darstellung des Migrationsprozesses aus [KH18]

Der vorgestellte Migrationsprozess (siehe Abbildung 3.5) zielt darauf ab, in einer Iteration den gesamten Monolithen zu modernisieren. Mittels fünf definierter Prozessschritte wird Schritt für Schritt der Monolith in eine Microservice-Architektur überführt. Im ersten Schritt (P1) werden Schnittstellen und ihre Operationen für den Zugriff von externen Softwaresystemen für den Monolithen konzipiert, ohne "fragwürdige Entwurfsentscheidungen" des Monolithen zu übernehmen. Die Autoren unterscheiden zwischen Schnittstellen und Operationen, die innerhalb einer Schnittstelle möglich sind. In diesem Prozessschritt erfolgt noch keine Entwicklung der entworfenen Schnittstellen. Damit fachliche und auf die Nutzerbedürfnisse zugeschnittene Schnittstellen entworfen werden, wird ein *Zieldomänenmodell* entwickelt. In diesem Modell wird der erwartete Zustand der Schnittstellen abgebildet. Anschließend erfolgt eine statische Analyse des Quellcodes, um die tatsächlichen Zugriffe von externen Softwaresystemen abzubilden. Als Beispiele nennen die Autoren die Methoden im Quellcode oder auch die Datenbanktabellen. Ausgehend von dem Ist-Zustand werden dann die bereits verwendeten Schnittstellen als *provisorische Schnittstellen* deklariert. Semantisch identische provisorische Schnittstellen werden dann zusammengeführt. Durch das Zieldomänenmodell und die statische Analyse liegen nun Ist- und Soll-Zustand vor. Abschließend werden im ersten Prozessschritt die Schnittstellen und Operationen des Ist-Zustands mit dem Soll-Zustand verknüpft.

Der zweite Prozessschritt (P2) sieht die Implementierung der festgelegten Schnittstellen vor. Um den Aufwand der Implementierung möglichst gering zu gestalten, wird der Monolith zunächst adaptiert. Neue Schnittstellen werden über *Adaptoren* an die bestehende Funktionalität des Monolithen gebunden. Die Identifikation der Adaptoren kann durch die in Prozessschritt 1 erstellte Verknüpfung zwischen Ist- und Soll-Zustand unterstützt werden. Dennoch ist es möglich, dass neue Funktionalität in den Monolithen gebracht werden muss, um den Soll-Zustand abzubilden. Wichtig während und nach der Durchführung dieses Prozessschrittes ist das Testen des Monolithen. Die Autoren heben hervor, dass gerade das Testen in Legacy-Softwaresystemen häufig vernachlässigt wird.

Bestehen die neuen Schnittstellen des Monolithen, werden in dem dritten Prozessschritt (P3) die abhängigen Softwaresysteme (nachfolgend auch Clients) migriert. Nach dem Prozessschritt wird die Funktionalität des Monolithen nur noch über die implementierten Schnittstellen aufgerufen. Zur Unterstützung der Migration der Clients wird eine sogenannte *Transition Documentation* erstellt. In dieser Dokumentation befinden sich textuelle Beschreibungen und Quellcodeausschnitte, auf die Entwicklungsteams bei der Anpassung ihres Clients zurückgreifen können. Detailliert wird die Transition der alten Schnittstelle auf die neue Schnittstelle mit ihren Operationen dargestellt.

Im Rahmen der Client-Migrationen wurden ebenfalls Adaptoren in die Clients integriert, um auch dort den Aufwand für Anpassungen zu minimieren. Während der Migration stellten sich wenige neue Schnittstellen als zu fein-granular heraus, weshalb diese angepasst wurden. Um die Migration der Clients weiterhin zu überwachen, wurde die statische Analyse aus Prozessschritt 1 kontinuierlich ausgeführt.

Der Prozessschritt 4 (P4) sieht eine Migration des Monolithen vor. Durch die Etablierung der neuen Schnittstelle als *Fassade* für die Kommunikation mit dem Monolithen kann dieser Prozessschritt parallel zu der Migration der externen Softwaresysteme erfolgen. Änderungen, welche die innere Architektur des Monolithen betreffen, können durch die Schnittstellen verschleiert werden. Im Kontext der bestehenden Arbeit wurde jedoch auf eine sequenzielle Migration gesetzt. Als Herausforderung nennen die Autoren die Identifikation der kohärenten Konzepte des Monolithen.

Abschießend erfolgt mit dem Prozessschritt 5 (P5) die eigentliche Migration des Monolithen in eine Microservice-Architektur. Durch die in den Prozessschritten P1 - P4 durchgeführten Arbeiten kann eine transparente Migration erfolgen.

Bewertung

Die Autoren Knoche und Hasselbring stellen einen manuellen Migrationsprozess vor, der systematische Extraktion von Softwarebausteinen aus dem Monolithen vorsieht, ohne diese direkt in eine Microservice-Architektur zu überführen. Der vorgestellte Migrationsprozess nimmt eine sehr eingeschränkte Sicht auf die Migration ein, die jedoch wegweisend für die Durchführung einer Migration ist. Aufgrund der stark eingeschränkten Sicht auf die Migration und das Ausbleiben formaler Softwareartefakte wird die Anforderung, einen systematischen und nachvollziehbaren Migrationsprozess (A1) zu beschreiben, als nur teilweise erfüllt angesehen. Die für die Extraktion wichtige Wiederherstellung einer Architekturbeschreibung (A2) ist erfüllt, denn sie wird im Rahmen einer Migrationsaktivität betrachtet. Ebenfalls nicht betrachtet ist eine Identifikation kohäsiver Softwarebausteine (A3), die gerade im Bezug auf die Extraktion von Softwarebausteinen von hoher Relevanz ist; kohäsive Komponenten werden als bereits vorhanden angenommen. Die betreffende Anforderung ist daher nicht erfüllt. Das Etablieren eines Domänenverständnisses (A4) bei dem Entwicklungsteam liegt ebenfalls nicht im Fokus der Autoren, weswegen auch diese Anforderung nicht erfüllt ist. Knoche und Hasselbring beschreiben, dass mit dem vorgestellten Migrationsprozess eine Trennung des Monolithen anhand der Geschäftsdomäne stattfinden soll. Konkrete Ansätze werden jedoch nicht genannt. Aus diesem Grund gilt die Anforderung des domänengetriebenen Entwurfs der Microservice-Architektur (A5) nur als teilweise erfüllt. Eine Anpassung des monolithischen Entwicklungsteams (A6) an den neuen Architekturstil ist nicht als Teil dieser Arbeit anzusehen, denn im Grunde erfolgt keine konkrete Migration in eine Microservice-Architektur; die Anforderung ist damit automatisch nicht erfüllt. Die Autoren beschreiben oberflächlich Konzepte, die den komplexen Betrieb einer Microservice-Architektur berücksichtigen (A7), doch die konkrete Verwendung dieser Konzepte bleibt aus. An dieser Stelle ist die betreffende Anforderung nur teilweise erfüllt.

3.2.7 Mazlami et al. - Extraction of Microservice from Monolithic Architectures

In [MC+17] wird ein semi-automatisiertes Verfahren zur Identifikation von Microservices innerhalb eines monolithischen Softwaresystems vorgestellt. Die Autoren haben hierfür einen Algorithmus entwickelt, der basierend auf drei *Kopplungsstrategien* (engl. coupling strategies) einen Graphen generiert (siehe Abbildung 3.6). Im ersten Schritt des Algorithmus wird der Monolith anhand seiner Meta-Informationen im Quellcode analysiert und in zwei verschiedene Graphen überführt. Anschließend wird der Graph genutzt, um die Bestandteile des Monolithen in Gruppen zusammenzufassen. Der letzte Graph stellt den Vorschlag für die Microservice-Kandidaten dar. Zur Erstellung der Graphen benötigt der Algorithmus keine manuelle Eingabe von Informationen durch den Anwender.

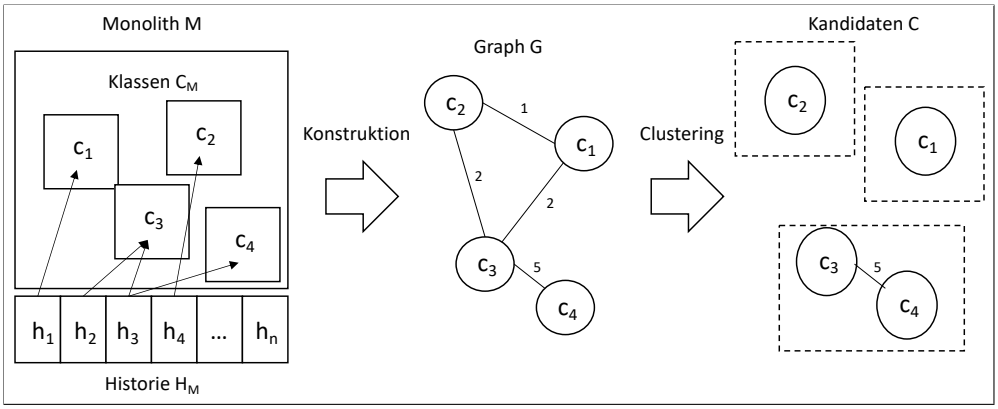

Abbildung 3.6: Beispielhafte Überführung eines Monolithen in Microservice-Kandidaten anhand des Algorithmus aus [MC+17]

Für die Erstellung der Microservice-Architektur sieht der Algorithmus drei Stufen vor, die über zwei Transitionen erreicht werden. Ausgangspunkt des Algorithmus ist der Quellcode des Monolithen inklusive seiner Meta-Informationen, die sich in einem Repository eines Versionierungswerkzeugs wie beispielsweise Git [Git-Doc] befindet. Die erste Stufe ist somit der rekonstruierte Monolith $M = \{C_M, H_M, D_M\}$, der als Tripel von drei Mengen dargestellt wird. In dem Tripel befinden sich die Menge aller Klassen C_M, die Historie der Änderungen H_M und beteiligte Entwickler D_M. In der Menge C_M befinden sich alle im Monolithen befindlichen Klassen, wobei die Annahme gestellt wird, dass Klassen einzigartig vorhanden sind. Gleichnamige Klassen dürfen somit innerhalb des Monolithen nicht existieren. Die Historie der Änderungen H_M stellt eine Sequenz an Ereignissen wie beispielsweise *Commits* der Entwickler in das Repository dar. Ein Ereignis $h_i = \{E_i, t_i, d_i\}$ besteht ebenfalls aus einem Tripel, wobei E_i die Menge an Klassen aus C_M bezeichnet, t_i den Zeitstempel des Ereignisses und d_i den auslösenden Entwickler. Der rekonstruierte Monolith M wird dann durch die *Construction-Transition* in den Graphen $G = \{E, V\}$ überführt, der durch die Kanten E und Knoten V beschrieben wird. Während der Ausführung der Construction-Transition wird eine der drei Kopplungsstrategien herangezogen. Unabhängig von der Kopplungsstrategie entsteht ein *ungerichteter*,

gewichteter Graph. Die Knoten V entsprechen den aus der Menge C_M bekannten Klassen. Die Kanten E verbinden die Knoten und werden über eine *Gewichtungsfunktion* mit einer Wertigkeit versehen. Aus der Funktion lässt sich die *Stärke der Kopplung* zwischen zwei Knoten E_i und E_j bestimmen; je höher der Wert, desto höher ist die Kopplung. Welchen Wert die Kanten einnehmen, hängt von der gewählten Kopplungsstrategie ab.

Ist der Graph G vollständig aufgebaut, kann die nächste Transition ausgeführt werden. Durch das *Clustering* wird der Graph in Empfehlungen für Microservices unterteilt. Die Empfehlungen R sind formell als Wald strukturiert, der aus einer beliebigen Anzahl an Bäumen besteht, die wiederum gerichtete Graphen S darstellen. Die Knoten eines Baumes entsprechen den Klassen C_M. Ein Baum korrespondiert mit einem Microservice-Kandidaten.

Zentral für den Algorithmus sind die drei unterschiedlichen Kopplungsstrategien, welche für den Aufbau des Graphen G verantwortlich sind. Jede Strategie führt zu einer anderen Ausprägung von G für denselben Monolithen M. Die Strategien sind wie folgt definiert:

Logische Kopplung Diese Strategie beruft sich auf das *Single Responsible Principle* (SRP) [Ma09]. Das SRP besagt, dass Komponenten eines Softwaresystems nur durch einen einzigen Grund verändert werden dürfen. Um dem gerecht zu werden, sind genau diese Softwarebausteine zusammenzufassen, die durch denselben Grund einer Änderung unterzogen werden. Die Strategie identifiziert die zusammengehörigen Bausteine über die Historie H. Wurden zwei Klassen $\{c_i, c_j\}$ im selben Ereignis h_i modifiziert, wird von einer logischen Kopplung zwischen den Klassen ausgegangen. Für jedes mögliche Tupel der Klassen C_M wird während der Construction-Transition diese Gewichtung auf Basis der Historie H_M ausgeführt. Es entsteht für jede Kante zwischen zwei Klassen eine Gewichtung.

Semantische Kopplung Die Grundlage der semantischen Kopplung stellt das *Bounded Context*-Konzept aus DDD dar. Gekoppelt sind im Kontext dieser Strategie die Klassenpaare, welche auf denselben *Dingen* bzw. *Domänenobjekten* agieren. Über den Wortgewichtungsalgorithmus *Vorkommenshäufigkeit Inverse Dokumentenhäufigkeit* (engl.Term-Frequency Inverse Document Frequency, TF-IDF) wird die Häufigkeit von Begriffen im Quellcode der Klassen C_M bestimmt und als Skalar-Vektor abgebildet. Basis der Berechnung sind *Identifizierer* und *Ausdrücke*, welche in den Variablen oder Methoden der Klasse verwendet werden. Betrachtet wird für die Gewichtung auf der Kante ein Klassenpaar c_i, c_j und ihr jeweiliger Vektor, um das Gewicht der Kante zu berechnen, werden beide Vektoren X, V über den Algorithmus der *Kosinus-Ähnlichkeit* auf ihre Ähnlichkeit geprüft.

Mitwirkerkopplung Das Ziel dieser Strategie ist der Entwurf einer Microservice-Architektur, welche einen minimalen Kommunikationsaufwand zwischen den für die Microservices verantwortlichen Entwicklungsteams gewährleistet. Die Bestimmung der Kanten zwischen einem

Klassenpaar c_i, c_j beruht auf der Historie der Änderungen H_M. Zunächst werden für alle Klassen die betreffenden Änderungen in Form von Mengen festgehalten. Anschließend erfolgt die Zuordnung der Entwickler D_M zu den jeweiligen Änderungen der Klasse. Es entsteht eine Abbildung der Entwickler auf die Klassen des Monolithen. Durch diese Abbildung können für den Graphen G die Gewichtungen der Kanten über die Kardinalität der Schnittmengen der Klassenpaare bestimmt werden.

Nach der Construction-Transition folgt das Aufteilen des Graphen G in die einzelnen Bäume und somit in Microservice-Kandidaten. Die *Clustering-Transition* entscheidet anhand der Gewichtung über das Eliminieren einer Kante des Graphen. Hohe Gewichtungen stehen für eine hohe Kopplung, daher sind die niedrigsten Werte die Ausgangspunkte. Um zu vermeiden, dass der Algorithmus alle Kanten des Graphen löscht, behelfen sich die Autoren mit zwei Bedingungen. Zum einen agiert der Algorithmus nur auf Teilgraphen, den sogenannten minimalen Spannbäumen (engl. minimal spanning trees). Zum anderen erwartet der Algorithmus eine manuelle Eingabe zur Bestimmung der Anzahl der Iterationen, da er selbst nicht in der Lage ist, automatisch das Partitionieren zu beenden.

Bewertung

Die Arbeit von Mazlami et al. stellt einen Migrationsprozess vor, der basierend auf einer Historie einer Quellcodeverwaltung automatisiert Microservice-Kandidaten ableitet. Durch die detailliert beschriebenen Algorithmen ist die Migration ausgehend von der Analyse bis hin zum Entwurf der Microservice-Architektur systematisch und nachvollziehbar. Jedoch ist die abschließende Implementierung der Microservices durch die Autoren nicht betrachtet. Damit liegt zwar eine Systematik und Nachvollziehbarkeit vor, doch die Migration wird nicht ganzheitlich betrachtet. So ist die Anforderung, einen systematischen und nachvollziehbaren Migrationsprozess (A1) zu beschreiben nur teilweise erfüllt. Die Algorithmen bauen auf Basis der Quellcodeverwaltung eine für die eigenen Zwecke gerechte Architekturbeschreibung auf; die konkrete Softwarearchitektur des Monolithen in Form formaler Softwareartefakte wird nicht wiederhergestellt. Dennoch gilt die Anforderung zur Wiederherstellung der Architekturbeschreibung (A2) als erfüllt, da eine spezielle Architekturbeschreibung automatisiert generiert wird. Weiterhin gilt die Identifikation kohäsiver Softwarebausteine (A3) als erfüllt. Die Entscheidung zur Kohäsion wird anhand der Historie getroffen. Nicht vorgesehen ist das Involvieren des Entwicklungsteams, weshalb auch kein Domänenverständnis etabliert (A4) wird. Die korrespondierende Anforderung ist damit nicht erfüllt. Die angewendeten Algorithmen zielen nicht zwangsläufig auf einen domänengetriebenen Entwurf der Microservice-Architektur (A5) ab, was jedoch nicht einen domänenorientierten Entwurf ausschließt. Die Anforderung wird daher als teilweise erfüllt angesehen. Weder die Anforderung der Reorganisation des monolithischen Entwicklungsteams (A6), noch die Anforderung der Berücksichtigung des komplexen Betriebes (A7) des

Tabelle 3.7: Ergebnisse der Bewertung bestehender Arbeiten

	A1 - Systematischer und nachvollziehbarer Migrationsprozess	A2 - Wiederherstellung der Architekturbeschreibung	A3 - Identifikation kohäsiver Softwarebausteine	A4 - Etablieren eines Domänenverständnisses	A5 - Domänengetriebener Entwurf der Microservice-Architektur	A6 - Reorganisation des monolithischen Entwicklungsteams	A7 - Berücksichtigung des komplexen Betriebs
[BH+18] Microservices Migration Patterns	◐	●	◐	●	○	○	●
[HR20] Migration to Microservices	◐	○	●	○	●	○	●
[Ne19] Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith	○	○	○	◐	●	◐	○
[BH+16] Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture	○	●	○	○	●	●	●
[GK+16] Service Cutter: A Systematic Approach to Service Decomposition	◐	●	●	○	◐	○	○
[KH18] Using Microservices for Legacy Software-Modernization	◐	●	○	○	◐	○	◐
[MC+17] Extraction of Microservices from Monolithic Architectures	◐	●	●	○	◐	○	○

neuen Architekturstils werden von den Autoren betrachtet. Beide Anforderungen sind daher nicht erfüllt.

3.3 Abgeleiteter Handlungsbedarf

In Tabelle 3.7 werden der Bewertungen zu den bestehenden Arbeiten aus Abschnitt 3.2 zusammenfassend dargestellt. Die Bewertungen werden in der Tabelle wie folgt abgebildet: Eine vollständig erfüllte Anforderung wird anhand des Symbols ● angezeigt, eine nur zum Teil erfüllte Anforderung hingegen durch das Symbol ◐. Wird die Anforderung gar nicht erfüllt, wird dies durch das Symbol ○ gekennzeichnet.

Durch die in Tabelle 3.7 gegebene Übersicht wird ersichtlich, dass keine der bestehenden Arbeiten alle Anforderungen (vgl. Abschnitt 3.1) erfüllt. Aus dieser Gegebenheit lässt sich ein Handlungsbedarf für die iterative Migration eines monolithischen Softwaresystems in eine Microservice-Architektur ableiten. Dieser Handlungsbedarf stellt gleichzeitig die Motivation für die Forschungsbeiträge der vorliegenden Arbeit dar.

Eine erfolgreiche iterative Migration muss durch eine geeignete Systematik unterstützt werden, damit das durchführende Migrationsteam in jeder Phase der laufenden Migration unterstützt wird. Weiterhin ist die Nachvollziehbarkeit für das Treffen von Entscheidungen in den einzelnen Migrationsaktivitäten relevant, weshalb der Einsatz formaler Softwareartefakte gefordert ist. Ein solches vollständiges Migrationsrahmenwerk, das diese geforderte Systematik und Nachvollziehbarkeit erfüllt, wird von keiner der bestehenden Arbeiten vollständig abgedeckt. Die Arbeit von Knoche und Hasselbring [KH18] liefert jedoch mit der vorgestellten Extraktion von *Monolithenbestandteilen* einen wichtigen Beitrag für die ersten Migrationsaktivitäten, auf den für die Gestaltung des Migrationsrahmenwerks zugegriffen werden kann. Jedoch muss für die vorgestellte Extraktion eine Erweiterung hinsichtlich der Bestimmung bestehender Anforderungen und der Identifikation der kohäsiven Softwarebausteine erfolgen. Weiterführend können aus den Arbeiten [BH+18], [Ne19] und [HR20] vereinzelt Migrationsmuster entnommen werden, die als feste Migrationsaktivität im Migrationsrahmenwerk vorgesehen werden. Insbesondere können die Vorbereitungen für den Betrieb der Microservices genutzt werden. Jedoch müssen für die verwendeten Migrationsmuster noch Vorarbeiten erfolgen, die nachvollziehbar in der Systematik abgebildet werden müssen. Es kann damit festgehalten werden, dass für die Bereitstellung eines Migrationsrahmenwerkes die bestehenden Arbeiten erweitert werden müssen.

Das Verständnis über die Geschäftsdomäne, welche die grundlegende Funktionsweise des Softwaresystems beeinflusst, ist für die Durchführung einer Migration unabdingbar. Denn sowohl bestehende als auch neue Funktionalität, die im Kontext einer Migration eingeführt werden kann, bedingt für eine korrekte Interpretation und auch Validierung nach der Migration ein tieferes Verständnis über die Geschäftsdomäne. Zwar greifen Arbeiten wie [BH+16], [KH18] oder [Ne19] auf DDD zurück, der die Geschäftsdomäne bei der Entwicklung in den Vordergrund setzt, aber das Etablieren des Verständnisses bei den einzelnen Softwareentwicklern liegt nicht im Fokus. Durch die hohe Relevanz des Verständnisses für den Erfolg der Migration wird daher ein weiterer Handlungsbedarf gesehen.

Damit eine hochgradig wartbare und wiederverwendbare Microservice-Architektur während der Migration etabliert wird, muss bei dem Entwurf eine Orientierung an der Geschäftsdomäne erfolgen. Wie bereits erwähnt, greifen hierfür die Arbeiten [BH+16, KH18] auf DDD und das damit verbundene Konzept der *Bounded Contexts* (vgl. Abschnitt 2.4.4) zurück. Mit DDD wird insbesondere die Granularität und Kohäsion eines Microservices betrachtet, weshalb für die Ausgestaltung der iterativen Migration ebenfalls auf DDD zurückgegriffen. Weiterführend muss jedoch die Anwendung von DDD durch das Migrationsrahmenwerk unterstützt werden, denn oft fehlt es den bestehenden Arbeiten an der systematischen Verwendung von DDD, sodass das Migrationsteam während des Entwurfs nicht geführt wird.

Das Reorganisieren des monolithischen Entwicklungsteams spielt nach der Migration eine entscheidende Rolle für die Wartung und den Betrieb der Microservices. Die Arbeit [BH+16] sieht eine Reorganisation anhand von Service-Teams vor, die über einen Querschnitt an Fähigkeiten verfügen, sodass letztendlich das Service-Team den gesamten Lebenszyklus des Microservices begleiten kann.

Die Struktur des Service-Teams nach [BH+16] stellt eine gute Grundlage für die weiterführende Betrachtung der Reorganisation dar. Aspekte wie das Berücksichtigen unterschiedlicher Erfahrungen zur Weiterentwicklung einzelner Softwareentwickler oder das Etablieren von Service-Teams bei kleinen monolithischen Entwicklungsteams, um *Interessenskonflikte* zu vermeiden, müssen noch betrachtet werden.

4 Migrationsrahmenwerk der iterativen Migration

Die Entwicklung eines Monolithen kann absichtlich oder unabsichtlich erfolgen. So verfolgt beispielsweise der Ansatz *MonolithFirst* [Fo15] das Ziel, zunächst einen Monolithen zu entwickeln, damit der Fokus auf das Verständnis der Geschäftsdomäne gelegt wird. Die Mehrheit der Monolithen entsteht jedoch unabsichtlich, gerade getrieben durch kurze Fristen bei der Softwareentwicklung [Su00] oder einfach, weil Organisationen nicht abschätzen können, wie stark das eigene Softwaresystem wachsen wird [Ru19]. Im Laufe der Weiterentwicklung steigt die Komplexität des Monolithen an, bis letztendlich die Effizienz und Effektivität der Softwareentwicklung abnimmt [FL14, Ri19]. Befindet sich die Entwicklungsabteilung in dieser Situation, muss eine Entscheidung bezüglich des Softwaresystems getroffen werden. Beschlossen wird meist die Migration des Softwaresystems und damit die Modernisierung der bislang monolithischen Architektur mittels des Architekturstils der Microservices. In der Praxis wurden bereits von mehreren Organisationen erfolgreich Migrationen durchgeführt [Ot15, Ha15, Iz16, Hm19]. Somit ist sowohl die Tragfähigkeit einer Microservice-Architektur als auch die Migration von monolithischen Softwaresystemen in den modernen Architekturstil selbst bewiesen. Doch aufbauend auf dem identifizierten Handlungsbedarf aus Kapitel 3, muss für die iterative Migration ein *Migrationsrahmenwerk* geschaffen werden, um das Entwicklungsteam bei dem Vorhaben zu unterstützen.

Der Entwurf des Migrationsrahmenwerks ist der zentrale Forschungsbeitrag dieser Arbeit und bildet gleichzeitig den Rahmen für die weiteren Beiträge (vgl. Abschnitt 1.5). Zum Entwurf des Migrationsrahmenwerks müssen mehrere Fragestellungen beantwortet werden: (1) *Welche Elemente müssen für eine iterative Migration definiert werden?* (2) *Welche Migrationsaktivitäten sind für die Modernisierung eines Inkrements notwendig?* (3) *Welche formalen Softwareartefakte unterstützen die Nachvollziehbarkeit der Systematik?* (4) *Wie wird der parallele Betrieb des geschaffenen Inkrements und des weiterhin bestehenden Monolithen sichergestellt?* Für das Migrationsrahmenwerk wird bei der Beantwortung der Fragestellungen die geforderte Flexibilität berücksichtigt, sodass das Entwicklungsteam eigene Softwareentwicklungsansätze einbringen kann. Eingesetzt werden verschiedene Methodiken der Softwaretechnik. So hilft das *Design Recovery* bei der Betrachtung des Ist-Zustandes des Monolithen [CC90]. Unterstützend wird auf die *statische Quellcodeanalyse* zurückgegriffen, die Abhängigkeiten zwischen Softwarebausteinen in der Mikroarchitektur offenlegt [Fe05]. Im Sinne der Flexibilität werden keine konkreten Konzepte oder Softwareartefakte von Softwareentwicklungsansätzen für die Ausgestaltung der Migrationsaktivitäten bei dem generellen Entwurf vorgesehen. Doch die Prinzipien von der *verhaltensgetriebenen Entwicklung* (engl. Behavior-Driven Development, BDD)

[Sm14] und des *domänengetriebenen Entwurfs* (engl. Domain-Driven Design, DDD) werden in die Systematik eingearbeitet.

Zu Beginn werden in Abschnitt 4.1 grundlegende Elemente der iterativen Migration definiert. Aus der Definition folgen die für das Migrationsrahmenwerk relevanten Elemente, welche beim Entwurf vorgesehen werden müssen. Anschließend wird in Abschnitt 4.2 die für das Migrationsrahmenwerk geforderte Systematik entworfen. Zum Entwurf der Systematik werden sequentiell ausgeführte Migrationsaktivitäten eingeführt. Um die Nachvollziehbarkeit der Systematik zu gewährleisten, werden in Abschnitt 4.3 Migrationsartefakte eingeführt. Abgeschlossen wird dieses Kapitel mit Abschnitt 4.4, der den parallelen Betrieb des Monolithen und der durch die Migrationsiteration geschaffenen Softwarebausteine behandelt. Die Betrachtung des parallelen Betriebs ist aufgrund der iterativen Vorgehensweise des Migrationsrahmenwerks notwendig.

4.1 Elemente einer iterativen Migration

Zunächst werden die *zentralen Elemente* einer iterativen Migration betrachtet. Darauf aufbauend werden *Definitionen* aufgestellt, die im weiteren Verlauf dieser Forschungsarbeit für den Entwurf des Migrationsrahmenwerks herangezogen werden. Die Beleuchtung des aktuellen Forschungsstands hat gezeigt, dass die Identifikation und explizite Benennung dieser Elemente noch nicht ausreichend betrachtet wurde (vgl. Abschnitt 3.3). Daher wird im Rahmen des Forschungsbeitrags diese Lücke geschlossen und eine Übersicht über die Elemente gegeben. Die Elemente werden anhand bestehender Arbeiten und dieser hier vorliegenden Forschungsarbeit identifiziert. Zu beachten ist, dass nicht zwangsläufig nur Arbeiten im Kontext iterativer Migrationen relevante Elemente beinhalten, weshalb auch ein erweiterter Forschungsgegenstand diesbezüglich betrachtet wird.

Das Formulieren der Definitionen erfolgt anhand unterschiedlicher Aspekte, welche die iterative Migration aus verschiedenen Blickwinkeln betrachten. Zunächst werden *organisationsbezogene Elemente* definiert, die das Ausführungsumfeld der iterativen Migration betreffen. Anschließend werden *prozessbezogene Elemente* betrachtet, welche die Ausführung einer iterativen Migration beschreiben. Abschließend werden *entitätenbezogene Elemente* betrachtet, welche bei der Ausführung einer iterativen Migration relevant sind.

4.1.1 Organisationsbezogene Elemente

Unter den *organisationsbezogenen Elementen* werden die Elemente verstanden, welche die Rahmenbedingungen der iterativen Migration betreffen. Eine Rahmenbedingung wird dabei durch die Organisation, in der die iterative Migration ausgeführt wird, vorgegeben. Die organisationsbezogenen

Tabelle 4.1: Organisationsbezogene Elemente einer iterativen Migration

Element	Kurzbeschreibung	Quellen
Organisation	Besitzt ein monolithisches Softwaresystem und stellt finanzielle und personelle Ressourcen für ein Migrationsvorhaben zur Verfügung. Die Kultur der Organisation kann durch die iterative Migration beeinflusst werden.	[DD+17, KH18]
Interessenvertreter	Wichtige Informationsquelle für Domänenwissen während der iterativen Migration.	[BH+18]
Migrationsteam	Eine aus dem Entwicklungsteam ausgegliederte Einheit, die nur Aufgaben innerhalb des Migrationsvorhabens übernehmen.	[SL13, BH+16]
Migrationsingenieur	Teammitglied des Migrationsteams.	[SL13]
Migrationsvorhaben	Eine durch das Entwicklungsteam getroffene Entscheidung zur Durchführung einer iterativen Migration.	[FB+19]

Elemente sind damit Elemente, die durch die Organisation in die iterative Migration eingebracht werden. Ihnen wird eine hohe Bedeutung zugemessen, da diese die Ausführung der iterativen Migration maßgeblich positiv oder negativ beeinflussen können. Alle organisationsbezogenen Elemente werden in Tabelle 4.1 zusammengefasst.

Das erste betrachtete Element ist die *Organisation* selbst, welche neben einem wirtschaftlichen Interesse an der Migration *finanzielle* und *personelle Ressourcen* zur Verfügung stellt [KH18]. Zudem verfügt die Organisation über eine *Kultur*, die gegebenenfalls durch die iterative Migration beeinflusst wird [DD+17]. Unter der Kultur wird die verfolgte Philosophie bei der Strukturierung der Organisation in Abteilungen und Kommunikationswege verstanden. Die personellen Ressourcen teilen sich in *Interessenvertreter* (engl. stakeholder) und Entwicklungsteam auf. Ein Interessenvertreter verfügt über ein hohes Wissen für geschäftsrelevante Vorgänge der Organisation und dient damit als wichtigste Quelle für Domänenwissen [BH+18]; die Bezeichnung Domänenexperte ist daher auch treffend [Ev04]. Das Entwicklungsteam setzt sich aus Softwareentwicklern und auch Softwarearchitekten zusammen, die über unterschiedliche Fähigkeiten verfügen [BH+16]. Bei dem Entwicklungsteam wird jedoch eine wichtige Unterteilung vorgenommen. Für die Ausführung der iterativen Migration wird ein dediziertes *Migrationsteam* geformt, dass orthogonal zu dem restlichen Entwicklungsteam und seinen Aufgaben stehen muss [SL13]. Darüber hinaus übernimmt das Migrationsteam die volle Verantwortlichkeit für Softwarebausteine innerhalb des monolithischen Softwaresystems, sobald diese in einem Inkrement der iterativen Migration betrachtet werden. Denn das restliche Entwicklungsteam darf, bedingt durch die Konsistenzerhaltung einer wiederhergestellten Architekturbeschreibung während einer Iteration, keine Softwareänderungen vornehmen. Ein Softwareentwickler wird im Kontext des Migrationsteams als *Migrationsingenieur* (engl. migration engineer) bezeichnet [SL13]. Abschließend wird als organisationsbezogenes Element noch das *Migrationsvorhaben* [FB+19] betrachtet, welches durch das Entwicklungsteam der Organisation vorgeschlagen wird. Die Organisation selbst muss das Migrationsvorhaben unterstützen, indem neben den finanziellen und personellen Ressourcen auch Weiterentwicklungen an dem monolithischen Softwaresystem reduziert werden.

Tabelle 4.2: Prozessbezogene Elemente einer iterativen Migration

Element	Kurzbeschreibung	Quellen
Iterative Migration	Besteht aus mehreren Migrationsiterationen, die ein einzelnes monolithisches Softwaresystem betreffen.	[BC+03b, Ri16, Ne19]
Migrationsiteration	Wird durch einen Migrationsauslöser gestartet und lässt das Migrationsteam die Systematik des Migrationsrahmenwerks durchlaufen.	[BC+03b, Bo06]
Systematik	Die sequentielle Verkettung von Migrationsaktivitäten für die ganzheitliche Durchführung einer Migrationsiteration.	[JA+13, TL+17, BH+18, HR20]
Migrationsaktivität	Basiert auf einer oder mehreren Aufgaben, die das Migrationsteam für die Generierung eines bestimmten Ergebnisses durchführen muss.	[AF+18]
Migrationsphase	Fasst Migrationsaktivitäten zusammen, die zur Erreichung eines gemeinsamen übergeordneten Zwischenziels der Migrationsiteration durchlaufen werden.	[GA05, Wa14, BH+18]
Migrationsauslöser	Ist eine funktionale oder nicht-funktionale Anforderung und kennzeichnet den Start einer Migrationsiteration.	[Fo04]

4.1.2 Prozessbezogene Elemente

Ein *prozessbezogenes Element* bezeichnet ein Element, welches für die Beschreibung der *Systematik* notwendig ist. Zusammenfassend werden die prozessbezogenen Elemente in Tabelle 4.2 dargestellt. Für die Identifikation der prozessbezogenen Elemente erfolgt eine Anlehnung an den *Rationale Unified Process* (RUP) [Kr04].

Die iterative Migration setzt eine iterative Vorgehensweise voraus, die prinzipiell aus Iterationen und Inkrementen besteht [BC+03b, Ri16, Ne19]. Eine iterative Migration setzt sich aus mehreren Iterationen zusammen. Zur besseren Abgrenzung von iterativen Softwareentwicklungsprozessen wie RUP werden die Iterationen als *Migrationsiterationen* bezeichnet. Innerhalb einer Migrationsiteration wird die *Systematik* des Migrationsrahmenwerks durchlaufen [JA+13]. Die Systematik beruht dabei auf *Migrationsaktivitäten*, die sequentiell ausgeführt werden [AF+18]. Eine Migrationsaktivität beschreibt eine oder mehrere durch das Migrationsteam auszuführende Aufgaben. Neben dem Migrationsteam können der Migrationsaktivität noch weitere Rollen wie beispielsweise spezifische Interessenvertreter zugeordnet werden, die bei der Ausführung unterstützen. Keine der Arbeiten hat die Systematik einer Migrationsiteration ganzheitlich betrachtet, sondern lediglich abgeschlossene Ausschnitte. Diese abgeschlossenen Ausschnitte könne jedoch als Gruppierungsmittel für Migrationsaktivitäten genutzt werden, weshalb das Element der *Migrationsphasen* [GA05, Wa14, BH+18] definiert wird. Eine Migrationsphase besteht somit aus mehreren Migrationsaktivitäten, die ein gemeinsames Zwischenziel der Systematik verfolgen. Solche Phasen sind auch bei RUP wiederzufinden [Bo06].

Eine Migrationsiteration benötigt einen bestimmten Auslöser, der dem Migrationsteam die Notwendigkeit des Startens einer Migrationsiteration signalisiert. Der Auslöser wird im Kontext der iterativen Migration als *Migrationsauslöser* bezeichnet. Geeignet als Migrationsauslöser sind Änderungen an dem monolithischen Softwaresystem, die in Form von funktionalen oder nicht-funktionalen Anforderungen an das Entwicklungsteam beziehungsweise bei einem bereits gestarteten Migrationsvorhaben dem Migrationsteam herangetragen werden. Denn die Änderungen sollten nicht mehr in

Tabelle 4.3: Entitätsbezogene Elemente einer iterativen Migration

Element	Kurzbeschreibung	Quellen
Monolith	Softwaresystem mit monolithischem Architekturstil und dem Paradigma der objektorientierten Programmierung zur Definition der Softwarebausteine der Mikroarchitektur.	[DG+17, Ne19, Ri19]
Monolithen-ausschnitt	Zusammenschluss von Softwarebausteinen des Monolithen, die innerhalb einer Migrationsiteration betrachtet werden.	[EC+16, BD+18]
Migrationsartefakt	Formale Softwareartefakte, die im Rahmen von Migrationsaktivitäten erstellt werden und gegebenenfalls nur im Kontext einer Migrationsiteration gültig sind.	[ISO-19506, Wa14]

den Monolithen übernommen werden [Fo04].

4.1.3 Entitätenbezogene Elemente

Bei der Ausführung von Migrationsaktivitäten kommen Elemente zum Tragen, welche bestimmte Entitäten der iterativen Migration bezeichnen. An dieser Stelle spricht man von den *entitätenbezogenen Elementen*. In Tabelle 4.3 werden die im Kontext der Forschungsarbeit definierten entitätenbezogenen Elemente zusammengefasst.

Ein zentrales Element ist das *monolithische Softwaresystem*–auch einfach nur *Monolith*–, das aus verschiedenen Softwarebausteinen besteht [DG+17, Ne19, Ri19]. Aufgrund der definierten Prämissen handelt es sich bei den Softwarebausteinen um Pakete oder Klassen, die in Abhängigkeit zueinander stehen. Dabei werden die Softwarebausteine zu einem Inkrement zusammengefasst, das letztendlich in eine Microservice-Architektur modernisiert werden soll. Um auch an dieser Stelle eine klare Abgrenzung zu dem Softwareentwicklungsprozess RUP zu schaffen, wird das Inkrement als *Monolithenausschnitt* bezeichnet [EC+16]. Alternativ fasst [BD+18] die Softwarebausteine als Monolithenkomponenten zusammen. Doch die Bezeichnung als Komponente ist durch die komponentenbasierte Entwicklung (engl. component-based development) anderweitig geprägt und suggeriert eine Berücksichtigung des *Separation of Concerns*.

Die Nachvollziehbarkeit der Systematik des Migrationsrahmenwerks beruht auf der Verwendung von formalen Softwareartefakten. Im Kontext der iterativen Migration werden die formalen Softwareartefakte als *Migrationsartefakte* bezeichnet [ISO-19506, Wa14]. Die Bezeichnung als Migrationsartefakt stellt die Verbindung des formalen Softwareartefakts zur iterativen Migration her und beschränkt damit implizit den Gültigkeitsbereich auf eine Migrationsiteration. Nicht alle der in der Migrationsiteration geschaffenen formalen Softwareartefakte finden nach Abschluss der Migrationsiteration Verwendung, weshalb eine solche Abgrenzung als sinnvoll erachtet wird.

4.1.4 Zusammenhang der Elemente

Um das Verständnis von Migrationen im Umfeld monolithischer Softwaresysteme und Microservices zu verbessern, werden die Elemente in der Abbildung 4.1 zusammenhängend dargestellt.

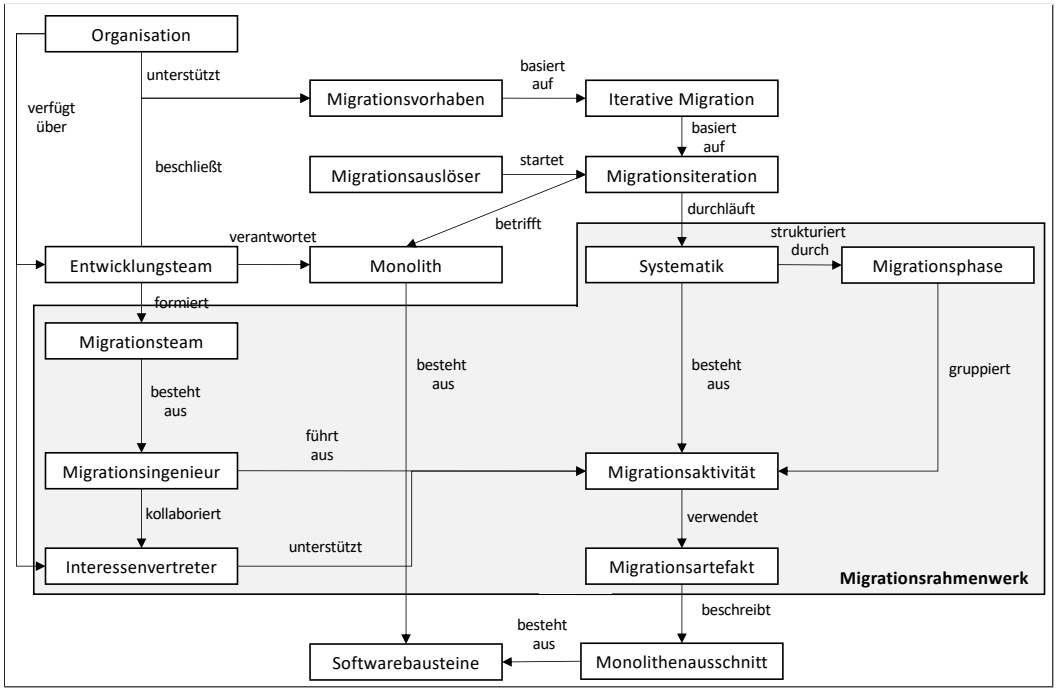

Abbildung 4.1: Zusammenhänge der Elemente einer iterativen Migration

Den Ausgangspunkt der Elemente stellt die *Organisation* dar, welche über ein *Entwicklungsteam* und verschiedene *Interessenvertreter* verfügt. Das Entwicklungsteam trägt die Verantwortung für die Wartung des *Monolithen*. Ausgehend von Ineffizienzen bei der Wartung ist das Entwicklungsteam in der Lage, ein *Migrationsvorhaben* zu beschließen. Die Organisation unterstützt dabei das Vorhaben mit finanziellen und personellen Ressourcen und schränkt die Weiterentwicklung des monolithischen Softwaresystems ein.

Das Migrationsvorhaben basiert auf der *iterativen Migration*, die anhand mehrerer *Migrationsiterationen* den Monolithen nach und nach modernisiert. Eine Migrationsiteration wird dabei durch einen *Migrationsauslöser* gestartet. Der Ablauf einer Migrationsiteration wird durch die *Systematik* definiert, welche aus *Migrationsaktivitäten*, die sequentiell angeordnet sind, besteht. Zur Strukturierung der Systematik werden *Migrationsphasen* verwendet. Ausgeführt wird die Migrationsaktivität durch *Migrationsingenieure*, die einem *Migrationsteam* angehören. Das Migrationsteam selbst wird aus dem *Entwicklungsteam* formiert. Um bei der Ausführung der Migrationsaktivitäten auf Domänenwissen zurückgreifen zu können, kollaborieren die Migrationsingenieure mit den *Interessenvertretern*.

Eine Migrationsaktivität beruht auf der Verwendung von *Migrationsartefakten*. Die Migrationsartefakte beschreiben dabei den *Monolithenausschnitt*, der innerhalb der Migrationsiteration betrachtet wird. Inhaltlich besteht der Monolithenausschnitt aus Softwarebausteinen der Mikroarchitektur des Monolithen.

Der Abbildung 4.1 können die Elemente entnommen werden, die bei dem Entwurf des Migrationsrahmenwerks von Relevanz sind. Ein zentrales Element ist dabei die Systematik mit den Migrationsaktivitäten und -artefakten. Durch die Migrationsphasen wird eine Übersichtlichkeit der Systematik im Migrationsrahmenwerk erreicht. Als Rollen werden das Migrationsteam mit den Migrationsingenieuren und auch die Interessenvertreter gesehen.

4.2 Systematik des Migrationsrahmenwerks

Der Entwurf der Systematik wird anhand dreier Migrationsphasen strukturiert, um eine bessere Übersichtlichkeit und Verständlichkeit zu gewährleisten. Zu Beginn des Entwurfs wird zuerst die Zielsetzung der Systematik definiert. Anschließend wird ein kurzer Gesamtüberblick über die Systematik gegeben. In einem abschließenden Abschnitt wird dieser Gesamtüberblick durch den Detailentwurf der Systematik verfeinert.

4.2.1 Zielsetzung

Bei der iterativen Migration eines monolithischen Softwaresystems besteht vor allem die Herausforderung eines *systematischen Vorgehens*, denn Migrationsvorhaben sind vielfältig gestaltbar und werden meist spontan beschlossen [KW+98]. Bei der Ausgestaltung des systematischen Vorgehens werden im Vorfeld Ziele definiert, welche das Migrationsteam bei der Durchführung einer Migrationsiteration unterstützen. Die nachfolgenden Ziele beziehen sich nur auf die Systematik und stehen neben der übergeordneten Zielsetzung dieser Forschungsarbeit (vgl. Abschnitt 1.5.1). Die dort vorgesehene Flexibilität des Migrationsrahmenwerks gilt ebenfalls für die entworfene Systematik.

So wird, um eine *einfache Anwendbarkeit* zu gewährleisten, auf automatisierte Migrationsaktivitäten verzichtet [CP07]. Dadurch müssen auch keine fehleranfälligen und komplexen Werkzeuge durch das Entwicklungsteam eingeführt beziehungsweise ausgeführt werden. Eine weitere Schwachstelle automatisierter Migrationsaktivitäten ist die fehlende Restriktierbarkeit auf einen bestimmten Ausschnitt, wie beispielsweise einen Monolithenausschnitt. Automatisierte Quellcodeanalyse-Werkzeuge betrachten den gesamten Monolithen. Die Systematik beruht damit nur auf manuellen Migrationsaktivitäten.

Mehrere Ziele betreffen die Migrationsiterationen. Zum einen soll eine ganzheitliche Betrachtung einer Migrationsiteration erfolgen, sodass die Systematik das Migrationsteam in jeder Situation unterstützen kann. Dies schließt die folgenden Aspekte ein: (1) *Interpretation des Migrationsauslösers*, (2) *Identifikation eines Monolithenausschnitts*, (3) *Entwurf der Microservices* und (4) *Vorbereitungen für den Betrieb der Microservices*. Für all diese Aspekte stellt die Systematik Migrationsaktivitäten bereit. Zum anderen sollen Migrationsiterationen in sich abgeschlossen sein und keine Abhängigkeiten zu anderen Migrationsiterationen haben; an dieser Stelle besteht ein Unterschied zu den Iterationen von RUP, die durch einen *Inkrementenplan* aufeinander aufbauen [Kr04]. Durch die Abgeschlossenheit ist das Migrationsteam in der Lage, bei Bedarf jederzeit die iterative Migration zu pausieren oder zu beenden. Auch kann das Migrationsteam entscheiden, ob ein Migrationsauslöser zu einem Start einer Migrationsiteration führt oder ob mehrere Migrationsiterationen parallel bearbeitet werden.

Zu jedem Zeitpunkt der Migrationsiteration soll die Systematik die Lauffähigkeit des monolithischen Softwaresystems gewährleisten. Auf diese Weise ist der Betrieb des verbleibenden monolithischen Softwaresystems sichergestellt und Benutzer sind nicht von der iterativen Migration betroffen.

Bei dem Entwurf der Microservices wird auf eine Orientierung an der Geschäftsdomäne der Organisation gesetzt, sodass die entstehenden Microservices eine hohe Autonomie und Wiederverwendbarkeit aufweisen [Ne15]. Entsprechend werden Migrationsaktivitäten vorgesehen, die ein Verständnis über die Geschäftsdomäne etablieren und dieses Domänenwissen in einen domänengetriebenen Entwurf der Microservices überführen. Vorgesehen wird jedoch auch die Bewertung des domänengetriebenen Entwurfs anhand nicht-funktionaler Anforderungen, um den effizienten Betrieb der Microservices in der Cloud sicherzustellen zu können [GB+17].

Als letztes Ziel wird die Reorganisation des Entwicklungsteams gesehen, welche für die Wartung des modernisierten Softwaresystems unabdingbar ist [HS17]. In der Systematik müssen damit Migrationsaktivitäten explizit für das Etablieren neuer Teamstrukturen vorgesehen werden. Begründet wird dies zum einen auch durch *Conway's Law*, welches besagt, dass eine Organisation ein Softwaresystem zwangsläufig nach dem Vorbild der eigenen organisatorischen Struktur und der darin gelebten Kommunikationswegen entwirft [Co68]. Angewandt auf die Systematik bedeutet dies, dass ein *Reverse Conway's Law* angewandt werden muss, sodass die Softwarearchitektur die Teamstruktur beeinflusst [Ne15].

4.2.2 Überblick

In diesem Abschnitt wird ein abstrahierter Überblick über die entworfene Systematik des Migrationsrahmenwerks gegeben. Weiterführend werden die Auswirkungen auf das monolithische Softwaresystem durch die Systematik aufgezeigt. Die Systematik kann der Abbildung 4.2 entnommen werden.

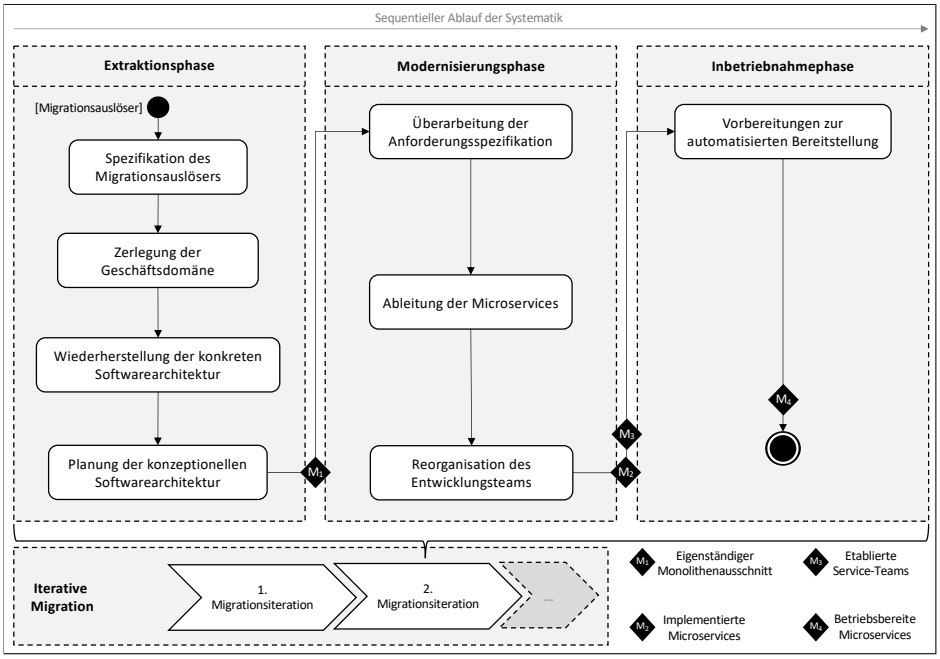

Abbildung 4.2: Generelle Migrationsaktivitäten der Systematik

Die Systematik gliedert sich in drei Migrationsphasen: (1) *Extraktionsphase*, (2) *Modernisierungsphase* und (3) *Inbetriebnahmephase*. Jede dieser Migrationsphasen verfolgt ein abgegrenztes Zwischenergebnis einer Migrationsiteration. So verfolgt die Extraktionsphase das Ziel, einen Monolithenausschnitt zu definieren und als eigenständig betriebenen Softwarebaustein bereitzustellen. Die Modernisierungsphase greift diesen eigenständigen Softwarebaustein auf und modernisiert diesen in eine Microservice-Architektur, sodass nach dieser Phase ein oder mehrere Microservices vorliegen. Zum Abschluss trifft die Inbetriebnahmephase Vorbereitungen für den Betrieb der neu geschaffenen Microservices. Die Zwischenergebnisse werden in Form von *Meilensteinen* festgehalten, die zum einen als Orientierungs- und zum anderen als Synchronisationspunkte für das Migrationsteam verwendet werden können.

Die Extraktionsphase schließt mit dem Vorhandensein eines eigenständigen Monolithenausschnittes (Meilenstein M_1) ab. Nach Abschluss der Modernisierungsphase werden zwei Zwischenergebnisse erreicht. Zum einen sind dies die implementieren Microservices (Meilenstein M_2) und zum anderen die etablierten Service-Teams (Meilenstein M_3). Die Inbetriebnahmephase schließt mit den betriebsbereiten Microservices (Meilenstein M_4) ab. Zusammenfassend werden die Meilensteine in Abbildung 4.3 dargestellt.

Der Einstieg in die Systematik findet nur über den Migrationsauslöser statt. Es werden keine Ergebnisse vorheriger Migrationsiterationen benötigt. Vereinzelt können jedoch Migrationsartefakte vorangegangener Migrationsiterationen wiederverwendet werden. Wie jedoch durch die Zielsetzung

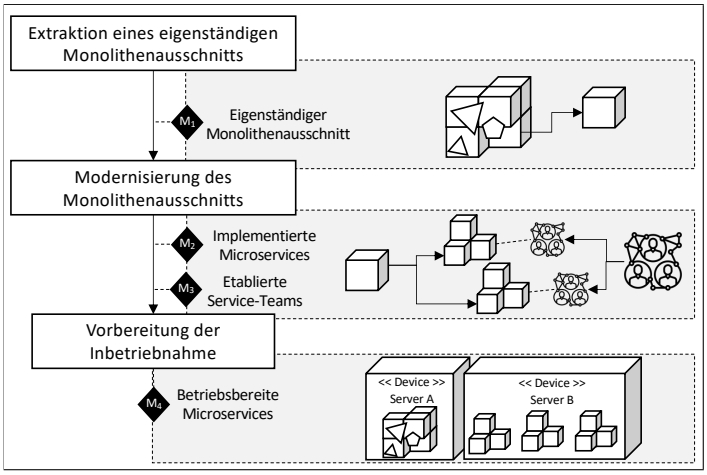

Abbildung 4.3: Als Meilensteine gekennzeichnete Zwischenergebnisse der Systematik

(vgl. Abschnitt 4.2.1) definiert, wird eine *Abgeschlossenheit der Migrationsiterationen* verfolgt. Folglich können zwar Migrationsartefakte wiederverwendet werden, doch das Erstellen aller relevanten Migrationsartefakte ist immer durch die Systematik gegeben. Im Gegensatz dazu sind die Migrationsphasen von Zwischenergebnisse vorangegangener Migrationsphasen abhängig. Somit müssen alle Migrationsphasen innerhalb einer Migrationsiteration durchlaufen werden.

Eindeutig der Abbildung 4.2 zu entnehmen ist die ungleich gewichtete Verteilung der Migrationsaktivitäten auf die einzelnen Migrationsphasen. Ein klarer Fokus liegt dabei auf der Extraktionsphase, welche die zentrale Herausforderung zur Bestimmung des Monolithenausschnitts darstellt. Die Migrationsaktivitäten werden sequentiell durchlaufen. Aus Gründen der Verständlichkeit wurde auf die Darstellung der mit den Migrationsaktivitäten verbundenen Migrationsartefakten verzichtet; diese werden in einem eigenen Abschnitt behandelt (vgl. Abschnitt 4.3).

Es existieren Meinungen darüber, dass eine Systematik im Rahmen einer Migrationsiteration das Migratonsteam einschränkt, da der Kontext des Migrationsvorhabens sich von Organisation zu Organisation zu stark unterscheidet [BH+18, HR20]. Doch im Vordergrund steht hier die Unterstützung des Migrationsteams, die ausgehend von dem eigenen Kontext die Systematik auf die Bedürfnisse anpassen kann. Der identifizierte Handlungsbedarf aus Abschnitt 3.3 zeigt, dass Migrationsansätze aus dem Bereich des *Situational Method Engineerings* zwar eine hohe Anpassbarkeit aufweisen, aber keine ganzheitliche Unterstützung für das Migrationsteam bei der Durchführung einer Migrationsiteration bieten.

Nicht abgebildet durch die Systematik wird die Implementierung der jeweiligen Zwischenergebnisse der Migrationsphasen. Das Vorsehen der Implementierung würde die Flexibilität des Migrationsteams maßgeblich bei der Wahl von Programmiersprachen oder sonstigen Programmierparadigmen einschränken. Zudem wurde der Ausschluss der Implementierung als Prämisse dieser Forschungsarbeit

festgelegt (vgl. Abschnitt 1.6.5). Durch die definierten Zwischenergebnisse werden dem Migrationsteam die Voraussetzungen zum Starten der nächsten Migrationsphase vorgegeben, sodass die Ergebnis der Implementierungsaktivitäten ausreichend definiert sind.

4.2.3 Entwurf der Migrationsphasen

Für die Ausgestaltung der Migrationsphasen wird der aktuelle Stand der Forschung aus Abschnitt 3.2 zugrunde gelegt. Verwendung finden vor allem die Arbeiten von Knoche und Hasselbring [KH18], Belalaie et al. [BH+16, BH+18], Henry und Ridene [HR20], Newman [Ne19] und Gysel et al. [GK+16].

Die vorgelagerte Extraktion eines Monolithenausschnitts, wie von Knoche und Hasselbring [KH18] vorgestellt, hat als Inspiration zum Entwurf der Extraktionsphase gedient. Auch kommen in der Extraktionsphase vereinzelte *Migrationsmuster* der Arbeiten von Balalaie et al. [BH+18], Henry und Ridene [HR20] und Newman [Ne19] zum Tragen. Relevant sind gerade die Muster, welche eine Betrachtung des Ist-Zustandes des Monolithen vorsehen. Insbesondere wird durch Newman die Datenhaltung detailliert betrachtet.

Als Inspiration für die Modernisierungsphase wird die Arbeit von Gysel et al. [GK+16] gesehen. Die vorgestellten *Kopplungskriterien*, die primär als nicht-funktionale Anforderungen zu interpretieren sind, zeigen die hohe Relevanz von nicht-funktionalen Anforderungen für den Entwurf der Microservice-Architektur. Daher wird im Rahmen der Modernisierungsphase auf eine Betrachtung nicht-funktionaler Anforderungen zurückgegriffen. Weiterhin ist die Betrachtung des monolithischen Entwicklungsteams, welche von Balalaie et al. [BH+16] durchgeführt wird, bei der Einführung einer Microservice-Architektur von Relevanz.

Die Inbetriebnahmephase orientiert sich an den Arbeiten von Balalaie et al. [BH+16, BH+18] und Henry und Ridene [HR20], die ein hohes Augenmerk auf den Betrieb der Microservices legen. Diese Arbeiten führen vereinzelte *Migrationsmuster* auf, welche die Microservices auf den eigentlichen Betrieb vorbereiten.

Extraktionsphase

Im Zentrum der *Extraktionsphase* steht die Identifikation und Bereitstellung eines in sich abgeschlossenen Monolithenausschnitts. Wichtig ist, dass im Kontext dieser Forschungsarbeit die eigentliche Extraktion mit der Implementierung und Bereitstellung des Monolithenausschnitts nicht betrachtet wird. Fokussiert wird stattdessen die Identifikation des Monolithenausschnitts. Zum Bestimmen eines Monolithenausschnitts werden verhaltensspezifische Aspekte des Softwaresystems in den Vordergrund gestellt. Unterstützend erfolgt die Betrachtung der *grobgranularen Geschäftsdomänenstruktur*, in der

das Softwaresystem platziert ist. Aufbauend auf den erhaltenen Informationen kann weiteres Verhalten spezifiziert werden, welches dem Monolithenausschnitt zugeordnet wird. Anschließend erfolgt die Wiederherstellung des Ist-Zustandes in Form einer Architekturbeschreibung. Der Ist-Zustand betrifft dabei nur das spezifizierte Verhalten und nicht den gesamten Monolithen. Zum Abschluss wird die Soll-Architektur entworfen, welche den Monolithenausschnitt als eigenständigen Softwarebaustein beschreibt.

Die Motivation zur Einführung der Extraktionsphase in die Systematik des Migrationsrahmenwerks beruht auf der eigenständigen Ausführbarkeit des Monolithenausschnitts. Aus der Eigenständigkeit folgt ein eigenständiger Betrieb des Monolithenausschnitts, der eine vom Monolithen unabhängige Skalierung ermöglicht. So ist das Migrationsteam beziehungsweise Entwicklungsteam in der Lage, in einer frühen Phase einer Migrationsiteration bereits Lastverteilungsprobleme für die ausgewählten verhaltensspezifischen Aspekte des Monolithenausschnitts aufzulösen. Weiterhin kann das Migrationsteam durch den sehr eingeschränkten Rahmen erste Erfahrungen im Bereich verteilter Softwaresysteme sammeln, welche bei der Betrachtung von Microservice-basierten Softwaresystemen von Nutzen sein können. Die Extraktionsphase greift die Problemstellungen $P2$ und $P3$ auf und ist als Teil des Beitrags $B2$ zu sehen (vgl. Abschnitte 1.4.2 und 1.4.3 und 1.5.3). Der eigentliche Forschungsbeitrag und damit auch die Auflösung der Problemstellungen werden in Kapitel 5 behandelt.

Das Vorsehen einer explizit vorgelagerten Extraktion eines Monolithenausschnitts beruht auf der Arbeit von Knoche und Hasselbring [KH18] (vgl. Abschnitt 3.2.6). Abzugrenzen ist die in dieser Forschungsarbeit entworfene Extraktionsphase von der Arbeit von Knoche und Hasselbring durch die ganzheitliche Betrachtung, die explizite Berücksichtigung des Migrationsauslösers und die Wiederherstellung der konkreten Softwarearchitektur, die dem Ist-Zustand entspricht. Die Autoren nehmen eine nur eingeschränkte Sicht auf die Durchführung einer Migrationsiteration ein, sodass wichtige Aspekte wie die Betrachtung des Migrationsauslösers oder auch die Wiederherstellung der Architekturbeschreibung vernachlässigt werden. Weiterführend nehmen die Autoren an, dass das monolithische Softwaresystem modularisiert wurde, sodass das Finden von geeigneten Ausschnitten eine triviale Herausforderung ist. Für die Extraktionsphase wird jedoch angenommen, dass das monolithische Softwaresystem nur *unzureichend modularisiert* wurde und deshalb eine Identifikation kohäsiv zum Verhalten stehender Softwarebausteine notwendig ist. Auch wird das Verhalten als Ausgangspunkt der Extraktionsphase gesehen, was in der Arbeit von Knoche und Hasselbring nur wenig Beachtung findet.

Der Ablauf der Extraktionsphase mit seinen Migrationsaktivitäten und Migrationsartefakten wird in Abbildung 4.4 aufgeführt. In den folgenden Abschnitten wird auf die Migrationsaktivitäten detailliert eingegangen.

Als Ausgangspunkt der Extraktionsphase wird das Verstehen des Migrationsauslösers gesehen, der in Form von neuer Funktionalität oder Änderungen am Softwaresystem an das Entwicklungsteam

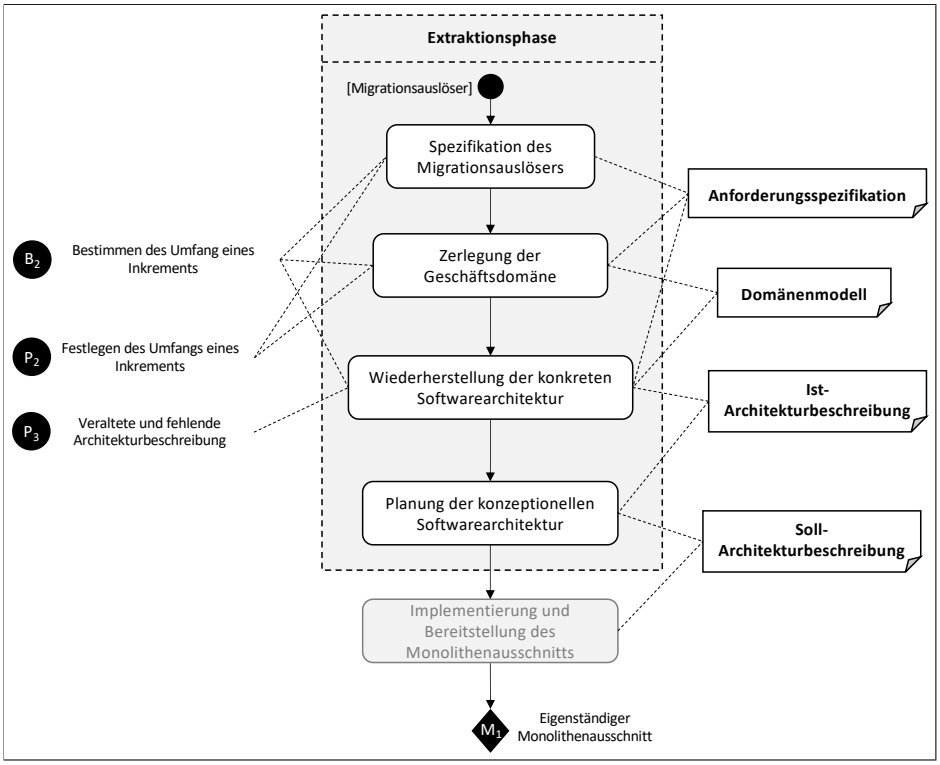

Abbildung 4.4: Entwurf der Extraktionsphase

herangetragen wird. Entscheidet sich das Entwicklungsteam ausgehend von dem Migrationsauslöser, eine Migrationsiteration zu starten, gilt es, den Migrationsauslöser zu spezifizieren. Dass der Migrationsauslöser ein geeigneter Einstieg in eine Migrationsiteration ist, zeigt das von Fowler [Fo04] vorgestellte *StranglerFigApplication*-Muster, das besagt, dass keine neue Funktionalität im Monolithen vorgesehen werden sollte. Bei dieser Spezifikation richtet sich die Extraktionsphase nach den Prinzipien von *BDD* [Sm14]. Folglich wird der Migrationsauslöser formal, aber dennoch für Interessenvertreter verständlich spezifiziert und einer *Anforderungsspezifikation* zugeordnet. Die Anforderungsspezifikation ist als Migrationsartefakt anzusehen, welches im Kontext der Migrationsiteration gültig ist. Die Formalisierung des Migrationsauslösers ist, neben der Verständlichkeit der Spezifikation [RN+13], auch für die Testbarkeit desselben notwendig. Dies gilt jedoch für die gesamte Anforderungsspezifikation, die in der Extraktionsphase erstellt wird, damit nach der Extraktion die Funktionsweise des Monolithen sichergestellt werden kann. Ausgenommen von den Auslösern sind jedoch *kritische Fehlerbehebungen*, denn die Behebung dieser Fehler muss meist in kürzester Zeit erfolgen. Eine Behebung nach einer erfolgten Migrationsiteration kann zu einem wirtschaftlichen Schaden für die Organisation führen, was die Akzeptanz des Migrationsvorhabens bei Schlüsselpositionen reduziert und gefährdet.

Im Anschluss an die Spezifikation des Migrationsauslösers gilt es, die *Geschäftsdomäne* zu betrachten.

Die Geschäftsdomäne erlaubt dem Migrationsteam, Aussagen über die Kohäsion von Verhalten zu treffen, denn zusammengehöriges Verhalten ist im selben *Problemraum* angesiedelt. Dieses Verständnis über Geschäftsdomänen im Zusammenhang mit Softwaresystemen basiert auf *DDD* [Ev04, Ve13]. Um den Problemraum bestimmen zu können, muss das Migrationsteam die Geschäftsdomäne auf einem hohen Abstraktionsniveau betrachten; man spricht auch von einer *grobgranularen Struktur* der Geschäftsdomäne. Im Kontext von DDD ist der Problemraum durch die sogenannten *Subdomänen* (vgl. Abschnitt 2.4.2) beschrieben. Die Subdomänen werden in einem *Domänenmodell* festgehalten. Auch hier ist die Formalisierung für das Verständnis des Entwicklungsteams von hoher Relevanz.

Bei der Wiederherstellung der *konkreten Softwarearchitektur* wird eine formale *Ist-Architekturbeschreibung* erstellt, welche die Softwarebausteine des Monolithen und die Abhängigkeiten zwischen diesen umfasst. Weiterführend gilt es, die *Datenhaltungsschicht* zu betrachten, um die Persistierung der Daten, die von den Softwarebausteinen ausgeht, berücksichtigen zu können. Die geforderte Ist-Architekturbeschreibung wird durch die Praktik des *Design Recoverys* [CC90] erreicht. Um dem iterativen Gedanken und der Zielsetzung der Systematik gerecht zu werden, zielt das Design Recovery auf einen in sich abgeschlossenen Monolithenausschnitt ab. Genauer bedeutet dies, dass nur die für den Monolithenausschnitt relevanten Softwarebausteine des Monolithen erhoben werden. Als Grundlage für das Design Recovery werden die zuvor erstellte Anforderungsspezifikation und das Domänenmodell gesehen. Als Informationsquellen für das Design Recovery werden das monolithische Softwaresystem und die Softwareentwickler herangezogen [RH06]. Die fertiggestellte Ist-Architekturbeschreibung entspricht dem definierten Monolithenausschnitt, der aus mehreren Softwarebausteinen bestehen kann. Wichtig ist, dass die Softwarebausteine des Monolithenausschnitts in die Hoheit des Migrationsteams übergehen und keine Änderungen durch andere Softwareentwickler außerhalb des Migrationsteams durchgeführt werden. Bei Fehlerbehebungen ist das Migrationsteam verantwortlich für die Implementierung und die nachgelagerte Überarbeitung der Ist-Architekturbeschreibung.

Aus der konkreten Softwarearchitektur kann das Migrationsteam anschließend die *konzeptionelle Softwarearchitektur* des Monolithenausschnitts ableiten. Festgehalten wird diese als formale *Soll-Architekturbeschreibung*. Der Soll-Zustand beschreibt den Monolithenausschnitt als eigenständigen Softwarebaustein neben dem Monolithen. Für den Monolithenausschnitt werden nur minimale syntaktische Anpassungen vorgesehen. Die Semantik beziehungsweise das Verhalten, welches durch den Monolithenausschnitt bereitgestellt wird, wird nicht überarbeitet. Damit ist dieselbe Funktionsweise des Softwaresystems auch nach der Extraktion sichergestellt. Die syntaktischen Anpassungen betreffen insbesondere die *lokalen Schnittstellen* (engl. Application Programming Interface, API) und die *Datenhaltungsschicht*. Eine lokale API entspricht dabei einem netzwerkunabhängigen Aufruf eines Softwarebausteins in einem anderen Softwarebaustein [LX+13]. Um die Extraktion und gleichzeitig die Möglichkeit des Verteilens auf der physischen Architektur zu ermöglichen, werden die lokalen APIs in *Web-APIs* überführt. Bei der Datenhaltung werden die Daten, welche durch

den Monolithenausschnitt betroffen sind, in eine eigene Datenhaltung überführt. Dies reduziert die Kopplung zu dem Monolithen [Ne15]. Eine vollständige Autonomie des Monolithenausschnitts muss jedoch noch nicht gegeben sein. Dadurch ergibt sich jedoch gleichzeitig die Notwendigkeit, eine *Datensynchronisation* zwischen dem Monolithen und dem Monolithenausschnitt vorzunehmen. Die Betrachtung der Fragestellung, wie ein paralleler Betrieb der beiden Softwarebausteine zu bewältigen ist, erfolgt in Abschnitt 4.

Modernisierungsphase

Die *Modernisierungsphase* setzt an bestehenden Monolithenausschnitt an und unterzieht diesen einem architekturellen Refactoring. An dieser Stelle der Migrationsiteration findet die Transition des Architekturstils auf die Microservice-Architektur statt. Während des Refactorings werden die Anforderungen, welche in der Anforderungsspezifikation der Extraktionsphase festgelegt wurden, an sich geänderte Bedürfnisse angepasst. Zur Implementierung und Wartung der Microservices werden weiterhin Service-Teams aus dem monolithischen Entwicklungsteam abgeleitet. Wie auch innerhalb der Extraktionsphase wird die Implementierung der Microservices nicht betrachtet.

Die Modernisierungsphase kapselt das architekturelle Refactoring des Monolithenausschnitts. Bei der Modernisierung wird der Ausschnitt in ein oder mehrere domänengetriebene Microservices unterteilt. Unter einem domänengetriebenen Microservice wird ein Microservice verstanden, der anhand von *Domänenwissen* [Ev04] die Funktionalität innerhalb des Monolithenausschnitts kohäsiv zusammenfasst. Folglich hängt die Anzahl der Microservices, die in der Migrationsiteration entworfen werden, von dem Umfang des Monolithenausschnitts und der Geschäftsdomäne ab. Aus der Orientierung an der Geschäftsdomäne folgen hochgradig autonome Microservices [Ne15]. Die Modernisierungsphase greift dabei zwei Problemstellungen der iterativen Migration auf. So werden die Problemstellungen $P4$ und $P5$ aufgelöst (vgl. Abschnitte 1.4.4 und 1.4.5). Weiterhin sind die Forschungsbeiträge $B3$ und $B4$ als Teil der Modernisierungsphase zu sehen (vgl. Abschnitte 1.5.4 und 1.5.5). Die konkrete Auflösung der Problemstellungen und Ausgestaltung der Beiträge ist dem Kapitel 6 zu entnehmen.

Einen wichtigen Beitrag zur Ausgestaltung der Modernisierungsphase wurde von Gysel et al. [GK+16] getätigt. Die Autoren leiten automatisiert und basierend auf einem *Kopplungskriterienkatalog* Microservices ab (vgl. Abschnitt 3.2.5). Ein Kopplungskriterium entspricht dabei einer nicht-funktionalen Anforderung. Gysel et al. sehen eine hohe Relevanz von nicht-funktionalen Anforderungen für einen effizienten Betrieb der Microservices. Die Modernisierungsphase greift diese Relevanz der nicht-funktionalen Anforderungen auf und nutzt diese zur Überarbeitung des domänengetriebenen Entwurfs der Microservice-Architektur. Es wird folglich davon ausgegangen, dass ein domänengetriebener Entwurf nicht zwangsläufig einen effizienten Betrieb ermöglicht. Im Gegensatz zu der bestehenden Arbeit von Gysel et al. sieht die Modernisierungsphase zunächst einen domänengetriebenen Entwurf

vor. Erst dann werden die nicht-funktionalen Anforderungen in den Entwurf integriert. So steht weiterhin die Orientierung an der Geschäftsdomäne im Vordergrund.

Die Abbildung 4.5 zeigt zusammenfassend die Migrationsaktivitäten und Migrationsartefakte der Modernisierungsphase auf. Nachfolgend werden die einzelnen Migrationsaktivitäten detailliert beschrieben. Von der Betrachtung ausgenommen wird, wie bereits angesprochen, die eigentliche Implementierung der Microservices.

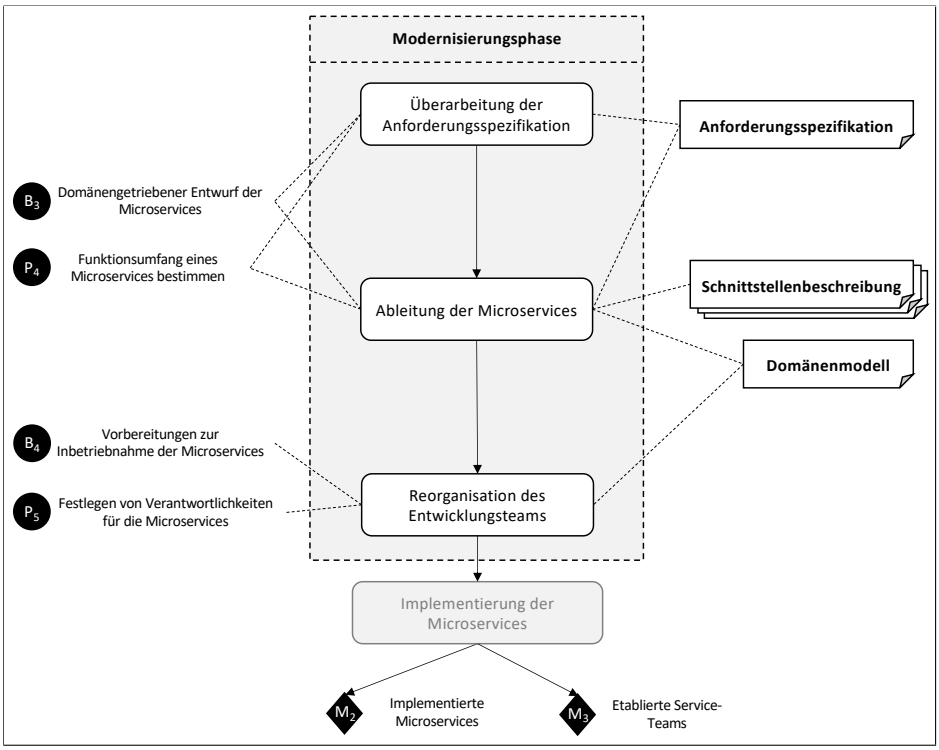

Abbildung 4.5: Entwurf der Modernisierungsphase

Eröffnet wird die Modernisierungsphase mittels einer Überarbeitung der *Anforderungsspezifikation*. Die Überarbeitung der dort spezifizierten Anforderungen ist aufgrund der natürlichen Evolution eines Softwaresystems notwendig. Neben geänderten Anforderungen werden auch diejenigen angepasst, die während der Softwareentwicklung falsch interpretiert wurden, was häufig zu Beginn einer Softwareentwicklung auftritt [WL01]. Ebenfalls wie in der Extraktionsphase orientiert sich die Modernisierungsphase im Kontext der Anforderungsspezifikation an *BDD*. Dadurch, dass diese Anforderungsspezifikation für den gesamten Lebenszyklus der Microservices gültig ist, wird eine hohe Qualität gefordert. Dies bedeutet neben der Formalisierung, dass zur Beschreibung der Anforderungen auf das Prinzip der *Specification by Example* [Ad11] zurückgegriffen wird. Durch die beschreibenden Beispiele wird das Verständnis der Anforderungen sowohl bei den Softwareentwicklern als auch Interessenvertretern verbessert. Ein weiteres Prinzip, das BDD bei der Ausgestaltung der Anforderungsspezifikation vorgibt, ist die automatisierte Testbarkeit der Anforderungen während der

Implementierung der Microservices. Auf diese Weise wird bereits bei der Implementierung auf die Korrektheit der Anforderungen geachtet.

Bei dem Entwurf der Microservice-Architektur gilt es, ausgehend von der Geschäftsdomäne, die Microservices abzuleiten. So ist sichergestellt, dass von domänengetriebenen Microservices gesprochen werden kann und die geforderten Qualitätsmerkmale wie eine hohe Wiederverwendbarkeit sichergestellt sind. Die verwendete Methodik dieser Migrationsaktivität beruht auf den Vorarbeiten von Hippchen et al. [HS+19], die ebenfalls ausgehend von einer durch BDD spezifizierten Anforderungsspezifikation einen domänengetriebenen Entwurf ableiten. Den Aspekt der Geschäftsdomäne decken die Autoren durch die Konzepte und Prinzipien von *DDD* [Ev04] ab. Die Verwendung von DDD nimmt dabei eine zentrale Rolle ein, die durch die weite Verbreitung von DDD im Kontext der Microservices [FL14, Ne15, NM+16, BP19] motiviert wird. Zum Ableiten der Microservices wird die *feingranulare Geschäftsdomäne*, strukturiert durch *Domänenobjekte*, aus den Anforderungen extrahiert und formal in einem *Domänenmodell* festgehalten; die feingranulare Sicht auf die Geschäftsdomäne ist als Verfeinerung zu der grobgranularen Betrachtung in der Extraktionsphase zu sehen. Die Domänenobjekte lassen sich in *kohäsiven Gruppen* zusammenfassen. Eine Gruppe entspricht dabei einem Microservice-Kandidaten. Das Resultat ist der domänengetriebene Entwurf der Microservice-Architektur, der auf den Anforderungen und Softwarebausteinen des Monolithenausschnitts aufbaut. Nachgelagert müssen für den domänengetriebenen Entwurf der Microservice-Architektur noch nicht-funktionale Anforderungen betrachtet werden, damit ein effizienter Betrieb sichergestellt wird [GK+16]. Ist der Entwurf der Microservice-Architektur festgelegt, kann abschließend ein Entwurf der Web-APIs und Datenhaltung erfolgen.

Als letzte betrachtete Migrationsaktivität in dieser Forschungsarbeit wird die Struktur des monolithischgetriebenen Entwicklungsteams an den neuen Architekturstil angepasst. Etabliert werden sogenannte *Service-Teams* [Ne15]. Für jeden Microservice wird ein eigenes Service-Team vorgesehen. Bei den Service-Teams muss darauf geachtet werden, dass verschiedene Kompetenzen vorhanden sind, sodass der gesamte Lebenszyklus des Microservices Team-intern betreut werden kann [BW+15]. Besonders zu behandeln ist die Etablierung von Service-Teams bei kleinen monolithischen Entwicklungsteams. Softwareentwickler sind so auf die Service-Teams zu verteilen, dass keine *Interessenkonflikte* entstehen. Ein Interessenskonflikt besteht beispielsweise dann, wenn Entscheidungen über Erweiterungen oder Anpassungen von Funktionalitäten mehrerer von dem Softwareentwickler betreuten Microservices getroffen werden müssen. Ein Softwareentwickler ist verleitet, die einfachere Entscheidung zu treffen, obwohl diese gegebenenfalls nicht den Prinzipien der Microservice-Architektur entspricht [BP19]. Sind die Service-Teams aufgestellt, kann die Implementierung der Microservices erfolgen. Mit Abschluss der Implementierung sind beide Meilensteine der Modernisierungsphase erreicht.

Inbetriebnahmephase

In der Inbetriebnahmephase werden die Grundsteine für das Verteilen (engl. deployment) und Betreiben (engl. operation) der neuen Microservices gelegt. Im Vordergrund dieser Phase steht die Erstellung von *Konfigurationsdateien* für Werkzeuge, die zur Inbetriebnahme der Microservices notwendig sind. Das Etablieren der Werkzeuge, sodass diese lauffähig dem Migrationsteam zur Verfügung stehen, wird nicht im Kontext dieser Forschungsarbeit betrachtet.

Motiviert wird die Inbetriebnahmephase durch die Problemstellung P6 und den Forschungsbeitrag B4 (vgl. Abschnitte 1.4.6 und 1.5.5). Die Problemstellung P6 der iterativen Migration beschreibt die höhere Komplexität des Betriebs eines Microservice-basierten Softwaresystems, während der Forschungsbeitrag B4 diese neue Komplexität durch explizites Vorsehen von *Cloud-nativen Prinzipien* eines Softwaresystems in einer Migrationsiteration aufzulösen versucht. Zur Auflösung der Problemstellung wird der eigene Forschungsbeitrag in Form der Inbetriebnahmephase in die Systematik des Migrationsrahmenwerks aufgenommen.

Bestehende Arbeiten wie [BH+16, BH+18, HR20] greifen ebenfalls den Betrieb der neuen Softwarebausteine während der Migration auf. Durch den Effizienzgewinn sehen Balalaie et al. [BH+16] die *Development & Operations-Prinzipien* (DevOps) als wichtige Treiber für eine Migration. Aus diesem Grund stellen die Autoren [BH+18] auch explizit mehrere *Migrationsmuster* zur Etablierung der DevOps-Prinzipien vor. Derselbe Ansatz ist auch in der Arbeit von Henry und Ridene [HR20] zu finden, die sogar das Etablieren der DevOps-Prinzipien an den Anfang einer Migration stellen. Die Inbetriebnahmephase dieser Forschungsarbeit ist jedoch am Ende der Migrationsiteration angesiedelt.

Die Abbildung 4.6 stellt zusammenfassend die Migrationsaktivitäten der Inbetriebnahmephase dar. In dieser Forschungsarbeit liegt im Vergleich zu den anderen Migrationsphasen nur ein geringer Fokus auf der Inbetriebnahmephase, was durch die Anzahl der Migrationsaktivitäten auch reflektiert wird.

Ein zentraler Aspekt Cloud-nativer Softwaresysteme, so Laszewski et al. [LA+18], wird in der *Automatisierung* gesehen. Im Kontext der Microservice-Architekturen haben sich für die Automatisierung die DevOps-Prinzipien durchgesetzt [BW+15]. Die Realisierung der Prinzipien basiert auf der Verwendung von Werkzeugen [Vi15]. Damit in der Migrationsiteration der Einsatz dieser Werkzeuge berücksichtigt wird, sieht die Inbetriebnahmephase die Erstellung von den durch die Werkzeuge geforderten Konfigurationsdateien vor. Die zu erstellenden Konfigurationsdateien hängen jedoch von der Wahl der Werkzeuge ab.

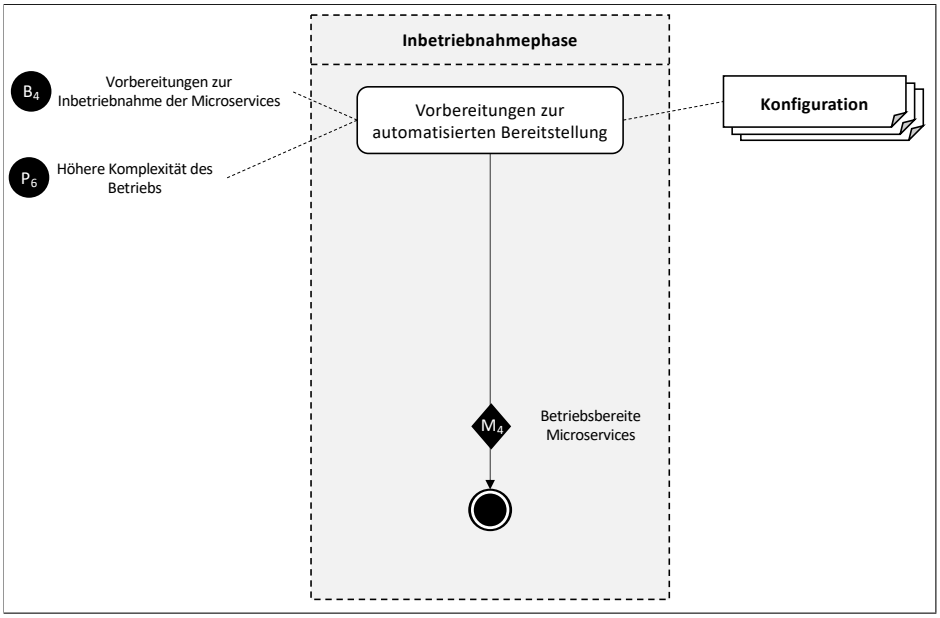

Abbildung 4.6: Entwurf der Inbetriebnahmephase

4.3 Migrationsartefakte des Migrationsrahmenwerks

In den Migrationsphasen werden verschiedene formale Migrationsartefakte erstellt, die als Eingabe und Ausgabe der Prozessschritte dienen. Mit den Migrationsartefakten wird eine funktionale und informative Sicht auf die Migrationsiteration eingenommen, was gerade dazu führt, dass innerhalb des Prozesses auf das Verständnis und die Kommunikation mit Interessenvertretern gesetzt wird [CK+92]. In den nachfolgenden Abschnitten wird zunächst auf die allgemeine Bedeutung der Migrationsartefakte für die Systematik eingegangen. Anschließend werden für jede Migrationsphase die darin auftretenden Migrationsartefakte im Detail betrachtet. Für jedes Migrationsartefakt werden die folgenden Fragen beantwortet: (1) *Was ist der Nutzen des Migrationsartefakts?* (2) *Welchen Inhalt umfasst das Migrationsartefakt?* (3) *Wie kann der Inhalt des Migrationsartefakts nachvollziehbar modelliert werden?* (4) *In welchen Migrationsaktivitäten tritt das Migrationsartefakt auf?*

Konkrete Migrationsartefakte, die durch formale Modellierungssprachen beschrieben sind, werden im Kontext dieses Kapitels nicht erörtert. Erst im Rahmen der weiteren Forschungsbeiträge und der Etablierung konkreter Softwareentwicklungsansätze erfolgt die Betrachtung der konkreten Modellierungsartefakte. So entspricht der Entwurf der Migrationsartefakte der allgemeinen Zielsetzung des generellen Migrationsrahmenwerks und gibt dem Migrationsteam einen Gestaltungsfreiraum für die Erstellung der Migrationsartefakte. Damit sind die in diesem Kapitel vorgestellten Migrationsartefakte als *Container* anzusehen, die in einer konkret ausgeprägten Systematik durch *konkrete Migrationsartefakte* befüllt werden. Die konkreten Migrationsartefakte erfüllen dabei immer den übergeordneten Zweck des Migrationsartefakts.

4.3.1 Methodik

Die *Migrationsartefakte*, nachfolgend auch kurz als Artefakte bezeichnet, stellen die Ergebnisse und Informationsquellen einzelner Migrationsaktivitäten dar und sind die Grundlage für die Nachvollziehbarkeit eines systematischen Vorgehens [Sc12]. So entsteht aus der expliziten Verknüpfung von Migrationsartefakt und Migrationsaktivität eine Leitlinie für das Migrationsteam. Für das Migrationsteam gilt es jedoch, die Erstellung der Migrationsartefakten vorzusehen, damit neben der Nachvollziehbarkeit auch die Chance einer weiteren Architekturerosion während der Implementierung des Monolithenausschnitts und der Microservices verringert wird. Die Migrationsartefakte schränken den Implementierungsspielraum der Softwareentwickler positiv ein [KR+11].

Ein Migrationsartefakt entspricht den Absichten der Softwareartefakte von Softwareentwicklungsprozessen und Softwarewartung [CK+92, SA+05]: Transportieren von Wissen zur Entwicklung oder Weiterentwicklung des Softwaresystems. Doch hinsichtlich der Gültigkeit und des Anwendungsbereiches unterscheiden sich die Migrationsartefakte von den genannten Softwareartefakten. Um diese Abgrenzung explizit darzustellen, werden die Softwareartefakte im Kontext der iterativen Migration als Migrationsartefakte bezeichnet. Eine alternative Bezeichnung wäre auch *Modernisierungsartefakt* [BB+13].

Ein Migrationsartefakt bildet entweder *strukturelle*, *verhaltens-* oder *laufzeitspezifische Aspekte* [Ga95] des bestehenden oder zukünftigen Softwaresystems ab. Um die Aspekte für die jeweiligen Zielgruppen der Interessenvertreter, zu denen auch das Migrationsteam selbst gehört, anzupassen, müssen die Migrationsartefakte ebenfalls das Prinzip der *architekturellen Blickwinkel* berücksichtigen (vgl. Abschnitt 2.2.2). Die verschiedenen Blickwinkel stellen die Grundlage für die Kommunikation mit den unterschiedlichen Interessenvertretern dar. Für die verhaltens- und laufzeitspezifischen Migrationsartefakte lassen sich die architekturellen Blickwinkel nur eingeschränkt anwenden. So können für die Konfigurationen der Inbetriebnahmephase keine unterschiedlichen Blickwinkel eingeführt werden.

Innerhalb einer Migrationsiteration können Migrationsartefakte auch über die Migrationsphasen hinweg verwendet werden. Bestimmte Artefakte wie beispielsweise das *Domänenmodell* werden auch iterationsübergreifend weiterentwickelt. Dieses Modell wird in jeder Migrationsiteration mit neuen Informationen über die grobgranulare Domänenstruktur angereichert, sodass bei Abschluss einer Migrationsiteration neues relevantes Domänenwissen modelliert ist. Kritisch ist dabei die Synchronisation des Migrationsartefakts, sobald mehrere Migrationsteams daran arbeiten. Lösung hierfür können Werkzeuge sein, welche das kollaborative Arbeiten an den Artefakten fördern [DF+07].

Bei der Erstellung der Migrationsartefakte kommt es auf die Effizienz und Effektivität an [CG+03, CI+03], weshalb nur ein minimaler Aufwand betrieben wird. Informationen, die nicht für die Mi-

grationsiteration relevant sind oder keinen Nutzen für Interessenvertreter haben, werden nicht in die Migrationsartefakte aufgenommen.

Das Verwenden von Artefakten bei der Migration von Softwaresystemen ist bereits durch die *modellgetriebene Migration* bekannt [RH06]. Eine Definition und Klassifizierung dieser Artefakte liefert dabei das von der *Object Management Group* (OMG) [ISO-19506] aufgestellte *Knowledge Discovery Meta-Model* (KDM). Die KDM liefert eine Vielzahl von Artefakten, die unterschiedliche Aspekte des bestehenden Softwaresystems betrachten [PG+11]. All diese Artefakte werden als Grundlage für das architekturelle Refactoring angesehen. Die Migrationsartefakte dieser Forschungsarbeit lassen sich zwar den Schichten und Paketen der KDM zuordnen, was den Nutzen der einzelnen Migrationsartefakte bestimmt, doch durch die komplexeren Metamodelle der KDM ist eine zu hohe Einschränkung bei der Modellierung des Migrationsteams gegeben. Für die Migrationsartefakte folgt daher nur eine Einordnung in die KDM-Schichten zur Festlegung der Semantik während die Syntax beziehungsweise Modellierungssprache von dem Migrationsteam frei wählbar bleibt. Empfohlen wird jedoch die Verwendung einer formalen Modellierungssprache wie die der *Unified Modeling Language* (UML), wobei auch generische Modellierungssprachen wie *NoUML* möglich sind [ML+12]. Formale Migrationsartefakte werden als Erfolgsfaktor für die iterative Migration gesehen [Sn06]. Durch ein konsistentes Erscheinungsbild wird zudem das Verständnis des Migrationsteams verbessert [RN+13].

4.3.2 Migrationsartefakte der Extraktionsphase

Die Migrationsartefakte der Extraktionsphase beziehen sich auf den Ist-Zustand des monolithischen Softwaresystems und den Soll-Zustand des Monolithenausschnitts. Abgebildet werden die beiden Zustände durch vier unterschiedliche Migrationsartefakte (Tabelle 4.4).

Das erste Migrationsartefakt ist die *Anforderungsspezifikation*. Sie umfasst funktionale und nichtfunktionale Anforderungen, die für das Bestimmen und Validieren des Monolithenausschnitts notwendig sind [Po10]. Die Anforderungsspezifikation wird noch einmal als Ausgangspunkt für die Anforderungen der Microservices in der Modernisierungsphase benötigt. Für die Erhebung der Anforderungen ist es wichtig, dass der Ist-Zustand abgebildet wird, damit am Ende der Extraktionsphase eine Verifizierung erfolgen kann und sichergestellt ist, dass durch den extrahierten Monolithenausschnitt auch weiterhin die Funktionalität des Softwaresystem bereitgestellt werden kann. Selbst wenn der Ist-Zustand nicht mehr den Bedürfnissen der Interessenvertreter entspricht, ist die Lauffähigkeit des Softwaresystems höher priorisiert. Am Ende der Extraktionsphase verliert die Anforderungsspezifikation durch die Überarbeitung in der Modernisierungsphase ihre Gültigkeit.

Ein weiteres Migrationsartefakt ist das *Domänenmodell*. Das Domänenmodell transportiert das Domänenwissen, welches durch den Monolithen abgebildet wird. Das Migrationsteam verschafft sich

über das Domänenmodell ein Verständnis über die Geschäftsdomäne. Das Domänenwissen beeinflusst maßgeblich die *Makro-* und *Mikroarchitektur* (vgl. Abschnitt 2.2.1) bei der Implementierung eines Softwaresystems und damit auch der Microservices [Ev04, Ne15]. Zudem hilft das Domänenwissen bei der Festlegung kohäsiver Softwarebausteine, aus denen der Monolithenausschnitt besteht. Aus diesem Grund ist das Domänenmodell wie auch die Anforderungsspezifikation in der Modernisierungsphase von Relevanz. Doch im Gegensatz zu der Anforderungsspezifikation ist das Domänenmodell iterationsübergreifend gültig. Die abgebildete grobgranulare Domänenstruktur im Domänenmodell ist als unabhängig von dem Ist-Zustand des Monolithen anzusehen und besitzt eine Allgemeingültigkeit für die Organisation.

Ausgehend von der Anforderungsspezifikation und dem Domänenmodell können nun die Softwarebausteine des Monolithen als Monolithenausschnitt zusammengefasst werden. Die betroffenen Softwarebausteine werden in einer *Ist-Architekturbeschreibung* festgehalten, sodass das Migrationsteam in der Lage ist, den Ist-Zustand des Monolithen während der gesamten Extraktionsphase zu sichten. Ebenso wie bei der Anforderungsspezifikation ist davon auszugehen, dass die Architekturbeschreibung des Monolithen veraltet oder gar nicht vorhanden ist [RH06]. Daher muss diese wiederhergestellt werden. Die Inhalte der Architekturbeschreibung müssen die verschiedenen *Blickwinkel* (vgl. Abschnitt 2.2.2) der Interessenvertreter abdecken.

Das letzte Migrationsartefakt der Extraktionsphase ist die *Soll-Architekturbeschreibung*. Festgehalten wird dort der Soll-Zustand, der den parallelen Betrieb des Monolithen und Monolithenausschnitts beschreibt. Wie auch in der Ist-Architekturbeschreibung müssen verschiedene *Blickwinkel* berücksichtigt werden. Das Migrationsteam nutzt die Soll-Architekturbeschreibung als Grundlage für die Implementierung des Monolithenausschnitts. Daher gilt es, die *APIs* und die *Datenhaltung* für den Monolithenausschnitt zu entwerfen [KH18, Ne19].

4.3.3 Migrationsartefakte der Modernisierungsphase

Zum Entwurf der domänengetriebenen Microservice-Architektur des Monolithenausschnitts werden in der Modernisierungsphase drei Migrationsartefakte eingeführt. Zusammenfassend werden die formalen Migrationsartefakte in Tabelle 4.5 dargestellt.

In der ersten Migrationsaktivität der Modernisierungsphase wird die *Anforderungsspezifikation* der Extraktionsphase aufgegriffen. Die grundlegenden Prinzipien, welche für die funktionalen und nichtfunktionalen Anforderungen in der Extraktionsphase vorgesehen wurden, gelten weiterhin im Kontext der Modernisierungsphase. Ein besonderes Augenmerk wird dabei aber auf die Verknüpfung der Anforderungsspezifikation mit der Implementierung gelegt, um so den Anforderungen einer *Living Documentation* zu genügen [Ma19]. Zwangsläufig müssen die Anforderungen so spezifiziert werden, dass sie für das automatisierte Testen der Microservices genutzt werden können. Damit steigt die

Tabelle 4.4: Migrationsartefakte der Extraktionsphase

Migrationsartefakt	Erläuterung	KDM-Schicht
Anforderungs-spezifikation	Umfasst die funktionalen und nicht-funktionalen Anforderungen des Ist-Zustandes des Monolithen. Genutzt werden die Anforderungen zur Bestimmung der Softwarebausteine für den Monolithenausschnitt und zur Verifizierung des Funktionsumfangs nach der Extraktion. Das Migrationsartefakt ist nach Abschluss der Extraktionsphase durch Überarbeitung der Anforderungen in der Modernisierungsphase ungültig.	Abstraktionsschicht
Domänenmodell	Beinhaltet die grobgranulare Domänenstruktur der Organisation, damit zum einen das Verständnis der Geschäftsdomäne gefördert wird und zum anderen die Frage der Kohäsion von Softwarebausteinen für den Monolithenausschnitt beantwortet werden kann. Das Migrationsartefakt ist iterationsübergreifend gültig.	Abstraktionsschicht
Ist-Architektur-beschreibung	Umfasst formale Modelle zur Beschreibung des Monolithen. Die formalen Modelle zeigen die Softwarebausteine, die dem Monolithenausschnitt zugeordnet werden. Nach der Extraktionsphase ist dieses Migrationsartefakt als obsolet anzusehen.	Softwarebaustein- und Ressourcen-schicht
Soll-Architektur-beschreibung	Umfasst formale Modelle, die zur Implementierung des Monolithenausschnitts dienen. Die Modelle beschreiben die parallele Ausführung des Monolithenausschnitts neben dem Monolithen. Auch die Soll-Architekturbeschreibung ist nach der Extraktionsphase obsolet.	Softwarebaustein- und Ressourcen-schicht

Bedeutung des Migrationsartefakts für den gesamten Lebenszyklus der zu entwerfenden Microservices und die Wartung der Anforderungen ist obligatorisch für die Weiterentwicklung.

Auch das *Domänenmodell* findet als Migrationsartefakt Verwendung in der Modernisierungsphase. Im Rahmen der Modernisierung wird das Domänenmodell mit *feingranularem Domänenwissen* und die *grobgranulare Struktur* mit weiteren Informationen angereichert. Aus dem neuen Domänenwissen müssen Verknüpfungen innerhalb des feingranularen Domänenwissens hervorgehen, damit das Migrationsteam eine Entscheidung bezüglich der Kohäsion von Funktionalitäten beziehungsweise Geschäftslogik treffen kann. So dient das Domänenmodell dem Entwurf der domänengetriebenen Microservices und der Reorganisation des monolithischen Entwicklungsteams. Durch starken Bezug zu dem Entwurf der Microservices und allgemein zur Implementierung dieser wird das Domänenmodell als Architekturbeschreibung gesehen.

Das letzte Migrationsartefakt der *Schnittstellenbeschreibung* umfasst für einen Microservice die nach außen angebotenen webbasierten APIs. Aus der Schnittstellenbeschreibung muss hervorgehen, welche Web-APIs der Microservice besitzt und wie diese implementiert werden sollen. Die Beschreibung der Web-APIs beruht auf dem gewählten Architekturstil. So wird beispielsweise eine *gRPC API* über eine *Protobuffer-Datei* spezifiziert [IK20]. Jeder entworfene Microservice muss über eine eigene Schnittstellenbeschreibung verfügen, damit eine Unabhängigkeit zwischen den Microservices gegeben ist.

Tabelle 4.5: Migrationsartefakte der Modernisierungsphase

Migrationsartefakt	Erläuterung	KDM-Schicht
Anforderungs-spezifikation	Umfasst die für die Microservices überarbeiteten funktionalen und nicht-funktionalen Anforderungen. Das Migrationsartefakt genügt den Anforderungen einer Living Documentation [Ma19], wodurch eine hohe Qualität von den spezifizierten Anforderungen gefordert wird. Die Anforderungsspezifikation ist ab der Modernisierungsphase für den gesamten Lebenszyklus der Microservices gültig und muss kontinuierlich gewartet werden.	Abstraktionsschicht
Domänenmodell	Stellt die feingranulare Domänenstruktur und angereicherte grobgranulare Domänenstruktur aus der Extraktionsphase dar. Die feingranulare Domänenstruktur modelliert die Funktionsweise der Geschäftslogik und offenbart, wie diese in den Microservices implementiert werden muss. Weiterhin wird darüber die Frage der Kohäsion von Geschäftslogik beantwortet. Aus der angereicherten grobgranularen Domänenstruktur folgt der Entwurf der Microservice-Architektur und die Aufteilung des monolithischen Entwicklungsteams in Service-Teams.	Softwarebaustein- und Abstraktionsschicht
Schnittstellen-beschreibung	Eine Schnittstellenbeschreibung umfasst die Web-APIs eines Microservices. Der Inhalt der Beschreibung hängt mit dem gewählten Architekturstil zusammen.	Softwarebausteinschicht

Tabelle 4.6: Migrationsartefakte der Modernisierungsphase

Migrationsartefakt	Erläuterung	KDM-Schicht
Konfiguration	Wird für die Konfiguration von DevOps-Werkzeugen benötigt.	Ressourcenschicht

4.3.4 Migrationsartefakte der Inbetriebnahmephase

In der Inbetriebnahmephase beschränken sich die Migrationsartefakte auf *Konfigurationsdateien*, die im Rahmen von den gewählten Werkzeugen notwendig werden, welche für die Bereitstellung oder den Betrieb der Microservices notwendig sind (Tabelle 4.6). Beispiele hierfür sind Containerisierungs-Werkzeuge wie Docker, Container-Orchestrierungen wie Kubernetes oder CI/CD-Pipelines wie Jenkins. Für jeden Microservice müssen die Konfigurationsdateien erstellt werden. Überwiegend sind diese Werkzeuge motiviert durch die *DevOps-Prinzipien* für die Automatisierung des Operation-Anteils der Softwareentwicklung. Eine konkrete Vorgabe für die Migrationsartefakte in der Betriebsphase existieren nicht, denn die Konfigurationsdateien sind abhängig von den gewählten Werkzeugen. Üblich ist die Verwendung der Auszeichnungssprache *YAML Ain't Markup Language* (YAML) oder der Objektnotation *JavaScript Object Notation* (JSON).

4.4 Paralleler Betrieb alter und neuer Softwarebausteine

Während der iterativen Migration entstehen in den einzelnen Migrationsiterationen Softwarebausteine, die zum einen selbst Funktionalität zur Verfügung stellen und zum anderen auf die Funktionalität im

Monolithen zugreifen. Unter den neuen Softwarebausteinen werden sowohl die Monolithenausschnitte als auch die Microservices verstanden, die zwar neben dem Monolithen stehen und einzeln betrieben werden, aber dennoch mit diesem interagieren müssen, falls die Abhängigkeiten nicht reduziert werden können. Es entsteht ein Bedarf zum *parallelen Betrieb*, der einige Anforderungen mit sich bringt [Fo04]. Anfragen der Clients (engl. client request) müssen zu den entsprechenden Softwarebausteinen weitergeleitet werden, die letztendlich in der Lage sind, die Anfrage zu beantworten. Zudem muss eine Kommunikation zwischen den neuen Softwarebausteinen und dem Monolithen hergestellt werden, da diese auf Funktionalität oder Daten des Monolithen zugreifen. Auch Daten, die gemeinsam vom Monolithen und den neuen Softwarebausteinen genutzt werden, müssen synchronisiert werden.

In den nachfolgenden Abschnitten wird die Umsetzung dieser Anforderungen diskutiert. Zunächst erfolgt die Betrachtung der Verteilung von Anfragen durch Clients auf die neuen Softwarebausteine und den Monolithen. Anschließend wird die Kommunikation zwischen dem Monolithen und den Softwarebausteinen behandelt, die neben dem Aufrufen von Funktionalität auch die Synchronisation von Daten beabsichtigt.

4.4.1 Verteilen der Client-Anfragen auf die Softwarebausteine

Nach der Implementierung des Monolithenausschnitts durch die Extraktionsphase gilt es, den Monolithenausschnitt für Client-Anfragen zur Verfügung zu stellen. Hierfür werden die in der Extraktionsphase entworfenen Web-APIs notwendig. Darüber hinaus müssen Client-Anfragen zu dem Softwarebaustein geleitet werden, der die Anfragen bearbeiten kann. Im Rahmen dieser Forschungsarbeit wird für diese Art der Integration von Softwarebausteinen während einer Migrationsiteration auf das *StranglerFigApplication*-Muster zurückgegriffen [Fo04]. Das Muster führt zum Treffen der Routing-Entscheidungen bezüglich Anfragen ein *API-Gateway* oder einen *Proxy* ein. Nachfolgend wird die Anwendung eines API-Gateways festgelegt, denn dieses kann im Gegensatz zu einem Proxy über eine für das Softwaresystem spezifische Logik verfügen, während der Funktionalitätsumfang eines Proxys beschränkt ist [MW16]. Das Aufsetzen des API-Gateways wird nicht im Rahmen der entworfenen Systematik vorgenommen und wird als gegeben angesehen (vgl. Abschnitt 1.6.3). Wichtig ist jedoch, dass die Bereitstellung des API-Gateways bereits vor der ersten Migrationsiteration erfolgt ist, sodass innerhalb der Migrationsiteration keine Unterbrechungen notwendig sind.

Die Abbildung 4.7 zeigt die Funktionsweise des API-Gateways bei einer eingehenden Anfrage. In einer Migrationsiteration wurde die kohäsive Funktionalität der Verwaltung von Dokumenten (in der Abbildung als *documents* bezeichnet), die im Sinne eines klassischen Dokumentenmanagementsystems (DMS) entworfen wurde, aus dem Monolithen in einen Monolithenausschnitt überführt. Die lokalen Schnittstellen wurden dabei durch Web-APIs ersetzt, die beispielsweise mit dem *Hypertext Transfer Protocol* (HTTP) arbeiten. Das API-Gateway entscheidet anhand der angefragten Funktionalität des Clients, zu welchem Softwarebaustein die Anfrage weitergeleitet wird. Eine Möglichkeit,

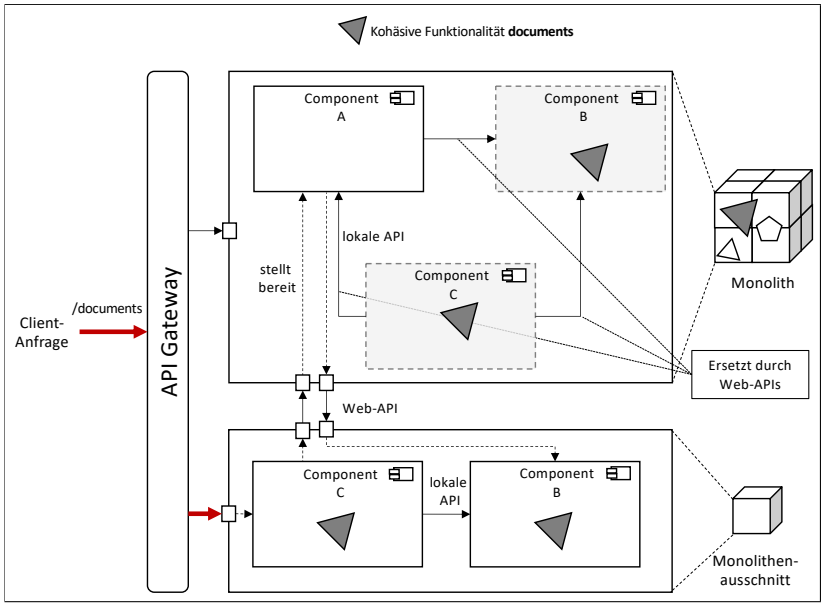

Abbildung 4.7: Einsatz eines API-Gateways zum Weiterleiten der Client-Anfragen zum entsprechenden Softwarebaustein

die Entscheidung im API-Gateway zu treffen, ist die Betrachtung des *URL-Pfades* (vgl. [BM+94]) einer *Representational State Transfer* API (REST) [Ne19]. Im Beispiel sendet der Client eine Anfrage an den Endpunkt "*/documents*", wofür das API-Gateway den Monolithenausschnitt als Empfänger definiert hat. Das API-Gateway ist somit die Grundlage für den parallelen Betrieb des Monolithen und des Monolithenausschnitts während der Extraktionsphase.

Das API-Gateway wird weiterhin in der Inbetriebnahmephase benötigt, denn die Microservices werden ebenfalls parallel zu einem weiterhin bestehenden Monolithen betrieben. Die Funktionsweise des API-Gateways bleibt dabei unberührt und die Weiterleitung erfolgt auf demselben Weg wie bei dem parallelen Betrieb in der Extraktionsphase. Zwar hat dies für das Migrationsteam zur Folge, dass keine zwei unterschiedlichen API-Gateways notwendig sind, aber auch gleichzeitig bedeutet, dass die Endpunkte der Web-APIs von Monolithenausschnitt zu Microservices unverändert bleiben sollten. Ansonsten muss zwangsläufig eine Konfiguration des API-Gateways erfolgen.

4.4.2 Nachrichtenbasierte Kommunikation zwischen Softwarebausteinen

Ausgehend von dem Beispiel der Abbildung 4.7, in dem auf die Verwendung von *REST APIs* gesetzt wird, gilt für die Softwarebausteine, dass durch die direkte Kommunikation beide Softwarebausteine über die Existenz des Kommunikationspartners Kenntnis haben müssen. Konkret bedeutet dies, dass in dem Quellcode der Softwarebausteine der REST-Endpunkt implementiert worden sein muss. Im Sinne eines *Cloud-nativen Softwaresystems* ist die Implementierung so vorzusehen, dass der

REST-Endpunkt durch eine in der Umgebung gelagerte Konfiguration definiert werden kann [Wi17]. Zwischen Softwarebausteinen entsteht eine Kopplung und der Bedarf einer *Orchestrierung* von Abläufen [BP19]. Zudem setzt die Kommunikation durch den REST-Architekturstil auf einem *synchronen Kommunikationsfluss* auf, der verschiedene Herausforderungen mit sich bringt [Be20]. So können bei lang andauernden REST-Anfragen *Transmission Control Protocol*-Zeitüberschreitungen (engl. Transmission Control Protocol timeouts, TCP timeouts) auftreten, die zum einen den angestoßenen Programmablauf abbrechen und zum anderen von dem Client berücksichtigt werden müssen. Hierfür muss die Erfahrung des Migrationsteams vorhanden sein.

Eine Alternative zu der Verwendung von REST APIs und der synchronen Kommunikation stellt die Verwendung des Architekturstils der *ereignisgetriebenen Architektur* (engl. event-driven architecture) [Be20] dar. Ein Ereignis beinhaltet die Daten in binärer Form. Für die Ereignisse existieren, im Gegensatz zu REST, keine Vorgaben über die Strukturierung der Daten. Um eine einfache Interpretation durch die Softwarebausteine zu ermöglichen, wird *JSON* empfohlen. Durch die Verwendung von Ereignissen, die über einen *Message Broker* veröffentlicht werden, müssen die Softwarebausteine nur Kenntnisse über den Message Broker haben und wissen, welche Ereignisse von Relevanz sind. Es entsteht eine lose gekoppelte Softwarearchitektur, die auf einer *asynchronen Kommunikation* aufbaut. Zudem ist asynchrone Kommunikation auf Basis der Ereignisse durch den Message Broker robuster [EF+03]. Denn Ereignisse, die nicht erfolgreich durch einen Softwarebaustein bearbeitet wurden oder bei denen Fehler bei der Übertragung stattgefunden haben, werden automatisch durch den Message Broker erneut zugestellt.

Diese Robustheit kann zur Sicherstellung einer Synchronisation von Daten zwischen dem Monolithen und den neuen Softwarebausteinen genutzt werden. Gleichzeitig wird damit das Migrationsmuster von Henry und Ridene [HR20] realisiert, welches die direkten Abhängigkeiten von dem Monolithenausschnitt beziehungsweise den Microservices zu dem Monolithen verringert. Durch die hohe Robustheit und die damit erreichte lose Kopplung ist bei der Synchronisation von Daten, welche durch eine Migrationsiteration notwendig wird, auf die ereignisgetriebene Architektur zurückzugreifen. Die Kommunikation zwischen den Softwarebausteinen und dem Monolithen wird in Abbildung 4.8 zusammenfassend dargestellt.

Wichtig für die Kommunikation zwischen den Bausteinen ist, dass der Einsatz von Ereignissen nicht den Einsatz von REST APIs zur synchronen Kommunikation ausschließt. Anhand der Anwendungsfälle muss für die Web-API der Architekturstil festgelegt werden.

4.5 Zusammenfassung

Eine iterative Migration setzt auf verschiedene grundlegende Elemente auf. Als Quelle zur Bestimmung der Elemente dienten zum einen bestehende Arbeiten und zum anderen das in dieser Forschungs-

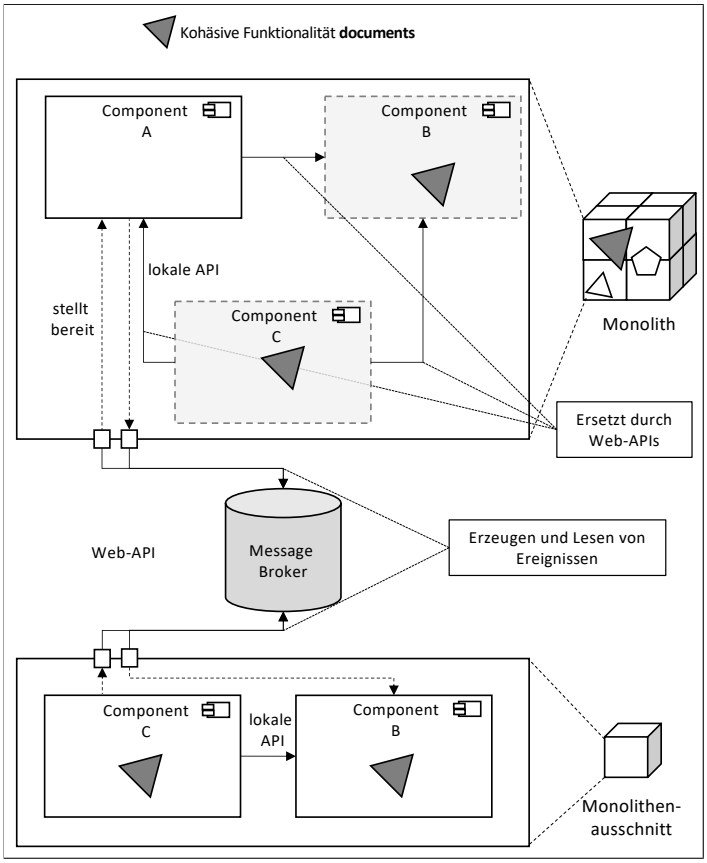

Abbildung 4.8: Einsatz des Message Brokers zur Realisierung einer ereignisgetriebenen Architektur

arbeit entworfene Migrationsrahmenwerk. Zur besseren Einordnung der Elemente wurden zunächst drei Aspekte festgelegt: (1) *organisationsbezogen*, (2) *prozessbezogen* und (3) *entitätsbezogen*. Ein organisationsbezogenes Element bezeichnet ein Element, welches die durch die Organisation vorgegebenen Rahmenbedingungen näher beschreibt. Ein prozessbezogenes Element wird als Bestandteil der *Systematik des Migrationsrahmenwerks* gesehen. Und zuletzt repräsentieren entitätenbezogene Elemente die Elemente, welche im Rahmen der Systematik durch das Migrationsteam betrachtet werden. Zu jedem der Aspekte wurden mehrere Elemente identifiziert und als Definition festgehalten.

Ausgehend von den Definitionen wurde die Systematik für das generelle Migrationsrahmenwerk entworfen. Die Systematik unterteilt eine Migrationsiteration in drei Migrationsphasen: (1) *Extraktionsphase*, (2) *Modernisierungsphase* und (3) *Inbetriebnahmephase*. Die Extraktionsphase definiert durch einen gegebenen *Migrationsauslöser* verhaltensspezifische und strukturelle Aspekte eines *Monolithenausschnitts*. Der Monolithenausschnitt besteht aus kohäsiv zusammenhängenden Softwarebausteinen des Monolithen. Durch die Kohäsion ist eine Eigenständigkeit des Monolithenausschnitts gegeben. Gleichzeitig ist die Kohäsion für die Modernisierungsphase notwendig, denn dort wird der Monolithenausschnitt in einen domänengetriebenen Microservice-Architekturentwurf überführt.

Hierfür werden zunächst die funktionalen und nicht-funktionalen Anforderungen des Monolithenausschnitts auf Korrektheit geprüft und gegebenenfalls überarbeitet. Anschließend erfolgt durch das Zurückgreifen auf Domänenwissen der Entwurf der Microservice-Architektur. Den Abschluss findet die Modernisierungsphase durch das Reorganisieren des monolithischen Entwicklungsteams in *Service-Teams*, die letztendlich für die implementierten Microservices verantwortlich sind. In der Inbetriebnahmephase werden für die implementierten Microservices Vorbereitungen für den Betrieb getroffen. Das Ziel ist es, die *Konfigurationsdateien* für spezifische Werkzeuge umzusetzen, welche den Betrieb der Microservices erleichtern.

In allen *Migrationsaktivitäten* der Migrationsphasen werden *Migrationsartefakte* eingesetzt. Ein Migrationsartefakt entspricht einem formalen Softwareartefakt, das dem Migrationsteam und Interessenvertretern während der Migrationsiteration als Informationsquelle für einen spezifischen Zweck dient. Dieser Zweck ist allgemein beschrieben. Im Rahmen des generellen Entwurfs des Migrationsrahmenwerks wurden noch keine konkreten Modellierungssprachen für die Migrationsartefakte eingeführt. So bleibt die Wahl der Modellierungssprache in der Obliegenheit des Migrationsteams und kann an entsprechende Softwareentwicklungsansätze angepasst werden. Für einige Migrationsartefakte ist die Gültigkeit an die Migrationsiteration gebunden, sodass nach Abschluss der Migrationsiteration das Migrationsartefakt obsolet ist. Aus diesem Grund wurde auch die Begrifflichkeit des Migrationsartefakts zur Abgrenzung gewählt.

Um den parallelen Betrieb des Monolithen und der Softwarebausteine, die während der Migrationsiteration entstehen, zu gewährleisten, werden zwei Konzepte verwendet. Zum einen wird für das Verteilen der Anfragen von Clients ein *API-Gateway* eingeführt. Das API-Gateway trifft Routing-Entscheidungen für die eingehenden Client-Anfragen und entscheidet, welcher Softwarebaustein die Anfrage bearbeiten kann. Zum anderen wird bei der Synchronisation von Datenbeständen zwischen Monolith und neuen Softwarebausteinen auf eine *ereignisgetriebene Architektur* gesetzt. Die Verwendung dieses Architekturstils eignet sich durch die hohe Robustheit der Kommunikation. Zusätzlich sorgt die ereignisgetriebene Architektur für eine lose Kopplung zwischen den Softwarebausteinen.

5 Entwurf der Extraktionsphase

Nachdem in Kapitel 4 die Grundlagen einer iterativen Migration sowie das Migrationsrahmenwerk und die darin enthaltene Systematik erarbeitet wurden, sind nun die einzelnen Migrationsphasen detailliert zu betrachten. Daher widmet sich dieses Kapitel dem detaillierten Entwurf der *Extraktionsphase* und beschreibt einen systematischen Ablauf basierend auf konkreten Migrationsaktivitäten zur Definition eines Monolithenausschnitts. Die konkreten Migrationsaktivitäten und auch die damit verbundenen Migrationsartefakte werden mit Softwareentwicklungsansätzen in Verbindung gebracht. Durch dieses Kapitel wird der zweite Forschungsbeitrag dieser Arbeit betrachtet. Für die Migrationsartefakte der Extraktionsphase aus Abschnitt 4.3.2 werden formale Modelle festgelegt und den einzelnen Migrationsaktivitäten zugeordnet. Zur Veranschaulichung der zu erstellenden Migrationsartefakte wird das *Dokumentenmanagementsystem-Szenario* (DMS) aus Abschnitt 1.5.6 herangezogen.

Mit diesem Kapitel werden zwei zentrale Fragestellungen der Extraktionsphase bearbeitet: (1) *Welche Migrationsaktivitäten sind zur Definition des Monolithenausschnitts notwendig?* (2) *Welche formalen Modelle unterstützen das Migrationsteam bei dem Treffen nachvollziehbarer Entscheidungen?* Die Beantwortung dieser Fragestellungen beruht auf der Verwendung von bekannten Softwareentwicklungsansätzen wie die der *verhaltensgetriebenen Entwicklung* (engl. Behavior-Driven Development, BDD) [Sm14] und des *domänengetriebenen Entwurfs* (engl. Domain-Driven Design, DDD) [Ev04] sowie auf den Praktiken des *Design Recovery* [CC90]. Neben der Beantwortung der beiden Fragestellungen werden innerhalb dieses Kapitels weitere Meilensteine formuliert, die zum besseren Verständnis des Ablaufs der Extraktionsphase dienen sollen. Erreichen jedes dieser Meilensteine ist obligatorisch für das Erreichen des in Abschnitt 4.2.2 formulierten Meilensteins M_1. Daher werden die in diesem Kapitel vorgestellten Meilensteine der Extraktionsphase als Submeilensteine bezeichnet. Zur Festlegung der Submeilensteine werden die in Abschnitt 4.3.2 definierten Migrationsartefakte zu Grunde gelegt, wodurch sich die Submeilensteine $M_{1.1}$ bis $M_{1.4}$ ergeben. Ein Submeilenstein beschreibt dabei die Fertigstellung eines Migrationsartefakts. Die Verknüpfung der Submeilensteine zu den jeweiligen Migrationsartefakten kann der Abbildung 5.1 entnommen werden.

Das vorliegende Kapitel ist wie folgt strukturiert: Zunächst diskutiert Abschnitt 5.1 die prozessuale Sicht auf die Extraktionsphase und gibt eine Übersicht über die darin vorzufindenden Migrationsaktivitäten und Migrationsartefakte. Der weitere Aufbau dieses Kapitels orientiert sich an den Submeilensteinen und damit an der sequentiellen Erstellung der Migrationsartefakte. In Abschnitt 5.2 wird die Spezifikation des Migrationsauslösers als formale Anforderung in der Anforderungsspezifikation behandelt. Abschnitt 5.3 befasst sich mit der Zerlegung der übergeordneten Struktur in die

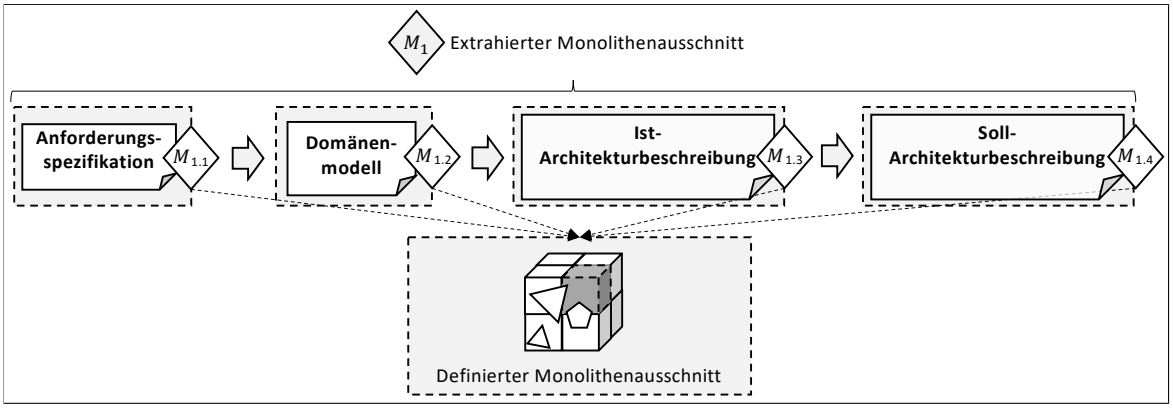

Abbildung 5.1: Übersicht der Submeilensteine innerhalb der Extraktionsphase zur Definition des Monolithenausschnitts

einzelnen Subdomänen der betreffenden Geschäftsdomäne in die einzelnen Subdomänen. Die erhobenen Subdomänen werden in einer Context Map festgehalten und dem Domänenmodell hinzugefügt. Auf Basis der gewonnenen Anforderungsspezifikation und des Domänenmodells kann die konkrete Softwarearchitektur des Monolithen in einer Ist-Architekturbeschreibung festgehalten werden. Die dafür notwendigen Migrationsaktivitäten und formalen Modelle werden in Abschnitt 5.4 behandelt. Zum Abschluss des Kapitels werden in Abschnitt 5.5 Migrationsaktivitäten und formale Modelle vorgestellt, die zur Erarbeitung der Soll-Architekturbeschreibung benötigt werden.

5.1 Übersicht über konkrete Migrationsaktivitäten und -artefakte

Die Übersicht umfasst die konkreten Migrationsaktivitäten und -artefakte und stellt diese in Verbindung zu den jeweiligen abstrakten Konzepten der Abschnitte 4.2.3 und 4.3.2 dar. Die detaillierte Ausarbeitung der konkreten Migrationsaktivitäten und die Verknüpfung dieser mit den konkreten Migrationsartefakten erfolgt im Kontext der weiteren Abschnitte dieses Kapitels. Auch die Definition der konkreten Migrationsartefakte wird in den jeweiligen Abschnitten behandelt. Zu Zwecken des besseren Verständnisses wird im weiteren Verlauf dieses Kapitels der Ablauf der Extraktionsphase nur noch als *Extraktion* bezeichnet.

5.1.1 Zugrundeliegende Herangehensweise der Extraktion

Eine für den Detailentwurf der Extraktion grundlegende, bestehende Arbeit ist die von Knoche und Hasselbring [KH18]. Die Autoren stellen einen Ablauf vor, der das Ziel verfolgt, den Monolithen zunächst in Ausschnitte zu zerlegen und diese als eigenständige Softwarebausteine zur Verfügung zu

stellen, ohne diese direkt in eine Microservice-Architektur zu überführen. Dadurch ist die Migration in eine Microservice-Architektur für das Migrationsteam einfacher zu beherrschen.

Wie bereits in dem abgeleiteten Handlungsbedarf in Abschnitt 3.3 festgestellt, liegt der Fokus des vorgestellten Ablaufs auf der Umwandlung der lokalen Schnittstellen in Web-Schnittstellen. Eine systematische und nachvollziehbare Identifikation der Stellen im Quellcode, die letztendlich die Grenze eines Ausschnitts repräsentieren und in eine Web-Schnittstelle überführt werden müssen, wird innerhalb des Ablaufs nicht thematisiert. Ausgehend von dem vorgestellten Ablauf sieht die mit dieser Forschungsarbeit vorgestellte Extraktion eine solche vorgelagerte Definition des Monolithenausschnitts vor, damit für die Umwandlung der lokalen Schnittstellen die einzelnen Quellcodepassagen nachvollziehbar sind. Gewährleistet wird dies durch Softwareentwicklungsansätze wie *BDD* [Sm14] und *DDD* [Ev04]. Weiterhin werden auch Praktiken wie die statische Quellcodeanalyse des *Design Recovery* [CC90] verwendet, um die *konkrete Softwarearchitektur* wiederherstellen zu können. Die Verwendung der Ansätze und Praktiken wird in den Abschnitten der Migrationsaktivitäten detailliert beschrieben.

5.1.2 Ganzheitlicher Entwurf der Extraktion

Der ganzheitliche Entwurf der Extraktion kann der Abbildung 5.2 entnommen werden. Grundlegend für den detaillierten Entwurf sind die festgelegten *abstrakten Migrationsaktivitäten* der Extraktionsphase durch das generelle Migrationsrahmenwerk. Aus den abstrakten Migrationsaktivitäten folgt eine bereits vordefinierte Extraktion, die sequentiell durchlaufen wird: (1) *Spezifikation des Migrationsauslösers*, (2) *Zerlegung der Geschäftsdomäne*, (3) *Wiederherstellung der konkreten Softwarearchitektur* und (4) *Planung der konzeptionellen Softwarearchitektur*. Den Abschluss einer abstrakten Migrationsaktivität kennzeichnet die Erstellung eines abstrakten Migrationsartefakts.

Die Extraktion beruht auf der Einführung von *konkreten Migrationsaktivitäten* und *-artefakten*. Diese gelten als eine konkrete Ausprägung der abstrakten Migrationsaktivitäten und -artefakte aus dem Gesamtentwurf der iterativen Migration. Die abstrakten Elemente werden zur Gruppierung der konkreten Migrationsaktivitäten und -artefakte der Extraktion verwendet, sodass eine bessere Lesbarkeit gegeben ist. Die Erstellung eines abstrakten Migrationsartefakts und damit die Durchführung einer abstrakten Migrationsaktivität wird anhand einer bestimmten Anzahl an konkreten Migrationsaktivitäten erreicht, die nachfolgend nur noch als Migrationsaktivität bezeichnet werden. Beispielsweise werden für die abstrakte Migrationsaktivität *Spezifikation des Migrationsauslösers*, die den Migrationsauslöser in das abstrakte Migrationsartefakt der *Anforderungsspezifikation* überführt, drei Migrationsaktivitäten festgelegt. Bei der Ausführung einer Migrationsaktivität werden konkrete Migrationsartefakte erstellt wie das *Gherkin Feature* bei der *Formalisierung einer funktionalen Anforderung*. Die konkreten Migrationsartefakte sind formale Modelle, die der Semantik des damit verbundenen abstrakten Migrationsartefakts entsprechen. So erfüllt das *Gherkin Feature* den Zweck der *Anforderungsspezifikation*.

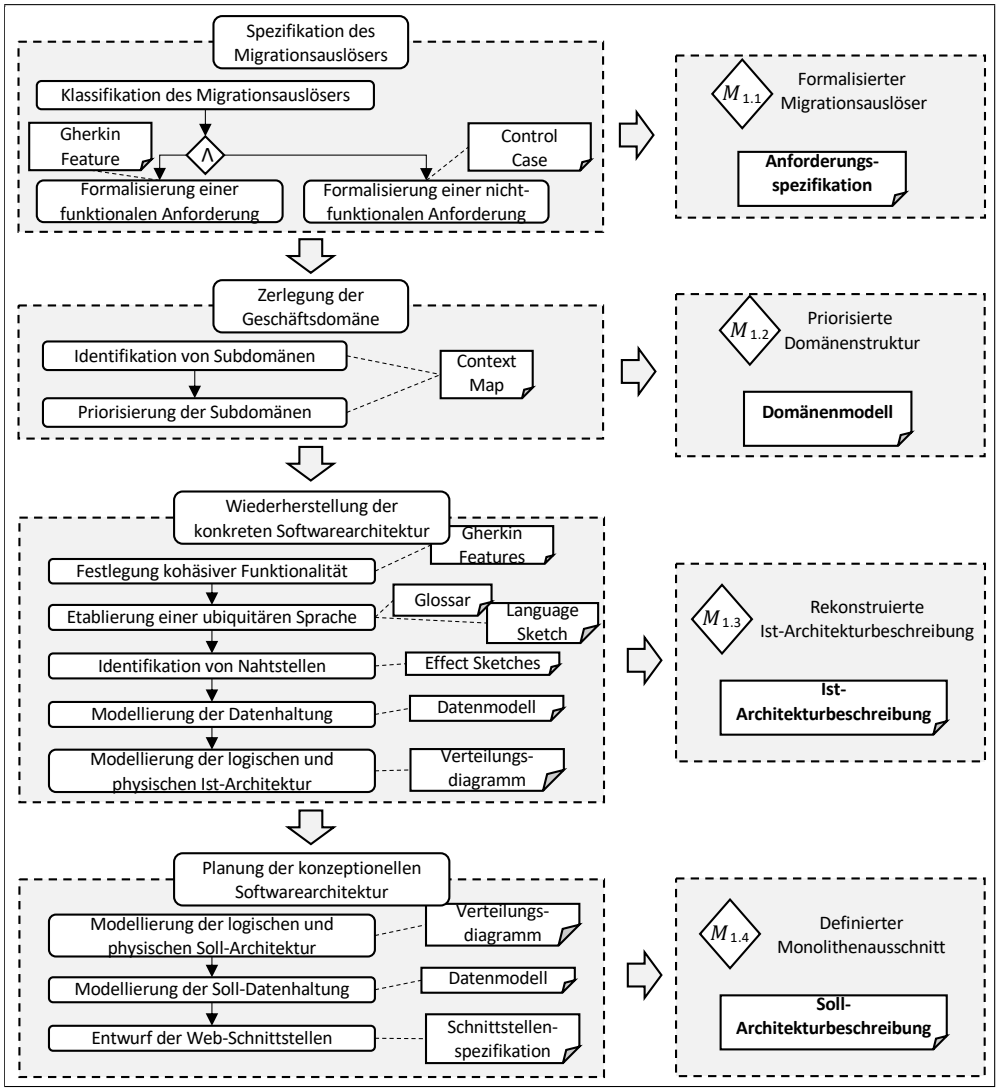

Abbildung 5.2: Übersicht der konkreten Migrationsaktivitäten und -artefakte der Extraktionsphase

In den jeweiligen Migrationsaktivitäten können weitere Softwareartefakte erstellt werden, die den Zweck einer *temporären Informationsquelle* erfüllen, um die Ausführung der Migrationsaktivität systematischer zu gestalten. Für temporäre Migrationsartefakte, die nicht an die abstrakten Migrationsartefakte gebunden sind, können auch informelle Modellierungssprachen verwendet werden. Diese temporären Softwareartefakte werden nicht den abstrakten Migrationsartefakten zugeordnet und nach Abschluss der Migrationsaktivität verworfen. Im Interesse der Übersichtlichkeit werden die temporären Softwareartefakte im Kontext der Systematik nicht in der Abbildung 5.2 dargestellt. Dies erfolgt in den Abschnitten der jeweiligen Migrationsaktivitäten.

5.2 Spezifikation des Migrationsauslösers

Den Beginn der Extraktion gekennzeichnet die abstrakte Migrationsaktivität *Spezifikation des Migrationsauslösers*, die dabei das Ziel verfolgt, den Migrationsauslöser formal aufzubereiten, damit dieser im weiteren Verlauf der Migrationsiteration verwendet werden kann. In den nachfolgenden Abschnitten werden die Migrationsaktivitäten zur Spezifikation detailliert beschrieben und Migrationsartefakte eingeführt, um den Migrationsauslöser formal festhalten zu können, sodass der Zweck der *Anforderungsspezifikation* erfüllt ist.

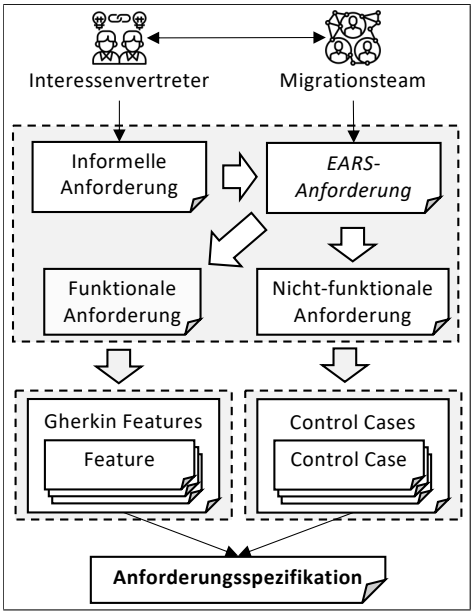

Abbildung 5.3: Sequentielle Erstellung von Softwareartefakten zur Spezifikation des Migrationsauslösers

Abbildung 5.3 zeigt die sequentielle Erstellung von Softwareartefakten zur Spezifikation des Migrationsauslösers in der Anforderungsspezifikation. Bei der *informellen Anforderung* und *EARS-Anforderung* handelt es sich um temporäre Softwareartefakte. Das *Gherkin Feature* beziehungsweise der *Control Case* sind Migrationsartefakte, die dem abstrakten Migrationsartefakt zugeordnet sind.

Die Extraktion sieht für die Spezifikation eine enge Kollaboration zwischen Migrationsteam und Interessenvertretern vor. Anforderungen, übermittelt durch Interessenvertreter, werden überwiegend in informeller Form an das Migrationsteam übermittelt und in unstrukturierter natürlicher Sprache verfasst sind. Hinzu kommen die von Aurum und Wohlin [AW05] identifizierten negativen Eigenschaften bei solchen informellen Anforderungen: (1) *unvollständig*, (2) *inkonsistent* und (3) *konkurrierend*. Welche Eigenschaften wie stark vertreten sind, hängt von den Fähigkeiten der Interessenvertreter ab. Aus den getroffenen Annahmen folgt die Notwendigkeit der Strukturierung der informellen Anforderung, sodass diese effizienter für Diskussionen genutzt und systematisch formalisiert werden

kann. Es folgt für die Extraktion eine Migrationsaktivität zur *Strukturierung und Klassifikation der informellen Anforderung*. Eine Klassifikation ist ebenfalls notwendig, um zwischen funktionalen (engl. functional requirement) und nicht-funktionalen Anforderungen (engl. non-functional requirement) als Migrationauslöser unterscheiden zu können. Abhängig von dem festgestellten Typ der Anforderung unterscheidet sich das Vorgehen der Extraktion. In der Migrationsaktivität wird auf den Strukturierungsansatz *Easy Approach Requirement Syntax* (EARS) zurückgegriffen. Dieser Ansatz wurde von Mavin et al. [MW+09] zur Verbesserung der Kollaboration und des Verständnisses von Anforderungen vorgestellt. Die strukturierte Anforderung wird weiterführend als *EARS-Anforderung* bezeichnet. Zur Klassifikation der EARS-Anforderung greift die Migrationsaktivität auf einen Ansatz von Böschen et al. [BB+16] zurück, der anhand bestimmter Schlüsselwörter Entscheidungen über den Typ der Anforderung trifft.

Zum Abschluss der Spezifikation des Migrationsauslösers wird die klassifizierte Anforderung mittels einer formalen Sprache festgehalten. Zur Formalisierung wird dabei zwischen zwei Migrationsaktivitäten unterschieden: (1) *Formalisierung einer funktionalen Anforderung* und (2) *Formalisierung einer nicht-funktionalen Anforderung*. Beide Migrationsaktivitäten setzen dabei auf eine andere formale Sprache. Eine besondere Anforderung wird dabei an die formale Sprache einer funktionalen Anforderung gestellt. Um nach der Extraktionsphase die Funktionalität des Monolithenausschnitts verifizieren zu können, wird eine automatisierte Testbarkeit und enge Verknüpfung zum Quellcode der formalen funktionalen Anforderung vorausgesetzt. Als formale Sprache für funktionalen Anforderungen wird auf *Gherkin* [Sm14] zurückgegriffen. Die nicht-funktionalen Anforderungen werden über *Control Cases* [ZP06] und *Planguage* [Gi05] formalisiert.

5.2.1 Strukturierung und Klassifikation der informellen Anforderung

In der ersten Migrationsaktivität der Extraktionsphase wird der Migrationsauslöser aufgenommen, strukturiert und anschließend einer Klassifikation in die zwei Klassen funktionale oder nicht-funktionale Anforderung unterzogen (Abbildung 5.4). Dazu wird zunächst die *informelle Anforderung* anhand von *Standardformulierungen der EARS* strukturiert [MW+09] und anschließend der Klassifizierungsansatz von Böschen et al. [BB+16] auf die *EARS-Anforderung* angewandt.

Die Überführung der informellen Anforderung in eine EARS-Anforderung ist an eine enge Zusammenarbeit zwischen Interessenvertreter und Migrationsteam geknüpft, denn in der Regel ist ein weiterer Informationsaustausch notwendig. EARS setzt dabei auf *fünf Standardformulierungen*, die jeweils durch einen bestimmten Zweck und Schlüsselwörter geprägt sind [MW+09]: (1) *ubiquitäre Anforderung*, (2) *ereignisgetriebene Anforderung*, (3) *ungewolltes Verhalten*, (4) *zustandsabhängige Anforderung* und (5) *optionale Funktionalität*. Über die Standardformulierungen werden informellen Anforderungen in natürlicher Sprache eine wiederkehrende und konsistente Struktur verliehen, was zum einen die Erstellung vereinfacht und zum anderen die Vollständigkeit unterstützt. Welche der

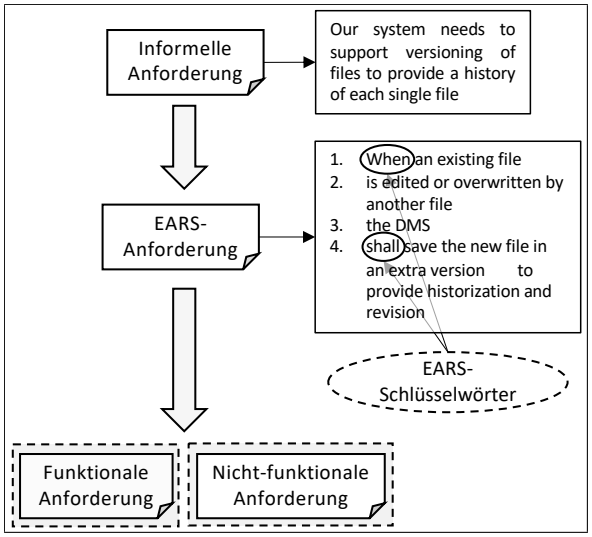

Abbildung 5.4: Strukturierung und Klassifikation der informellen Anforderung

Standardformulierungen verwendet werden muss, hängt von der betrachteten informellen Anforderung ab. Zur Unterstützung der Ausarbeitung der EARS-Anforderung sollte auf Praktiken des Requirement Engineerings wie das *Brainstorming* zurückgegriffen werden [TS+12]. Eine beispielhafte Überführung einer informellen Anforderung findet sich in Abbildung 5.4, in der die Standardformulierung der ereignisgetriebenen Anforderung entspricht.

Abschließend erfolgt die Klassifikation der EARS-Anforderung. Böschen et al. [BB+16] stellen einen Ansatz vor, der ausgehend von EARS-Anforderungen und den darin befindlichen Schlüsselwörtern der einzelnen Standardformulierungen Aussagen über die Klasse der Anforderung trifft. Schlüsselwörter wie "*when*", "*if*" oder "*whenever*" sind starke Zeichen für eine funktionale Anforderung. Findet sich keines der Schlüsselwörter wieder, ist zunächst die Überarbeitung der EARS-Anforderung durchzuführen, sodass möglichst Standardformulierungen verwendet werden. Ist dies jedoch nicht möglich, handelt es sich sehr wahrscheinlich um eine nicht-funktionale Anforderung. Die EARS-Anforderung der Abbildung 5.4 wird als funktionale Anforderung klassifiziert, da die Schlüsselwörter "*when*" und "*shall*" verwendet werden.

5.2.2 Formalisierung der strukturierten funktionalen Anforderung

Stellt sich die EARS-Anforderung als funktionale Anforderung heraus, muss diese für die Implementierung durch einen Softwareentwickler entsprechend aufbereitet werden [BT76]. In der mittels EARS formulierten Anforderung fehlen wichtige Implementierungsdetails, die später in der Anforderungsspezifikation vorhanden sein müssen. Welche Implementierungsdetails noch notwendig sind, kann aus der Schichtenarchitektur (vgl. Abschnitt 2.2.3) abgeleitet werden.

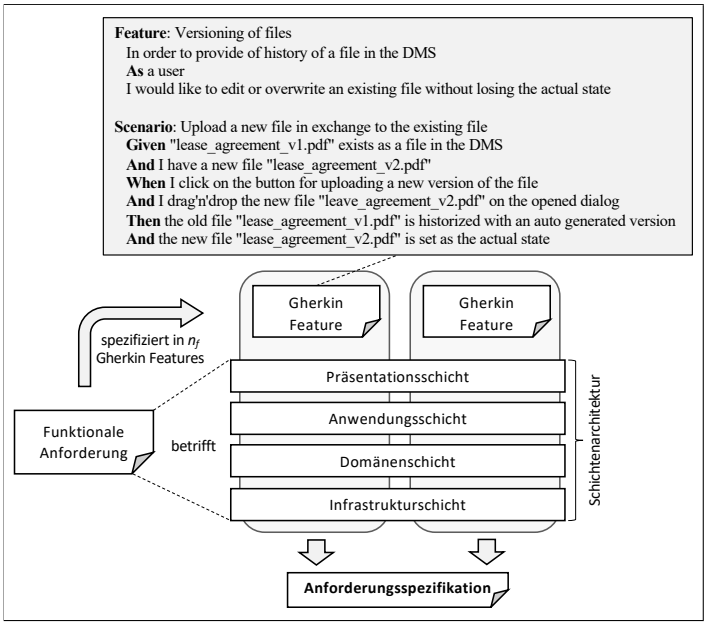

Abbildung 5.5: Erweitern der funktionalen Anforderung durch Gherkin Features für die Implementierung

Mittels der von *BDD* (vgl. Abschnitt 2.3) eingesetzten Beschreibungssprache *Gherkin*, die den Gedanken von Standardklauseln wie EARS aufgreift, wird die funktionale Anforderung für die Implementierung erweitert (Abbildung 5.5). Das Entwicklungsteam ist durch Gherkin in der Lage, die funktionale Anforderung in Form eines *Gherkin Features* mit Informationen für die einzelnen Schichten der Schichtenarchitektur anzureichern [WH+17]. Aus einer funktionalen Anforderung können n_f Gherkin Features abgeleitet werden. Weiterhin erfüllt die Anforderungsspezifikation nach BDD die Ansprüche einer *Living Documentation* durch die Verknüpfung der Anforderungen mit den Tests im Quellcode. Durch die Ausführbarkeit der Gherkin Features als automatisierte Tests wiederum kann nach der Extraktionsphase die Korrektheit der funktionalen Anforderung verifiziert werden. Die Eignung von BDD im Kontext von Microservice-Architekturen ist bereits durch Arbeiten in verschiedenen Domänen bekannt [RG15, HG+17, HS+19, AS+19].

5.2.3 Formalisierung der strukturierten nicht-funktionalen Anforderung

Ergibt die Klassifizierung der EARS-Anforderung eine nicht-funktionale Anforderung, so wird diese zur Integration in die *Anforderungsspezifikation* ebenfalls formalisiert. Für die nicht-funktionalen Anforderungen muss eine alternative Formalisierung gefunden werden, denn hierfür eignet Gherkin sich nicht. Wichtig für die Formalisierung der nicht-funktionalen Anforderungen ist zudem, dass zum Zeitpunkt der Erhebung kein Bezug zu Softwarebausteinen des Monolithen hergestellt werden muss. Das Migrationsteam kann in dieser Phase der Migrationsiteration den Migrationsauslöser nicht

mit den relevanten Softwarebausteine verknüpfen, da der Monolithenausschnitt noch nicht definiert wurde. Solche Einschränkungen existieren beispielsweise bei dem Ansatz der bestehenden Arbeit von Saadatmand et al. [SC+11], welche auf die UML zurückgreift oder bei dem von Soltani et al. [SA+12] entwickelten Ansatz, der *Feature-Bäume* zur Formalisierung verwendet.

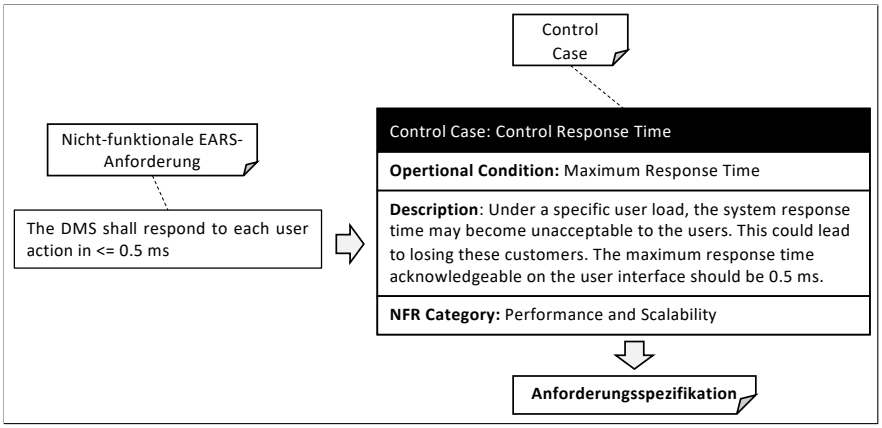

Abbildung 5.6: Formalisierung der nicht-funktionalen Anforderung in einen Control Case [ZP06]

Zou und Pavlovski [ZP06] stellen mit dem Konzept der *Control Cases* eine Vorlage zur Spezifikation von nicht-funktionalen Anforderungen vor, die zum einen der obigen Einschränkung gerecht wird und zum anderen eine Erweiterbarkeit durch eine erweiterte Control Case-Vorlage vorsieht, sobald der Monolithenausschnitt definiert wurde. Die einfache Version des Control Case beschreibt allgemeine Qualitätseigenschaften des zugrundeliegenden Softwaresystems. Die erweiterte Version der Control Cases sieht auch die Referenz auf die Softwarebausteine vor. Ein Vorteil der Control Cases ist, wie auch bei den Gherkin Features, die Verwendung der natürlichen Sprache, um leicht mit den Interessenvertretern diskutieren zu können. Um der Mehrdeutigkeit und Unvollständigkeit der natürlichen Sprache bei nicht-funktionalen Anforderungen entgegenzuwirken, findet die *Planguage* [Gi05] Verwendung. Die Planguage führt Schlüsselwörter ein, welche in die Beschreibung des Control Cases aufgenommen werden und als Kontrolle für das Migrationsteam dienen. Abbildung 5.6 zeigt beispielhaft die Überführung einer nicht-funktionalen Anforderung in einen Control Case.

5.3 Zerlegung der Geschäftsdomäne

Nachdem der Migrationsauslöser spezifiziert wurde, beabsichtigt die Systematik des Migrationsrahmenwerks die Zerlegung der Makrogeschäftsdomäne, damit Subdomänen zur Gruppierung von Funktionalität des Monolithen herangezogen werden können. Die Extraktion sieht für das Erreichen dieses Ziels zwei Migrationsaktivitäten vor, welche zunächst die Subdomänen identifizieren und diese anschließend einer Priorisierung unterziehen. Unabhängig von einer Migrationsaktivität wird

zur formalen Modellierung der Geschäftsdomäne und der dazugehörigen Subdomänen ein *Unified Modeling Language-Profil* (UML-Profil) vorgestellt.

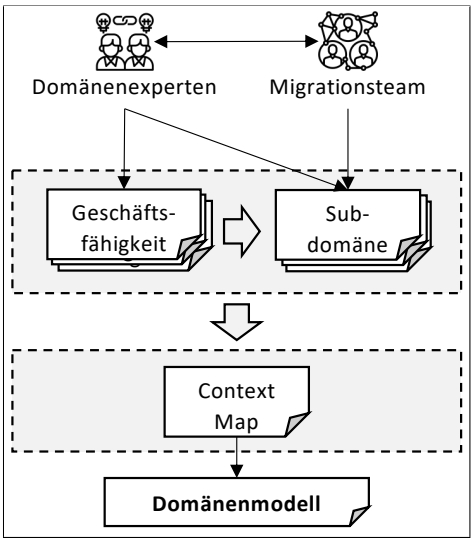

Abbildung 5.7: Zerlegen der Geschäftsdomäne in Subdomänen anhand der Geschäftsfähigkeiten

Abbildung 5.7 veranschaulicht den sequentiellen Verlauf der Migrationsaktivitäten in Verbindung mit den generierten Softwareartefakten. Das Domänenwissen der Domänenexperten ist zur Erfassung einer validen Struktur der grobgranularen Geschäftsdomäne von zentraler Bedeutung [In18]. Zur Erfassung und Zerlegung der Geschäftsdomäne wird auf den Softwareentwicklungsansatz *DDD* [Ev04] zurückgegriffen. DDD wurde bereits als Inspiration bei dem Entwurf der *Systematik des generellen Migrationsrahmenwerks* verwendet (vgl. Abschnitt 4.2.3). Für den detaillierten Entwurf der Extraktion werden nun konkrete Prinzipien, Konzepte und Methodiken übernommen.

Die in der Abbildung 5.7 dargestellte *Context Map* ist ein von DDD eingeführtes Modell zur Erfassung des *strategischen Entwurfs* der Organisation (vgl. Abschnitt 2.4.4). Konkret beschreibt der strategische Entwurf die Strukturierung der gesamten Domäne in abgeschlossene Bestandteile, die sogenannten Subdomänen [Ve13]. Dies entspricht der gewünschten Makrostruktur der Geschäftsdomäne.

Das in der Organisation vorzufindende Domänenwissen kann auf die verschiedenen Subdomänen verteilt werden. Die vorherrschende Komplexitität des Domänenwissens und die damit verbundene Geschäftslogik kann kohäsiv durch die Subdomänen gruppiert werden. Das Domänenwissen und die Geschäftslogik verknüpft Tune [Tu20] mit dem Begriff der *Geschäftsfähigkeiten* (engl. business capabilities) und sieht diese als geeigneten Ausgangspunkt für die Identifikation von Subdomänen an. Zu Beginn der Zerlegung der Geschäftsdomäne wird der Ansatz von Tune verwendet.

In der Regel existieren innerhalb der Organisation weder Dokumentationen über die Subdomänen noch über die einzelnen Geschäftsfähigkeiten. Aus diesem Grund erfolgt zunächst die Erarbeitung

der einzelnen Geschäftsfähigkeiten. Allgemein beschreibt Evans [Ev04] das Finden des strategischen Entwurfs als explorative Tätigkeit. Die Migrationsaktivität zur Identifikation der Subdomänen muss somit mit den Geschäftsfähigkeiten beginnen. Für die Zwecke der Geschäftsfähigkeiten, aber auch für die der Subdomänen, hat sich der von Brandolini [Br13b] vorgestellte Brainstorming-Ansatz *Event Storming* etabliert. In enger Zusammenarbeit erarbeiten Domänenexperten und das Migrationsteam gemeinsam die Geschäftsfähigkeiten und sind darüber hinaus in der Lage, diese in Subdomänen zu gruppieren.

Abschließend gilt es, die gefundenen Subdomänen zu priorisieren, um zum einen die Bedeutung der Subdomäne für die Organisation zu manifestieren und zum anderen eine mögliche Reihenfolge zur Migration der Funktionalität innerhalb des Monolithen zu finden; die Reihenfolge ist bei einer nicht-funktionalen Anforderung als Migrationsauslöser relevant. Die Priorisierung erfolgt anhand der von Vernon vorgestellten [Ve13] Klassen der Subdomänen.

5.3.1 Identifikation von Subdomänen

Als erste Migrationsaktivität zur Zerlegung der Geschäftsdomäne wurde die *Identifikation der Subdomänen* festgelegt. Die Aktivität sieht, anhand der Brainstorming-Methode Event Storming, eine iterative und explorative Erhebung der Subdomänen vor. Eine wichtige Einschränkung dabei ist, dass nur die Subdomänen erhoben werden, die durch das monolithische Softwaresystem betroffen sind, denn nur diese sind für die Durchführung der Migration von Relevanz. Das Vorgehen dieser Migrationsaktivität sieht die Durchführung von Workshops vor, in denen die Domänenexperten und das Migrationsteam kollaborativ an der Identifikation von Subdomänen arbeiten.

Im eigentlichen Sinne dient das Event Storming dem Finden von fachlichen Ereignissen innerhalb einer Geschäftsdomäne und der Verkettung dieser als sequentieller Ablauf. Ein fachliches Ereignis referenziert dabei auf das von DDD vorgestellte *Domänenereignis* (engl. domain event), das eine Veränderung des Zustands eines *Domänenobjekts* kennzeichnet (vgl. Abschnitt 2.4.1). Ein Domänenereignis tritt als Ergebnis im Kontext einer Geschäftsfähigkeit auf, weshalb von den Domänenereignissen ein Rückschluss auf die Geschäftsfähigkeiten möglich ist. Die Subdomänen, die das eigentliche Ziel dieser Migrationsaktivität sind, werden von Brandolini [Br13b] als Nebenprodukt der Domänenereignisse bezeichnet. Übergänge von einem Domänenereignis zu dem nächsten können eine fachliche Grenze darstellen, die letztlich den Subdomänen entspricht. Somit gilt es, für das Migrationsteam zunächst mit den Domänenexperten die Ereignisse zu bestimmen und anschließend die fachlichen Grenzen explizit festzulegen.

Die Abbildung 5.8 zeigt einen Überblick über die beteiligten Parteien und die darin involvierten Informationsquellen. Die primären Informationsquellen sind die partizipierenden Domänenexperten, die über das Domänenwissen verfügen. Ein wichtiger Faktor ist dabei die Diversität der Domänenexperten,

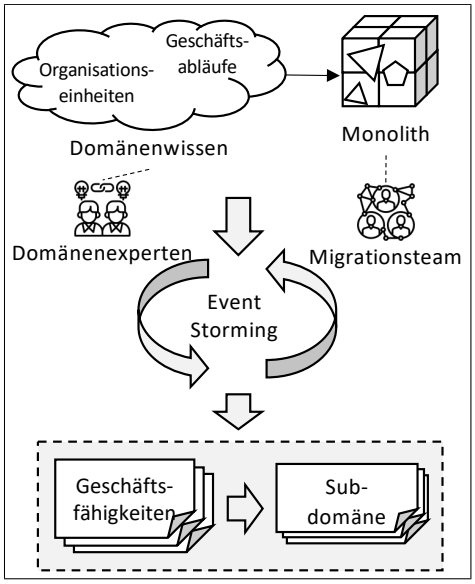

Abbildung 5.8: Einsatz des Event Stormings zur Identifikation der Subdomänen

die möglichst aus allen Bereichen der Organisation stammen. So ist sichergestellt, dass die Geschäftsdomäne aus allen Blickwinkeln betrachtet wird und ein breites Domänenwissen einfließt. Typische Informationsquellen sind Gorodinski [Go13] zufolge die bereits angesprochenen *Geschäftsfähigkeiten*, die in *Geschäftsabläufen* abgebildet und durch unterschiedliche *Organisationseinheiten* innerhalb der Organisation durchgeführt werden. Aber auch das bestehende monolithische Softwaresystem verfügt über Domänenwissen, denn die Geschäftsfähigkeiten wurden bereits implementiert [In18].

Eine Geschäftsdomäne derselben Organisation kann auf viele verschiedene Art und Weisen interpretiert werden, sodass sich eine beliebige Anzahl strategischer Entwürfe mit unterschiedlichen Subdomänen ergibt. Die Entwürfe können sich maßgeblich voneinander unterscheiden. Evans [Ev04] bezeichnet diesen Interpretationsspielraum als *Model Exploration Whirlpool*. Der strategische Entwurf ist damit keinesfalls als kanonisch anzusehen und sollte mit jeder Migrationsiteration erneut betrachtet werden; der strategische Entwurf beziehungsweise die Context Map ist als iterationsübergreifend anzusehen (vgl. Abschnitt 4.3.2).

Abbildung 5.9 zeigt das Ergebnis eines durchgeführten Event Stormings anhand des DMS-Szenarios. Das Beispiel zeigt vier Domänenereignisse, die anhand des Domänenwissens identifiziert werden. Zwischen den Domänenereignissen finden Kontextwechsel statt, die anhand von Verantwortlichkeiten der Organisationseinheiten bestimmt werden. Während das Ereignis "*file uploaded*" durch den Benutzer des DMS ausgelöst wird, werden die Ereignisse "*file type identified*" und "*values in file extracted*" automatisiert durch Softwarebausteine des DMS ausgelöst. Für diese Softwarebausteine ist zudem eine eigenständige Organisationeinheit "*AI Research & Development*" zuständig. Jeder Kontextwechsel wird anhand einer eigenständigen Subdomäne abgebildet.

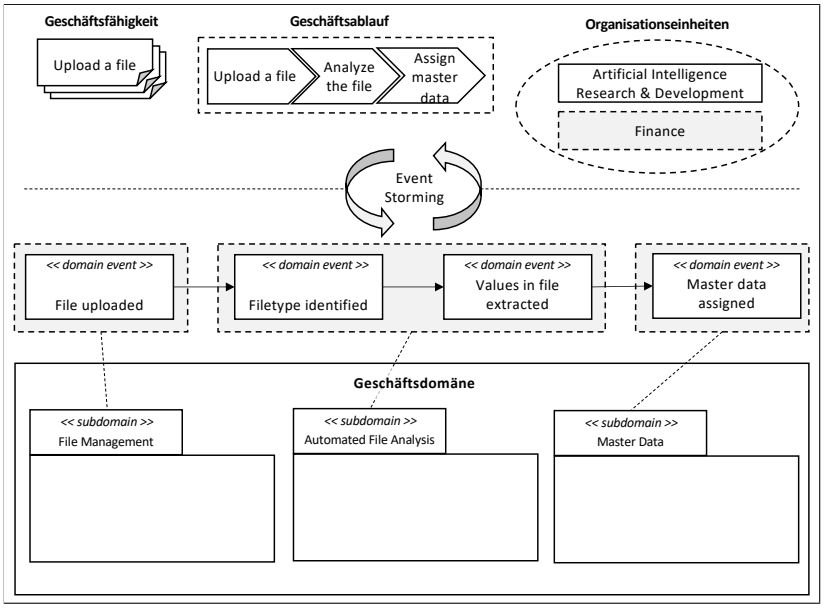

Abbildung 5.9: Durchführung des Event Stormings

5.3.2 Priorisierung der Subdomänen

In dieser Migrationsaktivität werden die identifizierten Subdomänen (vgl. Abschnitt 5.3.1) anhand ihrer Bedeutung für die Organisation priorisiert. Die Extraktion setzt auf die Klassifikation in die drei von Vernon [Ve13] vorgestellten Klassen: (1) Kern, (2) unterstützend und (3) generisch. Auch hier stellen die Domänenexperten, die vor allem die Kern-Subdomänen identifizieren können, die grundlegende Informationsquelle zur Entscheidung der Priorisierung dar [In18].

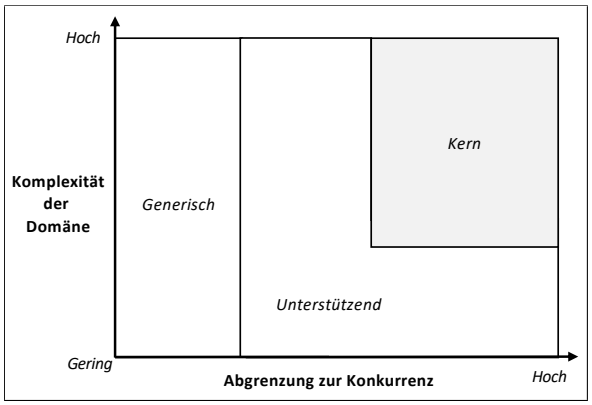

Abbildung 5.10: Priorisierung der Subdomänen nach [Tu20c]

Zur Unterstützung der Domänenexperten und des Migrationsteams stellt Tune [Tu20c] einen Ansatz zur einfachen Priorisierung anhand zweier Faktoren vor (Abbildung 5.10): (1) *Komplexität der Domäne* und (2) *Abgrenzung zur Konkurrenz*. Die Komplexität der Subdomäne beschreibt dabei, wie

schwer es ist, Lösungen für die in der Subdomäne beinhalteten Problemstellungen zu finden. Die Abgrenzung zur Konkurrenz beschreibt, ob neben der eigenen Organisation bereits Konkurrenten an Lösungen bezüglich der Subdomäne arbeiten. Je nach Ausprägung der beiden Faktoren entscheidet sich, welche Priorität der Subdomäne zugeordnet wird. Eine weitere Möglichkeit zur Priorisierung der Subdomänen ist das *Explore, Exploit, Sustain, Retire*-Modell von Humble et al. [HM+14], das in Verbindung mit dem Ansatz von Tune verwendet werden kann. Das Modell beschreibt den Lebenszyklus von Subdomänen von der Erforschung bis hin zur Stilllegung von Lösungen in vier Phasen. Jeder Phasenwechsel zieht eine neue Priorisierung der Subdomäne nach sich. Wenn beispielsweise eine Kern-Subdomäne von der *Sustain-Phase* in die *Retire-Phase* wechselt, sollte die Subdomäne nicht mehr als Kern priorisiert werden.

Die Priorisierung anhand der Bedeutung der Subdomäne für die Organisation ist bei der Durchführung der Extraktionsphase mit einer nicht-funktionalen Anforderung als Migrationsauslöser (vgl. Abschnitt 5.2.1) besonders relevant, denn dort wird als Monolithenausschnitt eine gesamte Subdomäne betrachtet. Daher ist auch die Neubewertung und gegebenenfalls die Umpriorisierung der Subdomäne bei jeder neuen Migrationsiteration notwendig, um keiner unbedeutenden Subdomäne in einer Migrationsiteration zu viel Bedeutung zuzumessen.

5.3.3 Modellierung der Context Map

Der *strategische Entwurf* wird in einer *Context Map* modelliert. Eine große Schwachstelle von *DDD* ist, dass für keines der vorgestellten Modelle eine formale Modellierungssprache vorgestellt wird. Somit ist zunächst kein einheitliches Erscheinungsbild der Context Map gewährleistet, was dazu führt, dass bei wechselnden Softwareentwicklern in dem Migrationsteam von Migrationsiteration zu Migrationsiteration Verständnisprobleme und Inkonsistenzen auftreten können. Und gerade bei dem *Domänenmodell*, welches iterationsübergreifend gültig ist, muss dies vermieden werden.

Zur Auflösung dieser Problematiken wird zur Vereinheitlichung eine formale Modellierungssprache eingeführt. Hippchen et al. [HS+19] greifen dafür auf die *UML* zurück und führen ein auf dem Komponentendiagramm basierendes UML-Profil für eine Context Map ein. Das vorgestellte UML-Profil umfasst alle von DDD genannten Bestandteile der Context Map. In Abbildung 5.11 wird ein Ausschnitt des UML-Profils anhand des DMS-Szenarios veranschaulicht.

5.4 Wiederherstellung der konkreten Softwarearchitektur

Nachdem die Geschäftsdomäne zerlegt wurde und damit die einzelnen Subdomänen bekannt sind, kann nun die Wiederherstellung der *konkreten Softwarearchitektur* erfolgen. Die dafür notwendigen Migrationsaktivitäten greifen auf Praktiken des *Reverse Engineerings* beziehungsweise des *Design*

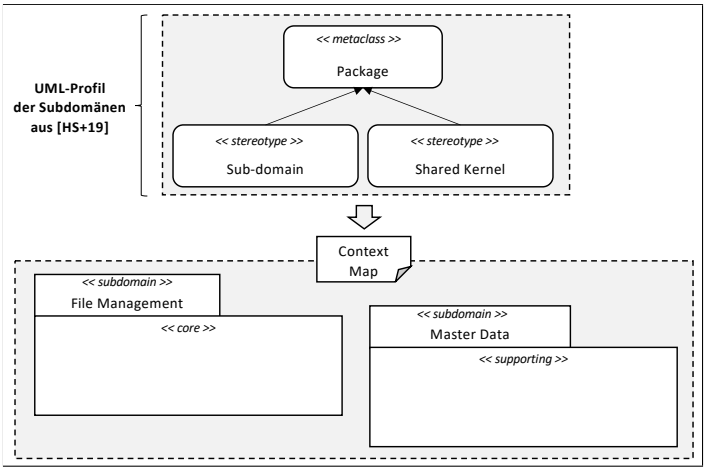

Abbildung 5.11: Context Map nach dem UML-Profil aus [HS+19]

Recoverys zurück. Durch die Praktiken wird die konkrete Softwarearchitektur in Form von formalen Migrationsartefakten in eine *Ist-Architekturbeschreibung* überführt. In der Abbildung 5.12 werden die Migrationsartefakte zusammenfassend dargestellt.

Die durchgeführten Migrationsaktivitäten bestimmen den Umfang des Monolithenausschnitts, der im Kontext der Migrationsiteration betrachtet wird. Für die Softwarebausteine des Monolithen wird auf die Prämisse verwiesen, dass der Monolith anhand der *objektorientierten Programmierung* (OOP) implementiert wurde, wodurch auf die bekannten Elemente der *Pakete*, *Komponenten*, *Klassen*, *Funktionen* und *Attribute* zurückgegriffen werden kann (vgl. Abschnitt 1.6.1). Zudem besteht eine Synergie zwischen DDD und objektorientierten Softwaresystemen, was die Anwendbarkeit von DDD vereinfacht [HG+17]. Die Softwarebausteine werden aus dem Quellcode erhoben, weshalb für die nachfolgenden Migrationsaktivitäten die Sicht aus der Architekturebene der Bausteine (vgl. Abschnitt 2.2.1) eingenommen wird. Relevant sind sowohl die Softwarebausteine der *Makroarchitektur* (Pakete und Komponenten) als auch die der *Mikroarchitektur* (Klassen, Funktionen und Attribute). Der Umfang des Monolithenausschnitts ist somit durch die für die Migrationsiteration relevanten Makro- und Mikrobausteine des Monolithen definiert.

Die erste Migrationsaktivität greift die Anforderungsspezifikation sowie das Domänenmodell auf und bestimmt, welche Funktionalität, die bereits im Monolithen implementiert wurde, in dem Monolithenausschnitt aufgenommen wird. Grundlegend wird die Funktionalität aufgenommen, die entweder kohäsiv mit dem Migrationsauslöser verbunden oder derselben Subdomäne zugeordnet ist. Die Frage der Kohäsion wird anhand der Geschäftsdomäne beantwortet. Kollaborativ erarbeiten das Migrationsteam und die Domänenexperten über das *Knowledge Crunching* [Ev04], eine Praktik von DDD, zunächst einen *Feature-Baum* [YD+08]. Dieser Feature-Baum dient zunächst als Diskussionsgrundlage der kohäsiven Funktionalität. Sobald die Funktionalität abschließend bestimmt wurde, wird der

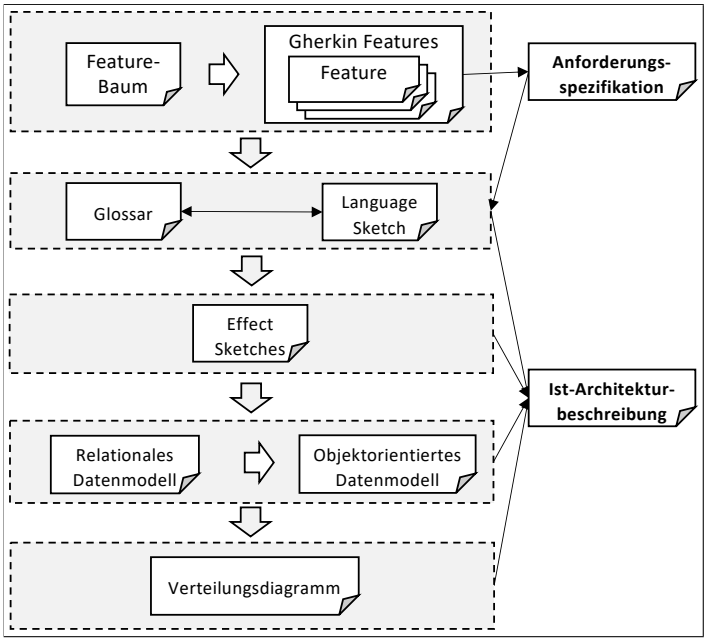

Abbildung 5.12: Erstellung der Migrationsartefakte zur Wiederherstellung der Ist-Architekturbeschreibung

Feature-Baum in formale *Gherkin Features* [Sm14] überführt und letztlich der *Anforderungsspezifikation* zugeordnet.

In der zweiten Migrationsaktivität wird die Etablierung einer *ubiquitären Sprache* (engl. ubiquitous language) thematisiert. Die ubiquitäre Sprache wurde von Evans [Ev04] als Konzept zur besseren Kommunikation zwischen Projektmitgliedern eingeführt. Diese Sprache umfasst alle relevanten Begrifflichkeiten, legt die Bedeutung fest und forciert die Verwendung dieser Begriffe im Kontext des Projekts. Um die Migrationsiteration effizienter durchführen zu können, werden die Begrifflichkeiten als *Glossar* [Ve13] und *Language Sketch* [AS+19] festgehalten. Die primäre Quelle der ubiquitären Sprache sind die Gherkin Features.

Die dritte Migrationsaktivität identifiziert ausgehend von den Funktionalitäten innerhalb der Gherkin Features die Softwarebausteine der Mikroarchitektur. Zurückgegriffen wird auf eine von Feathers [Fe05] vorgestellte Methodik der statischen Quellcodeanalyse, die *Nahtstellen* (engl. seams) zwischen Klassen offenlegt. Eine Nahtstelle entspricht dabei einer lokalen Schnittstelle zwischen den beiden Klassen, die gegebenenfalls bei der Extraktion des Monolithen in eine Web-Schnittstelle überführt werden muss. Die Nahtstellen werden in sogenannten *Effect Sketches* festgehalten.

Anschließend erfolgt durch die vierte Migrationsaktivität die Betrachtung der Datenhaltung. Ausgangspunkt zur Betrachtung der Datenhaltung sind die Softwarebausteine der Effect Sketches, die letztlich den Zugriff auf die Datenhaltung implementieren. Newman [Ne15, Ne19] zufolge ist gerade

die Datenhaltung für die Migration von monolithischen Softwaresystemen ein entscheidender Faktor, denn die Softwarebausteine des Monolithen stehen ebenfalls durch das Teilen von Daten in Abhängigkeit zueinander. Auch die Arbeiten von Balalaie et al. [BH+18] und Henry und Ridene [HR20] sehen die Datenhaltung als einen entscheidenden Faktor an. Die Abhängigkeiten werden in zwei formalen Modellen festgehalten. Zunächst wird das *relationale Datenbankschema* formal modelliert, aus dem dann anschließend ein *objektorientiertes Datenbankschema* abgeleitet wird.

Zum Abschluss der Wiederherstellung der konkreten Softwarearchitektur wird die physische Architektur des Monolithen im Kontext der festgestellten Softwarebausteine der Mikroarchitektur betrachtet. Erstellt wird ein formales *Verteilungsdiagramm* der *Unified Modeling Language* (UML).

5.4.1 Festlegung kohäsiver Funktionalität

Diese Migrationsaktivität befasst sich mit der Festlegung der kohäsiven Funktionalität, um den Umfang des Monolithenausschnitts bestimmen zu können. Die Erhebung dieser Funktionalität erfolgt in zwei Schritten: (1) *Diskussion anhand von Feature-Bäumen* und (2) *Spezifikation als Gherkin Features*. Ausgangspunkt zur Festlegung der kohäsiven Funktionalität sind die bereits bestehende *Anforderungsspezifikation* und die *Context Map*.

Der Begriff der *Kohäsion* für diese Forschungsarbeit baut auf der Definition von Vogel et al. [VA+11] auf: "*cohesion is a measure of how much a given building block is self-contained, semantically*". Für den Monolithenausschnitt wird die Frage der *semantischen Abgeschlossenheit* (engl. semantic self-containment) anhand der Geschäftsfähigkeiten beantwortet, die aus Sicht der Geschäftsdomäne miteinander verwandt sind. Ausschlaggebend für die Bestimmung, ob und inwiefern Geschäftsfähigkeiten miteinander verwandt sind, ist das Domänenwissen der Domänenexperten. Aber auch das Domänenwissen darüber, welches bereits in dem monolithischen Softwaresystem implementiert wurde, kann bei der Entscheidung unterstützen, denn die implementierte Funktionalität kann bereits im Quellcode gruppiert worden sein. Das prozessuale Vorgehen dieser Migrationsaktivität entspricht einem *hybriden Design Recovery-Ansatz* [DP09], denn durch die Domänenexperten wird ein *Top-Down-Ansatz* verfolgt, während durch das Migrationsteam und den Quellcode ein *Bottom-Up-Ansatz* verfolgt wird.

Die Frage, ob eine Funktionalität kohäsiv zu einer anderen Funktionalität steht, wird anhand eines zu bestimmenden zentralen Domänenobjekts beantwortet:

> **Zentrale Domänenobjekte** sind Domänenobjekte (vgl. Abschnitt 2.4.1), die im Kontext einer Subdomäne die Geschäftsfähigkeiten an sich binden, welche die Wertigkeit der Subdomäne für die Organisation bestimmen. Ein solches Domänenobjekt ist meist als Entität oder Aggregatswurzel zu verstehen. Eine Subdomäne verfügt meist über genau ein zentrales Domänenobjekt.

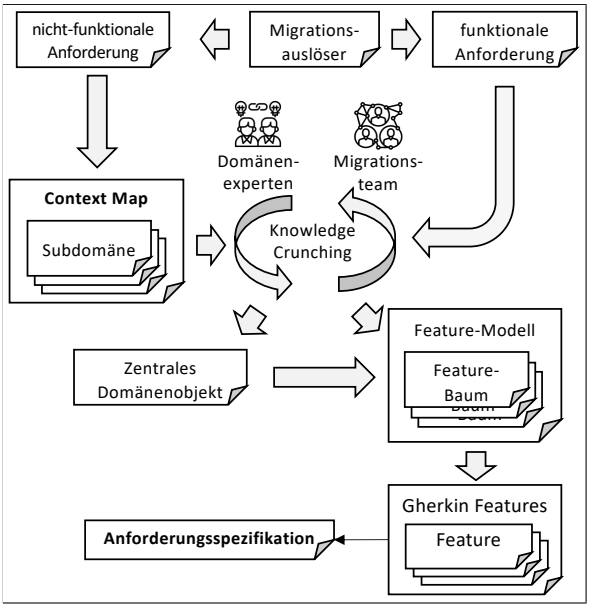

Abbildung 5.13: Ablauf zur Identifikation kohäsiver Funktionalität unter Einbezug des Migrationsauslösers

Identifiziert wird das zentrale Domänenobjekt über das von Evans [Ev04] vorgestellte *Knowledge Crunching* zwischen Migrationsteam und Domänenexperten. Der Typ des Migrationsauslösers entscheidet jedoch darüber, welche Informationsquelle im Knowledge Crunching verwendet wird. Betrachtet man als Migrationsauslöser eine funktionale Anforderung, erfolgt die Identifikation des zentralen Domänenobjekts über das bestehende *Gherkin Feature*, denn in diesen befinden sich bereits Domänenobjekte [HS+19]. Dagegen muss bei einer nicht-funktionalen Anforderung zunächst eine Subdomäne anhand ihrer Priorität bestimmt werden, bevor in Zusammenarbeit mit den Domänenexperten das zentrale Domänenobjekt bestimmt wird. Unterstützend kann bei einer funktionalen Anforderung das bestehende Gherkin Feature ebenfalls einer Subdomäne zugeordnet werden, was das Bestimmen des zentralen Domänenobjekts vereinfacht.

Das *Knowledge Crunching* wird weiterführend auch für die Identifikation der mit dem *zentralen Domänenobjekt* verbundenen Funktionalität angewandt. In den Diskussionen müssen die Domänenexperten und das Migrationsteam Verknüpfungen des zentralen Domänenobjekts in den Geschäftsfähigkeiten und dem Quellcode des Monolithen finden. Die Geschäftsfähigkeiten wurden bereits bei der Zerlegung der Geschäftsdomäne in Form von Domänenereignissen erhoben. Die Domänenereignisse referenzieren bereits auf ein Domänenobjekt, weswegen das Finden einer Verknüpfung meist trivial ist. Auch das Finden von Verknüpfungen im Quellcode gestaltet sich meist unproblematisch, denn gerade bei einem objektorientierten Softwaresystem ist meist eine zu dem zentralen Domänenobjekt äquivalente Klasse vorzufinden. Die Extraktion verfolgt zunächst eine leichtgewichtige Modellierung der kohäsiven Funktionalität durch *Feature-Bäume*, die sich als Grundlage für Diskussionen

mit den Domänenexperten eignen [YD+08]. Dass sich ein *Feature-Modell* zum Design Recovery eignet, zeigen Pashov und Riebisch [PR04], die eine Entität der Geschäftsdomäne als Wurzel des Baumes festlegen und davon abhängige Funktionalität zuordnet. Basierend auf diesem Ansatz werden die Feature-Bäume bei der Festlegung der kohäsiven Funktionalität erstellt und das zentrale Domänenobjekt als Wurzel des Baumes bestimmt.

Wurden die Feature-Bäume gemeinsam mit den Domänenexperten bestimmt, werden die Funktionalitäten in Gherkin Features überführt. Denn nach der Extraktion des Monolithenausschnitts müssen die Funktionalitäten wie gewohnt gewährleistet werden und durch die Testbarkeit der Gherkin Features kann dies verifiziert werden.

Ein beispielhafter Ablauf dieser Migrationsaktivität wird in Abbildung 5.14 anhand des DMS-Szenarios illustriert. Das zentrale Domänenobjekt "*File*" findet sich in der Funktionalität "*Upload a file*" wieder, weshalb diese Funktionalität als Blatt "*Upload*" dem Feature-Baum hinzugefügt wird. Alle Blätter des Feature-Baumes werden anschließend als Gherkin Features spezifiziert und in die Anforderungsspezifikation überführt.

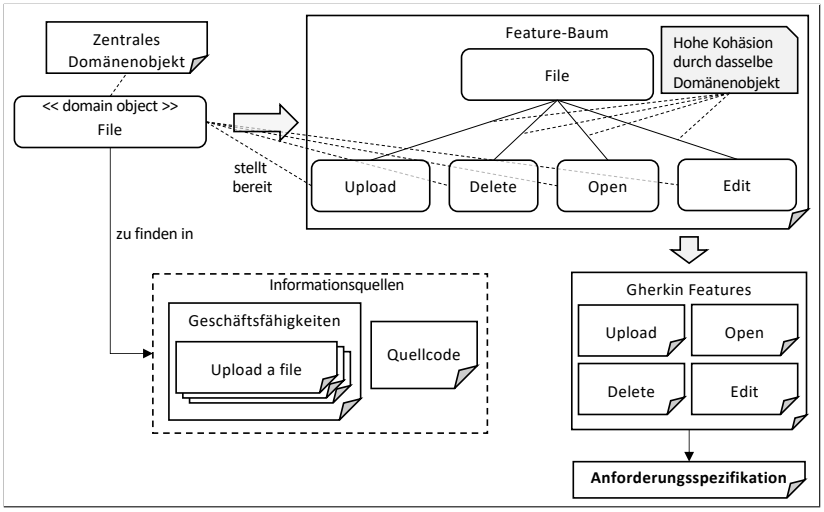

Abbildung 5.14: Modellierung der kohäsiven Funktionalität als Feature-Baum und Überführung in Gherkin Features

5.4.2 Etablierung der ubiquitären Sprache

Viele der Migrationsaktivitäten der Systematik des Migrationsrahmenwerks beruhen auf einer engen Zusammenarbeit zwischen dem Migrationsteam und anderen Interessenvertretern wie beispielsweise den Domänenexperten. Zur effizienten und effektiven Gestaltung der Diskussionen zwischen den Projektmitgliedern ist die Etablierung einer *ubiquitären Sprache* (engl. ubiquitous language) unabdingbar

[Ev04]. Missverständnisse, die während der Kommunikation durch mehrdeutige Begrifflichkeiten auftreten, gelten als Hauptgrund für das Scheitern von Softwareprojekten.

DDD [Ev04] führt zur Auflösung dieser Problematik das Konzept der ubiquitären Sprache ein. Teil dieser Sprache sind die Domänenobjekte, die im Kontext der Subdomänen und Funktionalitäten auftreten. So sind neben dem zentralen Domänenobjekt auch weitere Domänenobjekte für die ubiquitäre Sprache relevant. Die Domänenobjekte werden in dieser Migrationsaktivität als Begriffe aufgefasst, welche anhand eines *Glossars* mit einer Beschreibung versehen werden [Ve13]. Ergänzend zu dem Glossar verfolgen Abeck et al. [AS+19] die Modellierung der ubiquitären Sprache in Form eines *Language Sketchs*, um die Beziehungen der Begriffe grafisch darzustellen und dadurch ein einfacheres Verständnis über die ubiquitäre Sprache zu ermöglichen.

Der Geltungsbereich des Glossars und Language Sketchs beschränkt sich auf die aktuelle Migrationsiteration. Der Grund hierfür ist die auf die Extraktionsphase folgende Modernisierungsphase, durch die eine Überarbeitung der Anforderungen und damit auch der Begrifflichkeiten möglich ist. Nach der Modernisierungsphase ist das Glossar und der Language Sketch migrationsiterationsübergreifend gültig.

5.4.3 Identifikation von Nahtstellen

In dieser Migrationsaktivität wird die Identifikation der von der Anforderungsspezifikation betroffenen Softwarebausteine der logischen Mikroarchitektur vorgesehen. Die identifizierten Softwarebausteine entsprechen dem Monolithenausschnitt und werden bei der Extraktion aus dem Monolithen ausgegliedert. Die Softwarebausteine der logischen Mikroarchitektur entsprechen den Klassen der objektorientierten Programmierung.

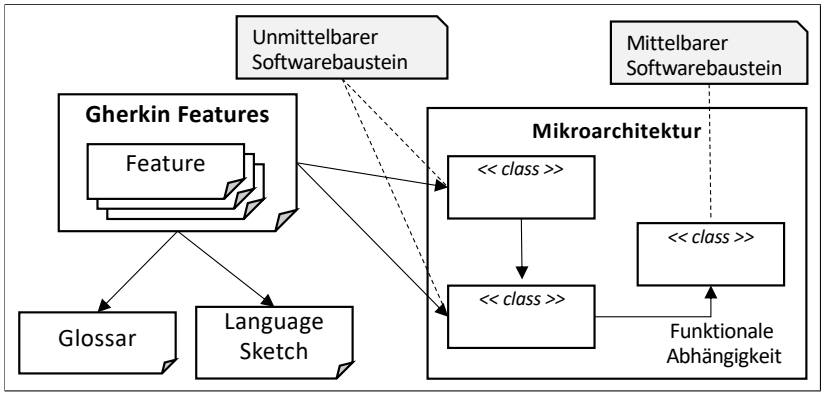

Abbildung 5.15: Typisierung der Softwarebausteine des Monolithenausschnitts

Diejenigen Gherkin Features, die bereits im Monolithen implementiert wurden, werden durch eine oder mehrere Klassen abgedeckt. Die Zuordnung eines Gherkin Features auf die Klassen bedarf ein

detailliertes Wissen über den Quellcode. Durch diese stark technische Sicht auf das monolithische Softwaresystem ist eine enge Zusammenarbeit mit den Softwareentwicklern, welche nicht unbedingt zum Migrationsteam gehören, notwendig. Wissen über die bereits implementierte Funktionalität sollte aus Gründen des Aufwands direkt von den verantwortlichen Softwareentwicklern extrahiert werden. Sollte das Wissen nicht mehr in der Organisation vorhanden sein, bedarf es einer Einarbeitung des Migrationsteams. Bei der Zuordnung von Gherkin Features zu den Klassen durch die darin explizit abgebildeten Domänenobjekte, die sich in den Klassen wiederfinden lassen, unterstützt die ubiquitäre Sprache.

Aus der Zuordnung aller Gherkin Features resultieren die Klassen, welche unmittelbar mit der Funktionalität des Monolithenausschnitts verbunden sind. Damit diese aus dem Monolithen extrahiert werden können, müssen die Abhängigkeiten zu anderen Klassen im Quellcode offengelegt werden. Die Abbildung 5.15 veranschaulicht die zwei Arten der Klassen. Zudem wird zur besseren Unterscheidung der beiden Arten von Klassen folgende Typisierung eingeführt:

Unmittelbare Softwarebausteine stellen beispielsweise Klassen dar, die unmittelbar die Funktionalität der Gherkin Features betreffen und damit auch direkt dem Monolithenausschnitt zugeordnet sind

Mittelbare Softwarebausteine stellen beispielsweise Klassen dar, die eine Abhängigkeit zu einem unmittelbaren Softwarebaustein besitzen, aber nicht die Funktionalität der Gherkin Features betreffen

O'Brien et al. [OS+05] definiert als relevante Abhängigkeit die sogenannte *funktionale Abhängigkeit* (engl. functional dependency). Zum Finden dieser funktionalen Abhängigkeiten stellt Feathers [Fe05] das Konzept der *Nahtstellen* (engl. seams) vor, die im Rahmen einer statischen Quellcodeanalyse bestimmt werden müssen. Eine Nahtstelle beschreibt dabei *"a place where you can alter behavior in your program without editing in that place"*. Eine solche *Stelle* entspricht einer funktionalen Abhängigkeit, die jedoch, wie Feathers beschreibt, in verschiedenen Ausprägungen vorliegen kann. Relevant für ein objektorientiertes Softwaresystem sind die sogenannten *Objektnahtstellen* (engl. object seams), die eine funktionale Abhängigkeit zweier Klassen durch das Verwenden von Methoden oder Attributen der abhängigen Klasse beschreiben. Konkret entsprechen die Objektnahtstellen [DR+00]: (1) *Vererbungen*, (2) *Übergabeparametern in Methodensignaturen* und (3) *Konstruktoraufrufen*. Abhängig von der vorzufindenden Nahtstelle müssen bei der nachgelagerten Extraktion des Monolithenausschnitts Änderungen an der Syntax in den einzelnen Klassen vorgenommen werden, sodass die lokale Schnittstelle zu einer webbasierten Schnittstelle migriert werden kann.

Die Identifikation der Nahtstellen erfolgt ausgehend von den unmittelbaren Softwarebausteinen. Für jede dieser Klassen werden sowohl die Nahtstellen zu unmittelbaren als auch mittelbaren Softwarebausteinen festgehalten. Die formale Modellierung der Nahtstellen erfolgt anhand der von Feathers [Fe05]

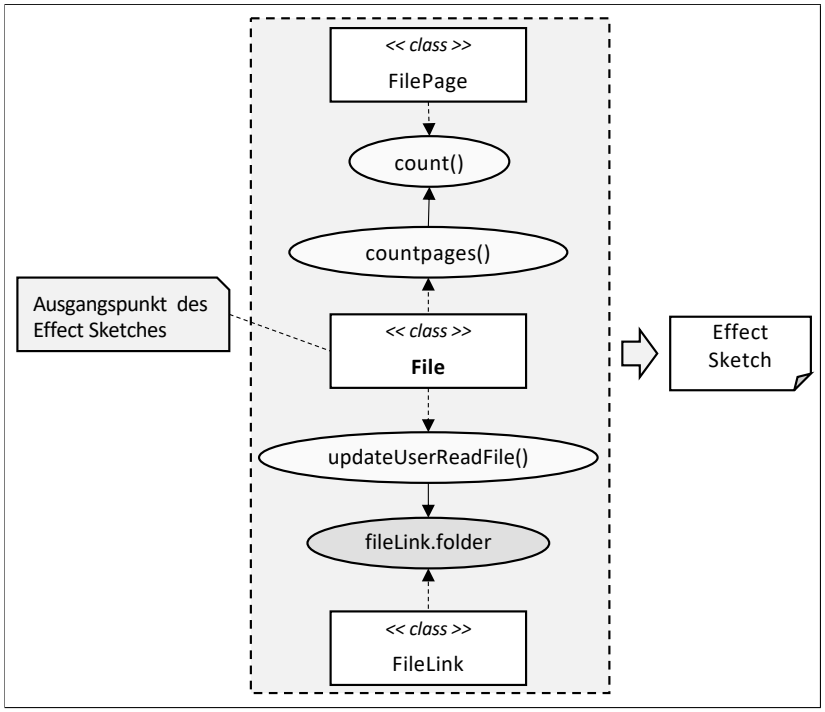

Abbildung 5.16: Modellierung eines Effect Sketches

konzipierten *Effect Sketches*. Ein Effect Sketch ist ein leichtgewichtiges Modell, das eine Klasse in den Fokus stellt und ausgehend davon die Nahtstellen in Form von Methoden- und Attributzugriffen darstellt. Somit wird für jeden unmittelbaren Softwarebaustein ein Effect Sketch modelliert. Alternativ kann die formale Modellierung der unmittelbaren und mittelbaren Softwarebausteine auch über ein *Klassendiagramm* der UML [BR+05] erfolgen. Doch der Vorteil des Effect Skechts gegenüber dem Klassendiagramm ist, dass die Nahtstellen feingranular auf Ebene der Methoden und Attribute dargestellt werden, sodass das Migrationsteam bei der Extraktion die Nahtstelle eindeutig im Quellcode referenzieren kann.

In Abbildung 5.16 wird ein Effect Sketch anhand des DMS-Szenarios dargestellt. Der Ausgangspunkt des Effect Sketches ist die Klasse "*File*", die über zwei Nahtstellen mit zwei weiteren Klassen verbunden ist. Die Nahtstelle zur Klasse "*FilePage*" beruht auf dem Aufruf der Methode "*count*", der in der Methode "*countPages()*" stattfindet. Eine weitere Nahtstelle besteht durch den direkten Zugriff auf Attribute des Objekts der Klasse "*FileLink*" in der Methode "*updateUserReadFile*". Zwischen diesen Klassen besteht somit eine Kopplung, die während der Extraktion berücksichtigt werden muss.

5.4.4 Modellierung der Datenhaltung

Eine weitere Abhängigkeit, die zwischen Softwarebausteinen besteht, ist laut O'Brien et al. [OS+05] die *Datenabhängigkeit* (engl. data dependency), welche den Austausch von Informationen zweier Softwarebausteine über Daten der Datenhaltung beschreibt. In der Regel verweist die Datenhaltung auf eine Datenbank, die im Kontext von monolithischen Softwaresystemen von allen Softwarebausteinen verwendet wird. An dieser Stelle spricht man von einer *geteilten Datenbank* (vgl. Abschnitt 2.2.3). Das Ziel dieser Migrationsaktivität ist es, die von dem Monolithenausschnitt betroffene Datenhaltung innerhalb dieser *geteilten Datenbank* offenzulegen und die Datenabhängigkeit der Softwarebausteine zu bestimmen.

Eine Datenbank legt Daten innerhalb eines *Datenbankschemas* ab, das wiederum durch *Datenbanktabellen* beschrieben ist [AH+95]. Die Struktur der Datenbanktabellen wird anhand von *Entitäten* bestimmt, welche mit dem Konzept der Domänenobjekte von DDD korrespondieren [Ev04].

Für die Softwarebausteine des Monolithenausschnitts müssen genau diejenigen Datenbanktabellen innerhalb des Datenbankschemas offengelegt werden, die zur Persistierung der Daten dienen. Diese identifizierten Datenbanktabellen werden dann im Rahmen der Extraktion des Monolithenausschnitts entsprechend der Soll-Architektur angepasst. Richardson [Ri19b] stellt verschiedene Datenbankmuster für Microservice-Architekturen vor, wobei Newman [Ne15] der Auffassung ist, dass pro Microservice eine eigene Datenbank verwendet werden sollte. Dies entspricht dem *Datebase-per-Service*-Muster. Um die Datenbank entsprechend auf dieses Muster vorzubereiten, müssten bei der Extraktion des Monolithen die identifizierten Datenbanktabellen aus der geteilten Datenbank ausgegliedert werden. Eine Besonderheit stellen dabei Datenbanktabellen dar, die sowohl von unmittelbaren als auch mittelbaren Softwarebausteinen verwendet werden. Diese Tabelle musst repliziert werden, sodass diese sowohl in der geteilten Datenbank des Monolithen als auch in der Datenbank des Monolithenausschnitts vorhanden ist. Die Daten innerhalb der Datenbanktabellen müssen bei einer Manipulation entsprechend synchronisiert werden (vgl. Abschnitt 4.4.2).

Der Ablauf dieser Migrationsaktivität orientiert sich an dem von Prückler und Schrefl [PS18] vorgestellten *Database Reverse Engineering*-Vorgehen. Das Vorgehen sieht die formale Modellierung des bestehenden Datenbankschemas in zwei Datenmodellen vor: (1) *relationales Datenmodell* und (2) *objektorientiertes Datenmodell*. Das relationale Datenmodell legt die technologieabhängige Struktur des Datenbankschemas anhand von Datenbanktabellen offen und stellt diese in Beziehung zueinander dar. Dagegen beschreibt das objektorientierte Datenmodell technologieagnostisch die Abhängigkeiten der Entitäten, die sich innerhalb der Datenbanktabellen befinden können. Eine Datenbanktabelle kann dabei mehrere Entitäten umfassen, was durch das relationale Datenmodell nicht offengelegt wird.

Bei der Extraktion des Monolithenausschnitts ist zunächst das relationale Datenmodell entscheidend. Extrahiert werden nur Datenbanktabellen und nicht einzelne Entitäten innerhalb dieser Tabellen. Dies

würde für die Extraktion des Monolithenausschnitts eine umfassendere Anpassung der Datenhaltung bedingen, was jedoch erst im Rahmen der Modernisierungsphase erfolgen soll. Zudem sollte bei der Modifikation der Datenhaltung auch eine Modernisierung auf neue Technologien angestrebt werden [Sn07]. Das objektorientierte Datenmodell wird damit erst in der Modernisierungsphase herangezogen. Es bietet sich dennoch an, das objektorientierte Datenmodell bereits in der Extraktionsphase zu erstellen, da das Migrationsteam sich zu diesem Zeitpunkt intensiv mit der Datenhaltung befasst hat und eine erneute Einarbeitung vermieden werden kann.

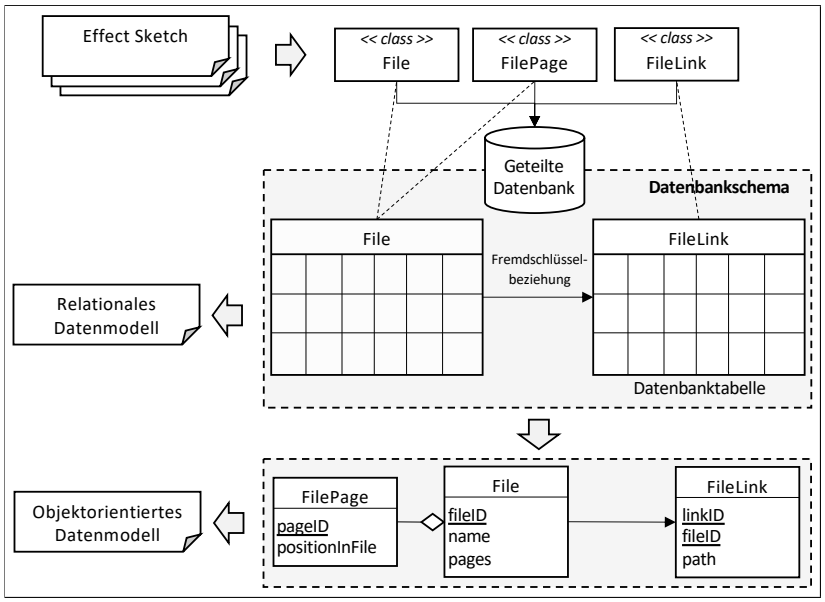

Abbildung 5.17: Modellierung der Datenhaltung des Monolithenausschnitts

Damit gezielt das Datenbankschema des Monolithenausschnitts betrachtet wird, werden alle Softwarebausteine der *Effect Sketches* herangezogen. Relevante Teile der Softwarebausteine sind die Quellcodepassagen, welche einen direkten Zugriff auf die Datenbanktabellen implementieren. Softwarebausteine, welche auf dieselbe Datenbanktabelle zugreifen, stehen in einer Datenabhängigkeit zueinander, was eine sehr hohe Kopplung der beiden Softwarebausteine zur Folge hat [OS+05]. Losgelöst von den Datenabhängigkeiten der Softwarebausteine gilt es auch die *Inklusionsabhängigkeiten* der Datenbanktabellen offenzulegen. Eine Inklusionsabhängigkeit beschreibt dabei eine *Fremdschlüsselbeziehung* zwischen Datenbanktabellen. Die formale Modellierung des relationalen Datenmodells erfolgt über ein Klassendiagramm der UML [VB+17].

Liegt das relationale Datenmodell vor, können aus diesem die Entitäten für das objektorientierte Datenmodell abgeleitet werden [PS18]. Das Finden der Entitäten kann auch hier durch die ubiquitäre Sprache unterstützt werden, denn Evans [Ev04] sieht eine Verbindung zwischen den Entitäten des objektorientierten Datenbankschemas und den Domänenobjekten. Eine Entität entspricht in der Regel einem Domänenobjekt. Zur formalen Modellierung des objektorientierten Datenmodells eignet sich

das *Entitäten-Relationen-Diagramm* (ER-Diagramm) [YL+08].

Abbildung 5.17 zeigt beispielhaft den Ablauf zur Modellierung der Datenhaltung anhand des DMS-Szenarios. Die Klassen der Effect Sketches nutzen die Datenbanktabellen "*File*" und "*FileLink*". Besonders hervorzuheben ist, dass sowohl die Klasse "*File*" als auch "*FilePage*" die Tabelle "*File*" verwenden. An dieser Stelle kommt der Vorteil des objektorientierten Datenmodells zum Tragen, denn dort wird die tabelleninterne Abhängigkeit zwischen "*File*" und "*FilePage*" explizit modelliert.

5.4.5 Modellierung der logischen und physischen Makroarchitektur

Bislang wurde durch die Extraktion die Mikroarchitektur des Monolithenausschnitts wiederhergestellt. Nun gilt es, den Monolithenausschnitt in der Makroarchitektur auf den Architekturebenen der Bausteine und Systeme (vgl. Abschnitt 2.2.1) zu betrachten. Für die Makroarchitektur werden neben den Softwarebausteinen auch die Hardwarebausteine relevant. Mit der Betrachtung der Makroarchitektur auf der Architekturebene der Systeme werden auch Abhängigkeiten zu *Drittsoftwaresystemen* offengelegt. Damit ergibt sich das Ziel dieser Migrationsaktivität, die von dem Monolithenausschnitt betroffenen Software- und Hardwarebausteine zu identifizieren, Abhängigkeiten zu Drittsoftwaresystemen offenzulegen und diese formal zu modellieren. Ein beispielhaftes Ergebnis dieser Migrationsaktivität kann der Abbildung 5.18 entnommen werden.

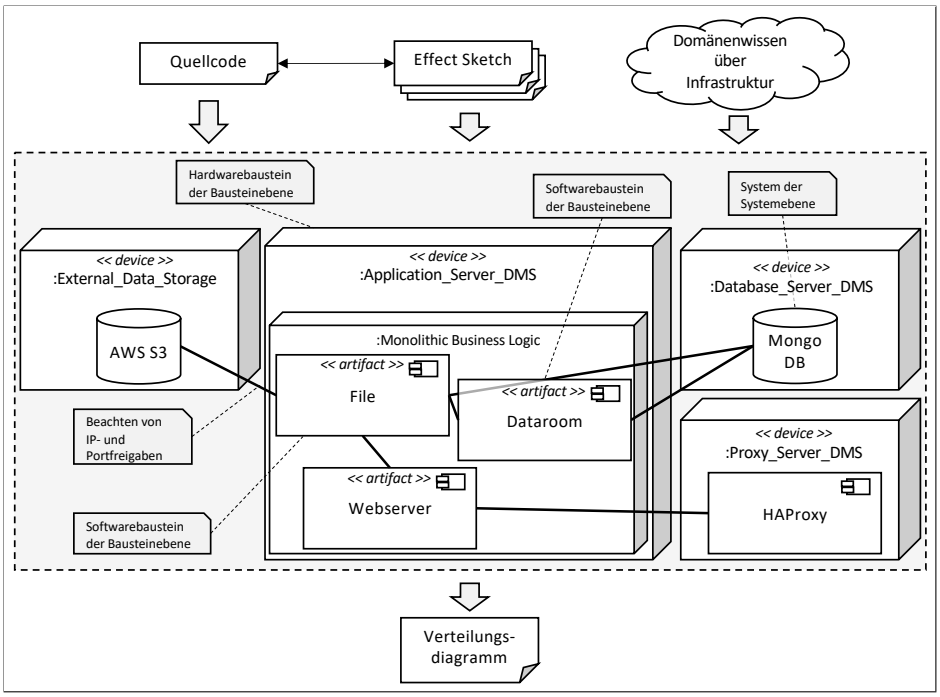

Abbildung 5.18: Formale Modellierung der Makroarchitektur

Im ersten Schritt dieser Migrationsaktivität werden die Softwarebausteine der Mikroarchitektur in Softwarebausteine der Makroarchitektur gruppiert. Verwendet wird das Konzept der *Komponente*, die einer Gruppierung von Klassen entspricht [RH06]. Welche Klassen des Monolithenausschnitts gruppiert werden, kann anhand des Quellcodes bestimmt werden. So wurden bespielsweise im Kontext des DMS-Szenario die Komponenten "*File*", "*Dataroom*" und "*Webserver*" identifiziert (Abbildung 5.18).

Weitere Softwarebausteine der Makroarchitektur sind Drittsoftwaresysteme. Ein Drittsoftwaresystem wird von dem monolithischen Softwaresystem zur Laufzeit benötigt; gleichzeitig kann dies bedeuten, dass auch der Monolithenausschnitt diese Drittsoftwaresysteme benötigt. Eine durch die *Twelve-Factor App* [Wi17] etablierte Bezeichnung für solche Drittsoftwaresysteme sind *Backing Services*, die auch im weiteren Verlauf dieser Forschungsarbeit verwendet wird. Die Backing Services werden als Softwarebausteine der Architekturebene der Systeme aufgefasst. Beispiele für solche Backing Services sind, neben der bereits betrachteten Datenbank, ein *binärer Objekt-Speicher* wie beispielsweise der Amazon Web Service *Simple Storage Service* (AWS S3) [Ama-Sim] oder ein *Proxy* wie der *HAProxy* [HAP-HAP] zur Verwaltung des Ingress-Netzwerks (Abbildung 5.18). Auch hier enthält der Quellcode wichtige Informationen über die Abhängigkeiten zu den Backing Services. Weiterhin müssen Systemadministratoren, welche die Backing Services betreuen, integriert werden.

Anschließend können zusätzlich zu den Softwarebausteinen, die im Verbund die logische Makroarchitektur beschreiben, die entsprechenden Hardwarebausteine identifiziert werden, auf denen diese Softwarebausteine betrieben werden. Somit wird zu diesem Zeitpunkt die *physische Makroarchitektur* betrachtet, die anhand der tatsächlichen Verteilung des monolithischen Softwaresystems abgeleitet werden kann. Der Quellcode liefert keine Informationen über die Laufzeitumgebung des Monolithen. Geeignete Quellen sind erneut die Systemadministratoren, welche die Hardwarebausteine betreuten und die *DevOps-Ingenieure*, die laut Bass [Ba17] die Softwarebausteine auf die Hardwarebausteine ausrollen. Zwar wird der Begriff des DevOps-Ingenieurs nicht in jeder Organisation verwendet, doch in dem monolithischen Entwicklungsteam lassen sich Verantwortliche für die Bereitstellung des Monolithen finden. Durch die Verknüpfung der Softwarebausteine mit den Hardwarebausteinen werden Abhängigkeiten offengelegt, welche über verschiedene Hardwarebausteine hinweg bestehen. Bei der Extraktion sind diese Abhängigkeiten insbesondere für die Verteilung des Monolithenausschnitts als eigenständigem Softwarebaustein auf einem neuen Hardwarebaustein von hoher Relevanz. Die Kommunikation erfolgt über die Verwendung von Web-Protokollen wie beispielsweise dem *Hypertext Transfer Protocol* (HTTP). Meist sind diese Kommunikationswege durch Restriktionen wie *Whitelisting* der *Internet Protocol-Adresse* (IP-Adresse) oder *Transmission Control Protocol-Ports* (TCP-Ports) gesichert [LC+16]. Die Kommunikation zwischen Hardwarebausteinen kann auf verschiedene Ebenen der *TCP/IP-Modells* [Ka17] beschränkt sein.

Grundlegend können die gesammelten Informationen dieser Migrationsaktivität der *physischen Sicht* des *4+1 Sichtenmodells* (vgl. Abschnitt 2.2.2) zugeordnet werden. Eine geeignete formale Form der

Modellierung stellen Booch et al. [BR+05] und Gomaa [Go11] vor. Beide sehen zur Abbildung der physischen Sicht das Verteilungsdiagramm (engl. deployment diagram) der UML als geeignet an, denn in diesem formalen Modell können sowohl die logische als auch physische Makroarchitektur beider Architekturebenen modelliert werden. Zudem eignet sich die Verwendung des Verteilungsdiagramms aufgrund der hohen Akzeptanz der UML bei Softwareentwicklern [RN+13]. Somit wird der Monolithenausschnitt abschließend für das *Design Recovery* in einem Verteilungsdiagramm modelliert.

5.5 Planung der Soll-Softwarearchitektur

Mit der Erreichung des Submeilensteins $M_{1.3}$ *Rekonstruierte Ist − Architekturbeschreibung* wurden alle notwendige Informationen der *logischen* und *physischen Ist-Architektur* erhoben, die den Monolithenausschnitt betreffen. Nun wird ausgehend von der Ist-Architekturbeschreibung mit ihren formalen Modellen die Soll-Softwarearchitektur abgeleitet. Die Soll-Softwarearchitektur zeigt zum einen die letztendlichen Softwarebausteine, die in den Monolithenausschnitt überführt werden und zum anderen auch die Verteilung dieser Softwarebausteine auf bestehende oder auch neue Hardwarebausteine. Bedingt durch die neu gewonnene Verteilbarkeit des Monolithenausschnitts, müssen die lokalen Schnittstellen (engl. Application Programming Interfaces, APIs) in Web-APIs überführt werden.

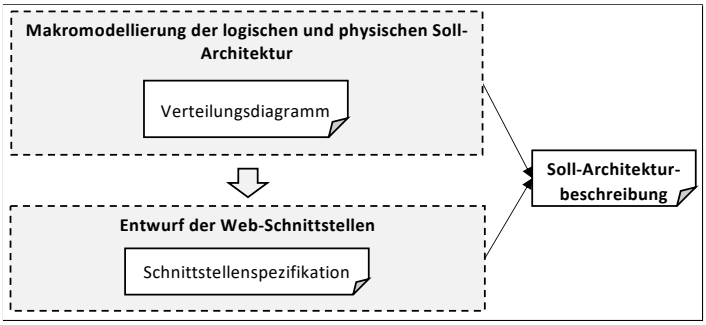

Abbildung 5.19: Migrationsaktivitäten und -artefakte für die Planung der Soll-Softwarearchitektur

Die Planung der Soll-Softwarearchitektur erfolgt im Rahmen zweier Migrationsaktivitäten (Abbildung 5.19). Zunächst erfolgt eine Makromodellierung der logischen und physischen Soll-Softwarearchitektur. Es wird von einer Makromodellierung gesprochen, da die Softwarebausteine der Makroarchitektur wie beispielsweise Komponenten verwendet werden. Das Model gibt, neben den Softwarebausteinen des Monolithenausschnitts, auch einen Überblick über die Verteilung auf die Hardwarebausteine. Ob der Monolithenausschnitt auf einem eigenständigen Hardwarebaustein betrieben wird, hängt von der Präferenz des Migrationsteams ab.

Aus der Modellierung der Soll-Softwarearchitektur folgt eine explizite Darstellung der APIs zwischen den Softwarebausteinen, die gegebenenfalls sogar über verschiedene Hardwarebausteine hinweg stattfindet. In der zweiten Migrationsaktivität erfolgt die Planung dieser Web-APIs zwischen den Softwarebausteinen des Monolithenausschnitts und der weiteren Subsysteme wie beispielsweise der Monolithen selbst. Wichtig festzuhalten ist, dass nicht nur APIs zu dem Monolithen betrachtet werden müssen, sondern auch die zu weiteren Subsystemen. Denn nach der ersten Migrationsiteration eines Migrationsvorhabens besteht das betrachtete Softwaresystem aus dem Subsystem des Monolithen und verschiedenen Microservices. Wird nun ein weiterer Monolithenausschnitt aus dem Monolithen extrahiert, können die Softwarebausteine dieses Monolithenausschnitts bereits Abhängigkeiten zu den Microservices besitzen; diese liegen zwar dann bereits als Web-API vor, sollten aber dennoch bei der Planung neuer Web-APIs berücksichtigt werden. Die Planung der Web-APIs erfolgt anhand drei möglicher Architekturstile: (1) *Representational State Transfer* (REST), (2) *gRemote Procedure Call* (gRPC) und (3) *ereignisgetriebene Architektur* (engl. event-driven architecture). Die Abwägung, welcher der drei Architekturstile verwendet wird, basiert auf den von Indrasiri und Kuruppu [IK20] und Bellemare [Be20] festgelegten Charakteristiken der einzelnen Architekturstile. Zu jedem der Architekturstile wird zudem eine geeignete formale Form zur Spezifikation vorgestellt.

5.5.1 Makromodellierung der logischen und physischen Soll-Architektur

Bislang wurde durch die Extraktion die konkrete Softwarearchitektur des Monolithen im Kontext des Monolithenausschnitts als eine *Ist-Architekturbeschreibung* wiederhergestellt. Nun gilt es, in dieser Migrationsaktivität aus der konkreten Softwarearchitektur einen detaillierten Entwurf der Soll-Softwarearchitektur abzuleiten, in dem der Monolithenausschnitt als eigenständiges Subsystem des Softwaresystems dargestellt wird. Die Soll-Softwarearchitektur wird ebenfalls, wie die Makroarchitektur der konkreten Softwarearchitektur, in einem Verteilungsdiagramm der UML formal modelliert.

Zum Entwurf der Soll-Softwarearchitektur wird zunächst die *Ist-Architekturbeschreibung* herangezogen. Alle von den *Gherkin Features* unmittelbar betroffenen Softwarebausteine werden dem Monolithenausschnitt zugeordnet. Die mittelbar betroffenen Softwarebausteine verbleiben in dem Monolithen, wobei die Nahtstellen zu den Softwarebausteinen des Monolithenausschnitts in Web-Schnittstellen überführt werden müssen. Der Entwurf der Web-Schnittstellen erfolgt im Rahmen der darauffolgenden Migrationsaktivität (vgl. Abschnitt 5.5.2). Neben den Softwarebausteinen der Ist-Architekturbeschreibung muss das Migrationsteam noch weitere Softwarebausteine, die innerhalb des Monolithen existieren, zur Gewährleistung von Querschnittsfunktionen wie beispielsweise einem *Webserver* oder *Logger* dem Entwurf hinzufügen. Solche Querschnittsfunktionen müssen auch weiterhin im Monolithen bestehen, weshalb diese Softwarebausteine zu replizieren sind.

Auch die *Backing Services* müssen in dem Entwurf der Soll-Softwarearchitektur auf den Monolithenausschnitt angepasst werden. So muss beispielsweise beim Auflösen der geteilten Datenbank eine dedizierte Datenbank für den Monolithenausschnitt bereitgestellt werden, welche die identifizierten Datenbanktabellen beinhaltet. Datenbanktabellen, die sowohl in der Datenbank des Monolithen als auch in der des Monolithenausschnitts vorhanden sein müssen, sollten explizit gekennzeichnet werden, um die Notwendigkeit einer Synchronisation zwischen den Datenbanken zu visualisieren.

Wie der Monolithenausschnitt betrieben wird, ist ebenfalls eine Fragestellung, die das Migrationsteam im Rahmen dieser Migrationsaktivität beantworten muss. Dabei besteht die Möglichkeit, dass der Monolithenausschnitt auf demselben Hardwarebaustein wie der Monolith betrieben wird. Dies reduziert den Aufwand die notwendigen Sicherheitsmaßnahmen wie das von Loui et al. [LC+16] vorgestellte *Whitelisting* umzusetzen. Eine weitere Möglichkeit ist jedoch auch, den Monolithenausschnitt auf einem eigenen Hardwarebaustein zur Verfügung zu stellen. Das führt dazu, dass die von Loui et al. vorgestellten Sicherheitsmaßnahmen vorgesehen werden müssen.

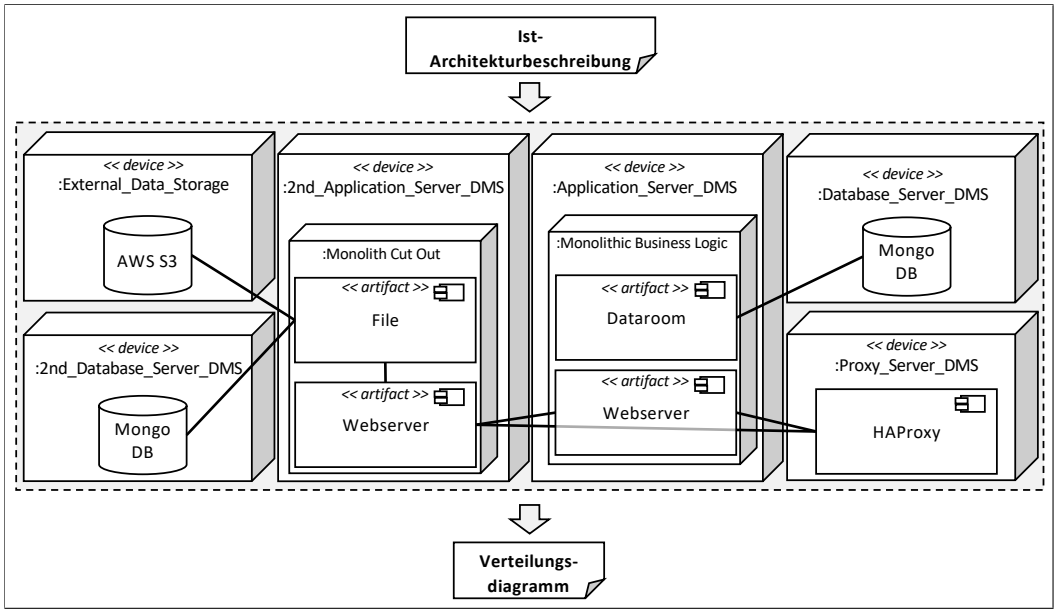

Abbildung 5.20: Entwurf des Monolithenausschnitts

Ein beispielhafter Entwurf der Soll-Softwarearchitektur anhand des DMS-Szenarios kann Abbildung 5.20 entnommen werden. Für das DMS-Szenario wurde ein eigener Hardwarebaustein für den Monolithenausschnitt vorgesehen. Ausgehend von der Ist-Architekturbeschreibung wurde die Komponente "*File*" mit den Klassen "*File*" und "*FilePage*" dem Monolithenausschnitt zugeordnet. Zudem wurde der Webserver repliziert. Die Datenhaltung erfolgt über eine eigene Datenbank, die auf einem eigenen Hardwarebaustein betrieben wird.

5.5.2 Entwurf der Web-Schnittstellen

Durch die Ausgliederung des Monolithenausschnitts entsteht der Bedarf, die funktionalen Abhängigkeiten und Datenabhängigkeiten auf eine Verteilbarkeit in der physichen Architektur anzupassen. Dies erfolgt durch die Wandlung der lokalen APIs in Web-APIs. Das Ziel dieser Migrationsaktivität ist der Entwurf dieser Web-APIs. Hierfür werden auf bestehende Architekturstile und Entwurfsmethoden zurückgegriffen.

Für die Kommunikation über Web-APIs werden drei Architekturstile etabliert: (1) *REST*, (2) *gRPC* und (3) *ereignisgetriebene Architektur* Die drei Stile beruhen auf unterschiedlichen Protokollen und verfolgen zudem unterschiedliche Kommunikationsmuster, deren Eignung je nach Anwendungsfall variiert [SS+19]. Ist Performance ein Faktor bei der Kommunikation zwischen zweier Softwarebausteinen, sollte auf gRPC gesetzt werden. Soll eine Web-API an externe Konsumenten veröffentlicht werden, bietet sich REST durch eine einfache Strukturierung in Ressourcen an. Eine detaillierte Abwägung der jeweiligen Architekturstile zeigen Indrasiri und Kuruppu [IK20] und Bellemare [Be20] auf.

Für die Architekturstile existieren bereits verschiedene Ansätze zum Entwurf der jeweiligen Web-API. Giessler [Gi18] stellt einen domänengetriebenen Ansatz zum Entwurf von REST-Schnittstellen vor. Hierfür werden *Domänenmodelle* mittels DDD modelliert und systematisch *REST-Ressourcen* anhand der Domänenobjekte abgeleitet. Durch die Orientierung dieses Ansatzes an den Domänenobjekten eignet er sich besonders im Kontext der Systematik dieser Forschungsarbeit. Einen weitereren domänengetriebenen Ansatz zum Entwurf der Web-APIs einer ereignisgetriebenen Architektur stellen Millet und Tune [MT15] vor. Die Autoren spezifizieren den Austausch von Informationen über *Domänenereignisse* (engl. domain events), die Zustandsänderungen an Domänenobjekten offenbaren. Unterstützend für den Entwurf der Domänenereignisse können die bereits in Abschnitt 5.3.1 durch das *Event Storming* identifizierten Domänenereignisse herangezogen werden. In Verbindung mit den Domänenereignissen stellt weiterführend Richardson [Ri19b] das *Saga*-Muster vor, das einen Geschäftsablauf anhand einer Sequenz an Domänenereignissen spezifiziert. Für den Entwurf einer gRPC-Schnittstelle existieren zu dem Zeitpunkt der Forschungsarbeit noch keine veröffentlichten systematischen Ansätze. Aber auch für diesen Architekturstil ist es möglich einen domänengetriebenen Ansatz zu verfolgen und Domänenobjekte als *gRPC-Services* zu definieren.

Nachdem die möglichen Architekturstile der Web-APIs aufgezeigt wurden, wird nun die Migrationsaktivität weiterführend spezifiziert. Weiterhin offen sind ein sequentielles Vorgehen bei dem Entwurf der Abhängigkeiten und die Strategien zum Auflösen der vorzufindenden Abhängigkeiten.

Als sequentielle Reihenfolge wird festgelegt, dass zunächst die Datenabhängigkeiten entworfen werden. Anschließend folgen die funktionalen Abhängigkeiten. Es ergibt sich aus der gewählten Reihenfolge keinen entscheidenden Vorteil bei dem Entwurf der Web-APIs. Es besteht die Möglichkeit,

dass sich aus dem *objektorientierten Datenmodell* (vgl. Abschnitt 5.4.4) die für die Web-API relevanten Domänenobjekte leichter ableiten lassen. Newman [Ne15] sieht zudem die Datenabhängigkeiten als zentrale Herausforderung bei der Extraktion eines Microservices aus einem Monolithen. Aus dieser zentralen Position der Datenabhängigkeiten bietet es sich daher an, diese vor den funktionalen Abhängigkeiten zu betrachten.

Ausgehend von dem Architekturstil eignen sich unterschiedliche Modelle zur Formalisierung des Entwurfs. Eine REST-API wird durch die *OpenAPI-Spezifikation* [Ope-Ope] in *Swagger* [Sma-Swa] festgehalten. Für gRPC wird auf das sowieso zu erstellende *ProtoBuffer* [IK20] zurückgegriffen. Die für die ereignisgetriebene Architektur entworfenen *Sagen* können über ein UML-Klassendigramm modelliert werden. Die Klassen bilden dabei die Domänenereignisse ab. Die Attribute einer Klasse spiegeln die Informationen wieder, die das Domänenereignis beinhaltet. Eine weitere Möglichkeit stellen Hippchen et al. [HS+19] vor. Ausgehend von dem Komponentendiagramm der UML führen die Autoren ein UML-Profil zur Modellierung einer *Context Choreographie* ein. Die Context Choreographie ist zwar im Kontext der Microservice-Architekturen angesiedelt, doch die Anwendung auf die Kommunikation zwischen Monolithenausschnitt und Monolithen ist ebenfalls möglich.

5.6 Zusammenfassung

Das Ziel dieses Kapitels war es, die *Extraktionsphase der Systematik* des Migrationsrahmenwerks detailliert zu entwerfen und damit die Problemstellungen P2 und P3 der iterativen Migration (vgl. Abschnitt 1.4) aufzulösen. Der detaillierte Entwurf der Extraktionsphase entspricht dem zweiten Forschungsbeitrag dieser Arbeit (vgl. Abschnitt 1.5.3). Vorgestellt wurde eine Extraktionssystematik, die aufbauend auf sequentiell ausgeführten konkreten Migrationsaktivitäten einen Monolithenausschnitt identifiziert und dokumentiert.

Die Grundlage für den detaillierten Entwurf der Extraktionsphase stellt der in Kapitel 4 vorgestellte Gesamtentwurf der Systematik des Migrationsrahmenwerks dar. Im Kontext der dort vorgestellten Extraktionsphase werden bereits vier abstrakte Migrationsaktivitäten beschrieben. Der Detailentwurf unterteilt die abstrakten Migrationsaktivitäten in konkrete Migrationsaktivitäten. Die semantische Ausrichtung der konkreten Migrationsaktivitäten richtete sich nach bestehenden und wohlbekannten Softwareentwicklungansätzen.

Um ein Verständnis über die Anforderungen beziehungsweise Funktionalitäten, die bereits im monolithschen Softwaresystem vorhanden waren, zu entwickeln, wurde auf *BDD* [Sm14] gesetzt. Die Frage der Kohäsion der Funktionalitäten wurde durch die Betrachtung der Domäne beziehungsweise des Domänenwissens beantwortet. Für die Betrachtung der Domäne wurde sich auf die Konzepte von *DDD* [Ev04] berufen. Ausgehend von den Anforderungen und der Funktionalität wurden die Software- und Hardwarebausteine des monolithischen Softwaresystems erhoben. Hierfür wurde auf das *Design*

Recovery [CC90] zurückgegriffen, das als Teil des *Reverse Engineerings* gesehen wird. Konkret wurde das Konzept der *Nahtstellen* (engl. seams) [Fe05] verwendet, um Abhängigkeiten zwischen Softwarebausteinen offenzulegen. Durch die Vorarbeiten von BDD und DDD konnte das Design Recovery gezielt auf die Softwarebausteine, die relevant für den vorgesehenen Monolithenausschnitt waren, angewandt werden.

Die Extraktion des Monolithenausschnitts sieht zunächst die Spezifikation des Migrationsauslösers vor, um einen Ausgangspunkt für die Identifikation des Monolithenausschnitts zu besitzen; es wird folglich die Erreichung des Submeilensteins $M_{1.1}$ *Formalisierter Migrationauslöser* verfolgt. Der Migrationsauslöser kennzeichnet den Start einer Migrationsiteration und kann entweder als funktionale oder nicht-funktionale Anforderung an das Entwicklungsteam herangetragen werden. Durch den informellen und unstrukturierten Charakter der eingehenden Anforderung sieht die Extraktion eine Strukturierung der Anforderung durch die *EARS* vor. Anschließend erfolgt eine Klassifikation der Anforderung in eine der beiden Anforderungstypen. Ausgehend von dem Anforderungstyp sieht die Extraktion eine Formalisierung der Anforderung als *Gherkin Feature* oder *Control Case* vor.

Nach der Spezifikation des Migrationsauslösers sieht die Extraktion eine detaillierte Betrachtung der Geschäftsdomäne der Organisation vor. Die Geschäftsdomäne wird in einzelne Subdomänen untergliedert, um Geschäftsfähigkeiten, welche in der Organisation vorhanden sind, in diesen Subdomänen zu bündeln. Durch die Verbindung von Geschäftsfähigkeit zu Funktionalität können die Funktionalitäten ebenfalls einer Subdomäne zugeordnet werden, wodurch ebenfalls die Frage der Kohäsion von Funktionalitäten beantwortet werden kann. Die Subdomänen werden durch die Brainstorming-Praktik *Event Storming* erhoben, entsprechend dem Geschäftswert für die Organisation priorisiert und letztendlich in einer *Context Map* formalisiert. Nach Fertigstellung der Context Map gilt der Submeilenstein $M_{1.2}$ *Priorisierte Domänenstruktur* als erreicht.

Mit dem Migrationsauslöser und der Geschäftsdomäne als Ausgangspunkt zur Bestimmung des Monolithenausschnitts erfolgt anschließend die Wiederherstellung der Ist-Softwarearchitektur des monolithischen Softwaresystems und damit auch die Erreichung des Submeilensteins $M_{1.3}$ *Rekonstruierte Ist-Architekturbeschreibung*. Durch den Migrationsauslöser werden nur diejenigen Bestandteile des Monolithen wiederhergestellt, die in Beziehung mit diesem stehen. Zunächst wird die kohäsive Funktionalität bestimmt, die entweder mit dem Migrationsauslöser als funktionale Anforderung kohäsiv verbunden oder einer gemeinsamen *Subdomäne* zugeordnet ist, falls der Migrationsauslöser eine nicht-funktionale Anforderung ist. Aus den festgelegten Funktionalitäten werden im nächsten Schritt die Begrifflichkeiten in einer *ubiquitären Sprache* festgehalten, um die Kommunikation während der Migration effizienter zu gestalten. Anschließend sieht die Extraktion die Ermittlung der Softwarebausteine vor, welche für die Erbringung der kohäsiven Funktionalität notwendig sind. Hierfür werden *Nahtstellen* auf Ebene der Klassen mit ihren Funktionen und Attributen identifiziert und in *Effect Sketches* formalisiert. Aus den Effect Sketches ergeben sich damit die Klassen, welche dem Monolithenausschnitt zugeordnet werden können oder in Abhängigkeit zu dem Monolithenausschnitt

stehen. Ausgehend von den Klassen erfolgt für die Extraktion die Betrachtung der Datenhaltung. Es werden ein *relationales* und *objektorientiertes Datenmodell* angefertigt, um zum einen die zu überführenden Datenbanktabellen festzulegen und zum anderen die Datenabhängigkeiten innerhalb der Datenhaltung offenzulegen. Abschließend für die Ist-Softwarearchitektur wird ein Verteilungsdiagramm erstellt, das die Softwarebausteine des Monolithenausschnitts in Verbindung mit den Hardwarebausteinen bringt. Weiterhin werden explitzit die sogenannten *Backing Services* festgehalten, die von den Softwarebausteinen des Monolithenausschnitts verwendet werden.

Die wiederhergestellte Ist-Softwarearchitektur wird abschließend für die Ableitung der Soll-Softwarearchitektur verwendet. Diese bildet den Zustand nach der Extraktion des Monolithenausschnitts ab und entspricht mit Fertigstellung dem Submeilenstein $M_{1.4}$ *Definierter Monolithenausschnitt*. Dieser Submeilenstein kennzeichnet, dass alle notwendigen Informationen über den Monolithenausschnitt vorliegen. Bestimmt werden die letztendlichen Softwarebausteine, die in dem Monolithenausschnitt vorliegen. Dabei muss darauf geachtet werden, dass Softwarebausteine wie beispielsweise ein Webserver in den Monolithenausschnitt repliziert werden müssen. Nur die Softwarebausteine, welche *Anwendungs-* oder *Domänenlogik* abbilden, werden aus dem eigentlichen Monolithen entfernt. Für den Monolithenausschnitt wird zudem noch festgelegt, ob dieser auf einem eigenständigen Hardwarebaustein betrieben wird oder nicht. Zuletzt sieht die Extraktion noch den Entwurf der Web-APIs vor. Die Nahtstellen der Ist-Softwarearchitektur und die Abhängigkeiten der Datenhaltung werden anhand dreier Architekturstile in eine Web-API überführt.

Abschließend gilt es festzuhalten, dass die entworfene Extraktionsphase beginnend mit der Spezifikation des Migrationsauslösers bis hin zur Planung der Web-APIs, die Konzeption des Monolithenausschnitts betrachtet. Der implementierte Monolithenausschnitt wird mit der Erreichung des Meilensteins M_1 *Extrahierter Monolithenausschnitt* gekennzeichnet.

6 Entwurf der Modernisierungsphase

Der durch die Extraktionsphase entworfene und bereitgestellte Monolithenausschnitt muss nun in Microservices modernisiert werden. Das nachfolgende Kapitel widmet sich dieser Modernisierung und definiert konkrete Migrationsaktivitäten und -artefakte zum Entwurf der *Modernisierungsphase*. Die Durchführung der Migrationsaktivitäten und der Modellierung der Migrationsartefakte wird anhand des *Dokumentenmanagementsystem-Szenarios* (DMS) aus Abschnitt 1.5.6 verdeutlicht.

Bei dem Entwurf der Modernisierungsphase werden die folgenden zentralen Fragestellungen betrachtet: (1) *Wie können veränderte Anforderungen und Bedürfnisse an das Softwaresystem berücksichtigt werden?* (2) *Wie kann der Umfang der Microservices bestimmt werden?* (3) *Wie kann ein effizienter Betrieb der Microservice-Architektur sichergestellt werden?* (4) *Welche strukturellen Anpassungen am Entwicklungsteam sind durch den Microservice-Architekturstil notwendig?* Die Ausgestaltung der Modernisierungsphase mit konkreten Migrationsaktivitäten und -artefakten basiert auf der Beantwortung der aufgeführten Fragestellungen. Dabei werden durch die Fragestellungen verschiedene Aspekte des Softwaresystems betrachtet. So bezieht sich Fragestellung (1) auf die natürliche Evolution des Softwaresystems und sie thematisiert den Bedarf zur Anpassung der in dem Monolithenausschnitt bestehenden Anforderungen. Mit Fragestellung (2) wird der Entwurf der Microservice-Architektur im Sinne der Kohäsion von Funktionalität betrachtet. Fragestellung (3) greift die Betrachtung von nicht-funktionalen Anforderungen von einem effizienten Betrieb einer Microservice-Architektur auf. Die Fragestellung (4) thematisiert die durch die Microservice-Architektur an Komplexität gewinnende Entwicklung und Wartung der geschaffenen Microservices in Verbindung mit dem Entwicklungsteam.

Zur Beantwortung dieser Fragestellungen werden die Migrationsaktivitäten und -artefakte der Modernisierungsphase auf den Softwareentwicklungsansätzen der *verhaltensgetriebenen Entwicklung* (engl. Behavior-Driven Development, BDD) [Sm14] und dem *domänengetriebenen Entwurf* (engl. Domain-Driven Design, DDD) [Ev04] basieren. Gerade DDD wird häufig zur Identifikation der Kohäsion und Festlegung der Service-Granularität verwendet [Ne15, PZ+17, RS+17, AS+19, MB+20]. Den kombinierten Einsatz von BDD und DDD zeigen Hippchen et al. [HG+17] bei der Entwicklung von Microservice-basierten Softwaresystemen auf, indem sie BDD für die Spezifikation des Softwaresystems und DDD für den Entwurf der Microservice-Architektur verwenden. Auch [HR20] sieht die Geschäftsdomäne als Grundlage zur Bestimmung des Umfangs der einzelnen Microservices, auch wenn DDD nicht explizit als Softwareentwicklungsansatz genannt wird. Ausgehend von den

Vorarbeiten von Hippchen et al. wird der Einsatz von BDD und DDD an die Modernisierungsphase angepasst.

In Abschnitt 6.1 wird eine prozessuale Sicht auf die Modernisierungsphase und damit eine Übersicht über die Migrationsaktivitäten und -artefakte gegeben. Anschließend erfolgt in Abschnitt 6.2 die Überarbeitung der Anforderungsspezifikation zur Anpassung der bestehenden Anforderungen an die neuen Bedürfnisse. Mit Abschluss der Anforderungsspezifikation kann der Entwurf der Microservice-Architektur erfolgen, der in Abschnitt 6.3 diskutiert wird. Zum Abschluss der Modernisierungsphase beschreibt Abschnitt 6.4 die Reorganisation des Entwicklungsteams anhand der Microservice-Architektur.

6.1 Übersicht über konkrete Migrationsaktivitäten und -artefakte

Im weiteren Verlauf wird der systematische Ablauf nur noch als *Modernisierung* bezeichnet. Zur Definition der Modernisierung wird zuerst die zugrundeliegende Herangehensweise diskutiert. Anschließend erfolgt die Festlegung der Modernisierung, die aus konkreten, sequentiell ausgeführten Migrationsaktivitäten besteht. Zu den Migrationsaktivitäten werden Migrationsartefakte angegeben, die als Quellen beziehungsweise als Ergebnisse einer Migrationsaktivität dienen. Ebenso wie bei der Extraktionsphase werden auch für die Modernisierungsphase Submeilensteine eingeführt, die der Synchronisation von den Zwischenergebnissen dienen.

6.1.1 Zugrundeliegende Herangehensweise der Modernisierung

Dem Entwurf der Modernisierungsphase werden einige bestehende Arbeiten zugrunde gelegt, um die Modernisierung selbst auf bereits etablierte Konzepte fußen zu lassen. Der wesentliche Beitrag ist dabei, diese bestehenden Arbeiten im Kontext der Modernisierung miteinander zu verbinden und so einen systematischen und nachvollziehbaren Ablauf zu gewährleisten.

Wie auch in der Extraktionsphase ist bei dem Entwurf der Modernisierungsphase die Betrachtung *verhaltensspezifischer Aspekte* des Softwaresystems zentral für den Einstieg. Denn ausgehend von der natürlichen Evolution des Softwaresystems, die zu kontinuierlichen Änderungen an den Anforderungen des Softwaresystems führt [NZ+04], wird eine Überarbeitung des Verhaltens in dem Monolithenausschnitt notwendig. Die Betrachtung des Verhaltens erfolgt auch bei Modernisierung durch die Anwendung des Softwareentwicklungsansatzes *BDD* [Sm14].

Damit das überarbeitete Verhalten anhand der Kohäsion in Microservices verteilt werden kann, wird auf den Softwareentwicklungsansatz *DDD* [Ev04] zurückgegriffen. DDD legt der Kohäsion von Verhalten die Geschäftsdomäne zugrunde. Die Verwendung von DDD bei der Migration zeigen auch

Newman [Ne19] und Henry und Ridene [HR20]. Die Schwachstellen der beiden Arbeiten liegen jedoch in der systematischen Anwendung von DDD.

Das Verbinden der beiden Softwareentwicklungsansätze BDD und DDD haben bereits Hippchen et al. [HG+17] näher beleuchtet. Die Autoren verbinden diese Ansätze im Kontext eines systematischen Softwarenentwicklungsprozesses zur Entwicklung domänengetriebener und microservicebasierter Softwaresysteme mit dem Ergebnis, dass beide Ansätze sich bei der Betrachtung verschiedener Aspekte des Softwaresystems ergänzen.

Der von Hippchen et al. [HG+17] vorgestellte Softwareentwicklungsprozess stellt somit eine geeignete Grundlage für den Detailentwurf der Modernisierung dar. Es gilt jedoch zu beachten, dass dieser Softwareentwicklungsprozess eine Neuentwicklung des Softwaresystems auf der "grünen Wiese" vorsieht, was einer Transferleistung auf den Kontext der Migration und den bestehenden Monolithenausschnitt bedarf. Neben dieser Transferleistung werden weitere zentrale Aspekte der Migration in der Modernisierung betrachtet, die bislang nicht von dem Softwareentwicklungsprozess abgedeckt werden. Zum einen gilt es, nicht-funktionale Anforderungen für den Entwurf der Microservice-Architektur zu berücksichtigen und zum anderen das monolithisch geprägte Entwicklungsteam auf den neuen Architekturstil anzupassen.

6.1.2 Ganzheitlicher Entwurf der Modernisierung

Die Grundlage für den ganzheitlichen Entwurf der Modernisierungsphase stellt die Systematik aus Abschnitt 4.2.2 und die darin festgelegten abstrakten Migrationsaktivitäten und -artefakte dar. Dabei werden die abstrakten Migrationsaktivitäten und -artefakte anhand konkreter Migrationsaktivitäten und -artefakte spezifiziert.

Der ganzheitliche Entwurf der Modernisierung kann der Abbildung 6.1 entnommen werden. Die abstrakten Aktivitäten (1) *Überarbeiten der Anforderungsspezifikation*, (2) *Ableitung der Microservices* und (3) *Reorganisation des Entwicklungsteams* geben den sequentiellen Rahmen vor.

Bei dem (1) *Überarbeiten der Anforderungsspezifikation* wird die bestehende Anforderungsspezifikation anhand der drei konkreten Migrationsaktivitäten *Festlegen der Geschäftsziele*, *Anpassung der Anforderungen* und *Anpassung der ubiquitären Sprache* überarbeitet. Alle drei Migrationsaktivitäten sind stark durch den Softwareentwicklungsansatz *BDD* geprägt. Bevor eine Anpassung der Anforderungen beziehungsweise der *Gherkin Features* erfolgt und diese damit auch vereinfacht wird, werden zunächst die–ebenfalls durch BDD bekannten–Migrationsartefakte der *Vision*, *Ziele* und *Fähigkeiten* erstellt und der Anforderungsspezifikation zugeordnet. Anschließend wird aufbauend auf dem neuen Wissen die *ubiquitäre Sprache* im *Glossar* und *Language Sketch* angepasst.

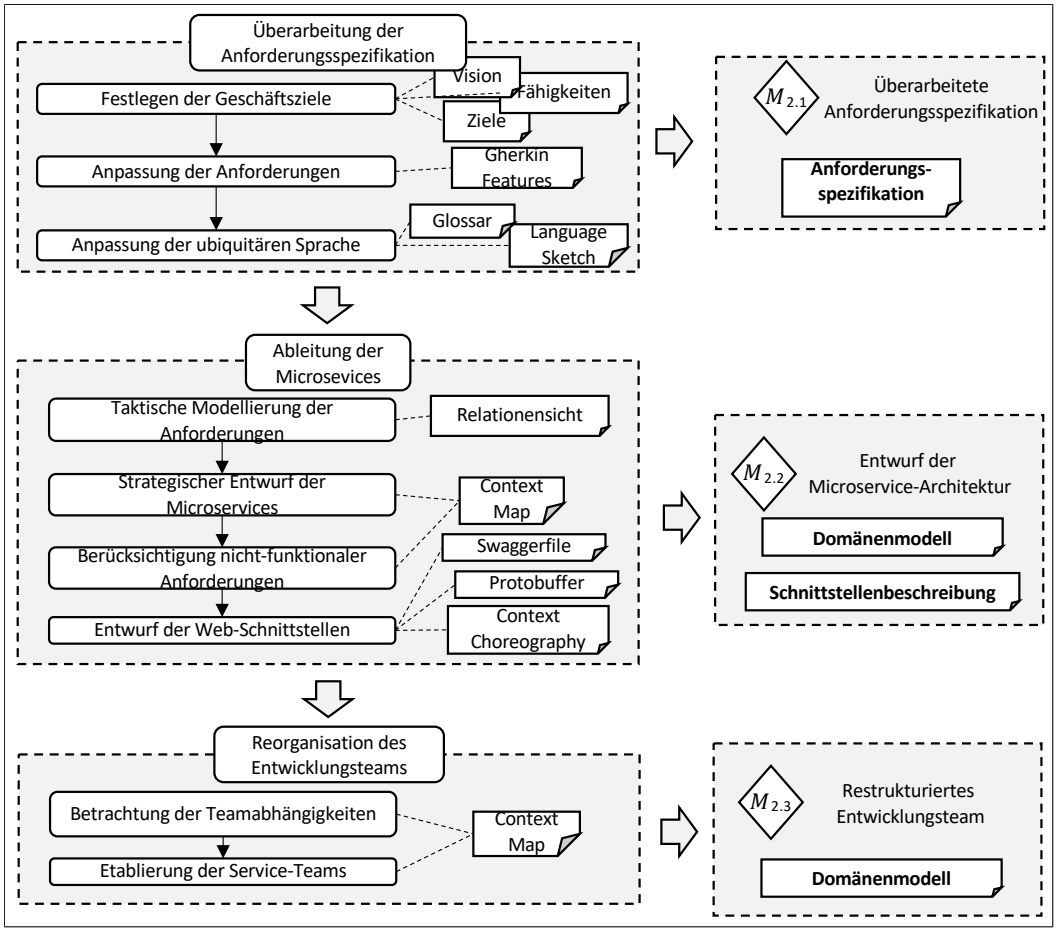

Abbildung 6.1: Übersicht der Migrationsaktivitäten und -artefakte der Modernisierungsphase

Das (2) *Ableiten der Microservices* erfolgt anhand einer *taktischen Modellierung der Anforderungen*, gefolgt von einem *strategischen Entwurf der Microservices*, der als Ergebnis eine starke Orientierung an der Geschäftsdomäne aufweist. Die Ergebnisse der beiden werden in formalen Modellen festgehalten und dem *Domänenmodell* zugeordnet. Der domänengetriebene Entwurf ignoriert Qualitätsmerkmale in Form von nicht-funktionalen Anforderungen, weshalb anschließend diese nicht-funktionalen Anforderungen berücksichtigt werden. Nachdem die Microservices festgelegt sind, erfolgt der *Entwurf der Web-Schnittstellen*, der sich auf verschiedene Architekturstile beruft. Je nach Architekturstil kommt eines der dargestellten Migrationsartefakte zum Tragen.

Die (3) *Reorganisation des Entwicklungsteams* stellt das Ende der Modernisierung dar. Hierfür wird zunächst eine *Betrachtung der Teamabhängigkeiten* durchgeführt und die Kommunikationswege, Abhängigkeiten und Machtverhältnisse zwischen den für die Microservices verantwortlichen *Service-Teams* festgelegt. Die Abhängigkeiten werden in der *Context Map* festgehalten. Bei der *Etablierung der Service-Teams* werden Entwickler des monolithischen Entwicklungsteams mit verschiedenen Fähigkeiten auf die Microservices aufgeteilt, wobei die Teamabhängigkeiten der Context Map ent-

scheidend für die Wahl der einzelnen Softwareentwickler ist.

Nach dem Abschluss einer übergeordneten beziehungsweise abstrakten Migrationsaktivität wird ein Submeilenstein der Modernisierung erreicht. Folglich umfasst die Modernisierung die drei Submeilensteine: $M_{2.1}$ *Überarbeitete Anforderungsspezifikation*, $M_{2.2}$ *Entwurf der Microservice-Architektur* und $M_{2.3}$ *Restrukturiertes Entwicklungsteam*. Die Zwischenergebnisse der einzelnen Submeilensteine werden in der Abbildung 6.1 dargestellt.

6.2 Überarbeitung der Anforderungsspezifikation

Es wird das Ziel verfolgt, die *Anforderungsspezifikation* an veränderte Bedürfnisse anzupassen. In der Extraktionsphase wurde der Ist-Zustand der Funktionalität aus dem monolithischen Softwaresystem extrahiert, ohne dabei die Korrektheit zu prüfen. Eine solche Korrektheitsprüfung erfolgt nun im Rahmen der abstrakten Migrationsaktivität *Überarbeitung der Anforderungsspezifikation*.

Der Bedarf zur Überarbeitung der Anforderungsspezifikation ergibt sich sowohl aus der *Software-Evolution* [NZ+04] als auch aus dem *Risikomanagement* für das Softwaresystem [DL03]. Getrieben durch den Wandel der Organisation ändern sich bereits implementierte Anforderungen an das Softwaresystem. Auch können Anforderungen bereits initial falsch implementiert worden sein, sodass eine Änderung notwendig wird. Das Realisieren der Änderungen ist mit hohen finanziellen Kosten als auch zeitlichem Aufwand verbunden [NZ+06, WB13]. Durch das Einschränken der Änderungen auf den Monolithenausschnitt werden Kosten und zeitlicher Aufwand minimiert. Bei veränderten Anforderungen gilt es, die Anforderung, welche durch den Migrationsauslöser aufgenommen wurde, erneut zu betrachten. Der Migrationsauslöser wurde zum Zeitpunkt der Extraktionsphase auf Basis der unveränderten Anforderungen erhoben und spezifiziert.

Die Migrationsaktivitäten zur Überarbeitung der Anforderungsspezifikation basieren wie auch in der Extraktionsphase auf dem Softwareentwicklungsansatz BDD. Neben den üblichen *Gherkin Features* greift die Modernisierungsphase auf weitere BDD-Konzepte zurück. In der ersten Migrationsaktivität werden die Geschäftsziele des ehemals monolithischen Softwaresystems festgelegt. Aus den Geschäftszielen können der Mehrwert des Softwaresystems abgeleitet und zu implementierenden Gherkin Features verifiziert werden. Zur Festlegung der Geschäftsziele stellt Smart [Sm14] die BDD-Konzepte *Vision*, *Ziele* und *Fähigkeiten* vor. Alle drei Konzepte werden im Kontext der ersten Migrationsaktivität für den spezifischen Rahmen des Monolithenausschnitts festgelegt.

In der zweiten Migrationsaktivität erfolgt die Anpassung der Anforderungen an veränderte Gegebenheiten. Die Anforderungen, die sich in der Anforderungsspezifikation befinden, werden zunächst anhand der Geschäftsziele verifiziert. Anschließend erfolgt in enger Zusammenarbeit mit den Interessenvertretern eine Überarbeitung der Anforderungen. Die Anforderungserhebung ist ein explorativer

Vorgang, der durch Praktiken des *Requirement Engineerings* ergänzt wird [TS+12]. Unter den Anforderungen werden sowohl funktionale als auch nicht-funktionale Anforderungen verstanden. Die funktionalen Anforderungen werden erneut als *Gherkin Features* festgehalten, während die nicht-funktionalen durch *Control Cases* spezifiziert werden. Am Ende dieser Migrationsaktivität liegt eine überarbeitete Anforderungsspezifikation vor.

Abschließend erfolgt in der letzten Migrationsaktivität die Anpassung der *ubiquitären Sprache* an die Anforderungsspezifikation. Hierfür werden das *Glossar* und der *Language Sketch* aus der Extraktionsphase herangezogen und an die neuen Begrifflichkeiten angepasst.

6.2.1 Festlegen der Geschäftsziele

Zu Beginn der Modernisierungsphase wird zunächst grundlegend die Diskussion bezüglich des *Geschäftswerts* des Monolithenausschnitts beziehungsweise der darin befindlichen Funktionalität geführt. Der Geschäftswert beschreibt dabei, ob die vorliegende Funktionalität das übergeordnete Problem der Organisation und Kunden löst. Um eine Aussage über den Geschäftswert treffen zu können, müssen zunächst die Geschäftsziele identifiziert werden, welche mit dem Monolithenausschnitts verfolgt werden. Die Festlegung der Geschäftsziele stellt die erste Migrationsaktivität der Modernisierungsphase dar.

Ein Konzept für das Festlegen der Geschäftsziele stellt Smart [Sm14] im Kontext von BDD vor. Smart führt, übergeordnet zu den Gherkin Features, die Softwareartefakte *Vision*, *Ziele* und *Fähigkeiten* ein, die in Kombination die Geschäftsziele bilden. Alle drei Softwareartefakte beziehen sich dabei auf das gesamte Softwaresystem, ohne dabei technische Details über die Implementierung von Funktionalitäten vorzugeben. Folglich sind die Softwareartefakte implizit auch für den Monolithenausschnitt gültig, da dieser als direkter Softwarebaustein des Softwaresystems anzusehen ist. Aus der Festlegung der Geschäftsziele folgt die Möglichkeit für das Migrationsteam, die Gherkin Features zu verifizieren und gegebenenfalls zu messen, ob diese den Geschäftszielen entsprechen.

Es wird genau eine *Vision* erstellt. Die Vision beschreibt, warum die Funktionalität des Monolithenausschnitts notwendig ist und was damit erreicht werden soll. Verfeinert wird die Vision durch mehrere *Ziele*, die den Mehrwert für den messbaren wirtschaftlichen Erfolg der Organisation beschreiben, der durch den Monolithenausschnitt entsteht. Die Ziele werden wiederum durch mehrere *Fähigkeiten* verfeinert. Eine Fähigkeit beschreibt dabei ein Verhalten, das den Interessenvertretern zur Verfügung gestellt werden muss, um ein bestimmtes Ziel zu erreichen. Abbildung 6.2 zeigt dabei ein Beispiel anhand des Dokumentenmanagementsystems. Die Beziehungen zwischen den Softwareartefakten von BDD stellt Smart [Sm14] über die in der Abbildung 6.2 vorzufindende Pyramide dar.

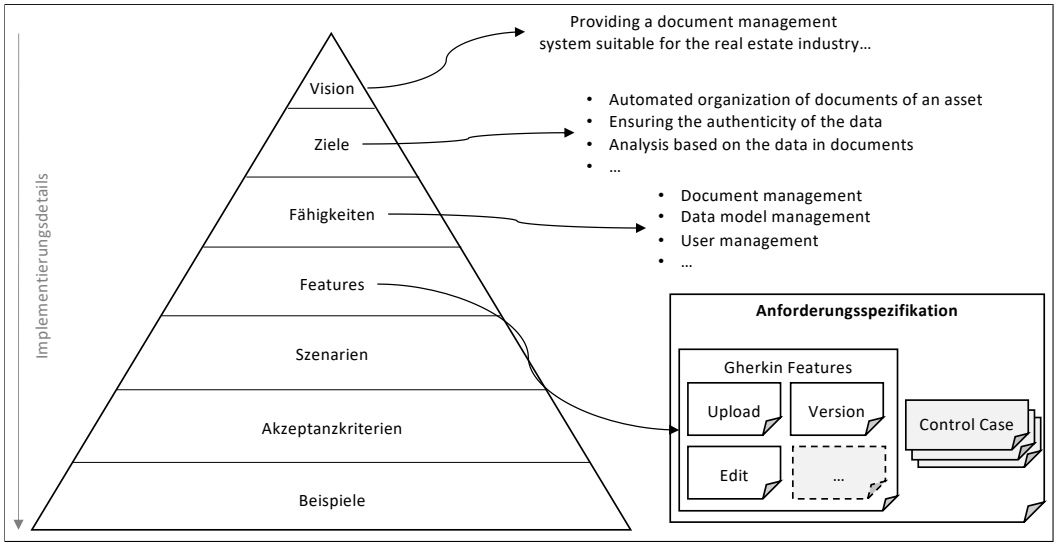

Abbildung 6.2: Festlegen der Geschäftsziele anhand der verhaltensgetriebenen Entwicklung nach Smart [Sm14]

6.2.2 Anpassung der Anforderungen

Nach der Festlegung der Geschäftsziele erfolgt die eigentliche Anpassung der Anforderungen. Die Anpassung bedingt jedoch zunächst eine Validierung, inwiefern die Anforderungen noch die tatsächlichen Bedürfnisse der Interessenvertreter widerspiegeln. Festzuhalten ist, dass sich die Bedürfnisse nicht zwangsläufig geändert haben müssen. Diese Migrationsaktivität verfolgt das Ziel, zunächst die bestehende Anforderungsspezifikation gegen die festgelegten Geschäftsziele und die tatsächlichen Bedürfnisse der Interessenvertreter zu validieren. Gegebenenfalls wird im Rahmen dieser Migrationsaktivität eine Anpassung der betreffenden Anforderungen durchgeführt.

Die Migrationsaktivität verfolgt einen leichtgewichtigen Ansatz zur Modernisierung, um den zeitlichen Aufwand zu minimieren. Auf aufwendige Konzepte aus den Bereichen *Software Changes* [CV10] und *Risikomanagement* (engl. risk management) [DL03] wird verzichtet, auch wenn diese nach der Migrationsiteration für systematische Änderungen etabliert werden sollten. Vorgesehen wird zunächst die Validierung der bestehenden Gherkin Features und Control Cases in der Anforderungsspezifikation in Anbetracht der festgelegten Geschäftsziele. Einen möglichen Ansatz zur Validierung stellen dabei Matts und Adzic [MA11] mit der *Feature Injection* vor, welche auf die von Smart [Sm14] vorgestellten *BDD-Pyramide* (Abbildung 6.2) zurückgreift. Die Validierung offenbart, ob die Anforderungen entweder den Geschäftszielen entsprechen oder Inkonsistenzen aufweisen. Entsprechen die Anforderungen den Geschäftszielen, wird im Sinne des Aufwands auf eine detaillierte Validierung verzichtet. Sobald eine Anforderung jedoch eine Inkonsistenz aufweist, erfolgt eine detaillierte Validierung.

Für eine detaillierte Validierung wird erneut eine Anforderungserhebung durchgeführt. Für die Anforderungserhebung haben sich in der Vergangenheit verschiedene Praktiken mit unterschiedlichen Intentionen etabliert [CB+10, TS+12]. Praktiken wie *Interviews* oder *Brainstorming* sehen dabei eine enge Einbindung der Interessenvertreter vor. Eine solche Einbindung beziehungsweise enge Zusammenarbeit ist fundamental für die Anpassung der Anforderungen, denn nur so können, Harker et al. [HE+93] zufolge, die tatsächlichen Bedürfnisse identifiziert werden. Durch die Migrationsartefakte der Gherkin Features und Control Cases besteht zudem die Synergie, dass durch die Verwendung der natürlichen Sprache in einer formalisierten Struktur die Einbindung von Interessenvertretern möglich ist und die Zusammenarbeit effektiv gestaltet werden kann. Gerade dies wird bei BDD als besonderes Merkmal hervorgehoben [Sm14]. Während der Durchführung einer Anforderungserhebung erfolgt simultan die Erstellung beziehungsweise Überarbeitung der Gherkin Features oder Control Cases.

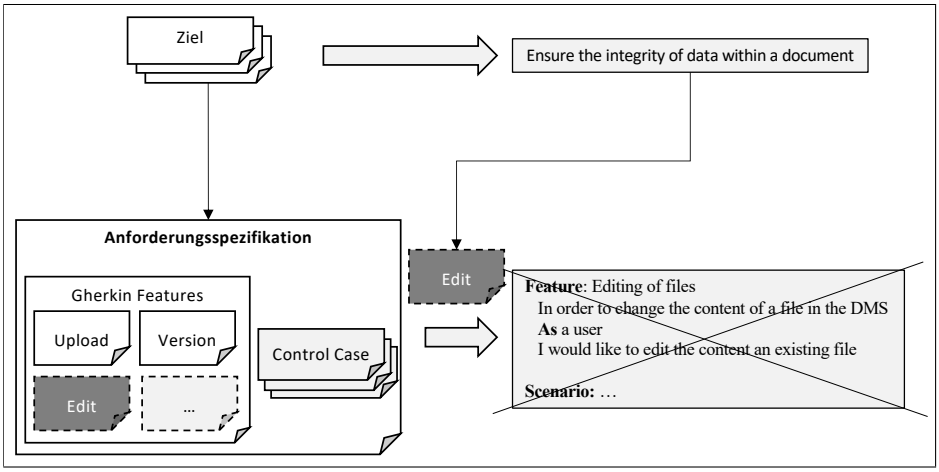

Abbildung 6.3: Verwerfen einer Anforderung aufgrund der festgelegten Ziele

In Abbildung 6.3 wird der Ablauf der Migrationsaktivität am Beispiel des Dokumentenmanagementsystems veranschaulicht. Die Funktionalität "*Edit*" steht im Konflikt mit den für den Monolithenausschnitt festlegten Geschäftszielen. Denn eine Fähigkeit fordert die "*Integrität der Daten innerhalb eines Dokumentes*". Durch die Funktionalität "*Edit*" ist es Nutzern des DMS jedoch möglich, Inhalte eines Dokuments zu verändern. Es folgt eine detaillierte Validierung der Anforderungen mittels einer erneuten Anforderungserhebung in enger Zusammenarbeit mit den Interessenvertretern. Das Ergebnis ist das Verwerfen der Funktionalität "*Edit*" aufgrund des fehlenden Bedarfs und des Konflikts mit den Geschäftszielen.

6.2.3 Anpassung der ubiquitären Sprache

Nach der Anpassung der Anforderung muss nun die *ubiquitäre Sprache*, die in der Extraktionsphase in Form der Migrationsartefakte *Glossar* und *Language Sketch* formalisiert wurde, an die neuen

Anforderungen angepasst werden. Die Anpassung der ubiquitären Sprache erfolgt im Rahmen dieser Migrationsaktivität.

Wie bereits in der Extraktionsphase beschrieben, wurden die Migrationsartefakte Glossar und Language Sketch mit dem Wissen angefertigt, dass in der Modernisierungsphase eine Überarbeitung der Anforderungen angestrebt und damit eine Anpassung der ubiquitären Sprache notwendig wird. Die in dieser Migrationsaktivität erarbeiteten Entwürfe des Glossars und Language Sketchs sind nicht mehr als temporär anzusehen und gelten somit auch nach der Migrationsiteration für die Kommunikation.

Zunächst erfolgt die Anpassung des Glossars. Begriffe der neuen Anforderungen werden mit den in dem Glossar vorhandenen Begriffen verglichen. In diesem Zuge werden neue Begriffe identifiziert und dem Glossar hinzugefügt. Weiterhin werden Begriffe, die zwar in dem Glossar existieren, aber nicht mehr in den Anforderungen vorzufinden sind, entfernt. Nach dem Vergleich erfolgt eine allgemeine Überprüfung der im Glossar vorzufindenden Begriffe, mit dem Ziel, die Begriffe der Domäne anzunähern und unpassende Bezeichnungen zu ersetzen. Denn häufig werden in den frühen Phasen eines Softwaresystems die Begriffe durch die Organisation oder auch die Softwareentwickler definiert, ohne das Domänenwissen einzubeziehen. Durch die generelle Bewertung der Begriffe können somit die unpassenden Begriffe überarbeitet werden.

Nach Abschluss des Glossars erfolgt die Überarbeitung des Language Sketchs entsprechend der neuen Begriffe. Bei dem Language Sketch wird ein starker Fokus auf die Beziehungen zwischen den Begriffen beziehungsweise den dahinter liegenden Domänenobjekten gelegt. Neue oder geänderte Beziehungen können erneut aus den Gherkin Features abgeleitet werden.

6.3 Ableitung der Microservices

Nach der Anpassung der Anforderungsspezifikation an die tatsächlichen Bedürfnisse sieht nun die Modernisierung das Ableiten der Microservices vor. Verfolgt wird ein zunächst *domänengetriebener Entwurf* der Microservice-Architektur, der ausgehend von den *Gherkin Features* der Anforderungsspezifikation eine taktische und strategische Modellierung der Geschäftsdomäne anstrebt. Anschließend werden, um einen effizienten und effektiven Betrieb zu gewährleisten, nicht-funktionale Anforderungen betrachtet und in dem domänengetriebenen Entwurf berücksichtigt. Für den Entwurf der Microservices werden abschließend Web-Schnittstellen entworfen.

Inspiriert ist diese Migrationsaktivität durch die Vorarbeiten von Hippchen et al. [HG+17, HS+19]. Die Autoren stellen ein systematisches Vorgehen zum Entwurf domänengetriebener Microservices vor. Dabei werden auch formale Modellierungsartefakte vorgestellt, die im Rahmen der Modernisierung als Migrationsartefakte verwendet werden. Die Abhängigkeiten zwischen den Migrationsartefakten können der Abbildung 6.4 entnommen werden.

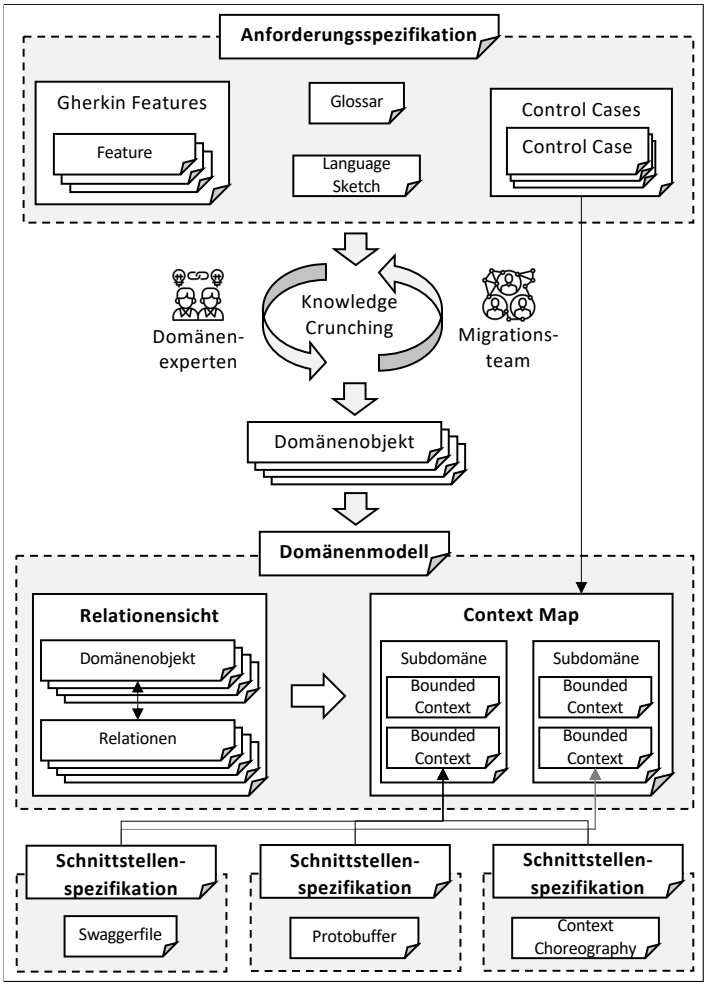

Abbildung 6.4: Ableiten der Microservices durch die Betrachtung der Domäne

Aus der *Anforderungsspezifikation*, welche die Gherkin Features, Control Cases, das Glossar und den Language Sketch umfasst, werden durch das *Knowledge Crunching* zwischen den Domänenexperten und dem Migrationsteam Domänenobjekte identifiziert. Diese Domänenobjekte dienen als Input für eine *taktische Modellierung* [Ev04] darstellt. Das Resultat der taktischen Modellierung ist die *Relationensicht*, welche die Domänenobjekte über *Relationen* miteinander verbindet. Die Relationensicht dient weiterführend als Grundlage für die *strategische Modellierung* [Ev04] die bereits in der Extraktionsphase zur Erstellung der *Context Map* genutzt wurde. In der Modernisierung wird die Context Map um sogenannte *Bounded Contexts* erweitert, die einen Teil der Relationensicht in sich kapseln und, Newman [Ne15] zufolge, einen geeigneten *Microservice-Kandidaten* repräsentieren. Für jeden dieser Bounded Contexts werden *Schnittstellenspezifikationen* als Entwurf der Web-Schnittstellen erstellt, die sich nach verschiedenen Architekturstilen richten. Dabei kann ein Bounded Context über mehrere Schnittstellenspezifikationen verfügen, abhängig davon, welche Architekturstile eingesetzt werden.

6.3.1 Taktische Modellierung der Anforderungen

Die Implementierung der Anforderungsspezifikation mittels domänengetriebener Microservices bedingt zunächst ein Verständnis über die Domänenobjekte und deren Beziehungen zueinander, die im Kontext der Anforderungen auftreten. Zum Aufbau dieses Verständnisses wird im Rahmen dieser Migrationsaktivität eine taktische Modellierung durchgeführt. Ausgangspunkt für die taktische Modellierung sind die Gherkin Features, das Glossar und der Language Sketch. Das Ergebnis der taktischen Modellierung ist ein formales Modell, welches dem Domänenmodell zugeordnet wird.

Die Praktik der taktischen Modellierung basiert auf DDD [Ev04]. Das Ziel der taktischen Modellierung ist die Abbildung der feingranularen Domänenstruktur durch Domänenobjekte und Relationen in einem *taktischen Modell*. Man spricht von einem taktischen Modell oder auch taktischen Entwurf aufgrund *taktischer Muster*, die während der Modellierung der Domäne verwendet werden [Ve13]. Ein taktisches Muster stellt eine Taktik zur Lösung eines Entwurfs- oder Implementierungsproblems bereit. Evans [Ev04] definiert weiterführend, dass die Modellierungssprache eines taktischen Modells, neben den festgelegten taktischen Mustern, nicht vorgegeben wird; das Modell muss lediglich dem Zweck des Wissenstransports gerecht werden. Eine fehlende einheitliche Modellierungssprache führt jedoch zu Inkonsistenzen bei der Modellierung und mindert so den Nutzen des Modells [RN+13]. Folglich ist auch die Erstellung des taktischen Modells nicht systematisch durchführbar, wenn den Migrationsteams freie Hand bei der Modellierungssprache gelassen wird. Um den systematischen Charakter des iterativen Migrationsprozesses auch für diese Migrationsaktivität zu gewährleisten, wird auf die von Hippchen et al. [HG+17] und Schneider et al. [SH+19] aufgestellten Modellierungsgrundlagen für die taktische Modellierung zurückgegriffen.

Evans [Ev04] beschreibt das Domänenmodell als Sammlung verschiedener Modelle, die unterschiedliche Aspekte der Domäne wiedergeben. Zwar unterscheidet Evans zwischen den taktischen und den strategischen Modellen, aber auch diese Modelle betrachten unterschiedliche Aspekte der Domäne. Für die taktische Modellierung führen Hippchen et al. [HG+17] zunächst das Konzept der *Domänensichten* ein. Eine Domänensicht betrachtet die Domäne aus einem bestimmten Blickwinkel und wird durch eine definierte Menge an möglichen Modellierungssprachen dargestellt. Die Modellierungssprachen orientieren sich an der *Unified Modeling Language* (UML). Vorgestellt werden zunächst nur die *Relationensicht* und *Prozesssicht* als Domänensichten. Im Fokus dieser Migrationsaktivität steht die Relationensicht, welche die Domänenobjekte der Anforderungsspezifikation gemeinsam mit den bestehenden Relationen zu anderen Domänenobjekten und damit die strukturellen Aspekte der Geschäftsdomäne darstellt. Als geeignete Modellierungssprache wird das UML-Klassendiagramm [BD09] vorgestellt. Schneider et al. [SH+19] stellen zur Erweiterung des UML-Klassendiagramms ein für die taktische Modellierung ausgelegtes UML-Profil vor, das die taktischen Muster als Stereotypen bereitstellt.

Wie auch in anderen Migrationsaktivitäten setzt die Modernisierung bei der taktischen Modellierung auf eine Kollaboration des Migrationsteams mit den Domänenexperten. Die Vorgehensweise ist dabei hochgradig explorativ. Evans [Ev04] verweist bei der taktischen Modellierung erneut auf das *Knowledge Crunching*. Bei der Modernisierung kann die Effizienz des Knowledge Crunching jedoch durch Informationsquellen wie die Gherkin Features, das Glossar und den Language Sketch verbessert werden, denn in allen drei Migrationsartefakten finden sich Domänenobjekte und Relationen zwischen den Domänenobjekten wieder. Hippchen et al. [HG+17] sehen gerade die Gherkin Features als eine primäre Quelle für die Modellierung der Relationensicht. Zusätzlich zeigt der Language Sketch bereits Relationen zwischen den Domänenobjekten auf. Wichtig ist jedoch, dass die Relationen aus dem Language Sketch im Rahmen der taktischen Modellierung aus einem implementierungsnahen Blickwinkel betrachtet werden, denn die Relationensicht wird DDD zufolge mit der Implementierung der Domänenschicht der Schichtenarchitektur verknüpft.

Dadurch, dass sowohl das Glossar als auch der Language Sketch aus den Gherkin Features abgeleitet werden, finden sich alle notwendigen Domänenobjekte in diesen wieder. Nach der Ableitung werden die Relationen betrachtet, die zunächst aus dem Language Sketch übernommen und dann anhand der Gherkin Features überarbeitet werden. Anschließend werden aus den Gherkin Features die Eigenschaften eines Domänenobjekts abgeleitet, die sich mit den Attributen und Methoden einer Klasse der objektorientierten Programmierung vergleichen lassen. Weiterhin dienen die Gherkin Features als Grundlage zur Modellierung der Relationensicht als *wissensreicher Entwurf* (engl. knowledge-rich design), womit Evans [Ev04] taktische Modelle bezeichnet, die neben Domänenobjekten und Relationen auch wichtige Geschäftsregeln wie Einschränkungen oder Bedingungen beinhalten. Die Geschäftsregeln wirken sowohl auf die Domänenobjekte als auch auf die Relationen ein. In Abbildung 6.5 wird der oben beschriebene Ablauf anhand des DMS-Szenarios illustriert.

Bei der taktischen Modellierung der Domänenobjekte muss darauf geachtet werden, dass nur die Domänenobjekte in die Relationensicht aufgenommen werden, die auch für die Erbringung der Gherkin Features von Relevanz sind [HS+19]. Es folgt eine bei der Implementierung beherrschbare Geschäftsdomäne. Dies ist in den Migrationsartefakten Glossar und Language Sketch nicht immer gegeben, was aber für diese beiden Migrationsartefakte nicht entscheidend ist. Die Entscheidung, inwiefern welche Domänenobjekte von Relevanz sind, trifft das Migrationsteam gemeinsam mit den Domänenexperten durch das Knowledge Crunching.

Die Modellierung der Relationensicht erfolgt parallel zu dem Knowledge Crunching und basiert auf dem von Schneider et al. [SH+19] vorgestellten UML-Profil. Die erstellte Relationensicht wird dem abstrakten Migrationsartefakt des Domänenmodells zugeordnet.

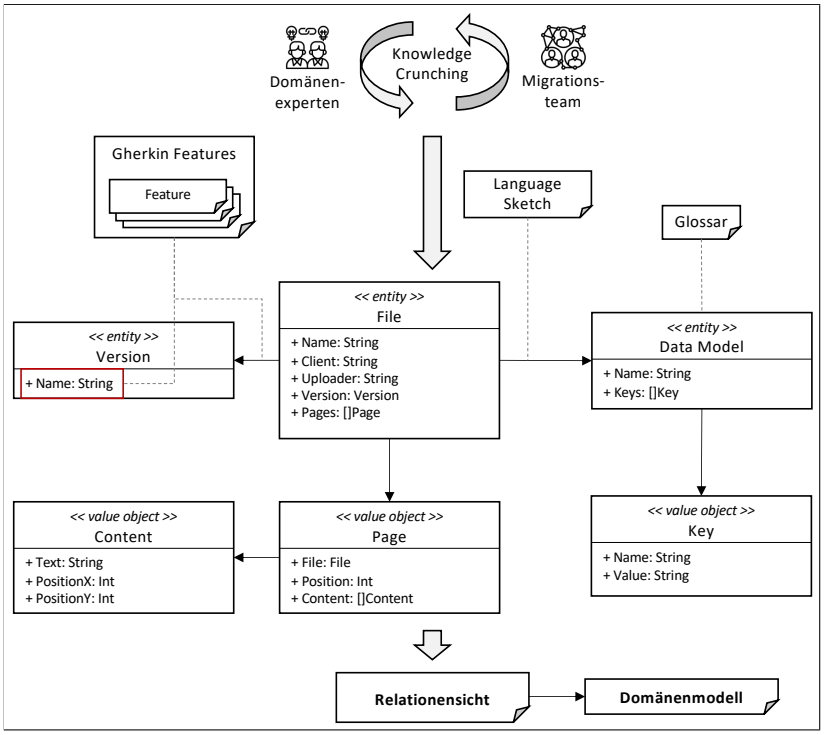

Abbildung 6.5: Taktische Modellierung der Domänenobjekte in einer Relationensicht

6.3.2 Strategischer Entwurf der Microservices

Im Anschluss an die taktische Modellierung erfolgt die Erweiterung der bereits erfolgten strategischen Modellierung während der Extraktionsphase. In der Extraktionsphase wurden bislang nur die Subdomänen der Organisation anhand von Geschäftsfähigkeiten identifiziert und die Anforderungen der Anforderungsspezifikation einer Subdomäne zugeordnet. Mit dieser Migrationsaktivität wird nun der bereits vorhandene *strategische Entwurf* um das Konzept der *Bounded Contexts* erweitert, sodass ein domänengetriebener Entwurf der Microservice-Architektur ableitbar ist. Die Grundlage zur Bestimmung der Bounded Contexts stellt die Relationensicht dar.

Ein Bounded Context repräsentiert eine explizite Grenze, die den Gültigkeitsbereich des Domänenwissens definiert (vgl. Abschnitt 2.4.4). Das Domänenwissen innerhalb eines Bounded Contexts muss eine hohe Kohäsion aufweisen, um auch eine *Autonomie* gewährleisten zu können [Ve13]. Genau durch diese Autonomie des Domänenwissens ist Newman [Ne15] der Auffassung, dass ein Bounded Context einem geeigneten Kandidaten für einen einzelnen Microservice entspricht. Aus der Anreicherung der *Context Map* mit Bounded Contexts entsteht somit der domänengetriebene Entwurf der Microservice-Architektur.

Die Grundlage zur Bestimmung der Bounded Contexts ist im Kontext der Modernisierung das taktische Modell der Relationensicht [HS+19]. In der Relationensicht befindet sich bereits das Domänenwissen,

welches durch den bisherigen Monolithenausschnitt erbracht wurde und nun durch Microservices bereitgestellt werden soll. Ergänzend wird der Ansatz von Tune [Tu19] zur Identifikation von Bounded Contexts genutzt. Tune sieht die Verkettung der Domänenereignisse, die im Rahmen des *Event Stormings* der Extraktionsphase (vgl. Abschnitt 5.3.1) identifiziert wurden, als ein Zustandsautomat, wobei Übergänge in weitere Zustände nur durch *zentrale Domänenereignisse* (engl. pivotal domain events) stattfinden. Diese zentralen Domänenereignisse stellen die Grenze des Bounded Contexts da.

Es erfolgt eine Klassifikation der Domänenereignisse, die sich in der Relationensicht befinden. Dabei wird ein Domänenereignis als zentrales Domänenereignis gekennzeichnet, sobald es eine bedeutende Änderung des Zustands eines Domänenobjekts beschreibt [Tu19]. In der Relationensicht finden sich meist an diesen Domänenereignissen Beziehungen zu mehreren Domänenobjekten, was zeigt, dass die Zustandsänderung durch das Domänenereignis für verschiedene Domänenobjekte von Relevanz ist. Mit der Deklaration eines Domänenereignisses als zentrales Domänenereignis wird nach und nach die bisherige Relationensicht in mehrere Relationensichten unterteilt. Denn die Relationensicht wird an der Stelle des zentralen Domänenereignisses so aufgeteilt, dass lose gekoppelte Relationensichten entstehen. Jeder dieser Relationensichten werden die entsprechenden Domänenobjekte, die über das zentrale Domänenereignis verbunden waren, zugeordnet. Das Domänenereignis selbst wird dem Bounded Context zugewiesen, der dieses auch auslöst. Beispielhaft ist dies in Abbildung 6.6 anhand des DMS-Szenarios dargestellt. Das Domänenereignis "*File uploaded*" kennzeichnet, dass eine Datei dem DMS hinzugefügt wurde. Gleichzeitig kann das Hinzufügen einer neuen Datei einer neuen Version entsprechen, weshalb die Verbindung der Domänenobjekte zu dem Ereignis besteht.

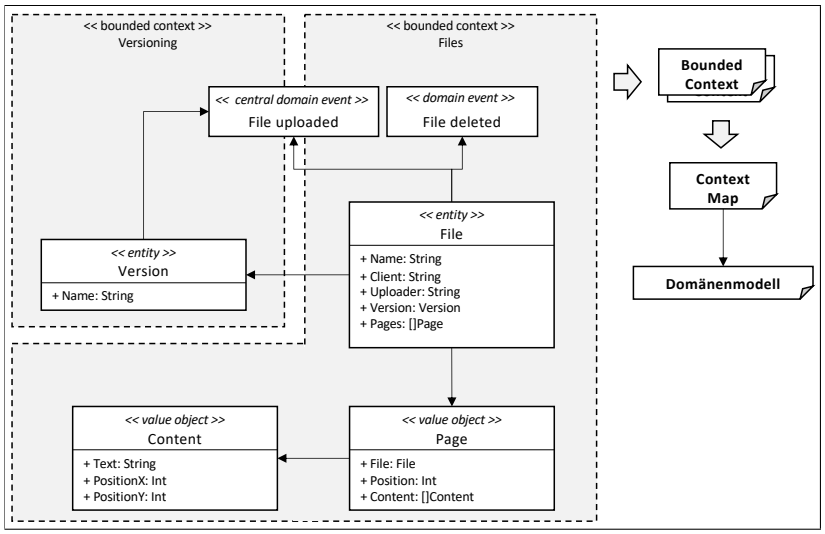

Abbildung 6.6: Identifikation eines zentralen Domänenereignisses zur Bounded Context-Bestimmung

Zwischen den entstehenden Bounded Contexts müssen zudem die Abhängigkeiten bestimmt werden, die zum einen den Informationsaustausch über das zentrale Domänenereignis kennzeichnen und zum

anderen das Machtgefüge zwischen den *Service-Teams*, die für die Microservices verantwortlich sind, definieren (Abbildung 6.7). Eine Sammlung an verschiedenen Abhängigkeitsarten liefert dabei Vernon [Ve13]. Zunächst werden die Bounded Contexts anhand des Datenflusses als *Upstream* oder *Downstream* klassifiziert. Anschließend erfolgt die Festlegung der Abhängigkeitsart. Ein Vorgehen zur Festlegung der Abhängigkeit stellen Hippchen et al. [HS+19] vor, die neben dem Datenfluss auch Abhängigkeiten zwischen den Bounded Contexts als Kommunikationswege der Service-Teams sehen.

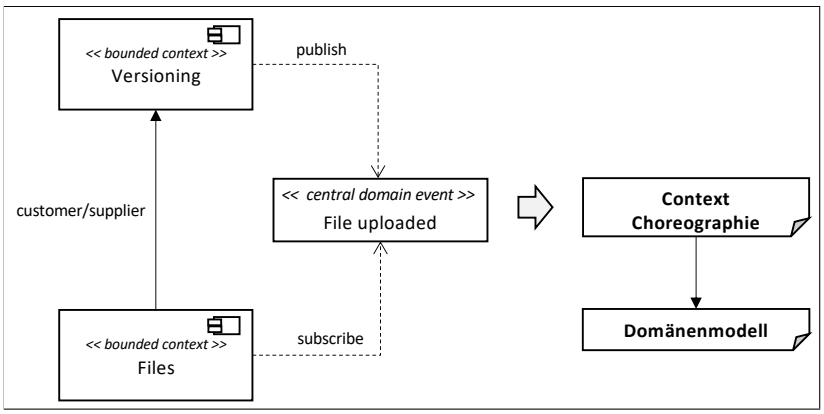

Abbildung 6.7: Klassifikation der Abhängigkeit zwischen zwei Bounded Contexts

Auch in dieser Migrationsaktivität wird für die formale Modellierung der Bounded Contexts in der Context Map auf das von Hippchen et al. [HS+19] vorgestellte UML-Profil zurückgegriffen. Das UML-Profil bietet die Möglichkeit, die identifizierten Bounded Contexts den bereits vorhandenen Subdomänen zuzuordnen und die festgelegten Relationen darzustellen. Alternativ kann auf das von Tune [Tu20b] vorgestellte *Bounded Context Canvas* zurückgegriffen werden, eine tabellarische Vorlage zur textuellen Definition eines Bounded Contexts. Das Bounded Context Canvas hat gegenüber der Context Map den Vorteil, dass es die Möglichkeit zur Beschreibung des Bounded Context bietet, während die Context Map an die Modellierungssprache gebunden ist. Die Context Map hingegen bietet den Vorteil, dass alle Bounded Contexts mit ihren Abhängigkeiten in einem formalen Modell vorliegen und die Erstellung der Context Map systematisch durch das UML-Profil erfolgen kann. Damit bleibt die Context Map das bevorzugte Mittel zur Modellierung der Bounded Contexts und die Canvas kann als Ergänzung erstellt werden.

Neben der Context Map führen Hippchen et al. [HS+19] ein weiteres Modell zum strategischen Entwurf hinzu. Mit der sogenannten *Context Choreography* (Abbildung 6.7) werden Domänenereignisse, die Bounded Context-übergreifend zum Austausch von Informationen dienen, dargestellt. Im Rahmen dieser Migrationsaktivität bietet es sich an, die Bounded Context und die ausgetauschten zentralen Domänenereignisse zu modellieren. Dadurch kann zum einen ein schneller Überblick über die kollaborierenden Bounded Contexts und zum anderen ein formales Artefakt für den systematischen Entwurf der Web-Schnittstellen geschaffen werden.

6.3.3 Überarbeitung der Context Map anhand nicht-funktionaler Anforderungen

In dieser Migrationsaktivität erfolgt die Überarbeitung der Context Map anhand nicht-funktionaler Anforderungen, die für den effizienten Betrieb der Microservices notwendig sind. Denn bislang wurde der Entwurf der Microservice-Architektur nur anhand der Domäne entschieden. Hierfür werden Problemstellungen, wichtige nicht-funktionale Anforderungen und deren Auswirkungen auf die *Context Map* aufgezeigt.

Ein domänengetriebener Entwurf einer Microservice-Architektur legt den Fokus auf die Gruppierung des Domänenwissens in Bounded Contexts anhand der Kohäsion, die durch die Domäne vorgegeben wird. Durch die hohe Kohäsion des Bounded Contexts, wird gleichzeitig auch eine hohe Autonomie des damit bezeichneten Microservices erreicht. Doch ein domänengetriebener Entwurf muss nicht gleichzeitig für den Betrieb der Microservices optimal sein. Verschiedene Problemstellungen können an dieser Stelle auftreten:

Häufige Kommunikation Wird häufig ausgeführte Funktionalität auf unterschiedliche Bounded Contexts verteilt, führt dies dazu, dass zwischen den Bounded Contexts häufig eine Kommunikation über Web-Schnittstellen stattfinden muss. Ein Aufruf einer Web-Schnittstelle bringt immer eine *Latenz* mit sich. Bei einer häufigen Kommunikation sollte diese Latenz im Sinne der Ausführungszeit reduziert werden.

Zeitkritisch Wird eine zeitkritische Funktionalität auf mehrere Bounded Contexts verteilt, führt auch hier die Kommunikation über Web-Schnittstellen zu einer Latenz, welche die Ausführungszeit beeinträchtigt. Die Latenz sollte im Sinne der Ausführungszeit vermieden werden.

Geringer Umfang Ein Bounded Context mit geringem Umfang weist eine geringe und einfach zu beherrschende Komplexität auf. Doch für die gesamte Microservice-Architektur erhöht dieser Bounded Context die Gesamtkomplexität. Hierzu gehört die *Verteilung* (engl. deployment), der *Betrieb* (engl. operations) und die *Team-Struktur*, denn der entstehende Bounded Context benötigt ein eigenes Service-Team, das organisiert werden muss. Die Gesamtkomplexität sollte an dieser Stelle reduziert werden.

Hohe Änderungsrate Innerhalb eines Bounded Contexts kann sich das Domänenwissen durch eine unterschiedliche Änderungsrate voneinander unterscheiden, auch wenn eine hohe Kohäsion vorliegt. Um ein konstantes Verhalten des Domänenwissens mit der geringen Änderungsrate zu gewährleisten, sollte eine Trennung zu dem volatilen Domänenwissen vorgesehen werden.

Hohe Gesamtkomplexität Eine hohe Anzahl an Bounded Contexts steht für eine hohe Anzahl an Microservices. In Kombination mit der Skalierbarkeit der Microservices führt das zu einer hohen Gesamtkomplexität des Deployments und des Betriebs. Weiterhin muss beachtet werden, dass durch die hohe Anzahl der skalierten Microservices die ausführenden Hardwarebausteine

einer hohen Last unterliegen [UN+16]. Für eine bessere Ausnutzung der Hardwarebausteine sollte die Gesamtkomplexität reduziert werden.

Anhand dieser Problemstellungen wird die Evaluation der domänengetriebenen Microservice-Architektur durchgeführt. Um eine systematische Evaluation zu gewährleisten, werden die Problemstellungen der nicht-funktionalen Anforderung der *Leistungseffizienz* (engl. performance effiency) gegenübergestellt, die im Rahmen des *ISO-25010-Standards* [ISO-25010] definiert wird. Die Leistungseffizienz setzt sich weiterführend durch drei Charakteristiken zusammen: (1) *Zeitverhalten*, (2) *Ressourcenauslastung* und (3) *Kapazität*. Alle drei Charakteristiken müssen von dem Migrationsteam zur Evaluation des vorliegenden Entwurfs berücksichtigt werden. Bestehende Arbeiten zeigen, dass für den Entwurf einer Microservice-Architektur auch weitere nicht-funktionale Anforderungen wie *Skalierbarkeit*, *Fehlertoleranz* oder auch *Interoperabilität* von hoher Relevanz sind [Ne15, BW+15, GK+16, BP19], doch diese nicht-funktionalen Anforderungen werden grundlegend durch den Architekturstil der Microservice-Architektur gefördert. Dagegen ist die Berücksichtigung der Leistungseffizienz nicht direkt forciert.

Die Leistungseffizienz muss dabei aus zwei Blickwinkeln betrachtet werden, die Ebert [Eb97] im Kontext der nicht-funktionalen Anforderungen als *benutzerorientierte* (engl. user-oriented viewpoint) und *entwicklungsorientierte Sicht* (engl. development-oriented viewpoint) bezeichnet. Das Unterscheiden der beiden Sichten ist für die Berücksichtigung der Charakteristiken der Leistungseffizienz relevant, denn während das (1) Zeitverhalten die funktionalen Anforderungen der Benutzer betrifft, sind die (2) Ressourcenauslastung und die (3) Kapazität mehr durch die Entwicklungsabteilung und insbesondere die DevOps-Engineers beeinflusst. Dementsprechend muss das Migrationsteam den Entwurf mit den Interessenvertretern und dem Entwicklungsteam unter den obigen Gesichtspunkten diskutieren.

Entscheidungshilfen für das Migrationsteam sind dabei *Best Practices-Muster* [NM+16, TL+18] und *Anti-Muster* (engl. anti-pattern) [Al15, Ri16, KM+17, TL+20], die sich im Kontext der Microservice-Architekturen etabliert haben. Zu beachten ist jedoch, dass die Evaluation nur die vorhandenen Bounded Contexts und damit die domänennahen Microservices betrifft. Technische Aspekte einer Microservice-Architektur wie *API-Gateways* oder *Load Balancer*, die zu den Best Practice-Mustern gehören, sind in diesem Entwurf nicht relevant, denn der Fokus liegt weiterhin auf der zu erbringenden Domäne des Softwaresystems.

Als Migrationsartefakte werden in dieser Migrationsaktivität die Anforderungsspezifikation und die Context Map herangezogen. Gherkin Features innerhalb der Anforderungsspezifikation müssen mit den Interessenvertretern im Hinblick auf die Leistungseffizienz diskutiert werden, um feststellen zu können, ob funktionale Anforderungen ein bestimmtes *Zeitverhalten* aufweisen müssen. Ist dies der Fall, wird das notwendige Zeitverhalten in Form von *Control Cases* spezifiziert. Die Context Map gibt einen Überblick über die Bounded Contexts und damit über die domänengetriebenen Microservices.

Entsprechend der durchgeführten Änderungen im Rahmen dieser Migrationsaktivität wird die Context Map angepasst.

6.3.4 Entwurf der Web-Schnittstellen

Nachdem nun der Entwurf der Microservice-Architektur feststeht, erfolgt mit dieser Migrationsaktivität der Entwurf der Web-Schnittstellen (engl. Application Programming Interface, API) zur Gewährleistung der *Interoperabilität* der entworfenen Microservices.

Durch die *strategische Modellierung* wird der Monolithenausschnitt durch mehrere Microservices substituiert. Daraus folgt ein Bedarf zum Entwurf neuer Web-APIs, die vor allem die Kommunikation zwischen den Microservices selbst sicherstellen. Aber auch die Web-APIs, die im Rahmen der Extraktionsphase zwischen Monolith und Monolithenausschnitt definiert wurden, müssen in dieser Migrationsaktivität erneut betrachtet und gegebenenfalls an die überarbeiteten Anforderungen und die Softwarearchitektur angepasst werden. Es ist wichtig, dass diese Betrachtung erfolgt, denn die Kommunikation zu dem Monolithen muss auch nach der Modernisierungsphase weiterhin gegeben sein, um die Funktionsfähigkeit des gesamten Softwaresystems zu gewährleisten.

Zum Entwurf werden, wie auch in der Extraktionsphase, die drei Architekturstile (1) *Representational State Transfer* (REST), (2) *gRemote Procedure Call* (gRPC) (3) *ereignisgetriebene Architektur* betrachtet. Je nach Anwendungsfall ist ein bestimmter Architekturstil geeigneter als die anderen [IK20, Be20]. Zu Beginn der Migrationsaktivität werden die bestehenden Web-APIs zwischen Monolithen und Monolithenausschnitt betrachtet. Diese Web-APIs müssen zum einen anhand der neuen Anforderungen, der *taktischen Modellierung* und der Microservice-Architektur angepasst werden. Denn sowohl durch die Anforderungen als auch durch die taktische Modellierung können sich die Domänenobjekte, welche über die Web-APIs ausgetauscht werden, verändert haben. Zu beachten sind Bezeichnungen von Domänenobjekten und auch strukturelle Änderungen an den Domänenobjekten selbst wie das Hinzufügen neuer Attribute. Zum anderen muss die Entscheidung getroffen werden, welche der Microservices die Web-APIs des Monolithenausschnitts implementieren, damit die Kommunikation zum Monolithen weiterhin gegeben ist. Anschließend müssen ausgehend von den Anforderungen und der Relationensicht die Web-APIs für die Kommunikation zwischen den Microservices entworfen werden. Die Anforderungen stellen dabei die *verhaltensspezifischen Aspekte* der Kommunikation dar, während die Relationensicht die *strukturspezifischen Aspekte* wie die Eigenschaften der Domänenobjekte preisgeben.

Die formale Modellierung der Web-APIs als Migrationsartefakte hängt zunächst von dem gewählten Architekturstil ab. Die bereits in der Extraktionsphase eingeführten Modelle (vgl. Abschnitt 5.5.2) finden auch hier Anwendung. Eine REST-API wird durch die *OpenAPI-Spezifikation* und *Swagger*

dokumentiert, gRPC wird weiterhin durch ein *ProtoBuffer* festgehalten und die ereignisgetriebene Architektur anhand eines UML-Klassendiagramms oder einer *Context Choreography* spezifiziert.

6.4 Reorganisation des Entwicklungsteams

Nachdem nun der Entwurf der Microservice-Architektur festgelegt wurde, erfolgt eine Reorganisation des Entwicklungsteams. Die Reorganisation sieht die Anpassung des monolithischen Entwicklungsteams an die neuen Bedürfnisse der Microservice-Architektur so vor, dass für jeden Microservice ein *Service-Team* etabliert wird. Hierfür werden zunächst *Teamabhängigkeiten* festgelegt, welche das Mächteverhältnis zwischen den Service-Teams bestimmt. Anschließend werden aus dem monolithischen Entwicklungsteam Softwareentwickler entsprechend ihrer fachlicher Fähigkeiten und Qualifikationen einem Service-Team zugewiesen.

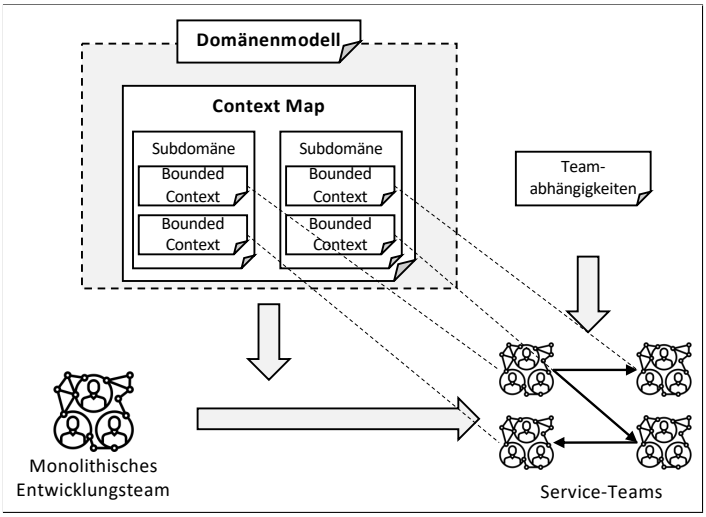

Abbildung 6.8: Organisation des Entwicklungsteams entlang der Bounded Contexts und entsprechend der Teamabhängigkeiten

Die Reorganisation des monolithischen Entwicklungsteams ist ebenso wichtig für die Etablierung einer Microservice-Architektur wie die Auftrennung des Monolithenausschnitts in Microservices selbst. Denn um eine Skalierbarkeit im Sinne der Weiterentwicklung des Softwaresystems zu gewährleisten, muss auch das Entwicklungsteam von einer beherrschbaren Komplexität sein [HS17]. Verfolgt wird die durch Newman [Ne15] beschriebene Abbildung eines Service-Teams auf einen Microservice, was dem umgekehrten Gedanken eines *Conway's Law* [Co68] entspricht. Bei der Zuordnung der Teammitglieder zu einem Service-Team wird auf divers verteilte Fähigkeiten und Rollen geachtet, sodass das gesamte notwendige Wissen zur Betreuung innerhalb dieses Service-Teams vorhanden ist. Die Fähigkeiten und Rollen werden durch Bass et al. [BW+15] festgelegt.

Hippchen et al. [HS+19] greifen die Abbildung von dem Microservice auf ein Service-Team auf und verknüpfen die *Context Map* mit der Verwaltung des Service-Team-getriebenen Entwicklungsteams. Aus der Verwendung eines formalen Modells zur Verwaltung der neuen autonomen Service-Teams folgt ein Überblick, der eine Vielzahl von Service-Teams beherrschbarer gestaltet. Weiterführend sehen Hippchen et al. *Teamabhängigkeiten* als weiteren Bestandteil der Context Map. Die Teamabhängigkeiten basieren auf Mustern von Beziehungen zwischen Bounded Contexts, die von Vernon [Ve13] im Kontext von DDD vorgestellt wurden.

6.4.1 Betrachtung der Teamabhängigkeiten

Das Ziel dieser Migrationsaktivität ist die Betrachtung der *Teamabhängigkeiten* zwischen den Service-Teams, die später für die jeweiligen Microservices verantwortlich sind. Hierzu wird die *Context Map* herangezogen und um Beziehungen zwischen den Bounded Contexts erweitert. Die Klassifikation der Beziehungen basiert auf den von Vernon [Ve13] vorgestellten Beziehungstypen.

Die Beziehungen zwischen den Bounded Contexts repräsentieren verschiedene Kommunikationsaspekte der Microservice-Architektur. Im Ursprung kann die Beziehung zwischen den Bounded Contexts als Web-Schnittstelle zwischen zwei Microservices verstanden werden [Ne15]. Um verschiedene Machtverhältnisse zwischen den Bounded Contexts beziehungsweise den Microservices darstellen zu können, führt Vernon [Ve13] verschiedene Beziehungstypen ein (Tabelle 6.1). Vernon definiert für die Kommunikation die zwei Kommunikationsteilnehmer *Upstream* und *Downstream*. Der sogenannte Upstream stellt die Schnittstelle zur Verfügung. Konsumiert wird die Schnittstelle durch den Downstream. Je nach Beziehungstyp werden den Kommunikationsteilnehmern unterschiedliche Einflussmöglichkeiten auf die Art und Weise der Kommunikation gegeben. So ist beispielsweise bei der Beziehung "*Kunde/Lieferant*" der Downstream befähigt, die Schnittstellenspezifikation entsprechend der eigenen Bedürfnisse anzupassen.

Hippchen et al. [HS+19] zeigen, dass in der Context Map die Beziehungen zwischen den Bounded Contexts nicht nur technisch zu interpretieren sind, sondern auch organisatorisch betrachtet werden müssen. Denn aus der Verknüpfung von Microservice zu Bounded Context und Microservice zu Service-Team folgt ebenfalls die transitive Verknüpfung Bounded Context zu Service-Team. So werden, Hippchen et al. zufolge, die Beziehungen zum einen als Teamabhängigkeiten und zum anderen als Kommunikationswege verstanden. Das dadurch offenbarte Machtgefüge zwischen den beteiligten Service-Teams legt in Diskussionen eindeutig fest, welches Service-Team maßgeblich die Anforderungen an die Schnittstellen zwischen den Microservices spezifiziert. Unter den Kommunikationswegen werden klare Verantwortlichkeiten mit festen Ansprechpartnern verstanden, sodass bei Änderungen oder Weiterentwicklungen an den Microservices die richtigen Parteien in die Diskussionen inkludiert werden.

Tabelle 6.1: Ausschnitt der wichtigsten Beziehungstypen zwischen Bounded Contexts nach [Ve13] und [HS+19]

Beziehungstyp	**Beschreibung**	**Aufwand**
Partnerschaft	Kooperation zwischen zweier Bounded Contexts zur Fehlerminimierung.	Sehr hoch
Geteilter Kernel	Explizit geteilte Funktionalität (beispielsweise als Quellcode in einer Library) zweier Bounded Contexts.	Mittel bis hoch
Kunde/Lieferant	Ein Bounded Context (Lieferant) stellt Funktionalität zur Verfügung, die jedoch durch den konsumierenden Bounded Context (Kunden) maßgeblich beeinflusst wird.	Hoch bis sehr hoch
Konformist	Ein Bounded Context (Konformist) konsumiert die Funktionalität eines Bounded Context, ohne diesen beeinflussen zu können.	Gering bis mittel
Getrennte Wege	Es findet keine Kooperation zwischen den beiden Bounded Contexts statt.	Gering
Antikorruptionsschicht	Explizite Schicht zwischen zwei Bounded Contexts, welche für Translationen von Datenstrukturen zuständig ist.	Gering

Im Rahmen der Modernisierung wird die Context Map nun weiterführend auch als Modell zur Verwaltung der neuen und komplexeren Entwicklungsteamstruktur verwendet. Daher werden die Beziehungen zwischen den Bounded Contexts aus der Sicht der Service-Teams klassifiziert, um dem Verwaltungszweck der Context Map gerecht zu werden. Die technischen Web-Schnittstellen werden bereits durch Migrationsartefakte wie der *Context Choreography* abgebildet. Die Context Map bietet somit auch einen schnellen Überblick über die bestehenden Service-Teams, die Abhängigkeiten, die zwischen diesen bestehen, und über gelebte Kommunikationswege. Für die Organisation ist es wichtig, dass die Kommunikationswege durch die Organisation begünstigt werden. In diesem Zusammenhang kann auf *Conway's Law* [Co68] verwiesen werden, das die Kommunikationswege der Organisation mit der Softwarearchitektur des Softwaresystems verbindet.

Die Vorgehensweise dieser Migrationsaktivität sieht zunächst die Klassifikation der Bounded Contexts als Upstream oder Downstream vor. Dabei wird für jede der bestehenden Beziehungen eine solche Klassifikation durchgeführt. Die Entscheidung, ob ein Bounded Context im Kontext der jeweiligen Beziehung den Upstream oder den Downstream repräsentiert, wird anhand des Kommunikationsflusses der Web-Schnittstelle getroffen. Aus diesem Grund wurden bereits die Web-Schnittstellen entworfen (vgl. Abschnitt 6.3.4).

Ausgehend von den klassifizierten Kommunikationspartnern muss anschließend der Beziehungstyp zwischen den Bounded Contexts festgelegt werden. Vernon [Ve13] liefert zu seinen vorgestellten Beziehungstypen gleichzeitig eine Handlungsempfehlung dazu, welcher Beziehungstyp in verschiedenen Szenarien eingesetzt werden sollte. Ergänzend führen Hippchen et al. [HS+19] zu den Beziehungstypen noch einen *Aufwandsindikator* ein (Tabelle 6.1). Dieser Indikator gibt Aufschluss darüber, wie hoch der Aufwand für die Einhaltung der Teamabhängigkeit und der Kommunikationswege ist. Der Aufwandsindikator kann ebenfalls für die Entscheidung bezüglich des Beziehungstyps herangezogen werden.

Das hauptsächliche Migrationsartefakt dieser Migrationsaktivität ist die Context Map. Das UML-Profil von Hippchen et al. [HS+19] sieht bereits die Modellierung der unterschiedlichen Beziehungstypen vor. Dadurch, dass das UML-Profil auf gerichtete Kanten setzt, kann der Kommunikationsfluss und damit die Upstream- und Downstream-Klassifikation anhand der Pfeilrichtung modelliert werden.

6.4.2 Etablierung der Service-Teams

In der letzten Migrationsaktivität der Modernisierungsphase werden abschließend die *Service-Teams* etabliert. Die Anzahl der Service-Teams richtet sich nach den Bounded Contexts beziehungsweise Microservices, die in der Context Map vorzufinden sind. Den Service-Teams werden die Entwickler des bestehenden monolithischen Softwaresystems nach *funktionaler Position* und *fachlicher Expertise* zugeordnet.

Die Notwendigkeit der Etablierung der Service-Teams beruht auf *Conway's Law* [Co68]. Dieses besagt, dass eine Organisation stets den Entwurf der Softwarearchitektur für das eigene Softwaresystem vorsieht, der genau die organisatorischen Strukturen widerspiegelt. Ein Beispiel hierfür ist gerade das Entwicklungsteam eines monolithischen Softwaresystems, das meist in Frontend- und Backend-Team strukturiert ist, wie auch der Monolith selbst. Für die Microservice-Architektur ist es daher wichtig, dass das monolithisch geprägte Entwicklungsteam an die neue Softwarearchitektur im Sinne des *Conway's Law* angepasst wird. Dies verhindert zudem eine Architekturerosion, die durch fehlende Struktur im Entwicklungsteam entstanden wäre. Newman [Ne15] greift im Kontext der Microservice-Architektur ebenfalls *Conway's Law* auf und bezeichnet die Reorganisation des Entwicklungsteams als *Conway's Law in Reverse*.

Einen weiteren Grund für die Etablierung von Service-Teams sehen Hasselbring und Steinacker [HS17] in der Skalierungsfähigkeit des Entwicklungsteams. Autonome und kleine Entwicklungsteams sind agiler und flexibler in der Weiterentwicklung des zugrundeliegenden Microservices. Zudem sinkt die Verwaltungs- und Entwicklungseffizienz eines monolithischen Entwicklungsteams bei steigender Entwickleranzahl. Denn durch die hohe Autonomie der Microservices sind die Service-Teams in der Lage losgelöst von anderen Service-Teams den Microservice weiterzuentwickeln [TV+17]. Lediglich Entscheidungen auf einem hohen Abstraktionsniveau, die vor allem die Web-Schnittstellen betreffen, müssen koordiniert werden. Die Abhängigkeiten, die darüber entscheiden mit welchen Service-Teams diskutiert werden muss, können der Context Map entnommen werden.

Zur Etablierung der Service-Teams wird zunächst ausgehend von der Context Map und den darin befindlichen Bounded Contexts die Anzahl der notwendigen Service-Teams festgelegt. Jeder Bounded Context resultiert in ein Service-Team. Bei der Etablierung des Service-Teams gilt es, verschiedene Faktoren zu beachten: (1) *fachliche Ausrichtung*, (2) *Erfahrung* und (3) *Interessenkonflikt*.

Tabelle 6.2: Rollen eines Service-Teams nach [BW+15]

Rolle	**Beschreibung**
Teamleitung	Verwaltet die Ressourcen des Service-Teams und verteilt gegebenenfalls Aufgaben.
Teammitglied	Übernimmt die typischen Entwicklungsaufgaben wie Analyse, Entwurf, Implementierung und Testen.
Besondere Rollen	
Service Owner	Verantwortet den Microservice und ist für die Kommunikation mit Externen zuständig. Zudem diskutiert er die allgemeinen Anforderungen an den Microservice mit dem restlichen Service-Team und gestaltet aktiv die Vision mit.
Reliability Engineer	Befasst sich mit dem Betrieb des Microservices und ist weiterhin auch für die Überwachung von Metriken bezüglich der Laufzeit verantwortlich. Außerdem ist der Reliability Engineer der erste Ansprechpartner bei Störungen im Betrieb.
Gatekeeper	Übernimmt die Freigabe neuer Versionen eines Microservices, die letztendlich dem Benutzer zur Verfügung gestellt werden. Eine primäre Tätigkeit ist dabei das Testen.
DevOps Engineer	Betreibt die DevOps relevanten Werkzeuge und integriert den Microservice in diese.

Diese drei Faktoren beziehen sich auf die Teammitglieder, die dem Service-Team zugewiesen werden sollen. Die fachliche Ausrichtung bezieht sich auf die Rolle, die das Teammitglied einnehmen wird. Die Rollen innerhalb eines Service-Teams basieren dabei auf den von Bass et al. [BW+15] vorgestellten Rollen (Tabelle 6.2). Bruce und Pereira [BP19] sprechen an dieser Stelle von *Cross-Functional Teams*.

In Kombination zu den notwendigen Rollen ist es weiterhin wichtig, dass die *Erfahrung* des jeweiligen Teammitglieds betrachtet wird. Das Ziel ist es, in den verschiedenen Rollen unterschiedliche Erfahrungsgrade abzubilden. So ist es möglich, Aufgaben anhand ihrer Komplexität den verschiedenen Teammitgliedern zuzuweisen und gleichzeitig ein Mentoring für unerfahrene Teammitglieder zu gewährleisten. Weiterhin können durch *Review-Gespräche* zwischen erfahrenen Softwareentwicklern und weniger erfahrenen Softwareentwicklern Schwachstellen in der Implementierung festgestellt und behoben werden.

Der *Interessenskonflikt* beschreibt einen zentralen Faktor bei der Etablierung der Service-Teams und der Festlegung der jeweiligen Teammitglieder. Ein Interessenskonflikt entsteht, sobald ein Teammitglied, unabhängig von der fachlichen Ausrichtung und Erfahrung, in zwei oder mehr Service-Teams vertreten ist. Somit hat das Teammitglied die Verantwortung für zwei oder mehr Microservices. Betrifft eine Änderung oder neue Anforderung ein solches Teammitglied, besteht die Möglichkeit, dass eine technische Entscheidung getroffen wird, die zwar die schnellste und einfachste Lösung repräsentiert, aber konträr zu den Prinzipien der Microservice-Architektur steht. Insbesondere ist der Interessenkonflikt bei hochrangigen Teammitgliedern aufgrund ihrer Entscheidungsgewalt kritisch zu

bewerten. Bruce und Pereira [BP19] sehen einen solchen Interessenkonflikt, auch als *1:n-Ownership* bezeichnet, als Anti-Muster für die Microservice-Architektur an. Doch auf dieses Anti-Muster muss dann zurückgegriffen werden, sollte die Organisation nur über wenige Softwareentwickler verfügen. In diesem Fall muss bei der Etablierung der Service-Teams darauf geachtet werden, dass ein Teammitglied nicht in zwei unmittelbar abhängigen Service-Teams leitend vertreten ist. Die Context Map muss an dieser Stelle herangezogen werden.

Neben den Faktoren, welche die Wahl der einzelnen Teammitglieder für das Service-Team beeinflussen, existieren weitere Faktoren, die das gesamte Service-Team betreffen. Newman [Ne15] hält fest, dass die *Geolokalisierung* der Teammitglieder innerhalb des Service-Teams für die Effektivität und Effizienz der Kommunikation ein wichtiger Faktor ist; daher sollten alle Teammitglieder derselben Geolokalisierung zugeordnet sein. Ein weiterer Faktor eines Service-Teams ist die Teamgröße, die Bass et al. [BW+15] zufolge möglichst klein gewählt werden sollte. Ein Richtwert stellt dabei das von Amazon vorgestellte *Two-Pizza-Team* dar, welches aus der Anzahl an Teammitgliedern besteht, die durch zwei Pizzen gesättigt werden können.

Zur Wahrung der Übersicht der einzelnen Service-Teams kann die Context Map entsprechend mit Notationen der UML erweitert werden. So bietet es sich an, durch Notizelemente an den Bounded Contexts die jeweiligen Service-Teams zu dokumentieren.

6.5 Zusammenfassung

Mit diesem Kapitel wurde ein detaillierter Entwurf der *Modernisierungsphase* der Systematik des Migrationsrahmenwerks umgesetzt. Dabei wurden die Problemstellungen P4 und P5 (vgl. Abschnitt 1.4) behandelt. Ein zentraler Teil dieses Kapitels ist der eigene Forschungsbeitrag B3 (vgl. Abschnitt 1.5.4), der eine systematische Identifikation von Microservices während einer Migrationsiteration verfolgt. Zur Erreichung des Forschungsbeitrags wurden Migrationsaktivitäten eingeführt, die sequentiell durchlaufen werden müssen. Zudem wurden formale Migrationsartefakte vorgestellt, welche die Entscheidungen durch das Migrationsteam nachvollziehbar gestalten und sowohl die Analysephase als auch die Entwurfsphase der Microservice-Entwicklung abdecken.

Für die konkreten Migrationsaktivitäten der Modernisierung wurden verschiedene bestehende Ansätze in den Entwurf aufgenommen. So wurde für die Überarbeitung der Anforderungsspezifikation, wie auch in der Extraktionsphase, auf *BDD* [Sm14] zurückgegriffen. Der vom Monolithenausschnitt ausgehende Entwurf der Microservice-Architektur, basiert zunächst auf *DDD* [Ev04], um die Orientierung an der Geschäftsdomäne zu gewährleisten. Beim Ableiten der Microservices fand ein systematisches Vorgehen Beachtung, das bereits durch Hippchen et al. [HS+19] vorgestellt wurde. Die Anwendung der jeweiligen Ansätze wurde an die Bedürfnisse des Migrationsteams während der Migrationsiteration angepasst, damit auch die Migrationsartefakte, welche durch die Extraktionsphase

und die Migrationsaktivitäten der Modernisierungsphase modelliert wurden, bei der Modernisierung berücksichtigt werden.

Zu Beginn der Modernisierung wird die *Anforderungsspezifikation* für die Microservice-Architektur überarbeitet. Infolgedessen, dass die Anforderungsspezifikation ab der Modernisierungsphase für den gesamten Lebenszyklus der Microservices gültig ist, wird die Anforderungsspezifikation zunächst erweitert. Die Erweiterung sieht die Einarbeitung einer *Vision*, mehrer *Ziele* und mehrer *Fähigkeiten* [Sm14] vor. Diese dienen dem Migrationsteam zur Orientierung bei der Anpassung der vorhandenen Anforderungen, aber auch bei der Bewertung neuer Anforderungen. Anschließend werden die bestehenden Anforderungen auf Korrektheit geprüft. In Zusammenarbeit mit den Interessenvertretern überarbeitet das Migrationsteam die *Gherkin Features*. Die Änderungen an den Gherkin Features werden danach auf das Glossar und den Language Sketch übertragen. Nach der Überarbeitung der Anforderungsspezifikation und den damit verbundenen Migrationsartefakten gilt der Submeilenstein $M_{2.1}$ *Überarbeitete Anforderungsspezifikation* als erreicht.

Die nächsten Migrationsaktivitäten sehen den eigentlichen Entwurf der Microservice-Architektur vor. Anhand der überarbeiteten Anforderungsspezifikation erfolgt eine *taktische Modellierung* [Ev04], welche die feingranularen Domänenstrukturen in Form eines angepassten UML-Klassendiagramms, der *Relationensicht*, aufnimmt. Aus der feingranularen Struktur gehen die Domänenobjekte hervor, welche für die Erbringung der Geschäftsfähigkeiten innerhalb der Subdomänen zuständig sind. Ausgehend von der Relationensicht ist das Migrationsteam in der Lage, *Bounded Contexts* abzuleiten, die den Microservices entsprechen. Das Bestimmen, welche Domäneobjekte welchem Bounded Context zugeordnet werden, beruht auf den Ansätzen von Hippchen et al. [HS+19] und Tune [Tu19]. Die Frage der Kohäsion von Domänenobjekten wird dort durch die *Beziehungen* zwischen den Domänenobjekten und Domänenereignissen beantwortet. Die Bounded Context stellen den domänengetriebenen Entwurf dar, der jedoch noch durch die Betrachtung nicht-funktionaler Anforderungen überarbeitet werden muss. Jede Beziehung zwischen den Bounded Contexts wird dabei in Frage gestellt, was dazu führen kann, dass Bounded Contexts zusammengelegt werden oder ein Bounded Context in mehrere Bounded Contexts unterteilt wird. Nach der Berücksichtigung der nicht-funktionalen Anforderungen steht der finale Entwurf der Microservice-Architektur fest. Damit kann der Entwurf der Web-Schnittstellen erfolgen. Relevant sind auch hier alle Beziehungen zwischen den Bounded Contexts und die Web-Schnittstellen, die bereits durch den Monolithenausschnitt bedient werden. Die Web-Schnittstellen des Monolithenausschnitts werden nur minimal angepasst, sodass der Anpassungsaufwand auf Seiten des Monolithen gering bleibt. Bei dem Enwurf der neuen Web-Schnittstellen, die insbesondere die Kommunikation zwischen den Microservices abbilden, sind verschiedene Architekturstile in Betracht zu ziehen. Mit dem Abschluss der Schnittstellenbeschreibungen gilt der Submeilenstein $M_{2.2}$ *Entwurf der Microservice-Architektur* als erreicht.

Zum Abschluss der Modernisierung wird noch das monolithische Entwicklungsteam an die entworfene Microservice-Architektur angepasst, sodass jeder Microservice ein eigenverantwortliches

Service-Team besitzt. Eingangs werden die *Teamabhängigkeiten* bestimmt, damit bei der Weiterentwicklung der Microservices das Mächteverhältnis zwischen den Service-Teams explizit festgelegt ist. Dabei wird jede Beziehung, die innerhalb der Context Map zwischen den Bounded Contexts besteht, anhand vordefinierter *Beziehungstypen* [Ve13, HS+19] klassifiziert. Abschließend erfolgt die Aufstellung der Service-Teams. Bei der Aufteilung der Teammitglieder werden verschiedene Faktoren berücksichtigt, sodass insbesondere *Interessenskonflikte* bei dem Treffen von Entwicklungsentscheidungen minimiert werden. Durch das Etablieren der Service-Teams ist der Submeilenstein $M_{2.3}$ *Restrukturiertes Entwicklungsteam* erreicht.

7 Entwurf der Inbetriebnahmephase

Das nachfolgende Kapitel befasst sich mit der Erreichung des Meilensteins M_4 *Betriebsbereite Microservices* und den dafür in der *Inbetriebnahmephase* notwendigen Migrationsaktivitäten und -artfakten. Die Inbetriebnahmephase verfolgt das Ziel, Vorbereitungen für den Betrieb der entwickelten Microservices zu treffen und zeitgleich die neue Komplexität des Betriebs für die Service-Teams zu reduzieren. Zurückgegriffen wird hierfür auf die Prinzipien *Cloud-nativer Softwaresysteme* [GB+17, Da19], zu denen auch die *Development & Operations-Prinzipien* (DevOps) [BW+15, BH+16, BH+18, HR20] gehören. Um einen geeigneten Rahmen für diese Forschungsarbeit zu gewährleisten, beschränkt sich die vorliegende Forschungsarbeit im Kontext der Inbetriebnahmephase auf die Etablierung einiger DevOps-Prinzipien.

Das Etablieren von DevOps-Prinzipien ist mehrheitlich werkzeuggetrieben [Ba17]. Wie in der Prämisse P3 (vgl. Abschnitt 1.6.3) festgehalten, ist die Einführung der in der Inbetriebnahmephase durch Devops notwendigen Werkzeuge nicht Teil der Betrachtung. Die konkreten Werkzeuge, auf die in den einzelnen Migrationsaktivitäten verwiesen wird, werden als gegeben vorausgesetzt. Unter gegeben wird dabei die Einsatzbereitschaft beziehungsweise Lauffähigkeit der Werkzeuge verstanden, sodass die in der Inbetriebnahmephase erreichten Ergebnisse unmittelbar Verwendung finden. Ein Ergebnis referenziert dabei auf Migrationsartefakte, welche während den Migrationsaktivitäten erstellt werden. Die Migrationsartefakte werden zur Konfiguration der DevOps-Werkzeuge benötigt. Exemplarisch wird die Durchführung der Betriebsphase anhand des *Dokumentenmanagementsystem-Szenarios* (DMS-Szenario) aus Abschnitt 1.5.6 gezeigt.

Wie auch bei den anderen Migrationsphasen stellen sich für die Inbetriebnahmephase zentrale Fragestellungen: (1) *Wie kann reproduzierbar ein Microservice ausgeführt werden?* (2) *Wie kann die Bereitstellung der Microservices unterstützt werden?* Abhängig von diesen Fragestellungen werden die Migrationsaktivitäten und -artefakte der Inbetriebnahmephase gewählt. Weiterführend hängt von den Fragestellungen die Wahl der DevOps-Werkzeuge ab. Fragestellung (1) ist besonders im Hinblick auf die Bereitstellungs- und Fehlerreproduzierbarkeit von hoher Relevanz. Bei der Skalierung eines Microservices, aber auch bei der Bereitstellung in unterschiedlichen Infrastrukturen, muss gewährleistet werden, dass der gleiche Softwarebaustein instanziiert wird. Nur so können die *Service-Teams* das Verhalten des Softwarebausteins voraussagen. Dies gilt ebenfalls für die Analyse von Fehlverhalten eines Softwarebausteins in bestimmten Situationen. Die Fragestellung (2) thematisiert die eigentliche Reduktion der Komplexität bei der Bereitstellung der Microservices. An dieser Stelle

werden insbesondere die DevOps-Prinzipien bedeutsam, denn diese verfolgen einen hohen Automatisierungsgrad für die Bereitstellung der Microservices, um das Service-Team dahingehend zu entlasten [BW+15]. Eine Automatisierung wird aus verschiedenen Gründen notwendig. Zum einen sollen neue Versionen eines Microservices unabhängig und schnell zur Verfügung gestellt werden. Zum anderen werden für den Betrieb der Microservices, die grundsätzlich ein verteiltes Softwaresystem darstellen, weitere Betriebskonzepte wie *Service-Registry* oder *Service Discovery* [BH+16] notwendig. Damit ein Service-Team unabhängig von diesen Betriebskonzepten arbeiten kann, werden durch DevOps-Werkzeuge Abstraktionen geschaffen.

Der Einsatz von DevOps-Prinzipien ist nicht nur im Kontext der Microservice-Architekturen etabliert [BW+15, Ne15, BP19], sondern auch im Kontext von Migrationen in eine Microservice-Architektur [BH+16, BH+18, Ne19, HR20]. Newman [Ne19] ist der Auffassung, dass im Rahmen einer Migration die DevOps-Prinzipien explizit berücksichtigt werden müssen, um die neue Komplexität im Betrieb abfedern zu können. Die Inbetriebnahmephase dieser Forschungsarbeit greift die obigen Fragestellungen (1-2) durch den Einsatz bestimmter Konzepte auf. Zunächst wird die *Containerisierung* der Microservices [JN+16] verfolgt. Daran anknüpfend wird auf eine *Container-Orchestrierung* [Kh17] zurückgegriffen, welche die grundlegenden Betriebskonzepte wie *Service-Registry* und *Service-Discovery* für die Service-Teams vereinfachen. Abschließend wird die Bereitstellung eines Microservices durch die *Continuous Integration* (CI) und *Continuous Delivery* (CD) [BW+15] für das Service-Team automatisiert. In den Migrationsaktivitäten, die jeweils die Konzepte realisieren, werden konkrete Werkzeuge mit ihren Konfigurationsdateien vorgestellt.

Das vorliegende Kapitel ist wie folgt strukturiert: Abschnitt 7.1 vermittelt einen Überblick über die Inbetriebnahmephase und die darin vorgesehenen Migrationsaktivitäten und -artefakte. Abschließend wird in Abschnitt 7.2 die zentrale Migrationsaktivität detailliert beschrieben und Migrationsartefakte in Abhängigkeit von konkreten Werkzeugen eingeführt.

7.1 Übersicht über konkrete Migrationsaktivitäten und -artefakte

Ausgehend von der Herangehensweise für die Inbetriebnahmephase festgelegte Herangehensweise wird ein systematischer Ablauf bestimmt, der konkrete Migrationsaktivitäten in einer sequentiellen Reihenfolge definiert. Nachfolgend wird der systematische Ablauf nur noch als *Inbetriebnahme* bezeichnet. Es erfolgt weiterführend eine Zuordnung der Migrationsartefakte zu den -aktivitäten. Zum konsistenten Entwurf der Inbetriebnahme werden wie auch bei der Extraktion und der Modernisierung Submeilensteine zur Kennzeichnung von Zwischenergebnissen definiert.

7.1.1 Zugrundeliegende Herangehensweise der Inbetriebnahme

Für den Detailentwurf der Inbetriebnahme werden die bestehenden Arbeiten von Balalaie et al. [BH+18] und Henry und Ridene [HR20] zugrunde gelegt. Die Autoren der beiden Arbeiten führen *Migrationsmuster* ein, die zum einen die grundlegenden Betriebskonzepte verteilter Softwaresysteme und zum anderen die von der Inbetriebnahme angestrebte Unterstützung der *Service-Teams* bei der Bereitstellung der Microservices abdecken. Zur Ausgestaltung der Inbetriebnahme werden spezifische Migrationsmuster übernommen und durch die Migrationsaktivitäten abgebildet.

Henry und Ridene [HR20] sehen mit dem Migrationsmuster "*Warm-Up und Skalierung*" den Start in eine iterative Migration des Softwaresystems. Zu Beginn werden die Werkzeuge, welche für den Betrieb der Microservices notwendig sind, durch das Migrationsteam eingeführt und für die weitere Verwendung bereitgestellt. Die Inbetriebnahme der Systematik des Migrationsrahmenwerks grenzt sich von diesem Migrationsmuster insofern ab, dass, neben dem bereits ausgeführten Verzicht auf die Etablierung der Werkzeuge, eine klare Sequenz an Migrationsaktivitäten gegeben wird, welche für die Vorbereitung des Microservices notwendig sind. Das Migrationsmuster von Henry und Ridene dagegen zeigt lediglich den Bedarf dieser Vorbereitungen auf. Detaillierter befassen sich Balalaie et al. [BH+18] mit den Vorbereitungen und stellen feingranularere Migrationsmuster vor. So sieht das Migrationsmuster "MP_1 *Continuous Integration ermöglichen*" eine Integration der Microservices in eine *CI-Pipeline* vor. Ein weiteres wichtiges Migrationsmuster, welches den Betrieb maßgeblich beeinflusst, ist "MP_2 *Einführen eines Konfigurationsservers*". Das Migrationsmuster führt einen designierten Server zur Verteilung der Konfiguration ein. Für die Inbetriebnahme wird ein verwandter Ansatz gewählt, der jedoch nicht auf einen designierten Server zurückgreift, sondern hierfür ein *Container-Orchestrierungswerkzeug* verwendet. Siehe hierzu auch das Migrationsmuster "MP_{14} *Bereitstellung in einer Container-Orchestrierung*". Ein zentrales Migrationsmuster der Autoren, welches in der Inbetriebnahme übernommen wird, ist "MP_{13} *Containerisierung der Microservices*". Durch die *Containerisierung* ist die Reproduzierbarkeit der Microservice-Instanzen gegeben.

Die Migrationsaktivitäten, die für die Inbetriebnahme entworfen werden, berufen sich somit teilweise auf bestehende Migrationsmuster und bringen diese in eine sequentielle Form. Weiterführend muss festgehalten werden, dass die Migrationsaktivitäten die Vorbereitung der Microservices feingranularer betrachten als die obigen Migrationsmuster. Zudem werden die Vorbereitungen für den Betrieb als letzte Migrationsphase der Migrationsiteration ausgeführt, während Henry und Ridene [HR20] diese Aspekte zu Beginn einer Migrationsiteration sehen.

7.1.2 Ganzheitlicher Entwurf der Inbetriebnahme

Dem abstrakten Entwurf der Inbetriebnahmephase aus dem generellen Migrationsrahmenwerk zufolge besteht diese nur aus der abstrakten Migrationsaktivität (1) *Vorbereitungen zur automatisierten Bereit-*

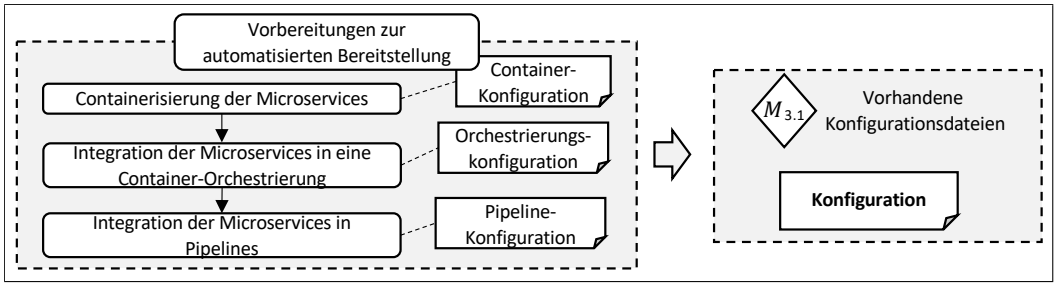

Abbildung 7.1: Übersicht der Migrationsaktivitäten und -artefakte der Inbetriebnahmephase

stellung. Zur Vorbereitung der Microservices für den automatisierten Betrieb werden die drei konkreten Migrationsaktivitäten *Containerisierung der Microservices*, *Integration der Microservices in eine Container-Orchestrierung* und *Integration der Microservices in Pipelines* eingeführt (Abbildung 7.1). Durch die *Containerisierung* wird die Grundlage für ein reproduzierbares und vorhersagbares Bereitstellen der Microservices gelegt, indem durch eine *Container-Konfiguration* ein ausführbares *Container-Image* eines Microservices erstellt wird. Um die Komplexität während der Bereitstellung und der Ausführung der neuen Microservices beherrschen zu können, wird nach der Containerisierung eine Integration in eine *Container-Orchestrierung* vorgesehen. Die Container-Orchestrierung nimmt dem Migrationsteam die Aufgabe ab, die grundlegenden Konzepte eines verteilten Softwaresystems realisieren zu müssen. Solche Konzepte werden von Balalaie et al. [BH+18] als eigene Migrationsmuster vorgesehen. Eine *Orchestrierungskonfiguration* wird für jeden einzelnen Microservice zur Integration in die Container-Orchestrierung erstellt. Anschließend wird die Orchestrierungskonfiguration bei der Integration der Microservices in eine *Continuous Integration/Continuous Delivery-Pipeline* (CI/CD) benötigt, welche die Microservices automatisiert bereitstellt. Auch für die Integration der Pipeline wird eine Konfigurationsdatei erstellt, die im Kontext des Betriebs als *Pipeline-Konfiguration* bezeichnet wird.

7.2 Vorbereitungen zur automatisierten Bereitstellung

Nachfolgend wird die durch die Systematik des Migrationsrahmenwerks vorgegebene Migrationsaktivität anhand konkreter Migrationsaktivitäten und -artefakte detailliert beschrieben. Das Ziel der Migrationsaktivitäten ist das Treffen von Vorbereitungen, um die in der Migrationsiteration implementierten Microservices automatisiert bereitstellen zu können.

Die Containerisierung der Microservices basiert auf dem Einsatz des Werkzeugs *Docker* [Doc-Ref]. Mit Docker ist das Migrationsteam in der Lage, *Container-Images* zu erstellen, die sich jederzeit und reproduzierbar ausführen lassen. Ein solches Container-Image ist die Grundlage für die Verwendung einer *Container-Orchestrierung*. Im Kontext dieser Forschungsarbeit wird *Kubernetes* (K8s) [Lin-Kub] als Werkzeug verwendet. Das Container-Image eines Microservices wird durch Kubernetes

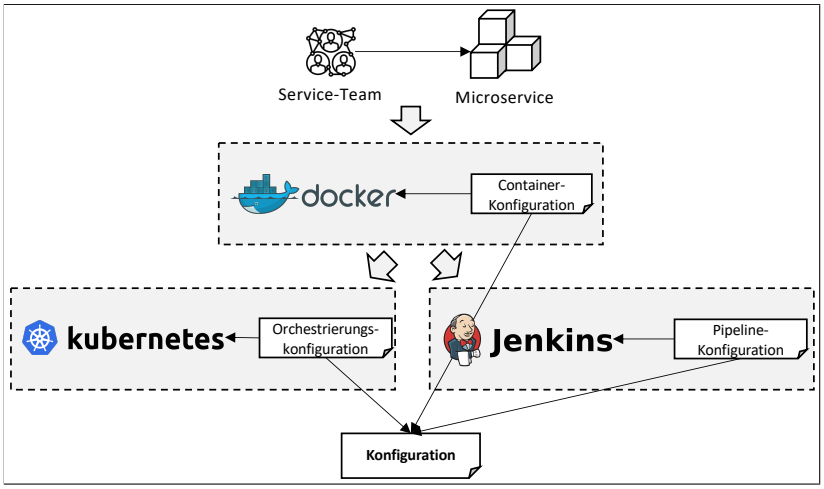

Abbildung 7.2: Erstellen der Konfigurationsdateien zur Vorbereitung eines Microservices

ganzheitlich verwaltet. Hierzu zählt auch die Skalierung des Microservices, was im Grunde dem mehrmaligen Starten desselben Container-Images entspricht. Damit das Erstellen des Container-Images und die Auslieferung in die Container-Orchestrierung automatisiert abläuft, wird als *CI/CD-Pipeline* das Werkzeug *Jenkins* verwendet. Jenkins greift auf die Container-Konfiguration zurück, erstellt das neue Container-Image und lässt dieses durch die Container-Orchestrierung bereitstellen.

7.2.1 Containerisierung der Microservices

Die Bereitstellung eines Microservices als lauffähiger Softwarebaustein wird zunächst durch das Mittel der *Containerisierung* vereinfacht. Folglich ist das Ziel dieser Migrationsaktivität, die Containerisierung der einzelnen Microservices zu erreichen. Dabei muss jedes Service-Team für die Microservices eine eigene Container-Konfiguration erstellen.

Den Nutzen der Containerisierung haben bereits Balaleie et al. [BH+18] durch das Migrationsmuster "MP_{13} *Containerisierung der Microservices*" hervorgehoben. In der vergangenen Zeit hat sich gezeigt, dass die Containerisierung einen großen Einfluss auf den gesamten Entwicklungsprozess von Microservices hat. Softwareentwickler sind nicht mehr gezwungen, Laufzeitumgebungen spezifischer Programmiersprachen lokal auf der eigenen Hardware laufen zu lassen, sondern sind durch die Containerisierung in der Lage, hardwareunabhängig eine konstante Entwicklungsumgebung zu schaffen. Doch neben der Art, wie Microservices entwickelt werden, beeinflusst die Containerisierung insbesondere den Betrieb. Microservices können beliebig oft als laufende Container auf beliebiger Hardware mit reproduzierbarem Verhalten gestartet werden. Um sich genau dies während einer Migrationsiteration zunutze machen zu können, sieht die Inbetriebnahme die Erstellung einer *Container-Konfiguration* vor.

Die Migrationsaktivität sieht vor, dass für jeden neuen Microservice eine Container-Konfiguration erstellt wird. Wie diese Container-Konfiguration zu gestalten ist, steht in Abhängigkeit zu dem gewählten Werkzeug. Durchgesetzt hat sich im Kontext der Microservices das Werkzeug *Docker*. Über ein sogenanntes *Dockerfile* [Doc-Ref] konfiguriert das Service-Team den lauffähigen Container, der auch als *Docker-Image* bezeichnet wird.

```
1  # Building Stage
2  FROM golang:1.15.2 AS BUILDER
3
4  WORKDIR /go/src
5
6  COPY ./src .
7  RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
8
9  # Production Stage
10 FROM alpine:latest AS PRODUCTION
11
12 WORKDIR /root/
13 COPY --from=BUILDER /go/src/app .
14
15 RUN groupadd -g 3333 appGroup
16 RUN useradd -u 3333 -g appGroup -s /bin/sh -m appUser
17
18 USER 3333:3333
19
20 CMD ["./app"]
```

Quelltext 7.1: Beispielhaftes Dockerfile eines Microservices

Für eine effiziente Verwendung des Docker-Images müssen verschiedene Prinzipien in dem Dockerfile berücksichtigt werden. Diese werden anhand eines beispielhaften Dockerfiles (Listing 7.1) aufgezeigt. Aus Gründen der Speichergröße eines Docker-Images sollte auf einen *mehrstufigen Bauprozess* von Docker [Doc-Use] zurückgegriffen werden. Dadurch wird die Speichergröße des Images minimiert, was sich positiv auf die Startzeiten des Images auswirkt. Konkret bedeutet dies, dass der Microservice bei einer geringen Speichergröße schneller zur Verfügung steht. Gerade im Kontext des Lastausgleiches durch Skalierung ist eine schnelle Verfügbarkeit neuer Microservice-Instanzen kritisch. Im Quelltext 7.1 wird dies durch die Zeilen 2, 10 und 13 erreicht. Weiterhin ist die Erstellung und Verwendung eines expliziten Benutzers im Docker-Image aus Gründen der Sicherheit notwendig. Standardmäßig werden in einem Docker-Image alle Interaktionen durch den Standardadmin ausgeführt. Im Falle einer Kompromittierung des laufenden Docker-Containers ist ein Administratorzugriff möglich. Durch die Zeilen 15, 16 und 18 des Listings 7.1 wird ein expliziter Nutzer ohne Administrationsrechte erstellt und verwendet. So kann durch eine Kompromittierung kein weiterer Schaden entstehen.

Das durch Docker eingeführte Dockerfile wird im Kontext der Inbetriebnahme als formales Migrationsartefakt verstanden. Jeder Microservice verfügt über ein eigenes Dockerfile.

7.2.2 Integration der Images in eine Container-Orchestrierung

Microservice-basierte Softwaresysteme, die aus einer Vielzahl von Softwarebausteinen bestehen, die durch eine Containerisierung als Container-Images betrieben werden, müssen durch die hohe Komplexität mittels einer *Container-Orchestrierung* verwaltet werden. Dabei übernimmt die Container-Orchestrierung, neben der Verwaltung der *Container*, auch grundlegende Konzepte eines verteilten Softwaresystems. Die Inbetriebnahme greift aufgrund der hohen Beliebtheit auf *Kubernetes* [Lin-Kub] als Container-Orchestrierung zurück. Zur Integration der Microservices in Kubernetes müssen die *Service-Teams* verschiedene Konfigurationsdateien erstellen, was das Ziel dieser Migrationsaktivität darstellt.

Durch Kubernetes werden den Service-Teams die grundlegenden Konzepte für das Betreiben eines verteilten Softwaresystems standardmäßig zur Verfügung gestellt. Alle notwendigen Konzepte können aus den Migrationsmustern von Balalaie et al. [BH+18] entnommen werden. Unter anderem ist dies eine auf dem *Domain Name System-basierte (DNS) Service Discovery*, die ausgehend von DNS-Einträgen die laufenden Container auffindbar macht.

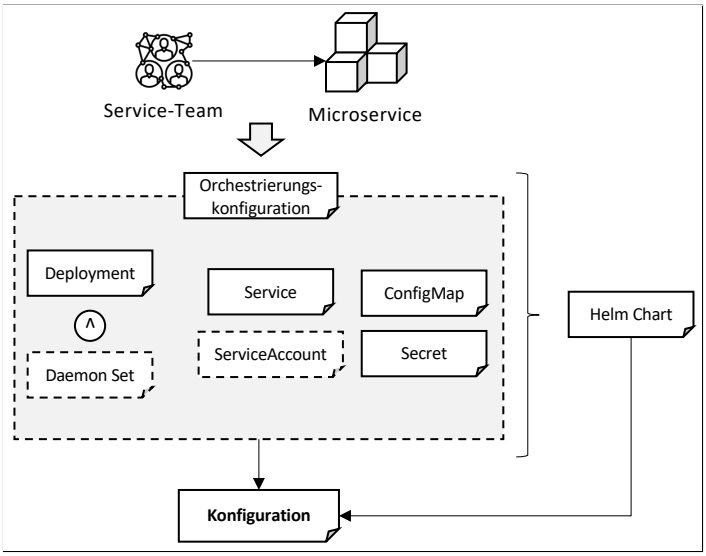

Abbildung 7.3: Konfiguration zur Integration des Microservices in Kubernetes

Um einen Microservice in Kubernetes betreiben zu können, wird auf die von Kubernetes eingeführten *Ressourcen* zurückgegriffen (Tabelle 7.1). Für jeden Microservice der Migrationsiteration müssen bestimmte Kubernetes-Ressourcen erstellt werden. Zu den rudimentären Ressourcen gehören das *Deployment*, der *Service*, die *Config Map* und das *Secret*. Während das Deployment und der Service

Tabelle 7.1: Standardressourcen von Kubernetes [Lin-Kub]

Ressource	Kurzbeschreibung
Pod	Eine logische Gruppe von (Docker) Containern, die über eine IP-Adresse verfügt. In einem Pod wird somit der eigentliche Microservice ausgeführt.
Service	Gruppiert Pods zu einer logischen Einheit. Über den Service kann die Gruppe an Pods angesprochen werden. Dabei können Algorithmen zur Verteilung der Anfragen auf die Pods angewendet werden.
Deployment	Ist für die Bereitstellung einer Pod-Klasse zuständig und verwaltet gleichzeitig den Zustand. Das Deployment wird auch zur Skalierung der Pods verwendet.
Daemon Set	Übernimmt, wie auch das Deployment, die Bereitstellung und Verwaltung einer Pod-Klasse. Der Unterschied zu einem Deployment liegt in der Skalierung, denn ein Daemon Set stellt pro verfügbarer Hardware nur eine Pod-Instanz zur Verfügung.
Config Map	Umfasst Umgebungsvariablen, die bei einer Ausführung eines Pods injiziert werden. Durch die Config Map wird die Konfiguration des Microservices in der Umgebung gehalten [Wi17].
Secret	Umfasst ebenfalls Umgebungsvariablen, die jedoch einen schützenswerten Charakter besitzen.
Service Account	Logischer Benutzer, der Pods ausführt und Zugriffe auf weitere Kubernetes-Ressourcen beschränkt.

beschreiben, welcher Container ausgeführt wird und wie dieser innerhalb von Kubernetes zu erreichen ist, definieren die Config Map und das Secret die Konfiguration des ausgeführten Containers. Weiterführende Kubernetes-Ressourcen wie dedizierte *Service Accounts* sind zwar nicht notwendig, aber sie erhöhen die Sicherheit bei der Ausführung der Microservices. Im Rahmen dieser Migrationsaktivität wird sich jedoch zunächst nur auf die rudimentären Kubernetes-Ressourcen berufen. Somit liegen am Ende dieser Migrationsaktivität für jeden Microservice mindestens vier Kubernetes-Ressourcen vor (Abbildung 7.3).

Um den Umgang mit den Kubernetes-Ressourcen für das Service-Team weiterführend zu vereinfachen, sieht die Inbetriebnahme die Einführung des Werkzeugs *Helm* [Hel-Doc] vor. Helm fasst zur besseren Verwaltung der Kubernetes-Ressourcen die Ressourcen in einem sogenannten *Helm Chart* zusammen (Abbildung 7.3). Die Idee hinter Helm ist es, die Helm Charts als *Pakete* anzusehen, die in Kubernetes hinzugefügt, entfernt oder auch aktualisiert werden können. Eine Interaktion über Helm bezieht immer das gesamte Paket ein, sodass alle Kubernetes-Ressourcen für die Ausführung des Microservices vorhanden sind.

Sowohl die Kubernetes-Ressourcen als auch das Helm Chart, welche beide als Migrationsartefakte dieser Migrationsaktivität angesehen werden, greifen auf die *YAML Ain't Markup Language* (YAML) zurück.

7.2.3 Integration der Microservices in Pipelines

Die Containerisierung und Integration in eine Container-Orchestrierung sind die notwendigen Schritte zum Ausführen der neuen Microservices. Nun gilt es, die Komplexität der Containerisierung und der Bereitstellung des Container-Images in der Container-Orchestrierung für das *Service-Team* zu reduzieren. Hierfür durchgesetzt haben sich *CI/CD-Pipelines* [BH+18, HR20]. Solche Pipelines durchlaufen mehrere Schritte und greifen hierfür auf die *Container-Konfiguration* und *Orchestrierungskonfiguration* zurück. In dieser Migrationsaktivität werden nun die notwendigen Konfigurationen erstellt, welche für die Integration in eine solche CI/CD-Pipeline notwendig sind. Die Inbetriebnahme verwendet *Jenkins* [Jen-Jen] als Werkzeug.

Der Einsatz einer Microservice-Architektur verleiht den Service-Teams die Agilität und Unabhängigkeit, Änderungen oder neue Funktionalität zu jeder Zeit zu publizieren. Damit ein Service-Team nicht bei neuen Versionen des Microservices manuell das Container-Image erstellen und dieses in die Container-Orchestrierung integrieren muss, werden diese Phasen in einer CI/CD-Pipeline abgebildet. Dies führt auch zu der durch DevOps gewünschten Automatisierung bei der Bereitstellung [BW+15, BH+16]. Eine weitere wichtige Phase der CI/CD-Pipeline ist das automatisierte Testen des Microservices, denn nur bei einer erfolgreichen Testphase soll die neue Version des Microservices in die Container-Orchestrierung übernommen werden.

```
1  pipeline {
2      agent any
3      stages {
4          stage('Build image') {
5              steps {
6                  echo 'Starting to build docker image'
7
8                  script {
9                      def image = docker.build("${env.CONTAINER_REGISTRY
                           }/ms-versioning:${env.VERSION}")
10                     image.push()
11                 }
12             }
13         }
14         stage('Test image') {
15             ...
16         }
17         stage('Deploy image') {
18             ...
19         }
20     }
21 }
```

Quelltext 7.2: Beispielhaftes Jenkinsfile eines Microservices

Ebenso wie bei der Containerisierung der Microservices werden nun für die Integration der Microservices in eine CI/CD-Pipeline Konfigurationsdateien notwendig. Wie diese Konfigurationsdateien aussehen, hängt von dem Werkzeug zur Bereitstellung einer CI/CD-Pipeline ab. Jenkins verwendet sogenannte *Jenkinsfiles* (Listing 7.2) zur Konfiguration einer Pipeline. Die Konfiguration sollte aus mindestens drei Phasen bestehen: (1) *Container-Image bauen*, (2) *Testen des Container-Images* und (3) *Integration des Container-Images*. Bei dem (1) Bauen des Container-Images muss auf eine sinnvolle *Versionierung* geachtet werden, damit die Container-Orchestrierung eindeutig auf Container-Images verweisen kann. Damit ist die Verwendung von Versionen wie "*latest*", die im Umfeld von Docker durchaus üblich ist, zu vermeiden. Zudem verliert das Service-Team die Übersicht über die vorhandenen Container-Images. Die Inbetriebnahme nutzt zur Bestimmung einer Container-Image-Version das *Semantic Versioning* [Sem-Sem].

Als Migrationsartefakt dieser Migrationsaktivität wird die Konfigurationsdatei des jeweiligen Pipeline-Werkzeuges gesehen. So ist beispielsweise bei der Verwendung von Jenkins als Werkzeug das *Jenkinsfile* das formale Migrationsartefakt.

7.3 Zusammenfassung

In Kapitel 7 wurde der konkrete Entwurf der *Inbetriebnahmephase* zur Vorbereitung der Bereitstellung der Microservices vorgestellt, die im Rahmen der durchlaufenen Migrationsiteration entwickelt werden. Die Inbetriebnahme befasst sich insbesondere mit der Auflösung der Problemstellung P6 (vgl. Abschnitt 1.4). Somit ist der Forschungsbeitrag B4 (vgl. Abschnitt 1.5.5) als Teil dieses Kapitels anzusehen. Zur Vorbereitung der Inbetriebnahme der Microservices wurden die drei konkreten Migrationsaktivitäten entworfen.

Aufbauend auf den bestehenden Arbeiten von Balalaie et al. [BH+18] und Henry und Ridene [HR20], welche beide die Bereitstellung der Microservices in Form von *Migrationsmustern* thematisieren, wurden für die Inbetriebnahme konkrete Migrationsaktivitäten und Migrationsartefakte abgeleitet. Zur Veranschaulichung der Migrationsartefakte wird in den Migrationsaktivitäten auf konkrete Werkzeuge verwiesen. Verwendet werden *Docker* [Doc-Ref], *Kubernetes* [Lin-Kub], *Helm* [Hel-Doc] und *Jenkins* [Jen-Jen].

Zu Beginn der Inbetriebnahme wird die *Containerisierung* der Microservices durchgeführt. Die *Service-Teams* konfigurieren die Containerisierung mit der Erstellung eines *Dockerfiles*. Für die Erstellung der Dockerfiles durch die Service-Teams werden weiterführend auch Prinzipien vorgestellt. Anschließend wird für den containerisierten Microservice die Integration in die *Container-Orchestrierung* Kubernetes vorgesehen. Damit ein Microservice in Kubernetes ausgeführt werden kann, müssen verschiedene Standardressourcen erstellt werden. Auch die Bündelung der Kubernetes-Ressourcen in

ein *Helm-Chart*, das den Service-Teams als weitere Abstraktionsschicht die Bereitstellung der Microservices in Kubernetes erleichtert, findet in der Inbetriebnahme Beachtung. Zum Abschluss der Migrationsaktivität und auch der Inbetriebnahme wird eine Integration in *CI/CD-Pipelines* vorgestellt. Als Werkzeug wird hierfür *Jenkins* vorgestellt. Die Service-Teams erstellen sogenannte *Jenkinsfiles*, welche die Phasen der CI/CD-Pipelines beschreiben.

8 Validierung der Beiträge

Mit dem hier vorgestellten Migrationsrahmenwerk kann systematisch in Migrationsiterationen ein bestehendes monolithisches Softwaresystem in eine Microservice-Architektur überführt werden. Die Systematik des Migrationsrahmenwerks unterstützt Softwarearchitekten und Softwareentwickler in den Phasen der Anforderungserhebung, des Design Recoverys, der Extraktion von Monolithenausschnitten, dem Entwurf von domänengetriebenen Microservices und der Restrukturierung des Entwicklungsteams. Durch das iterative Vorgehen und auch durch die Einschränkung einer Migrationsiteration auf einen beherrschbaren Monolithenausschnitt ist ein störungsfreier Betrieb des bislang monolithischen Softwaresystems gewährleistet.

Die Validierung dieser Forschungsarbeit richtet sich nach dem aktuellen Stand der empirischen Validierung für Entwurfsprozesse. Beschrieben und klassifiziert wird die Validierung der Forschungsbeiträge durch vier Validierungstypen, welche durch Durdik [Du16] im Kontext des architektonischen Wissensmanagements aufgestellt wurden. Giessler [Gi18] nutzt diese Validierungstypen zur Bewertung eines systematischen und nachvollziehbaren Entwurfsprozesses für ressourcenorientierte Microservices. Die prozessbehaftete Migration kann mit einem softwaretechnischen Entwurfsprozess gleichgestellt werden, wodurch sich die Anwendbarkeit der Validierungstypen für das Migrationsrahmenwerk begründen lässt. Aus diesem Grund orientiert sich das Vorgehen zur Validierung an der Arbeit von Giessler. Die Grundlage der vier Validierungstypen *Machbarkeit* (Typ 0), *Eignung* (Typ I), *Anwendbarkeit* (Typ II) und *Kosten/Nutzen* (Typ III) von Durdik fußt auf den Validierungstypen aus der *modellbasierten Leistungsvorhersage* aus [BR08] und [Ko08].

Jeder Validierungstyp bescheibt, auf welche Art und Weise die Validierung durchzuführen ist, um den Zweck dieses Validierungstyps zu erfüllen. Daher ergeben sich für die Validierung der Forschungsbeiträge folgende Vorgehensweisen: (1) Vergleich einer Migration mit und ohne Migrationsrahmenwerk, (2) Bewertung der eigenen Forschungsbeiträge anhand eines Anforderungskatalogs, (3) Anwendung der Systematik an einem durchgängigen und realitätsnahen Beispiel durch Kontrollgruppen, (4) Anwendung der Systematik im Kontext eines Migrationsprojekts im Rahmen einer Industriekooperation. Die Vorgehensweisen lassen sich den vier Validierungstypen zuordnen.

Die (2) Bewertung der eigenen Forschungsbeiträge anhand eines *Anforderungskatalogs* erfolgt im Rahmen der Typ 0-Validierung. Die Anforderungen dieses Katalogs werden darauf geprüft, ob diese erfüllt sind. Das Ergebnis wird gleichzeitig mit den Ergebnissen aus der Bewertung zum Forschungsstands des Abschnitts 3.3 verglichen. Für die Durchführung einer Typ I-Validierung

ist ein durchgängiges Beispiel von Nöten. Zur Bereitstellung des Beispiels soll innerhalb einer Kooperation zwischen Industriepartner und studentischen Projektteams eine Softwareentwicklung zur Konstruktion eines monolithischen Softwaresystems erfolgen. Die Typ I-Validierung zeigt dabei die (3) Anwendung der Systematik an einem durchgängigen und realitätsnahen Beispiel durch Kontrollgruppen. Abschließend erfolgt durch die Typ II-Validierung, ebenfalls gestützt durch eine Industriekooperation, ein (1) Vergleich einer Migration mit und ohne Migrationsrahmenwerk und die (4) Anwendung der Systematik im Kontext eines realen Migrationsprojekts.

In Abschnitt 8.1 werden die Validierungstypen detailliert im Kontext des iterativen Migrationsprozesses beschrieben. Anschließend erfolgt in Abschnitt 8.2 eine Auflistung der durchgeführten Studien für die Validierung der Forschungsbeiträge. Strukturiert werden die Studien anhand der Validierungstypen. Um die Gültigkeit der Validierung bewerten zu können, wird in Abschnitt 8.3 auf die Bedrohungen der Validität eingegangen. Der Abschnitt 8.4 behandelt die Durchführung der Typ 0-Validierung und präsentiert die erreichten Ergebnisse. Die Durchführung der Typ I-Validierung wird detailliert in Abschnitt 8.5 vorgestellt. Abschließend zeigt Abschnitt 8.6 die Durchführung der Typ II-Validierung.

8.1 Typen der empirischen Validierung

Zur Validierung architektureller Entscheidungsfindungen stellt Durdik [Du16] vier Validierungstypen vor. Giessler [Gi18] überträgt die Anwendung der Validierungstypen auf einen Entwurfsprozess zum Ableiten ressourcenorientierter Microservices aus einem Domänenmodell. Ausgehend von dem Validierungstyp kann eine Aussage über die Generalisierbarkeit der Forschungsbeiträge getroffen werden. Man spricht an dieser Stelle von der *externen Validität* [CP+82]. Je höher die externe Validität ist, desto generalisierbarer sind die Forschungsergebnisse, welche dann in anderen Kontexten Anwendung finden können [WR+12]. Die Abbildung 8.1 zeigt die externe Validität in Relation zu den Validierungstypen.

Folgende Validierungstypen finden im Kontext des iterativen Migrationsprozesses Anwendung:

Typ 0 (Machbarkeit) Die Typ 0-Validierung stellt den Eintrittspunkt in die Validierung dar und gilt als einfachste Form zum Validieren von Ansätzen. Ausgehend von einfachen, fiktiven Beispielen können Autoren die eigenen Ansätze durch geeignete Maßnahmen auf ihre Machbarkeit prüfen. Geeignete Maßnahmen im Kontext der Entwurfsprozesse können Umfragen [Ko08] oder textuell beschriebene Vergleiche zwischen Situationen sein, welche die Soll-Situation (systematischer iterativer Migrationsprozess vorhanden) mit einer hypothetischen Situation vergleicht, in der kein vergleichbarer Migrationsprozess existiert. Es bietet sich an, hypothetische Situationen aus dem Stand der Forschung abzuleiten, um den eigenen Entwurfsprozess dem Stand der Forschung gegenüberzustellen. Der Personenkreis einer Umfrage-getriebenen Typ

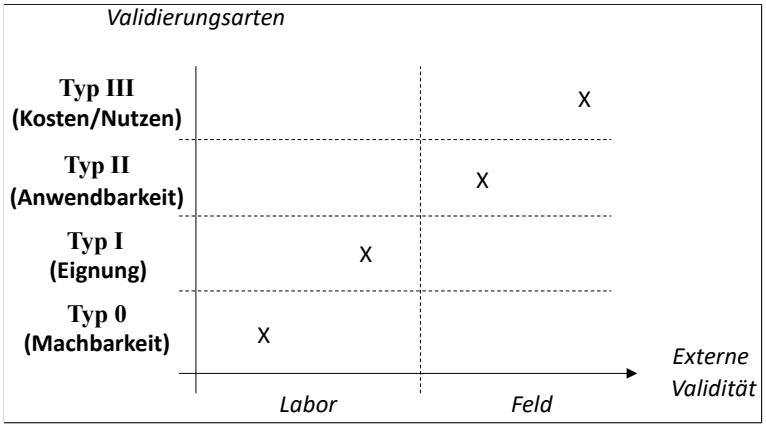

Abbildung 8.1: Verknüpfung der externen Validität und den Validierungstypen aus [Gi18]

0-Validierung sollte möglichst nah an der eigentlichen Zielgruppe des Migrationsprozesses liegen, um ein grundlegendes Verständnis der Thematik und aussagekräftige Ergebnisse zu gewährleisten. Der Aufwand für diese Art der Validierung ist üblicherweise gering und hat den Vorteil, dass kein industrienaher Kontext notwendig ist. Gleichzeitig bedeutet dies, dass die externe Validität der Typ 0-Validierung durch die Laborsituation am geringsten ist (siehe Abbildung 8.1). Studien für diesen Validierungstyp können durch die Autoren der Forschungsbeiträge selbst durchgeführt werden.

Typ I (Eignung) Mit der Typ I-Validierung wird der aufgestellte Entwurfsprozess auf seine Eignung geprüft. Im Gegensatz zu der Typ 0-Validierung basiert die Eignungsprüfung auf einem durchgängigen Beispiel, welches dennoch einen fiktiven Charakter besitzen darf. Je näher das Beispiel an der Realität liegt, desto höher ist die externe Validität. Auch die Typ I-Validierung kann durch die Autoren der Forschungsbeiträge selbst durchgeführt werden. Das Ziel der Eignungsprüfung ist es, zu zeigen, dass die einzelnen Bestandteile des Migrationsprozesses sich für das eigentliche Ziel eignen. Als Bestandteile werden die *Migrationsaktivitäten*, *sequentielle Ordnung* der Prozessschritte und generierten *Migrationsartefakte* verstanden. Bewertet werden die Ergebnisse der Typ I-Validierung durch den Vergleich mit einer *no-approach-Situation* [Du16], also einer Situation ohne einen systematischen Entwurfsprozess. Häufig erfolgt diese Bewertung über die Ausführung eines Fragebogen-gestützten Interviews mit den teilnehmenden Kontrollgruppen. Der Aufwand zur Durchführung einer Typ I-Validierung beläuft sich ähnlich wie bei der Typ 0-Validierung auf ein üblicherweise geringes Maß. Die externe Validität der Typ I-Validierung ist jedoch höher (siehe Abbildung 8.1), auch wenn die Studie durch die Autoren ausgeführt wird.

Typ II (Anwendbarkeit) Mittels der Typ II-Validierung wird die Anwendbarkeit des Migrationsprozesses durch die letztendlichen Anwender geprüft. Während die Typ 0- und Typ I-Validierungen durch die Autoren des Migrationsprozesses selbst durchgeführt werden können, müssen für

die Typ II-Validierung zwangsläufig reale Kontrollgruppen herangezogen werden. Im Rahmen dieser Forschungsarbeit sind die Probanden der Kontrollgruppe *Softwarearchitekten*, *Softwareentwickler* und *Studierende der Informatik*. Letztere eigenen sich als Zielgruppe für Validierungen, wie [Ti00] bestätigt. Weiterhin unterscheidet sich die Typ II-Validierung von der Typ I-Validierung durch die Notwendigkeit eines Prototypen, der in einem realitätsnahen Projektkontext betrieben wird. Irrelevant ist dabei die Tatsache, ob der Prototyp im Rahmen eines Forschungsprojektes oder eines realen Praxisprojektes betrachtet wird. Nützlich ist die Praxisnähe, da somit auch eine Anwendbarkeit im realen Praxiskontext erprobt wird. Alle Bestandteile des Migrationsprozesses werden für die Validierung einem für den Forschungsbereich geläufigen Ansatz gegenübergestellt. Das Ergebnis der Validierung kann mittels eines Bewertungsbogens festgehalten werden [Gi18]. Intensiver gestalten sich der Aufwand und die Kosten für die Durchführung einer Typ II-Validierung. Begründet ist dies durch das Einbeziehen von praxisnahen Kontrollgruppen und die Notwendigkeit eines realitätsnahen Beispiels.

Typ III (Kosten/Nutzen) Die Typ III-Validierung beruht auf der Messung der tatsächlichen Kosten, die durch den Einsatz des Entwurfsprozesses eingespart werden können. Der Einsatz der Typ III-Validierung gilt als besonders aufwendig und kostenintensiv. Das zugrundeliegende Softwaresystem muss mindestens zweimal entwickelt werden, um die notwendigen Informationen der Validierung sammeln zu können. Zum einen gilt es, das monolithische Softwaresystem ohne den Einsatz des Migrationsprozesses zu migrieren, unter der Prämisse, dass die dadurch entstehenden (höheren) Kosten für die Wartung oder auch Störungen des Softwaresystems akzeptiert werden. Dem entgegen steht die Migration des monolithischen Softwaresystems unter dem Einsatz des iterativen Migrationsprozesses, unter der Prämisse, dass die höheren initialen Kosten für die Planung der Iterationen und die Erstellung von Entwürfen hingenommen werden. Die jeweiligen Kosten der durchgeführten Migrationsarbeiten gilt es dann über einen längeren Zeitraum gegenüberzustellen, um den letztendlichen Nutzen des iterativen Migrationsprozesses zu bestimmen. Somit ist der Einsatz einer Typ III-Validierung zeit- und kostenintensiv. Durdik [Du16] zufolge ist der Einsatz einer Typ III-Validierung im Rahmen des Forschungsbereiches der Entwurfsentscheidungen unüblich und ist allgemein nicht zu finden.

8.2 Gegenstand der Validierung

Zur Validierung des Migrationsrahmenwerks und der damit verbundenen Forschungsbeiträge wurden primär die Validierungstypen Typ 0, Typ I und Typ II berücksichtigt. Auf die Durchführung einer ganzheitlichen Typ III-Validierung wurde aus Kosten- und Zeitgründen verzichtet. Bedingt durch die vergangenen Migrationsbestrebungen der Evana AG können jedoch oberflächlich Aufwände, Erfahrungen und Problemstellungen aufgezeigt werden, welche die Vorzüge des Migrationsrahmenwerks erneut hervorheben. Zusammenfassend stellen sich die durchgeführten Validierungen wie folgt dar:

Typ 0 (Machbarkeit) Eine Typ 0-Validierung wurde durch den Vergleich eines Migrationsvorhabens mit und ohne Migrationsrahmenwerk durchgeführt. Gleichzeitig erfolgte eine Bewertung der Forschungsbeiträge im Vergleich zu bestehenden Arbeiten aus dem Forschungsgegenstand der iterativen Migration. Zur Bewertung wurde der *Anforderungskatalog* aus Abschnitt 3.1 herangezogen. Das Ergebnis der Typ 0-Validierung wird in Abschnitt 8.4 dargestellt.

Typ I (Eignung) Die Durchführung der Typ I-Validierung beruhte auf zwei Phasen eines beispielhaften Softwareprojektes. Um die Systematik des Migrationsrahmenwerks einsetzen zu können, musste zunächst ein beispielhaftes monolithisches Softwaresystem geschaffen werden. Die Grundlage für dieses Softwaresystem lieferte der Industriepartner Evana AG mit seiner SaaS-Lösung *Evana360*. Erst nach der Konstruktion des beispielhaften Softwaresystems konnte der Einsatz der Systematik erprobt werden. Durchgeführt wurden die beiden Phasen des Softwareprojektes durch studentische Kontrollgruppen in Kooperation mit Domänenexperten der Evana AG. Somit wurde für die Typ I-Validierung auf Studierende zurückgegriffen. Die Erprobung des Migrationsprozesses zunächst mit Hilfe von Studierenden entspricht auch der Empfehlung von Tichy [Ti00], dem zufolge eine Validierung mit Studierenden noch vor der Validierung mit der letztendlichen Zielgruppe erfolgen sollte. Die Kontrollgruppen stammten dabei aus dem Sommersemester (SoSe) 2019, dem Wintersemester (WiSe) 2019/2020 und dem SoSe 2020. Um das geistige Eigentum der Evana AG zu wahren, wurde die Funktionalität des beispielhaften Softwaresystems durch die Domänenexperten eingeschränkt. Das nachempfundene Softwaresystem wurde unter dem Namen *simple DocumentManagementSystem* (sDMS) entwickelt. Die Ergebnisse der Typ I-Validierung und die Betrachtung der Eignung des iterativen Migrationsprozesses werden in Kapitel 8.5 diskutiert.

Typ II (Anwendbarkeit) Die Validierung der Anwendbarkeit des Migrationsrahmenwerks wurde durch die Industriekooperation mit der Evana AG durchgeführt. Ein dediziertes Migrationsteam, bestehend aus einem Softwarearchitekten und vier Softwareentwicklern, hat die Systematik im Kontext eines realen Migrationsauslösers demonstriert. Ein besonderer Fokus wurde während der Validierung auf die Extraktionsphase gelegt, deren Ergebnisse in Abschnitt 8.6 präsentiert werden.

8.3 Bedrohungen und Einschränkung der Validität

Jede Validierung unterliegt verschiedenen Bedrohungen und Einschränkungen, welche die Gültigkeit der Ergebnisse beeinflussen können. Eine Klassifizierung der Validität und ihre Bedrohungen wurden von Wohlin et al. [WR+12] auf Basis von [CC79] vorgenommen. Die Autoren kommen zu dem Schluss, dass vier Typen von Validitäten existieren: (1) *Ergebnisvalidität*, (2) *Interne Validität*, (3)

Konstruktionsvalidität und (4) *Externe Validität*. Nicht jede Validität wird durch eine Studie im gleichen Maße betrachtet. Selbiges gilt für die Bedrohungen der Validität.

Ergebnisvalidität Der Verlauf einer Studie lässt sich durch das Anpassen von Eingabeparametern (auch *Variablen* genannt) beeinflussen. Um möglichst unterschiedliche Ergebnisse von einer Studie zu erreichen, können die Variablen in ihren Werten angepasst werden. Variablen die angepasst werden, bezeichnet man auch als *Faktoren* (engl. factors). Die Werteänderung einer Variablen bzw. eines Faktors bezeichnet man als *Behandlung* (engl. treatment). Die Ergebnisvalidität befasst sich dabei mit der Beziehung zwischen Faktor und Resultat der Studie im Rahmen einer bestimmten Behandlung. Bedroht wird die Validität durch eine falsche Schlussfolgerung bezüglich der Beziehung zwischen Faktor und Ergebnis.

Interne Validität Unter der internen Validität wird die Interpretation von *Ursache* und *Wirkung* innerhalb der Studie verstanden. Die Ursache wird durch Variablen und ihre Werte beschrieben, die letztendlich im Verlauf der Studie für eine bestimmte Wirkung sorgen. Da Variablen durch unterschiedliche Behandlungen unterschiedliche Werte einnehmen, variiert das Ergebnis. Somit kann davon ausgegangen werden, dass eine bestimmte Behandlung von Variablen (Ursache) für ein bestimmtes Ergebnis (Wirkung) verantwortlich ist. Es muss sichergestellt sein, dass die Ursache kausal mit der Wirkung in Abhängigkeit steht. Als eine Bedrohung für die interne Validität wird eine falsche *kausale Abhängigkeit* zwischen einer Ursache und einer Wirkung gesehen.

Konstruktionsvalidität Die Konstruktionsvalidität befasst sich mit dem *Aufbau der Studie*. Ist die Ursache und Wirkung einer Studie in kausaler Abhängigkeit, gilt es sicherzustellen, dass die entwickelte Methodik korrekt in der Studie abgebildet wurde. Dies muss bei der Konstruktion der Ursache (Variablen und Behandlung) und der Abbildung der Wirkung (Ergebnisse der Studie) beachtet werden. Man spricht auch von der korrekten Abbildung der Theorie auf den Kontext der Studie, in dem Beobachtungen gemacht werden können. Bedroht wird die Konstruktionsvalidität durch eine falsche Abbildung der Theorie auf die Studie.

Externe Validität Die externe Validität beschreibt die Generalisierbarkeit der entwickelten Methodik. Durch die Studie wird die Methodik innerhalb eines speziellen Kontexts durchgeführt. Für die Anwendung der Methodik ist die Frage, inwiefern die Konzepte und auch Ergebnisse der Studie außerhalb des Kontexts Gültigkeit finden, von großem Interesse. Bedrohungen für die externe Validität sind die falschen Probanden, falsche Umgebungsbedingungen oder realitätsferne Zeitfenster bei der Durchführung der Studie.

Tabelle 8.1: Bewertung der vorliegenden Forschungsarbeit

	A1 - Systematischer und nachvollziehbarer Migrationsprozess	A2 - Wiederherstellung der Architekturbeschreibung	A3 - Identifikation kohäsiver Softwarebausteine	A4 - Etablieren eines Domänenverständnisses	A5 - Domänengetriebener Entwurf der Microservice-Architektur	A6 - Reorganisation des monolithischen Entwicklungsteams	A7 - Berücksichtigung des komplexen Betriebs
[BH+18] Microservices Migration Patterns	◐	●	◐	●	○	○	●
[HR20] Migration to Microservices	◐	○	●	○	●	○	●
[Ne19] Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith	○	○	○	◐	●	◐	○
[BH+16] Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture	○	●	○	○	●	●	●
[GK+16] Service Cutter: A Systematic Approach to Service Decomposition	◐	●	●	○	◐	○	○
[KH18] Using Microservices for Legacy Software-Modernization	◐	●	○	○	◐	○	◐
[MC+17] Extraction of Microservices from Monolithic Architectures	◐	●	●	○	◐	○	○
Vorliegende Forschungsarbeit	●	●	●	●	●	●	◐

8.4 Typ 0-Validierung - Machbarkeit

Die Typ 0-Validierung dieser Forschungsarbeit beruht auf der Bewertung des Migrationsrahmenwerks mit den damit verbundenen Forschungsbeiträgen anhand des Anforderungskatalogs aus Abschnitt 3.1. Die dort abgebildeten Anforderungen (A1) *Systematischer und nachvollziehbarer Migrationsprozess*, (A2) *Wiederherstellung der Architekturbeschreibung*, (A3) *Identifikation kohäsiver Softwarebausteine*, (A4) *Etablieren eines Domänenverständnisses*, (A5) *Domänengetriebener Entwurf der Microservice-Architektur*, (A6) *Reorganisation des monolithischen Entwicklungsteams* und (A7) *Berücksichtigung des komplexen Betriebs werden nachfolgend auf ihre Erfüllung geprüft*. Wichtig für die Erfüllung einer Anforderung ist, dass im Vergleich zu bestehenden Arbeiten eine Verbesserung der Situation für das Migrationsteam erreicht wird. Das Ergebnis der Validierung wird in Tabelle 8.1 festgehalten. Die Bewertungen der einzelnen Anforderungen werden nachfolgend detailliert erörtert.

Systematischer und nachvollziehbarer Migrationsprozess (A1) Die Anforderung A1 bezeichnet einen Migrationsprozess als systematisch, sobald ein Migrationsrahmenwerk mit einer festgelegten Reihenfolge an Migrationsaktivitäten vorhanden ist. Eine Migrationsaktivität entspricht dabei einer Aktivität, welche durch das Migrationsteam durchgeführt werden muss. Weiterführend muss anhand von Migrationsaktivitäten die iterative Migration ganzheitlich betrachtet sein, um das Migrationsteam in jedem Aspekt einer Migrationsiteration zu unterstützen. Die Nachvoll-

ziehbarkeit des Migrationsprozesses ist definiert durch den Einsatz formaler Softwareartefakte, die es dem Migrationsteam erlauben, Entscheidungen auf Basis der Inhalte der Softwareartefakte zu treffen und diese nachhaltig nachvollziehbar zu gestalten. Wie bereits im Handlungsbedarf (vgl. Abschnitt 3.3) festgehalten, hat sich durch die Betrachtung des Forschungsgegenstandes gezeigt, dass ein ganzheitlicher Migrationsprozess oder auch ein Migrationsrahmenwerk nicht gegeben ist. So verbessert die hier vorliegende Forschungsarbeit durch den Entwurf eines Migrationsrahmenwerks beziehungsweise einer Migrationssystematik (vgl. Abschnitt 4.2) die Situation bei der Durchführung einer Migrationsiteration. Eine Bewertung muss noch bezüglich der Ganzheitlichkeit und der Nachvollziehbarkeit der Systematik erfolgen. Die entworfene Systematik betrachtet die Durchführung einer Migrationsiteration beginnend mit der Spezifikation eines Migrationsauslösers und abschließend mit der Vorbereitung der Microservices für den Betrieb. In den bestehenden Arbeiten wie beispielsweise [RH06, KH18] stellen neue Anforderungen an das monolithische Softwaresystem die Ausgangssituation der Migration dar. Dies deckt sich mit dem initialen Betrachten des Migrationsauslösers der Systematik. Daher wird davon ausgegangen, dass durch die Betrachtung des Migrationsauslösers als erste Migrationsaktivität der allgemeine Start einer Migrationsiteration gefunden wurde. Ebenfalls von großer Relevanz ist die Rekonstruktion der Architekturbeschreibung, die durch eigene Migrationsaktivitäten erarbeitet wird. So kann der Umfang der Migrationsiteration bestimmt werden, der letztendlich durch die Migrationsaktivitäten zum Entwurf der Microservice-Architektur benötigt wird. Auch vorgesehen ist die Reorganisation des monolithischen Softwaresystems, die nach [HS17] für einen effizienten Einsatz der Microservice-Architektur notwendig ist. Abschließend werden, wie bereits erwähnt, Vorbereitungen für den Betrieb der Microservices getroffen. Die Nachvollziehbarkeit ist durch die durchgängige Verwendung formaler Migrationsartefakte gegeben. Ergebnisse, die durch das Migrationsteam generiert werden, sind in diesen Migrationsartefakten festgehalten. Gleichzeitig werden die Migrationsartefakte genutzt, um Entscheidungen abzuleiten.

Wiederherstellung der Architekturbeschreibung (A2) Die Architekturbeschreibung hat während und nach der Migrationsiteration eine hohe Bedeutung. Während der Migrationsiteration dient sie als Informationsquelle für das Migrationsteam. Insbesondere werden darüber die Softwarebausteine identifiziert, welche dem Monolithenausschnitt zugeordnet werden. Nach der Migrationsiteration dient die Architekturbeschreibung zur Verifikation der Funktionalität, die bereits im Vorfeld durch den Monolithen bereitgestellt wurde. In der Extraktionsphase aus Kapitel 5 wird eine konkrete Wiederherstellung der Architekturbeschreibung angestrebt. Als Quelle dienen sowohl Softwareentwickler als auch der Quellcode des monolithischen Softwaresystems. Zum Tragen kommen dabei Ansätze wie das *Design Recovery* mit der statischen Quellcodeanalyse. Die Besonderheit der Extraktionsphase ist, dass nur der für die Migrationsiteration relevante Ausschnitt in Form einer Architekturbeschreibung rekonstruiert wird. Damit greift die Extraktionsphase den zeitlichen und finanziellen Faktor einer Migration

auf und sieht eine *ressourcenschonende Wiederherstellung* vor.

Identifikation kohäsiver Softwarebausteine (A3) Die dritte Anforderung A3 thematisiert die Identifikation von Softwarebausteinen, die nach der Definition von Vogel et al. [VA+11] gemeinsam eine hohe Kohäsion aufweisen. Notwendig ist dies zur Bestimmung des Umfangs beziehungsweise der Softwarebausteine, die zusammen in einer Migrationsiteration betrachtet werden. Der Zusammenschluss der Softwarebausteine wird anhand des Migrationsrahmenwerks dieser Forschungsarbeit als Monolithenausschnitt bezeichnet. Die Extraktionsphase sieht in mehreren Migrationsaktivitäten die Identifikation vor. Der Beantwortung der Frage zur Kohäsion von Softwarebausteinen legt die Extraktionsphase den *domänengetriebenen Entwurf* (engl. Domain-Driven Design, DDD) zugrunde. Infolgedessen wird zunächst die grobgranulare Struktur der Geschäftsdomäne mit ihren Subdomänen bestimmt. Durch das Domänenwissen, welches in den Subdomänen gebündelt wird, können anschließend Funktionalitäten, die bereits durch den Monolithen bereitgestellt werden, den Subdomänen zugeordnet werden. Damit liegen zunächst die kohäsiven Funktionalitäten vor. Mit diesen Funktionalitäten ist das Migrationsteam in der Lage, die Softwarebausteine zu bestimmen, welche für das Erbringen der Funktionalitäten zuständig sind.

Etablieren eines Domänenverständnisses (A4) Kritisch für den Erfolg eines Softwareprojektes ist das Verständnis der Softwareentwickler über die Geschäftsdomäne [Ev04]. Das Etablieren dieses Domänenverständnisses hat die Anforderung A4 für das Migrationsrahmenwerk zur Folge. Die Systematik des hier entworfenen Migrationsrahmenwerks sieht die Etablierung dieses Verständnisses als zentrale Aufgabe an. Das Verständnis ist dabei immer im Kontext des Monolithenausschnitts anzusehen. In den Migrationsphasen der Extraktion und Modernisierung ist der Softwareentwicklungsansatz DDD an mehreren Stellen verankert. Dadurch sind die Berührungspunkte des Migrationsteams und nachher auch der Service-Teams mit der Geschäftsdomäne vielfältig. Umgesetzt werden strategische und taktische Modellierungen [Ev04, Ve13], die ein Verständnis über die grob- und feingranulare Geschäftsdomäne vermitteln. Weiterhin wird eine Projektsprache, die ubiquitäre Sprache, in Form eines Glossars und Language Sketchs (vgl. Abschnitt 5.4.2) festgehalten. Beide Migrationsartefakte liefern einen schnellen und einfachen Überblick über die Funktionsweise der Geschäftsdomäne.

Domänengetriebener Entwurf der Microservice-Architektur (A5) Die Anforderung A5 verlangt bei der Durchführung einer Migrationsiteration den Entwurf einer domänengetriebenen Microservice-Architektur. Durch die Orientierung an der Geschäftsdomäne wird sich eine hohe Autonomie und Wiederverwendbarkeit der Microservices versprochen [HG+17]. Beide Aspekte sind dabei zentrale Eigenschaften einer Microservice-Architektur [Ne15], weshalb die Erfüllung dieser Anforderung als besonders wichtig erachtet wird. Der Entwurf der Microservice-Architektur für den Monolithenausschnitt erfolgt in der Modernisierungsphase. Dort werden die Domänenobjekte der Relationensicht (vgl. Abschnitt 6.3.2) in die Bounded

Contexts aufgeteilt. Die Zuordnung der Domänenobjekte in die Bounded Contexts erfolgt über die Betrachtung zentraler Domänenereignisse. Die Domänenereignisse entkoppeln die Domänenobjekte voneinander. So liegt eine hohe Autonomie des Bounded Contexts vor. Gleichzeitig liegen diejenigen Domänenobjekte zusammen, die nicht durch Domänenereignisse entkoppelt werden können. Der daraus resultierende Entwurf der Microservice-Architektur richtet sich somit nach der Geschäftsdomäne. Ein weiterer wichtiger Schritt der Modernisierungsphase ist jedoch auch die Überarbeitung dieses Entwurfs anhand nicht-funktionaler Anforderungen, um einem effizienten Betrieb gerecht zu werden.

Reorganisation des monolithischen Entwicklungsteams (A6) Eine Microservice-Architektur verfolgt nicht nur die Verbesserung der Skalierung des Softwaresystems, sondern auch die Skalierbarkeit des Entwicklungsteams [HS17]. Monolithische Entwicklungsteams sind mit zunehmender Größe nur noch mit großem Aufwand zu verwalten. Dagegen sind kleine Entwicklungsteams, die *Service-Teams*, durch die geringe Anzahl an Softwareentwicklern besser zu überblicken. Zudem ist es möglich, kleinen Entwicklungsteams eine hohe Autonomie zu ermöglichen. Um diese grundlegende Anforderung an das Migrationsrahmenwerk zu erfüllen, sieht die Systematik in der Modernisierungsphase die Reorganisation des Entwicklungsteams vor. Dabei wird für jeden entworfenen Microservice ein Service-Team etabliert. Die entsprechende Migrationsaktivität berücksichtigt bei dem Etablieren der Service-Teams zum einen Interessenskonflikte, die insbesondere bei kleinen monolithischen Softwareentwicklungsteams entstehen und zum anderen ein breites Spektrum an Fähigkeiten und Erfahrungsstufen.

Berücksichtigung des komplexen Betriebs (A7) Der Betrieb eines monolithischen Softwaresystems ist durch die geringe Anzahl der Softwarebausteine, die eigenständig betrieben werden, für das Entwicklungsteam einfach zu beherrschen. Durch die Einführung der Microservices wird die Anzahl der eigenständigen Softwarebausteine maßgeblich erhöht, wodurch die Komplexität des Betriebs des Softwaresystems steigt. Diese Komplexität muss für das Entwicklungsteam reduziert werden, was durch die Anforderung A7 gefordert wird. Das Ziel ist es, Konzepte bei der Implementierung vorzusehen und weitere Vorbereitungen vor der Inbetriebnahme zu treffen. Die Vorbereitungen werden durch die Systematik des Migrationsrahmenwerks in der Inbetriebnahmephase vorgesehen. Dort wird eine Containerisierung und die Integration der Microservices in eine Container-Orchestrierung und Continuous Integration/Continuous Delivery-Pipeline (CI/CD) durchgeführt. Dies sind laut Balalaie et al. [BH+18] wichtige Schritte zur Reduzierung der Komplexität. Nicht weiter betrachtet wurden Prinzipien von Cloud-nativen Softwaresystemen [GB+17, Da19] während des Entwurfs und der Implementierung der Microservices.

8.5 Typ I-Validierung - Eignung

Die Prüfung inwieweit die Systematik des Migrationsrahmenwerks geeignet ist, im Rahmen einer Typ I-Validierung durchgeführt. Die Grundlage der Validierung stellte die SaaS-Lösung *Evana360* der Evana AG dar. Dadurch konnte ein realitätsnahes und durchgängiges Beispiel für die Eignungsfeststellung bereitgestellt werden. Ergebnisse dieser Studie wurden bereits in den einzelnen Kapiteln zur Migrationssystematik (siehe Kapitel 5 bis 7) zur besseren Erläuterung verwendet. Die Studie selbst wurde in den Semestern *WiSe 2018/2019*, *SoSe 2019* und *WiSe 2019/2020* im Rahmen von Seminar- und Praktikumsarbeiten durchgeführt.

Bevor die eigentliche Durchführung der Studie erfolgen konnte, musste zunächst ein monolithisches Softwaresystem realisiert werden. Hierfür wurde ein kohärenter Ausschnitt der Evana360-Lösung betrachtet. Es wurde nur ein stark eingeschränkter Ausschnitt betrachtet, um den Aufwand, auch unter Berücksichtigung des Stundenkontingents der Studierenden, auf ein Minimum zu reduzieren. Zudem konnte durch die Einschränkung das geistige Eigentum der Evana AG gewahrt werden. Das zu realisierende Softwaresystem wurde im Rahmen der Forschungsgruppe Cooperation & Management (C&M) als *sDMS* bezeichnet.

Die Vorgehensweise bei der Migration des Monolithen wurde auf Basis der Charakteristik einer Typ I-Validierung bestimmt. Die Projektteams migrierten den Monolithen basierend auf den Migrationsphasen *Extraktion*, *Modernisierung* und *Inbetriebnahme*. Am Ende der Studie wurde mit den Autoren ein Interview geführt, um die Einschätzungen der Studierenden festzuhalten. Durch die hohe Diversität der Probanden konnte die Eignung der Migrationssystematik besser beobachtet werden. Studierende befassen sich in der Regel zum ersten Mal mit der Migration eines Softwaresystems.

8.5.1 Vorgehensweise zur Konstruktion des Monolithen

Bei der Durchführung der Typ I-Validierung wurde zum einen auf die *Realitätsnähe der Studie* und zum anderen auf die *Beherrschbarkeit durch die Projektteams* geachtet. Für die Konstruktion des monolithischen Softwaresystems bedeutete dies, dass keine trivialen Anwendungsfälle (wie bspw. eine Todo-Listenverwaltung) betrachtet wurden. Weiterhin musste der Umfang des zu konstruierenden Softwaresystems in einem für die Studierenden zumutbaren zeitlichen Rahmen stattfinden. Bewertungsgrundlage dafür war die Leistungspunktbewertung (LP-Bewertung) des Praktikums, das die Studierenden belegt hatten. Das Praktikum wurde mit 5 LP bewertet, was einem zeitlichen Aufwand von *150 Stunden* entspricht. Bei einer durchschnittlichen Größe von 4 Studierenden pro Projektteam bedeutete dies ein Kontingent von *600 Stunden*.

Zusammenfassend wird die Konstellation der Studie in Abbildung 8.2 dargestellt. Die von der Evana AG getätigten Beiträge werden als *realer Kontext* zusammengefasst. Sämtliche von den Projektteams durchgeführte Tätigkeiten finden sich in dem *experimentellen Kontext* wieder.

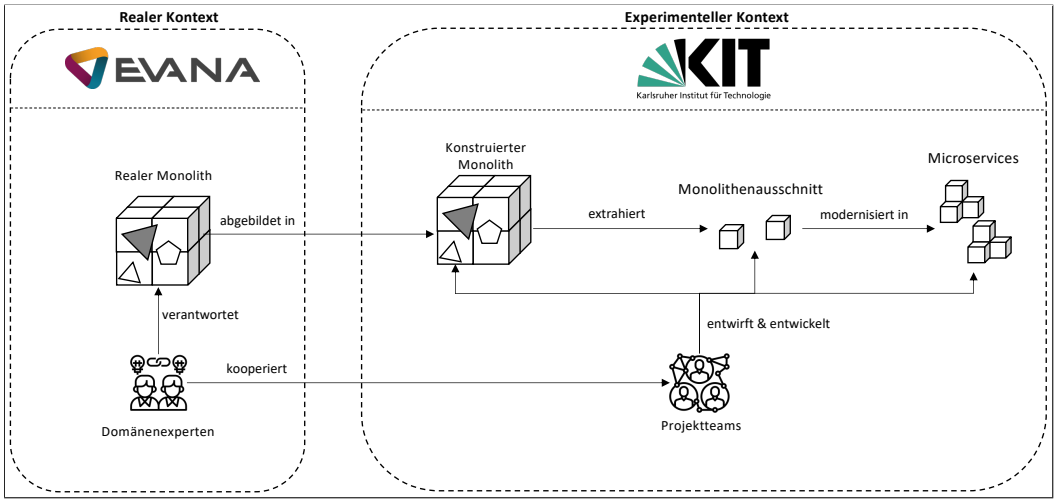

Abbildung 8.2: Kontext der Typ I-Validierung in Kooperation mit der Evana AG

Aufbauend auf der Kooperation mit der Evana AG wurde die Konstruktion des monolithischen Softwaresystems *sDMS* von Interviews mit den Domänenexperten begleitet. Zunächst galt es dem Projektteam den Kontext des angestrebten Monolithen näher zu bringen. Hierzu erfolgte eine Präsentation der SaaS-Lösung *Evana360*, damit die Projektteams sich einen Eindruck über das reale Softwaresystem verschaffen konnten. Anschließend wurde gemeinsam ein *Minimum Viable Product* (MVP) definiert, das die Bedingungen (1) nicht-triviale Funktionalität und (2) beherrschbarer Umfang erfüllt. Die Festlegung des MVPs erfolgte durch die Erstellung von Features, die aufbauend auf den Konzepten der *verhaltensgetriebenen Entwicklung* (engl. Behavior-Driven Development, BDD) und der Beschreibungssprache *Gherkin* erstellt wurden (vgl. Abschnitt 2.3). Bei der Nutzung der Features erfolgte die Einschränkung, dass diese nicht für das Testen des zu entwickelnden Softwaresystems verwendet werden, was dem eigentlichen Sinn der BDD-Features widersprechen würde. Diese Einschränkung erfolgte auf Basis des zugrundeliegenden *Evana360*, welches nicht auf der Verwendung von BDD beruht. Der Nutzen der *BDD-Features* zum Abbilden und Abstimmen der gewünschten Funktionalität blieb jedoch erhalten.

Nach Abnahme der BDD-Features konnte die Analysephase beendet werden. Anschließend wurde der Entwurf des Monolithen besprochen. Relevant für die Diskussion war nur noch die Mikroarchitektur des Softwaresystems, da die Makroarchitektur auf dem monolithischen Architekturstil aufsetzt. An dieser Stelle konnten die Domänenexperten auf Basis des realen Softwaresystems eine Mikroarchitektur vorgeben. Stark geprägt durch die Schnittstellentechnologie *GraphQL* [Gra-Gra] beruft sich die Evana AG auf die in Abbildung 8.3 dargestellte Mikroarchitektur. Basierend auf den Aussagen

der Domänenexperten und eigenen Erfahrungen zeichnet sich diese Mikroarchitektur durch die hohe Kopplung von Domänenkonzepten aus, was für die Durchführung der Studie als Vorteil einzustufen ist. Um die festgelegte Mikroarchitektur mit Leben zu füllen, griffen die Projektteams auf Konzepte von *DDD* zurück. Zwar entsprach auch dies nicht dem Vorgehen der Evana AG, doch das Verständnis und die Einarbeitung neuer Projektteammitglieder über die Semester hinweg stand im Vordergrund, was mit den Konzepten von DDD gewährleistet werden konnte. Somit wurde zunächst ein *Domänenmodell* aus den BDD-Features extrahiert. Anschließend war es den Projektteams möglich, die gewonnenen Erkenntnisse in dem monolithischen Softwaresystem abzubilden.

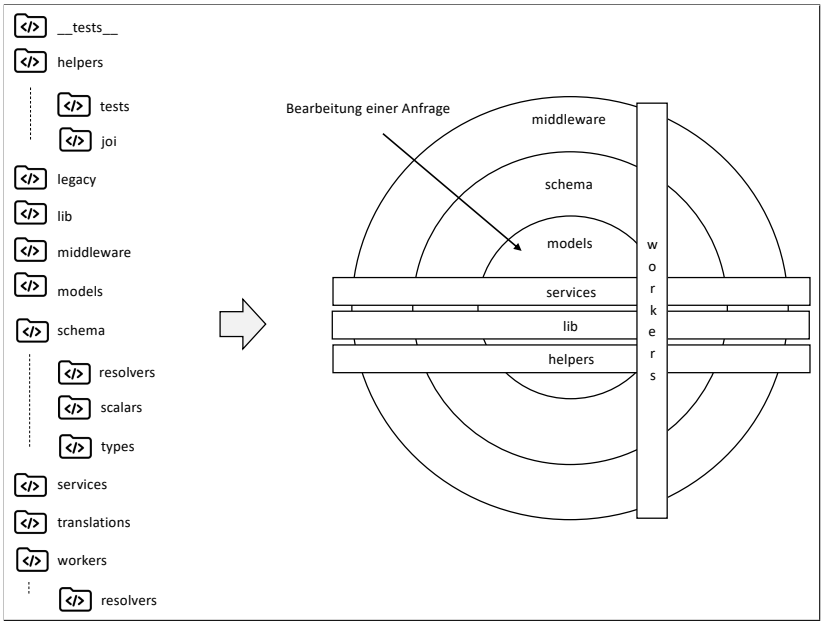

Abbildung 8.3: Mikroarchitektur der monolithischen SaaS-Lösung Evana360

Das gewählte Vorgehen zur Konstruktion des Monolithen ist in Abbildung 8.4 festgehalten. Der beschriebene Ablauf wurde im Semester *WiSe 2018/2019* mit einem Projektteam durchlaufen. Das konstruierte *sDMS* wurde in den nachfolgenden Semestern den Migrationsiterationen unterzogen.

8.5.2 Durchführung der Migrationsiterationen

Aufbauend auf dem konstruierten *sDMS* wurden in den Semestern *SoSe 2019* und *WiSe 2019/2020* jeweils eine Migrationsiteration durch ein Projektteam ausgeführt. Um das Migrationsvorhaben möglichst realitätsnah zu gestalten, durften die Studierenden nicht auf die Architekturbeschreibungen zurückgreifen. Somit lag zum Zeitpunkt der Migrationsvorhaben nur der Quellcode des sDMS vor. Neben dem Quellcode konnten die Studierenden jedoch noch auf die Domänenexperten der Evana AG zurückgreifen.

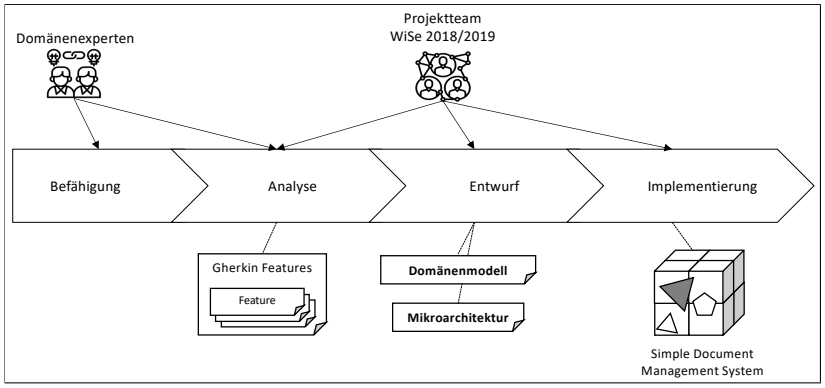

Abbildung 8.4: Prozesshafte Darstellung der Konstruktion des monolithischen Softwaresystems

Der Projektkontext der Studierenden hat sich wie folgt zusammengesetzt (Abbildung 8.5). Zu Beginn jedes Semesters wurde den Studierenden das Dokumentenmanagementsystem (DMS) *Evana360* vorgestellt. So konnten die Studierenden einen Überblick über die bestehenden Funktionalitäten des DMS und der Geschäftsdomäne erhalten. Damit die Studierenden jedoch nicht auf ein bereits vertieftes Wissen während der iterativen Migration zurückgreifen konnten, wurde die Vorstellung auf ein Minimum reduziert. Gezeigt wurden die Benutzeroberfläche und die grundlegenden Funktionalitäten der *Dateiverwaltung* und des *Datenraums*. Beide Projektteams haben aufgrund einer notwendigen Vergleichbarkeit dieselbe Vorstellung durch die Domänenexperten erhalten. Der Aufbau eines vertieften Wissens bezüglich der Geschäftsdomäne und des Softwaresystems sollte durch die Systematik des Migrationsrahmenwerks erfolgen. Die Migrationsvorhaben der beiden Semester wurden als unabhängig voneinander angesehen, sodass jedes Projektteam ein neues Migrationsvorhaben beschlossen und umgesetzt hat. Dadurch war es möglich, die Systematik des Migrationsrahmenwerks auf denselben Monolithen anzuwenden und Rückmeldungen zur Systematik, welche durch das Projektteam im *SoSe 2019* erzielt wurden, für das Projektteam im *WiSe 2019/2020* zu übernehmen und Anpassungen zu erproben. Bei einem zusammenhängenden Migrationsvorhaben hätten die Rückmeldungen des vorangegangenen Projektteams und die daraus resultierenden Verbesserungen nicht beobachtet werden können. Pro Semester wurde genau eine Migrationsiteration durchgeführt. Zur Durchführung griffen die Studierenden ohne Hilfestellungen durch den Autoren auf das Migrationsrahmenwerk zurück. Auf Hilfestellungen wurde explizit verzichtet, um Schwachstellen in Migrationsaktivitäten oder auch fehlende Migrationsaktivitäten feststellen zu können. Beiden Projektteams wurde zum Starten einer Migrationsiteration ein Migrationsauslöser durch die Domänenexperten der Evana AG genannt. Dem Projektteam im *SoSe 2019* wurde folgende neue funktionale Anforderung genannt: *Sperren von Ordnern in einem Datenraum*. Das Projektteam im *WiSe 2019/2020* bearbeitete die funktionale Anforderung *Versionierung von Dateien*. Ausschnitte der Ergebnisse der Migrationsiteration des Projektteams im *WiSe 2019/2020* wurden durchgängig in den Beitragskapiteln dieser Forschungsarbeit verwendet.

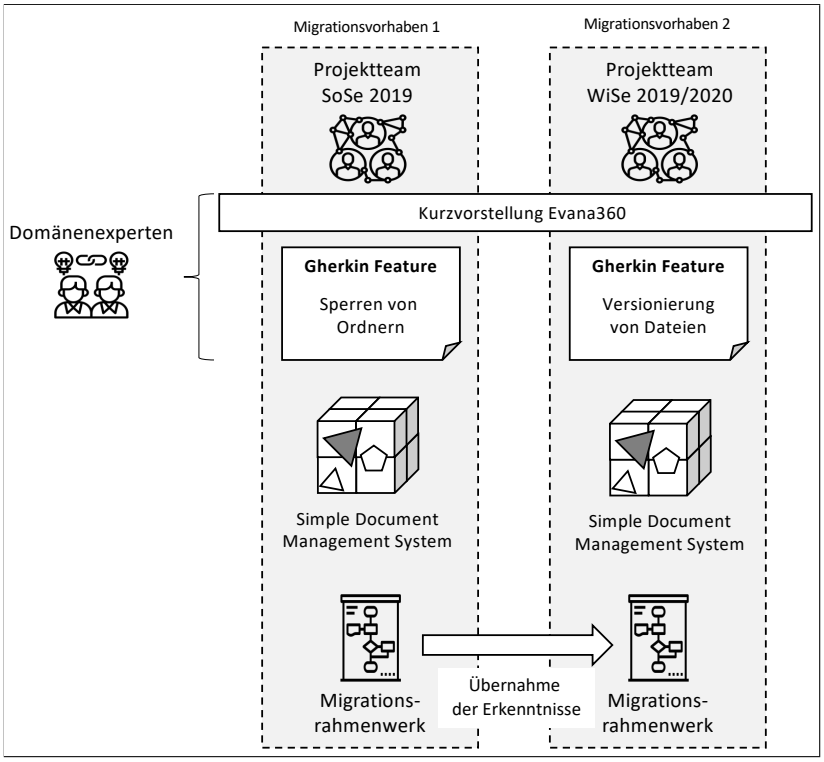

Abbildung 8.5: Projektkontext der beiden Semester SoSe 2019 und WiSe 2019/2020

8.5.3 Ergebnisse der Typ I-Validierung

Die Ergebnisse der Typ I-Validierung werden nun nachfolgend semesterweise vorgestellt. Bei den Ergebnissen geht es nicht um die Migrationsartefakte, die durch die Projektteams erstellt wurden, sondern um die Eignung der Systematik für die Durchführung einer Migrationsiteration. Zudem wird die Implementierung der einzelnen Softwarebausteine, wie auch im Migrationsrahmenwerk, nicht berücksichtigt. Der Fokus liegt auf den Analyse- und Entwurfsmigrationsartefakten. Von Relevanz ist damit der Grad der Unterstützung, den das Projektteam durch das Migrationsrahmenwerk erhalten hat. Gemessen wurde dies anhand zweier Metriken: (1) *Anzahl Rückfragen* und (2) *Anzahl Korrekturen*. Zur besseren Veranschaulichung der Rückfragen wurden diese in *Kategorien* eingeordnet. Beide Metriken sind bei einer hohen Anzahl als negativ zu bewerten, sodass bei geringen Rückfragen und Korrekturen eine hohe Eignung des Migrationsrahmenwerks vorliegt. Weiterführend werden die *Zeitaufwände für Rückfragen und Korrekturen* in Relation zu der Durchführung der eigentlichen Migrationsaktivitäten gestellt. Möglich ist dies durch den von den Studierenden gepflegten *Stundenzettel* bei C&M. Berücksichtigt werden Einträge des Typs "*Besprechung*", die nicht im Kontext von Workshops mit den Domänenexperten stattfanden.

Die Erhebung der Daten erfolgte auf Basis unterschiedlicher Kommunikationsmittel wie E-Mail, Slack, Microsoft Teams und GitLab. Ein Fragebogen zur Durchführung eines Interviews wurde nicht

Tabelle 8.2: Ergebnisse der Typ I-Validierung

	SoSe 2019	
# Studierende	4	
Stundenkontingent des Projektteams	600h	
Zeitaufwand für außerordentliche Besprechungen	98,5h	
	Rückfragen	**Korrekturen**
# Spezifikation des Migrationsauslösers	3	0
# Modellierung der Subdomänen	2	1
# Wiederherstellung der Architekturbeschreibung	5	3
# Extraktion des Monolithenausschnitts	5	2
# Überarbeitung der Anforderungsspezifikation	1	1
# Entwurf der Microservice-Architektur	3	2
# Vorbereitungen für die Inbetriebnahme	1	0
Gesamt	20	9
# Workshops mit den Domänenexperten	2	

vorgesehen.

In Tabelle 8.2 wird das Ergebnis des Projektteams vom *SoSe 2019* zusammengefasst. Die Studierenden stellten insgesamt 20 Rückfragen, die primär die Extraktionsphase betrafen. Von den 20 Rückfragen wurden 15 Rückfragen und somit 75% der Gesamtanzahl bezüglich der Migrationsaktivitäten in der Extraktionsphase formuliert. Dies zeigt eine klare Tendenz bei der Verteilung des Schwierigkeitsgrad innerhalb der Systematik. Die größten Herausforderungen sind somit in der Extraktionsphase angesiedelt. Dies korreliert auch mit der Anzahl der Korrekturen, die durch die Domänenexperten und den Autor dieser Forschungsarbeit durchgeführt werden mussten. Ausgehend von dieser Rückmeldung des Projektteams wurden die Erkenntnisse und Problemstellungen in die Extraktionphase übernommen. Es zeigte sich auch, dass die doch geringe Anzahl der Workshops mit den Domänenexperten mit der Anzahl der Rückfragen korrelierte. Der betriebene Zeitaufwand für außerordentliche Besprechungen, in denen Rückfragen thematisiert wurden, belief sich auf 98,5 Stunden. In Relation zu dem Gesamtkontingent von 600 Stunden nahmen die Rückfragen $16,4\%$ des zeitlichen Aufwands ein (Abbildung 8.6).

Die Durchführung der Migrationsiteration des *WiSe 2019/2020* beruhte auf der überarbeiteten Systematik. Die Rückmeldung des Projektteams im Vergleich zum vorangegangen Semester wird in Tabelle 8.3 festgehalten. Die Anzahl der Rückfragen hat sich durch die überarbeitete Systematik deutlich reduziert. Insgesamt wurden 6 Rückfragen gestellt, wobei die Mehrheit der Rückfragen immer noch die Extraktionsphase betrafen. Dennoch stellt dies eine Reduzierung der Rückfragen um 70% dar. Auch der Zeitaufwand, der für außerordentliche Besprechungen notwendig war, reduzierte sich auf 24,5 Stunden, was $4,08\%$ vom Gesamtkontingent des zeitlichen Aufwands ausmacht (Abbildung 8.7). Erhöht hatte sich dagegen die Anzahl der Workshops, die das Projektteam mit den Domänenexperten durchführte. Die überarbeitete Systematik gab den Studierenden noch expliziter die Einbindung der Domänenexperten vor, wodurch diese Steigerung zustande kam. Es wurden deutlich mehr Migrationsartefakte kollaborativ erstellt. Dies ist auch der Grund für die geringere Anzahl an

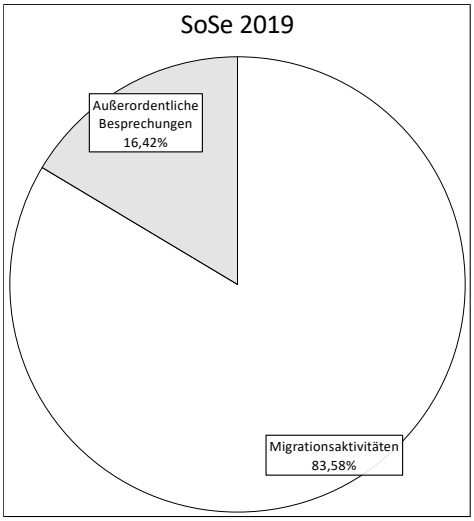

Abbildung 8.6: Zeitlicher Aufwand für außerordentliche Besprechungen

Tabelle 8.3: Ergebnisse der Typ I-Validierung

	SoSe 2019		**WiSe 2019/2020**	
# Studierende	4		4	
Stundenkontingent des Projektteams	600h		600h	
Zeitaufwand für außerordentliche Besprechungen	98,5h		24,5h	
	Rückfragen	**Korrekturen**	**Rückfragen**	**Korrekturen**
# Spezifikation des Migrationsauslösers	3	0	0	0
# Modellierung der Subdomänen	2	1	0	0
# Wiederherstellung der Architekturbeschreibung	5	3	2	1
# Extraktion des Monolithenausschnitts	5	2	2	1
# Überarbeitung der Anforderungsspezifikation	1	1	0	0
# Entwurf der Microservice-Architektur	3	2	2	1
# Vorbereitungen für die Inbetriebnahme	1	0	0	0
Gesamt	20	9	6	3
# Workshops mit den Domänenexperten	2		7	

Rückfragen und Korrekturen in den Migrationsaktivitäten die eine Zusammenarbeit vorsehen wie beispielsweise bei der Modellierung der Subdomänen.

Aus den gesammelten Daten lässt sich erkennen, dass die Überarbeitung der Systematik für das WiSe 2019/2020 den gewünschten Effekt hatte. Außerdem kann aus der geringen Anzahl an Rückfragen und Korrekturen geschlossen werden, dass die Systematik die Studierenden an den relevanten Stellen maßgeblich unterstützt. Die Reduzierung der außerordentlichen Besprechungen zeigt auch, dass die Studierenden sich nur wenig mit Aktivitäten außerhalb der Migrationsaktivitäten beschäftigen mussten, was auf eine effiziente Ausführung der Migrationsiteration schließen lässt. Weiterhin kann festgehalten werden, dass beide Migrationsteams durch die Systematik die gestartete Migrationsiteration abschließen konnten. Dies untermauert die Effektivität des Migrationsrahmenwerks.

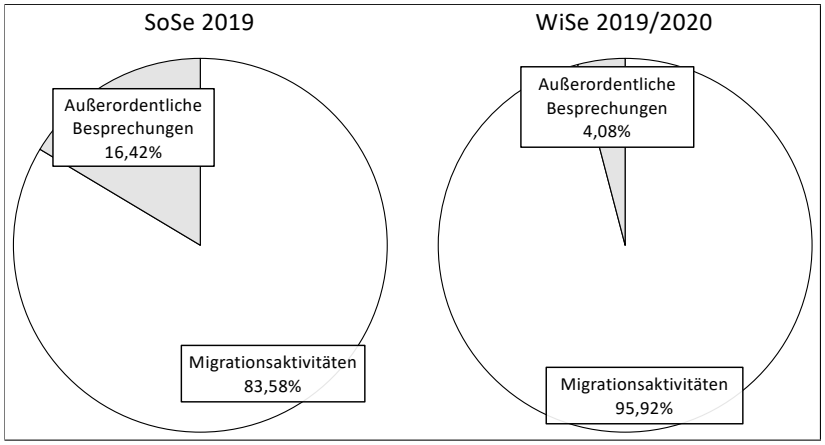

Abbildung 8.7: Vergleich des zeitlichen Aufwands für außerordentliche Besprechungen

8.5.4 Bedrohung und Einschränkung der Validität

Der Wert der hier durchgeführten Typ I-Validierung wird anhand der Bedrohungen und Einschränkungen der Validität aus Abschnitt 8.3 ermittelt.

Ergebnisvalidität Als positiv zu bewerten sind die Rahmenbedingungen der durchgeführten Studien. In beiden Semestern wurde die gleiche Anzahl an Studierenden in die Projektteams aufgenommen, sodass das Stundenkontingent identisch war. Zwar unterscheiden sich die Studierenden in ihren Fähigkeiten, doch diese Divergenz ist zu vernachlässigen, denn es ist nicht möglich Studierende mit denselben Fähigkeiten zu finden. Weiterhin erwartete die Studierenden die exakt selbe Ausgangssituation, sodass kein Wissensvorsprung vorherrschen konnte. Kritisch zu betrachten ist jedoch die Güte der Daten selbst, denn diese wurden über informelle Kanäle und unstrukturiert erhoben. Auch die Betrachtung der zeitlichen Aufwände für außerordentliche Besprechungen beruht auf einer hohen Selbstdisziplin von Studierenden, die solche Besprechungen in den Stundenzetteln vermerken mussten. Zusammenfassend kann die Ergebnisvalidität als nicht hoch eingeschätzt werden. Dennoch können aus den Daten Tendenzen festgehalten werden, die nicht zu vernachlässigen sind.

Interne Validität Im Vordergrund dieser Validitätsbewertung stehen die Studierenden als zentrale Variable der Studien. Neben den schwankenden Fähigkeiten von Studierendem zu Studierendem spielen noch weitere externe Einflussfaktoren eine Rolle. Die Ursache für die Reduzierung der Rückfragen und Korrekturen muss damit nicht unbedingt in der Verbesserung der Systematik liegen. Studierende können beispielsweise gerade gegen Ende der Vorlesungszeit durch Prüfungssituationen das Interesse an der Studie verlieren, wodurch die Fehlerrate steigt und das Stellen von Rückfragen sinkt. Weiterhin können auch persönliche Faktoren wie Ängste oder Schüchternheit zur Reduzierung der Rückfragen beigetragen haben. Doch all diese Einflüsse

lassen sich in studentischen Experimenten nicht auflösen und sind daher hinzunehmen. So wird davon ausgegangen, dass die Ursache und Wirkung der Studien in kausaler Abhängigkeit zueinander stehen.

Konstruktionsvalidität Es kann nicht ausgeschlossen werden, dass die an der Studie teilnehmenden Studierenden bereits Erfahrungen oder sonstige Kenntnisse im Kontext der iterativen Migration von monolithischen Softwaresystemen gesammelt hatten. Dennoch wurde bei der Durchführung der Migrationsiteration auf eine genaue Einhaltung der Migrationssystematik aus dem Migrationsrahmenwerk geachtet. Nicht ausgeschlossen werden kann auch eine vorsätzliche Verfälschung der Studien durch die Studierenden. Aus diesem Grund wurden die Studierenden nicht über die nachträglichen Untersuchungen informiert. Die Ergebnisse, welche die Studierenden durch die Ausführung der Systematik erzielt haben, konnten nicht verfälscht werden. In wöchentlichen Besprechungen wurden die Ergebnisse ausführlich diskutiert und von den Domänenexperten abgenommen.

Externe Validität Damit die Bedrohung der externen Validität möglichst gering blieb, basierten die Studien auf dem durch die Evana AG gegebenen Industriekontext. Zudem wurden die beiden Studien anhand unterschiedlicher Migrationsauslöser betrachtet, um verschiedene Aspekte in den Migrationsiterationen betrachten zu können. In der Regel werden in Organisationen die Migrationen von erfahrenen Softwareentwicklern und einem Softwarearchitekten durchgeführt. Bei den Studierenden wird davon ausgegangen, dass diese bislang nur wenig Erfahrung im Bereich der praxisnahen Softwareentwicklung gesammelt hatten. Aus diesem Grund fördert der erfolgreiche Einsatz von Studierenden die externe Validität des Migrationsrahmenwerks. Denn gelingt es unerfahrenen Studierenden, Migrationsiterationen anhand der Systematik abzuschließen, sollten erfahrene Softwareentwickler und Architekten noch weniger Probleme haben.

8.5.5 Zusammenfassung

Die Erhebung der Daten erfolgte im Kontext zweier Studien, die denselben Rahmenbedingungen unterlagen. Unter Berücksichtigung der Bedrohungen und Einschränkungen der Validität kann aufgrund der Studien festgehalten werden, dass der Einsatz des Migrationsrahmenwerks in einem industrienahen Kontext einen positiven Einfluss auf die Durchführung von Migrationsiterationen nimmt. Innerhalb einer Migrationsiteration konnte eine geringe Anzahl an Rückfragen, welche die Ausführung der Systematik betrafen, erzielt werden. Weiterhin waren nur noch wenige Korrekturen an den Ergebnissen des Migrationsteams durch die Domänenexperten notwendig.

8.6 Typ II-Validierung - Anwendbarkeit

Die Typ II-Validierung (Anwendbarkeit) beruht auf der Verwendung des Migrationsrahmenwerks im industriellen Kontext. Dieser Kontext wird durch den Kooperationspartner Evana AG gestellt. Im Fokus der Typ II-Validierung stehen die Forschungsbeiträge *Entwurf eines Migrationsrahmenwerks* (B1) und *Bestimmen des Umfangs eines Inkrements* (B2) (vgl. Abschnitt 1.5). Die Einschränkung auf diese beiden Forschungsbeiträge ist begründet durch den hohen Aufwand der Erhebung der Daten für die gesamte Systematik. Somit bleibt die Typ II-Validierung in einem beherrschbaren Rahmen. Anzumerken ist jedoch auch, dass die Darstellung der Ergebnisse insbesondere aus der Modernisierungsphase durch das Berücksichtigen einer *Vertraulichkeitsvereinbarung* (engl. Non-Disclosure Agreement, NDA) nicht sinnvoll möglich ist. Weiterer Gegenstand der NDA war zudem der Verzicht auf die Darstellung von Quellcode. Jeglicher dargestellter Quellcode wurde vereinfacht, damit vertrauliche Stellen nicht dargestellt wurden. Die Evana AG legt einen großen Wert auf die Vertraulichkeit der Ergebnisse aus dieser Validierung.

Betrachtet wird das bereits vorgestellte monolithische Softwaresystem *Evana360*. Als realer Migrationsauslöser wurde die *Versionierung von Dateien* in Evana360 betrachtet. Dieser Migrationsauslöser wurde bereits in der Typ I-Validierung durch das studentische Projektteam im *WiSe 2019/2020* (vgl. Abschnitt 8.5) näher beleuchtet. Dennoch war es ein Anliegen der Evana AG, diese neue funktionale Anforderung als Auslöser bei der Validierung zu verwenden, damit diese im Softwaresystem vorgesehen wird; dies unterstreicht die Realitätsnähe der Typ I-Validierung. Als Migrationsteam stellte Evana AG *einen Softwarearchitekten* und vier Softwareentwickler bereit, deren primäre Aufgabe die Durchführung der Migration war. Bei den Softwareentwicklern handelte es sich um *einen leitenden Softwareentwickler*, *zwei Senior-Softwareentwickler* und *einen Junior-Softwareentwickler*. Besondere Rollen wie *Development & Operations-Engineerings* (DevOps-Engineerings) waren nicht vertreten. Der Beschluss des Migrationsvorhabens wurde durch den Vorstand bestätigt, wobei ein *Zeitraum von 2 Monaten* für die Migrationsiteration angesetzt wurde.

Begonnen wird mit Abschnitt 8.6.1, der bisherige Migrationsversuche der Evana AG ohne das Migrationsrahmenwerk mit dem Migrationsvorhaben mit vorhandenem Migrationsrahmenwerk vergleicht. Anschließend werden mit Abschnitt 8.6.2 die Ergebnisse der Durchführung der Extraktionsphase präsentiert.

8.6.1 Bisheriges Migrationsvorgehen und Einsatz des Migrationsrahmenwerks

Die an die Typ II-Validierung geknüpfte Durchführung einer Migrationsiteration ist nicht das erste Migrationsvorhaben der Evana AG. Unabhängig von dieser Forschungsarbeit wurden bereits zwei Migrationsversuche an Evana360 vollzogen:

Ausgliederung der künstlichen Intelligenz als Microservices Der erste Migrationsversuch galt der Ausgliederung der Funktionalitäten, welche dem Bereich der künstlichen Intelligenz (KI) zugeordnet werden konnten, zu eigenständigen Microservices. Die Auslöser für das Vorhaben waren die nicht-funktionalen Anforderungen *Performance* und *Skalierbarkeit* [CN+12], denn der Hardwareressourcenbedarf der KI-Funktionalitäten überstieg die des eigentlichen Dokumentenmanagementsystems. Umgesetzt wurde das Vorhaben durch zwei Senior-Softwareentwickler in einem Zeitraum von 4 Monaten. Beide Softwareentwickler als auch die Leitung des Entwicklungsteams hatten keine Erfahrung mit Migrationen und den damit verbundenen Softwaretechnikansätzen. Folglich wurde die Migration ohne ein systematisches und nachvollziehbares Vorgehen durchgeführt. Die Phasen einer detaillierten Analyse und des Entwurfs wurden ignoriert und die Migration begann direkt mit der Implementierung von Softwarebausteinen. Durch die fehlende Entwurfsphase entstanden als Ergebnis der Migration eigenständig bereitstellbare Softwarebausteine, die jedoch über Web-Schnittstellen eine hohe Kopplung zum bestehenden Monolithen Evana360 beibehielten und nicht den üblichen Definitionen von Microservices [FL14, Ne15, NM+16, BP19] entsprachen. Weiterhin waren die Anforderungen durch die fehlende Analysephase nicht mit den Interessenvertretern abschließend diskutiert worden, weshalb die neuen Softwarebausteine nicht den eigentlichen Bedürfnissen entsprachen. Das Migrationsvorhaben wurde von der Evana AG als teilweise erfolgreich bewertet. Zwar konnten die Softwarebausteine unabhängig von dem Monolithen skaliert werden, doch durch das Ignorieren der Analyse- und Entwurfsphase wurden *technische Schulden* dem Softwaresystem hinzugefügt, die zum Zeitpunkt dieser Forschungsarbeit weiterhin bestehen.

Modernisierung des Monolithen in Microservices Ein zweiter Migrationsversuch wurde für die Modernisierung des gesamten Monolithen Evana360 gestartet. Aufgrund von Lastproblemen sollte der Monolith in einzelne Microservices aufgeteilt werden, um wie auch bei dem ersten Migrationsvorhaben die Last durch eine Skalierbarkeit des Softwaresystems auszugleichen. Die Umsetzung dieses Migrationsvorhabens erfolgte mittels vier Softwareentwickler über einen geplanten Zeitraum von drei Monaten. Die Erfahrungsstufen der Softwareentwickler waren wie folgt: ein leitender Softwareentwickler, zwei Senior-Softwareentwickler und ein Junior-Softwareentwickler. Um systematisch an diesen Migrationsversuch heranzugehen, wurde initial ein Vorgehen definiert (Abbildung 8.8). Die Systematik sah keine iterative Migration vor, sondern strebte einen *Big Bang* an. Somit wurden die einzelnen Phasen der Systematik auf den gesamten Monolithen angewandt. Während des Migrationsvorhabens traten verschiedene Problemstellungen auf, die letztendlich dazu geführt haben, dass das Migrationsvorhaben abgebrochen beziehungsweise das Ziel verändert wurde. Nachfolgend werden die Problemstellungen kurz aufgezählt: (1) *Unterbrechungen der Migration durch Störungen von außen*, (2) *unrealistischer Zeitrahmen der Migration für das gesamte Softwaresystem*, (3) *fehlende Erfahrung der Softwareentwickler*, (4) *keine Einbindung von Interessenvertretern*, (5) *Weiterentwicklung*

des Monolithen neben der Migration und (6) *fehlendes Verständnis über die Geschäftsdomäne*. Durchlaufen wurde lediglich die erste Phase der aufgestellten Systematik, die als Ergebnis eine unstrukturierte Liste an textuell beschriebenen funktionalen Anforderungen hatte. Es wurden damit keine formellen Softwareartefakte verwendet, wodurch eine Nachvollziehbarkeit nur wenig gegeben war. Nach Abschluss der ersten Phasen waren bereits $\frac{2}{3}$ des vorgegebenen Zeitrahmens erreicht.

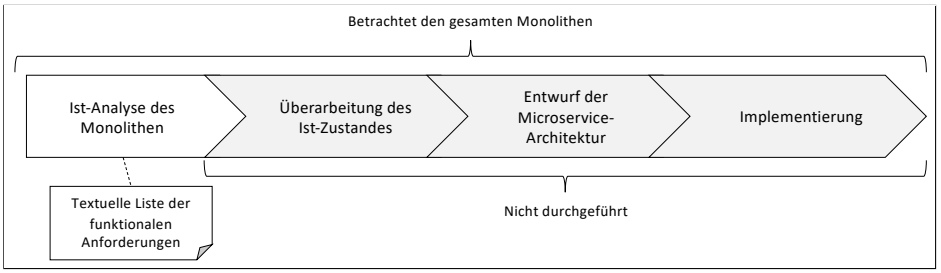

Abbildung 8.8: Systematisches Vorgehen des zweiten Migrationsversuches der Evana AG

Ausgehend von den beiden beschriebenen Migrationsvorhaben hatte die Evana AG ein hohes Interesse an einem systematischen und nachvollziehbaren Vorgehen, welches vor allem die Problemstellungen des letzten Migrationsvorhabens auflöst. Die aufgetretenen Problemstellungen werden als praxisnahe Problemstellungen angesehen, die durchaus in verschiedenen Organisationen auftreten werden. Aus diesem Grund erfolgt die Bewertung der Anwendbarkeit des Migrationsrahmenwerks im Vergleich zu dem letzten Migrationsvorhaben. Im weiteren Verlauf der Typ II-Validierung wird zur besseren Abgrenzung der Migrationsvorhaben das neue Migrationsvorhaben auf Basis des Migrationsrahmenwerks als Studie bezeichnet.

Zu Beginn der Studie wurden über einen Bewertungsbogen (vgl. Anhang A.1) die Wissensstände der einzelnen Teammitglieder des Migrationsteams erhoben. Das Ergebnis ist der Abbildung 8.9 und der Tabelle 8.4 zu entnehmen. Für die Beantwortung der Fragen in dem Fragenbogen wurden den Teammitglieder fünf Auswahlmöglichkeiten zur Bewertung vorgegeben. Diese fünf Auswahlmöglichkeiten lassen sich auf die Skala 1 bis 5 abbilden, wobei die Interpretation der Auswahlmöglichkeiten im Kontext der jeweiligen Frage variiert; bei der Auswertung der Frage wird die Interpretation der Skala immer mit angegeben. Betrachtet wurden dabei Themengebiete der Softwareentwicklung, welche die Migration und das Migrationsrahmenwerk betreffen. Ein Großteil der Teammitglieder hatte wenig bis gar keine Erfahrung mit iterativen Migrationen und den angewandten Softwareentwicklungsansätzen. Deutlich bekannter waren dagegen die Microservices, wobei der Bewertungsbogen keine detaillierten Konzepte hinter dem Architekturstil erfragt hat. Ebenfalls bekannter waren Modellierungssprache wie die Unified Modeling Language (UML). Für die Durchführung der Migrationsiteration und die Verwendung der Systematik kann daher abgeleitet werden, dass das Migrationsteam hohen Unterstützungsbedarf hatte.

Tabelle 8.4: Übersicht der Teammitglieder des Migrationsteams

Rolle	Anzahl
Softwareentwickler (Junior)	1
Softwareentwickler (Senior)	2
Softwareentwickler (Leitend)	1
Softwarearchitekt	1

Berufsjahre	Anzahl
Weniger als 1 Jahr	0
1 - 3 Jahre	1
3 - 5 Jahre	3
5 - 8 Jahre	1
8 - 10 Jahre	0
Über 10 Jahre	0

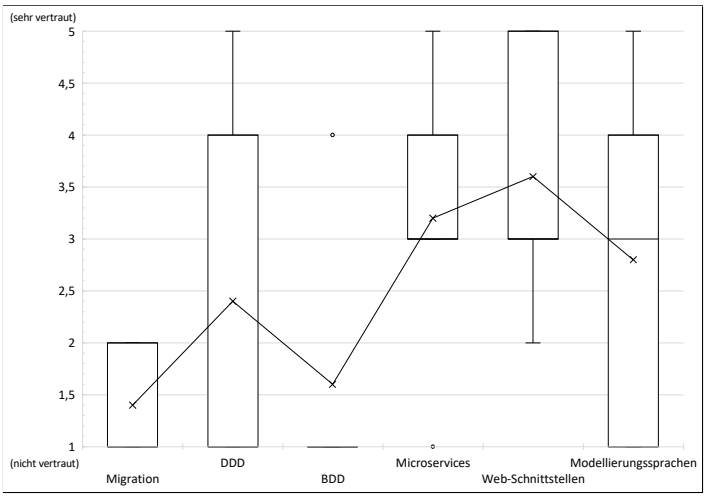

Abbildung 8.9: Ergebnisse der Wissensstände in relevanten Bereichen der Softwareentwicklung

Nach der Durchführung einer einzelnen Migrationsiteration wurde das Migrationsvorhaben beendet. Anhand des Bewertungsbogens wurde die Anwendbarkeit des Migrationsrahmenwerks durch das Migrationsteam bewertet. Die Darstellung der Ergebnisse erfolgt in fünf Kategorien, die ebenfalls so in dem Bewertungsbogen abgebildet wurden: (1) *Allgemeine Informationen zum Migrationsablauf*, (2) *Durchführung der Extraktionsphase*, (3) *Durchführung der Modernisierungsphase*, (4) *Durchführung der Inbetriebnahmephase* und (5) *Bewertung des Migrationsrahmenwerks*.

Allgemeine Informationen zum Migrationsablauf In dieser Kategorie wurden zunächst Informationen über das Migrationsvorhaben und den allgemeinen Verlauf der Migrationsiteration gesammelt. Aus den Antworten des Migrationsteams geht hervor, dass zwischen *7 - 10 Personen* an der Migrationsiteration beteiligt waren. Somit wurden mindestens zwei, jedoch maximal fünf Interessenvertreter beziehungsweise Domänenexperten in die Migrationsiteration integriert. Dies zeigt, dass die Migrationsaktivitäten dazu geführt haben, dass das Migrationsteam auf das Domänenwissen der Interessenvertreter zurückgegriffen hat. Weiterhin kann als eindeutiges Ergebnis festgehalten werden, dass das Migrationsteam die Migrationsiteration im vorgegebenen Zeitraum abgeschlossen hat, auch wenn Teammitglieder vereinzelt von dem Projekt abgezogen wurden. Die Unterbrechungen traten vor allem bei den höheren Erfahrungsstufen auf, die zusätzlich Verantwortung in anderen Projekten übernommen haben. An dieser Stelle

muss das Bewusstsein für die *Isolierung* und die *Auswahlkriterien* für Teammitglieder geschärft werden.

Durchführung der Extraktionsphase Das übergeordnete Ergebnis der Durchführung der Extraktionsphase ist, dass die vorgestellte Extraktion zum Entwurf eines Monolithenausschnitts genügt hat. Die Studie sah, wie bereits festgehalten, den Einsatz einer funktionalen Anforderung als Migrationsauslöser vor. Einheitlich waren die Teilnehmer der Meinung, dass nicht alle Aspekte für die Migrationsiteration von der Extraktion berücksichtigt werden. Jeder Teilnehmer sah die *Implementierung* als fehlend an und wünschte sich in dieser Phase Unterstützung. Dieses eindeutige Ergebnis kann mit einer nicht ausreichenden Aufklärung über die entworfene Extraktion zusammenhängen, da die Implementierung absichtlich nicht betrachtet wurde. Es zeigt jedoch, dass Migrationsteams auch bei der Implementierung Herausforderungen sehen und dort eine Systematik benötigen. Zwei der Teilnehmer wiesen auf die Bereitstellung beziehungsweise den Betrieb des Monolithenausschnitts hin. Ein weiterer Teilnehmer nahm Bezug auf die von Dritten zur Verfügung gestellten Bibliotheken, welche bei der Implementierung des Monolithen verwendet wurden. Das explizite Festhalten der Abhängigkeiten wurde gewünscht. In dieser Migrationsiteration wurden *sieben Softwarebausteine* des Monolithen als Umfang des Monolithenausschnitts gewählt. Der Umfang wurde von allen Teammitgliedern als passend und beherrschbar erachtet. Es zeigte sich, dass die ubiquitäre Sprache nicht konsequent eingehalten wurde. Zwei Teammitglieder empfanden den Aufwand als zu hoch und sahen keinen Nutzen darin. Die restlichen Teammitglieder konnten die Interessenvertreter nicht von den festgelegten Begrifflichkeiten überzeugen. Ein Teammitglied gab sogar an, dass durch die Nichtanwendung der ubiquitären Sprache Missverständnisse in der Kommunikation aufgetreten sind. In Abbildung 8.10 wird die Bewertung der Extraktionsphase anhand verschiedener Kriterien festgehalten. Allgemein kann festgehalten werden, dass die Anwendbarkeit der Extraktionsphase ausgehend von dieser Bewertung gegeben ist. Der Extraktionsphase wird jedoch ein hoher zeitlicher Aufwand zugesprochen.

Durchführung der Modernisierungsphase Auch für die Modernisierungsphase kann eindeutig festgehalten werden, dass die dahinter liegende Modernisierung für den Entwurf einer domänengetriebenen Microservice-Architektur verwendet werden kann. Denn alle Teammitglieder gaben an, dass sie eine entsprechende Microservice-Architektur entwerfen konnten. Bei der Überarbeitung der Anforderungsspezifikation war das Migrationsteam der Auffassung, dass die festgelegten Geschäftsziele durch BDD hilfreich waren (Abbildung 8.11). Ebenfalls als positiv zu bewerten ist das Verständnis der einzelnen Mitglieder des Migrationsteams bezüglich der Geschäftsdomäne, das insgesamt als hoch angesehen wird. Lediglich ein Teammitglied gab an, nur ein moderates Verständnis zu besitzen (Abbildung 8.11). Weiterhin zeigt die Abbildung 8.11 das Empfinden des Migrationsteams gegenüber dem abgeleiteten Entwurf. Dem Entwurf wird eine hohe Orientierung an der Geschäftsdomäne zugesprochen und scheint zugleich sehr gut

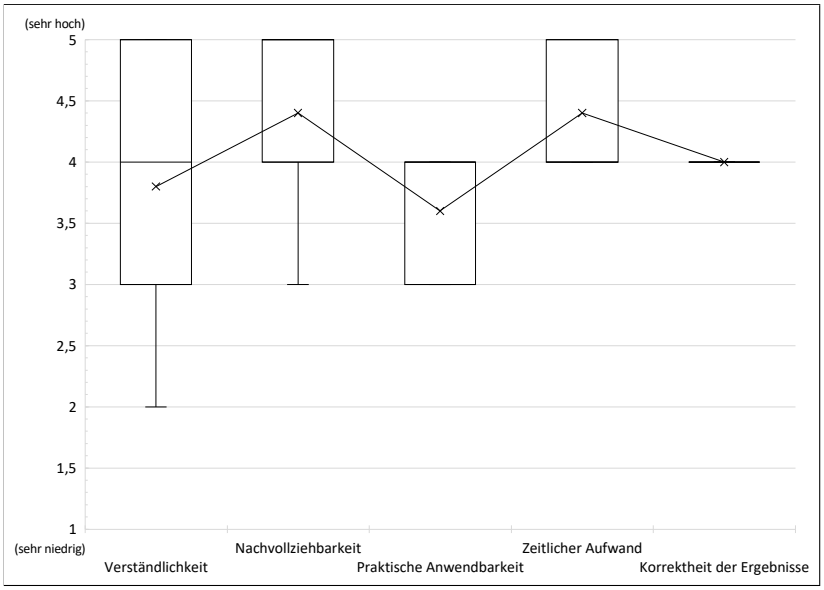

Abbildung 8.10: Ergebnisse der Bewertung der Extraktionsphase anhand verschiedener Kategorien

dokumentiert zu sein. Weniger gut bewertet ist die Effizienz des Entwurfs im Betrieb. Dies deckt sich auch mit der Aussage eines Teammitglieds hinsichtlich eines noch stärker zu setzenden Fokus auf die Einarbeitung von nicht-funktionalen Anforderungen in den Entwurf. Abschließend für die Validierung der Modernisierungsphase wurde hier anhand von vordefinierten Kategorien eine Bewertung vorgenommen (Abbildung 8.12). Der zeitliche Aufwand für die Modernisierungsphase wurde als sehr hoch empfunden. Ein Teammitglied begründete dies durch den hohen Modellierungsaufwand.

Durchführung der Inbetriebnahmephase Bei der Validierung wurde ein eher geringer Fokus auf die Inbetriebnahmephase gelegt. Dennoch wurde das Migrationsteam nach einer Bewertung anhand Kategorien (Abbildung 8.13) und Verbesserungsvorschlägen gefragt. Deutlich zu erkennen ist, dass der zeitliche Aufwand für die Inbetriebnahmephase als sehr gering empfunden wird. Gleichzeitig befindet sich die praktische Anwendbarkeit im moderaten Bereich. Zwei der Teammitglieder gaben an, dass wichtige Aspekte in der Inbetriebnahmephase nicht vorgesehen wurden. Ein weiteres Mitglied wünschte sich das Vorsehen wichtiger *Cloud-native Prinzipien* bei der Implementierung der Microservices, damit ein stabiler Betrieb gewährleistet werden kann. Positiv wurden die Verständlichkeit und Nachvollziehbarkeit gewertet, was möglicherweise mit der geringen Anzahl an Migrationsaktivitäten und der Nähe zur Praxis zusammenhängt.

Bewertung des Migrationsrahmenwerks Zum Abschluss der Validierung wurde das Migrationsteam noch um eine allgemeine Bewertung des Migrationsrahmenwerks gebeten. Zudem sollte eine Aussage bezüglich des Nutzen des Migrationsrahmenwerks bei der Durchführung einer

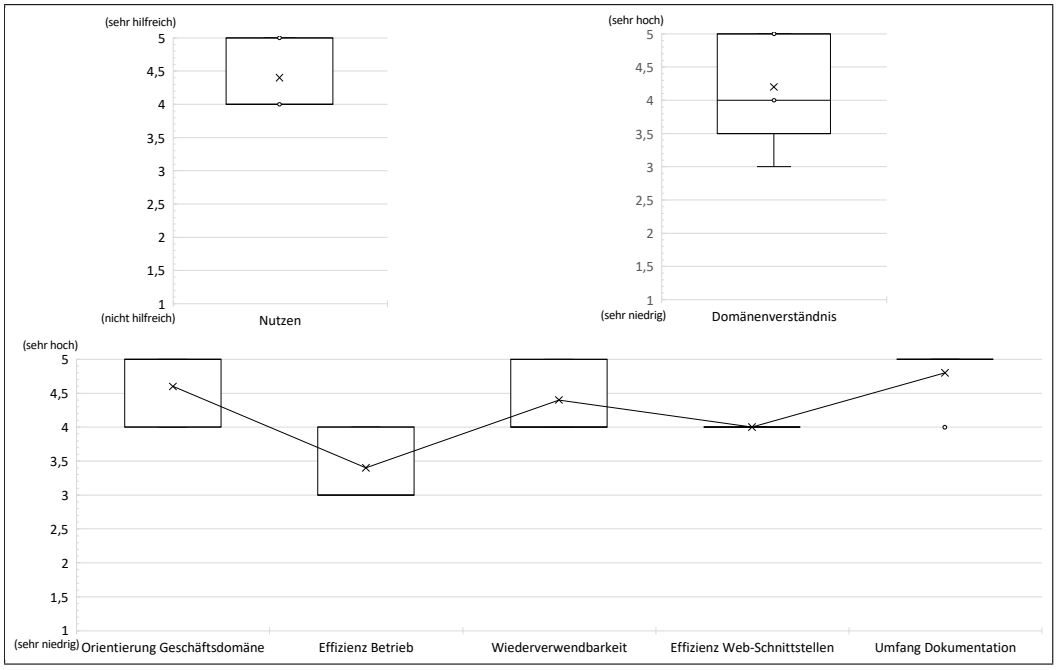

Abbildung 8.11: Ergebnisse der Bewertung verschiedener Aspekte der Modernisierungsphase

Migrationsiteration getroffen werden. Die allgemeine Bewertung kann der Abbildung 8.14 entnommen werden. Insgesamt beurteilte das Migrationsteam das Migrationsrahmenwerk als positiv. Lediglich die Vollständigkeit und Entscheidungsunterstützung wurden vereinzelt als moderat bewertet. Das Ergebnis der Bewertung bezüglich Vollständigkeit lässt sich aus den Bewertungen für die einzelnen Migrationsphasen ableiten. Hinsichtlich der Entscheidungsunterstützung wurden keine textuellen Angaben gemacht. Es ist zu vermuten, dass die geringe Erfahrung des Migrationsteams mit den angewandten Softwareentwicklungsansätzen dazu geführt hat. Besonders hervorzuheben ist die Bewertung der Technologie-Agnostizität, was der Zielsetzung des Migrationsrahmenwerks, das Migrationsteam bei der Umsetzung nicht einzuschränken, zugutekommt. Die allgemeine Nützlichkeit des Migrationsrahmenwerks wurde mit durchschnittlich $4,8$ Punkten bewertet. In künftigen Migrationsvorhaben wird damit das Migrationsrahmenwerk für die Studienteilnehmer von Relevanz sein.

8.6.2 Demonstration der Extraktionsphase

Der zweite Teil der Typ II-Validierung demonstriert die Anwendbarkeit des Migrationsrahmenwerks an der durch das Migrationsteam der Evana AG durchgeführten Migrationsiteration. Es soll gezeigt werden, dass die Durchführung der Systematik und die modellierten Migrationsartefakte das Migrationsteam bei dem Entwurf und Treffen von Entscheidungen ganzheitlich unterstützt. Erneut

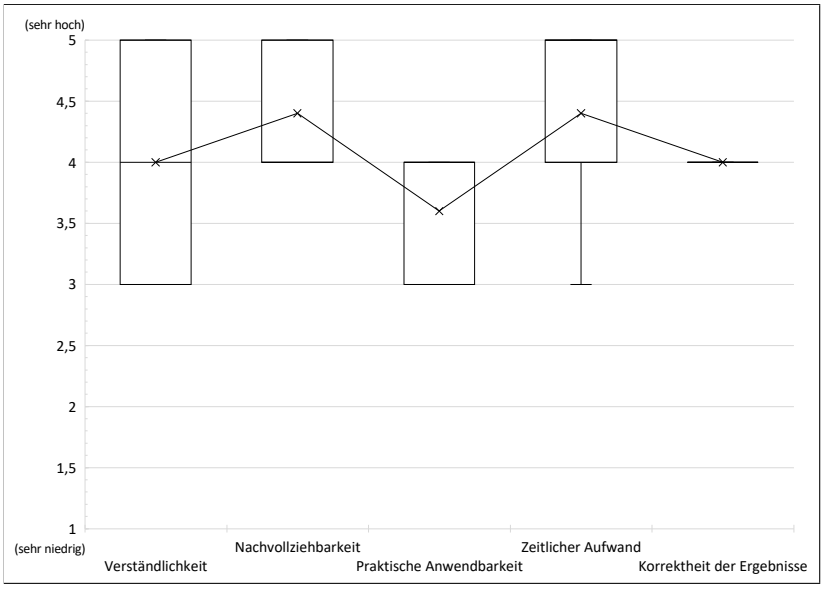

Abbildung 8.12: Ergebnisse der Bewertung der Modernisierungsphase anhand verschiedener Kategorien

wird festgehalten: das Migrationsteam betrachtet als neue funktionale Anforderung an Evana360 die *Versionierung von Dateien*.

```
1  Feature: Versioning of Files
2                As an asset manager
3                I would like to create a new version of my file
4                And restore them as needed
5                So that I can display a history of the file in Evana360
6                And change them in an audit-proof way
7
8          Scenario: Open a dialog to replace the file
9              Given I am on the page for uploading files
10              When I mark a single file
11              Then it opens a context menu where I can upload a new
                    version of the file
12
13         ...
```

Quelltext 8.1: Ausschnitt des Gherkin Features zur Versionierung und Historisierung der Dateien im Dokumentenmanagementsystem

Unter Einbezug von Interessenvertretern hat das Migrationsteam den Migrationsauslöser als *Easy Approach Requirement Specification-Anforderung* (EARS) formuliert (Abbildung 8.16). Anschließend wurden die EARS-Schlüsselwörter zur Klassifikation des Migrationsauslösers herangezogen. In der

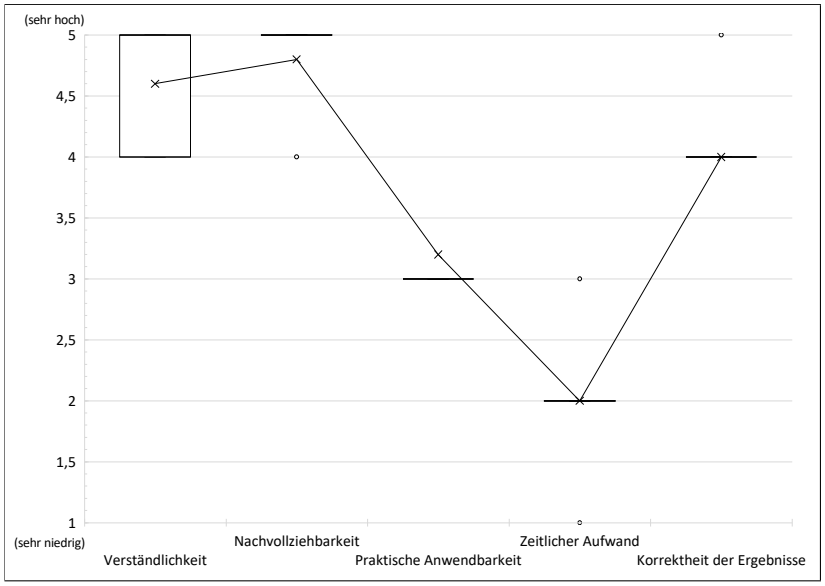

Abbildung 8.13: Ergebnisse der Bewertung der Inbetriebnahmephase anhand verschiedener Kategorien

Anforderung ließen sich die Wörter "*when*" und "*shall*" identifizieren, die dem Migrationsrahmenwerk zufolge eine funktionale Anforderung beschreiben. Es folgte die formale Spezifikation als *Gherkin Feature*. In dem Quellcode 8.1 wird ein kleiner Ausschnitt dieses Gherkin Feature dargestellt. Das gesamte Gherkin Feature findet sich im Anhang A.2 wieder. Auch hier wurden eine enge Kollaboration mit den Interessenvertretern gelebt und ausgehend von der EARS-Anforderung detaillierte Informationen über die Anforderung erhoben.

Wie in der nächsten Migrationsiteration vorgesehen, wurde in einem *Event Storming-Workshop* eine *grobgranulare Context Map* mit Subdomänen modelliert (Abbildung 8.17). Erarbeitet wurden konkret sechs Subdomänen, die zwar nicht alle unmittelbar mit dem Migrationsauslöser in Verbindung stehen, aber aus den Ergebnissen des Event Stormings konnten unproblematisch diese Subdomänen identifiziert werden. Anschließend erfolgte die Priorisierung der Subdomänen und die Zuordnung des Migrationsauslösers zu einer Subdomäne. Bedingt durch die enge Verknüpfung des Auslösers mit Dateien wird der Migrationsauslöser der Subdomäne "*File Management*" zugewiesen.

Nachdem die in der Migrationsiteration betrachtete Subdomäne festgelegt wurde, konnte das Migrationsteam kohäsive Funktionalität identifizieren. Als Ausgangspunkt definierte das Team das "*File*" als *zentrales Domänenobjekt*, welches zur weiteren Identifikation genutzt wurde. Unter Einbezug der Oberfläche und dem Quellcode von Evana360 konnten weitere Funktionalitäten gefunden werden. Diese wurden zunächst in Form eines *Feature-Baumes* festgehalten (Abbildung 8.18) und abschließend mit den Interessenvertretern diskutiert. Nach der Abnahme durch die Interessenvertreter konnte die Formalisierung der Funktionalitäten in *Gherkin Features* erfolgen (vgl. Anhang A.2).

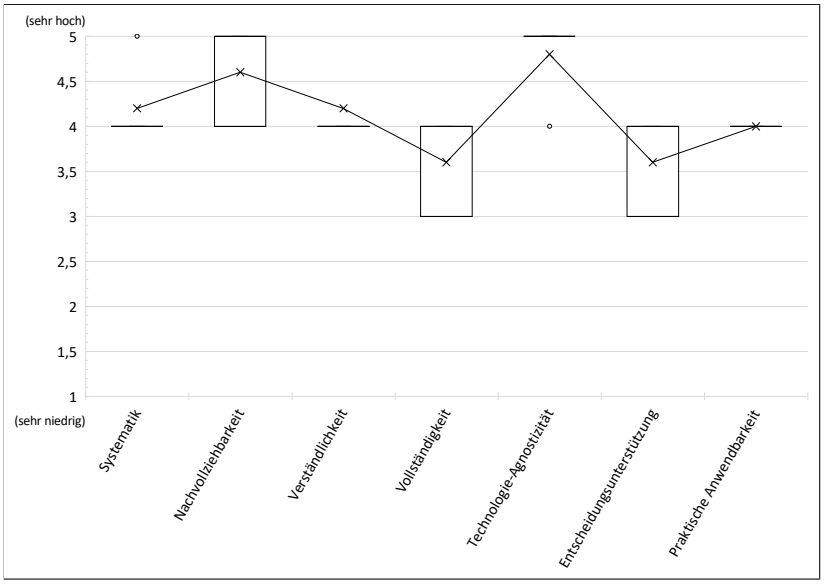

Abbildung 8.14: Ergebnisse der allgemeinen Bewertung des Migrationsrahmenwerks

Im Anschluss wurden aus den Gherkin Features relevante Begriffe für die ubiquitäre Sprache abgeleitet. Neben der Ausführung der Begriffe in einem *Glossar* fertigte das Migrationsteam einen *Language Sketch* an. Dieser kann der Abbildung 8.19 entnommen werden. In Workshops und Diskussionen wurde gerade der Language Sketch zur Darstellung der Abhängigkeiten von Begrifflichkeiten genutzt. Auch wenn, wie das Ergebnis der Bewertungsbögen aus Abschnitt 8.6.1 offenbarte, die ubiquitäre Sprache nicht eingehalten wurde, konnte aus dem Language Sketch ein Mehrwert generiert werden.

Mit der nächsten Migrationsaktivität erstellte das Migrationsteam die *Effect Sketches*. Hierfür wurde zunächst das zentrale Domänenobjekt "*File*" im Quellcode des Monolithen gesichtet. Da Evana360 dem Paradigma der objektorientierten Programmierung folgt, konnte das Migrationsteam die Klasse "*File*" finden. Das Finden der Klasse gestaltete sich aufgrund der Vertrautheit des Migrationsteams mit dem Monolithen einfach. Es stellte sich heraus, dass die gesamte Funktionalität, welche über die Gherkin Features abgebildet wurde, innerhalb dieser Klasse vorzufinden war. Ausgehend davon musste das Migrationsteam nur noch die *mittelbaren Softwarebausteine* (vgl. Abschnitt 5.4.3) identifizieren. Bei der Betrachtung der Nahtstellen wurde festgestellt, dass sich innerhalb der Klasse "*File*" unterschiedliche Schichtenlogik befindet (Abbildung 8.20). Für die Nahtstellen sind zunächst nur die Anwendungs- und Domänenschicht von Relevanz. Erst bei der Betrachtung der Datenhaltung wird der Quellcode der Infrastrukturschicht relevant. Weiterführend wurden verschiedene Typen von Nahtstellen entdeckt. Bei der Entwicklung des Monolithen wurden neben den Abhängigkeiten über Methodenaufrufe auch gleichzeitig Abhängigkeiten durch Zugriffe auf Klassenattribute entdeckt. Zudem wurde nicht auf die Einhaltung des *SOLID-Prinzips Dependency Inversion* [Ma02] geachtet.

Der erste Effect Sketch der Nahtstellenbetrachtung kann der Abbildung 8.21 entnommen werden.

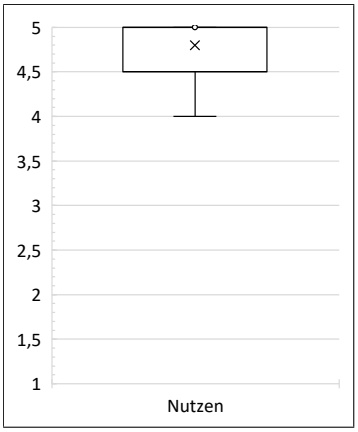

Abbildung 8.15: Ergebnisse der Bewertung zum Nutzen des Migrationsrahmenwerks

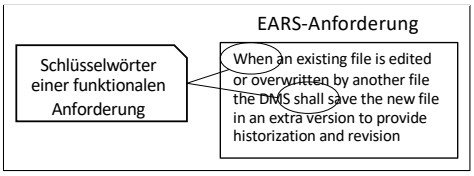

Abbildung 8.16: Versionierung von Dateien als EARS-Anforderung

Es bestehen verschiedene Abhängigkeiten zu vier konkreten Softwarebausteinen: (1) "*FileLink*", (2) "*Task*", (3) "*FilePage*" und (4) "*Model*". Bei "*Model*" liegt eine *Ableitungsbeziehung* zwischen den beiden Softwarebausteinen vor. Die restlichen Abhängigkeiten wurden im Verlauf der Extraktion für den Entwurf der Web-Schnittstellen des Monolithen notwendig.

In einem weiteren Effect Sketch wurden anschließend Abhängigkeiten von weiteren Softwarebausteinen zur "*File*"-Klasse erarbeitet. Denn der Effect Sketch aus Abbildung 8.21 zeigt lediglich die Abhängigkeiten ausgehend von der "*File*"-Klasse. Da jedoch weitere Abhängigkeiten in anderen Softwarebausteinen abgebildet werden können, musste das Migrationsteam andere Softwarebausteine sichten. Die Abbildung 8.22 fasst dieses Ergebnis zusammen. Dadurch, dass sowohl die "*FilePage*" als auch die "*FileLink*"-Klasse eine Abhängigkeit zu der "*File*"-Klasse besaß, lag eine *zyklische Abhängigkeit* vor. Ein neuer abhängiger Softwarebaustein wurde mit der "*Folder*"-Klasse gefunden. Die abgebildeten Abhängigkeiten wurden im späteren Verlauf für den Entwurf von Web-Schnittstellen des Monolithenausschnitts genutzt, damit der verbleibende Monolith auf die Funktionalitäten zurückgreifen konnte.

Wie von der Extraktion vorgesehen, wurde nach den Softwarebausteinen die Datenhaltung betrachtet. Hierfür wurde zunächst im Quellcode der "*File*"-Klasse nach Infrastrukturlogik gesucht. Wie bereits in Abbildung 8.20 festgehalten, wurden mehrere Stellen mit Infrastrukturlogik gefunden. Erarbeitet wurde, dass die gesamte "*File*"-Datenhaltung in einer "*files*"-Tabelle vorzufinden war. Zu und von dieser Tabelle beziehungsweise Entität gingen verschiedene Abhängigkeiten aus. Diese können

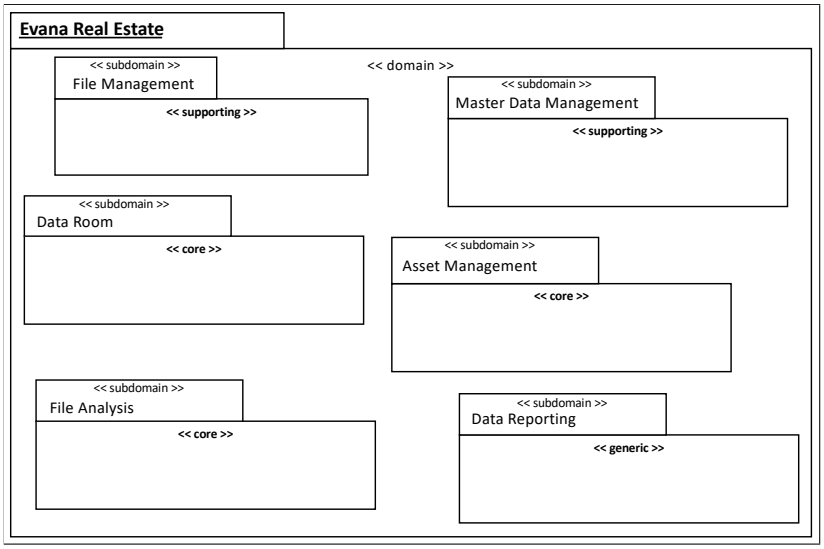

Abbildung 8.17: Context Map zur Erfassung der grobgranularen Domänenstruktur

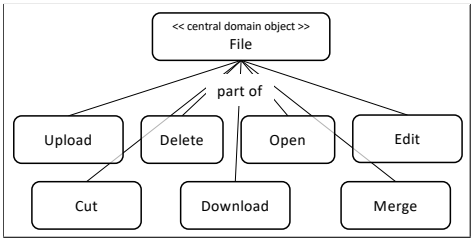

Abbildung 8.18: Mit dem zentralen Domänenobjekt verbundene Funktionalität

der Abbildung 8.23, einem *Entity Relationship-Diagramm* (ER), entnommen werden. Die Entität "*File*" besitzt eine Abhängigkeit zu der Entität "*Client*", einer Repräsentation eines Kunden im Evana360. Für den späteren Entwurf der Web-Schnittstellen bedeutete dies, dass der Monolith das Abfragen der Kundendaten ermöglichen muss. Die Entitäten "*FilePage*", "*FileLink*" und "*Task*" referenzierten auf eine"*File*". Aus diesem Grund musste bei dem Entwurf der Web-Schnittstellen für den Monolithenausschnitt das Abfragen von Dateiinformationen berücksichtigt werden. Entgegen der Erwartung des Migrationsteams, wurde bei der Entität "*Folder*" keine Datenabhängigkeit festgestellt, obwohl in der Anwendungsschicht eine Symbolabhängigkeit zwischen den beiden Klassen festgestellt wurde (Abbildung 8.22).

Ausgehend von den bekannten unmittelbaren und mittelbaren Softwarebausteinen wurde anschließend die logische und physische Ist-Architektur modelliert. Besonders wurde bei der Modellierung auf die Backing-Services geachtet, die für den Betrieb des Monolithen notwendig sind. Der Ist-Zustand wird in Abbildung 8.24 dargestellt.

Nach der Modellierung der logischen und physischen Ist-Architektur modellierte das Migrationsteam

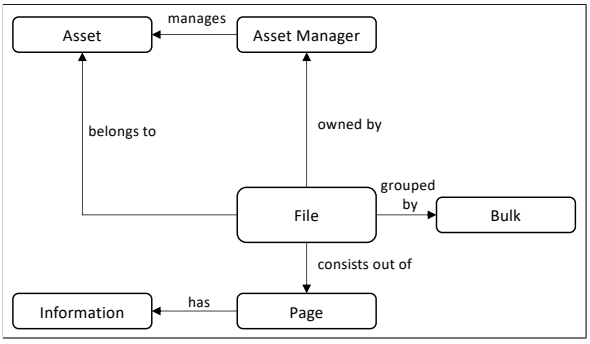

Abbildung 8.19: Language Sketch der Migrationsiteration

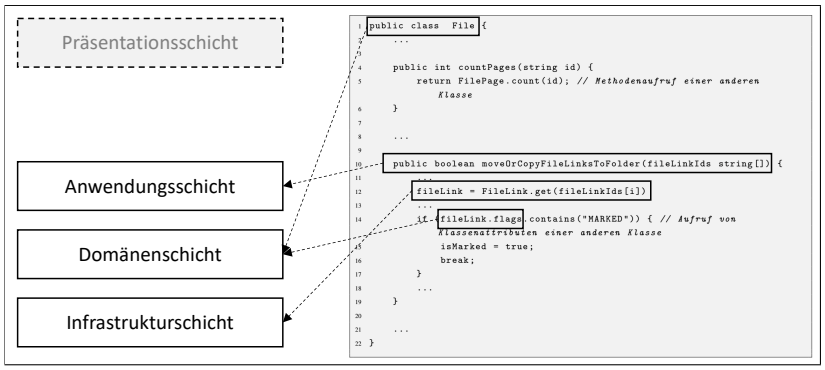

Abbildung 8.20: Einordnung der Quellcodezeilen der "*File*"-Klasse in die Schichtenarchitektur

die konzeptionelle Architektur für den Monolithenausschnitt. Aufgrund der hohen Kohäsion zwischen den Softwarebausteinen "*File*" und "*FilePage*" traf das Migrationsteam die Entscheidung, die beiden Softwarebausteine in den Monolithenausschnitt zu verlagern. Dies bedingte zusätzlich, dass die beiden Tabellen "*Files*" und "*FilePages*" ebenfalls in eine gemeinsame Datenbank ausgelagert wurden. Zudem wurden weitere Softwarebausteine, die zur Ausführung des Monolithenausschnitts notwendig waren, repliziert. Dazu gehörte auch die "*Model*"-Klasse, die im Monolithen durch Abhängigkeiten zu anderen Softwarebausteinen verbleiben musste.

Um die Verteilung auf der physischen Architektur zu ermöglichen, wurden anschließend die Web-APIs entworfen. Die grundlegenden Entwurfsentscheidungen der Web-APIs basierten auf den *ressourcenorientierten RESTful* Paradigmen und der *ereignisgetriebenen Architektur*. Für den Entwurf der ressourcenorientierten RESTful APIs wurde der Ansatz von Giessler et al. [Gi18] verwendet. Bei dem Entwurf der Ereignisse griff das Migrationsteam auf die Domänenobjekte und Domänenereignisse aus dem Event Storming zurück. Die Dokumentation der REST-API findet sich aus Gründen der Übersichtlichkeit im Anhang A.3 dieser Forschungsarbeit wieder.

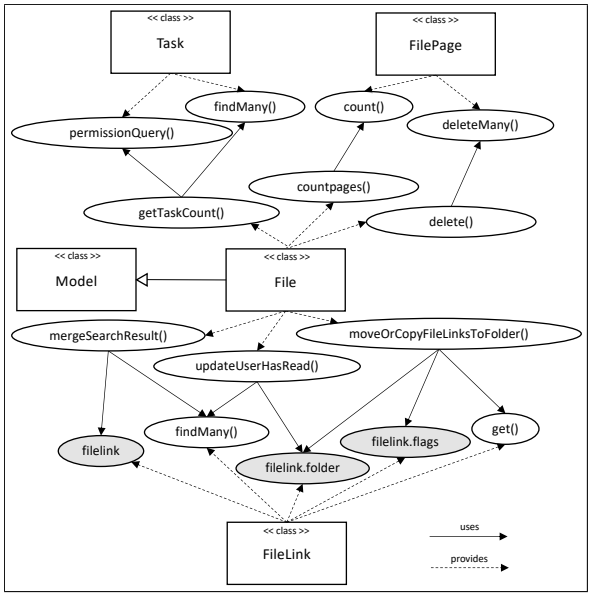

Abbildung 8.21: Abhängigkeiten der "*File*"-Klasse durch lokale Schnittstellen zu anderen Softwarebausteinen

8.6.3 Bedrohungen und Einschränkungen der Validität

Auch für die Typ II-Validierung existieren Bedrohungen und Einschränkungen der Validität:

Ergebnisvalidität Die Studie wurde im Rahmen einer Industriekooperation mit einem eingeschränkten Zeitraum und eingeschränkter Teilnehmerzahl durchgeführt. Vor und nach der Studie wurden von den fünf teilnehmenden Softwareentwicklern statistische Daten durch einen Bewertungsbogen erhoben. Kritisch zu betrachten ist die geringe Anzahl der Teilnehmer, was jedoch dem realen Kontext des Industriepartners geschuldet ist. Das Durchführen mehrerer Migrationsiterationen und das Blockieren weiterer Ressourcen bedeutete einen zu hohen Aufwand. Die Erfahrungsstufen und Fähigkeiten der Teilnehmer war breit aufgestellt, wodurch ebenfalls eine hohe Realitätsnähe zu iterativen Migrationen in realen Projektkontexten hergestellt werden kann. Es muss festgehalten werden, dass die heterogene Verteilung der Erfahrungsstufen und Fähigkeiten zwar das Ergebnis verfälschen kann, die externe Validität dagegen jedoch steigt [WR+12]. Ein weiteres Risiko ist die Ausgestaltung des Fragebogens, der zwar durch einen Testprobanden geprüft wurde, aber dennoch durch fehlinterpretierte Fragestellungen ein falsches Ergebnis verursachen kann. Die Ergebnisse weisen eine hohe Realitätsnähe auf, da die Extraktionsphase auf das Softwaresystem Evana360 angewendet wurde. Zusammenfassend kann für die Ergebnisvalidität festgehalten werden, dass zwar durch die geringe Anzahl der Teilnehmer die Aussagekraft der Daten fehlt, aber die Ergebnisse in einem realen Kontext generiert wurden.

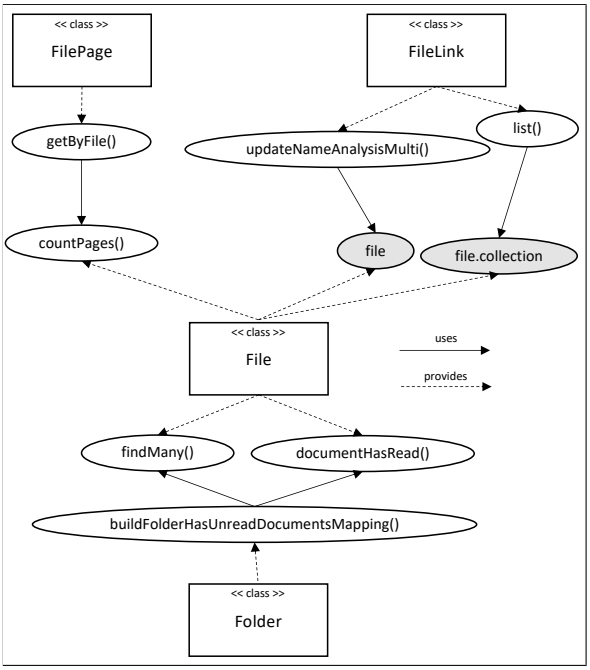

Abbildung 8.22: Abhängigkeiten mittelbarer Softwarebausteine zu der "*File*"-Klasse

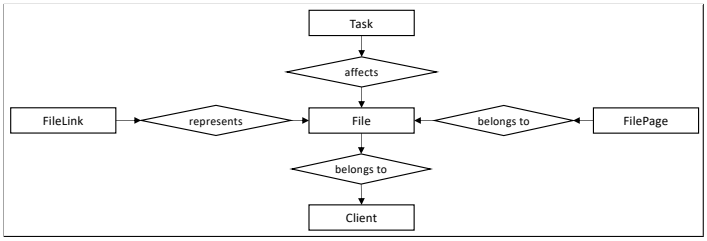

Abbildung 8.23: ER-Diagramm des Monolithenausschnitts

Interne Validität Bei der Betrachtung der erhobenen Daten müssen die bereits angesprochenen unterschiedlichen Berufserfahrungen und Kenntnisse der Studienteilnehmer über die im Migrationsrahmenwerk verwendeten Softwareentwicklungsansätze miteinbezogen werden. Zudem kann der industrielle Kontext auf die Konzentration und Motivation bei der Durchführung der Migrationsiteration Einfluss nehmen. Neben Unterbrechungen, die, folgt man den Daten, stattgefunden haben, können die Teilnehmer an verschiedenen Studientagen Frustration oder Desinteresse verspürt haben, was die Ergebnisse sowohl des Bewertungsbogens als auch die der Extraktionsphase beeinflusst. Doch dies ist ein unumgänglicher Faktor, der in Industriestudien hingenommen werden muss. Weiterhin spielt Voreingenommenheit eine große Rolle bei der Konfrontation von Softwareentwicklern mit neuen Methodiken. Festzustellen war dies beispielsweise bei der ubiquitären Sprache, die gerade von Interessenvertretern als überflüssig verspürt wurde. Entsprechend kann die kausale Abhängigkeit zwischen Ursache und Wirkung falsch interpretiert werden. Die interne Validität der vorgestellten Studie ist aus den genannten

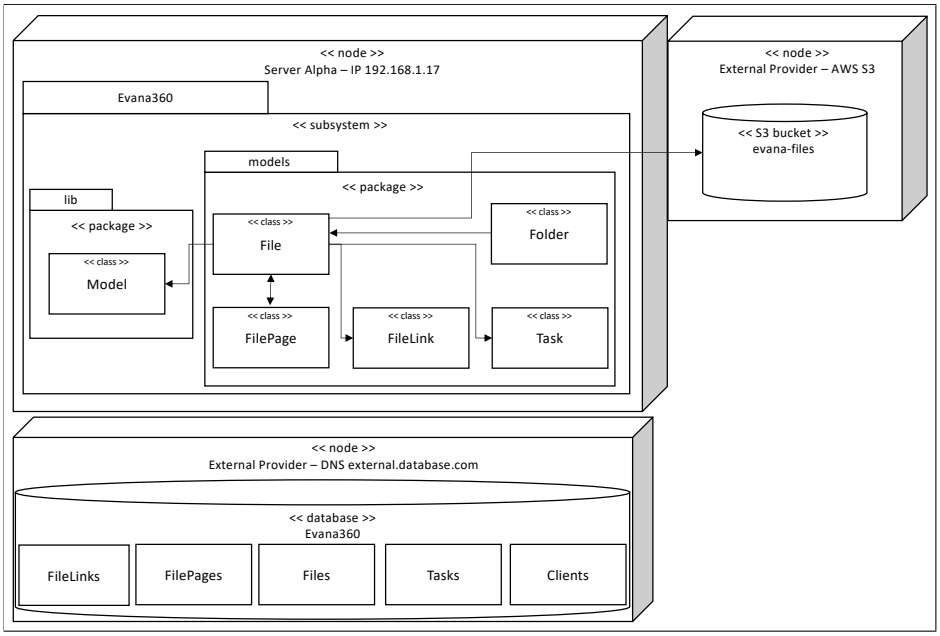

Abbildung 8.24: Verteilungsdiagramm der logischen und physischen Ist-Architektur

Gründen als moderat zu bewerten.

Konstruktionsvalidität Bei der Durchführung der Migrationsiteration wurde streng auf die Einhaltung der Systematik des Migrationsrahmenwerks geachtet. Abweichungen durch eigene Ideen der Teilnehmer wurde durch Kontrolle der Ergebnisse und gelegentliches Beiwohnen bei den Workshops unterbunden. Zwar ist die Softwareentwicklung und auch die Migration von Softwaresystemen ein kreativer Prozess, doch für die Korrektheit der Studiendaten ist eine Einhaltung der Vorgaben obligatorisch. Nicht zu verhindern war dagegen, dass Teilnehmer der Studie bereits verschiedene Erfahrungen im Bereich der Migrationen gesammelt hatten. Es war bei der Konstruktion der Studie zudem nicht möglich, auf eine größere Menge von Teilnehmern zurückzugreifen, da diese durch den Industriepartner zur Verfügung gestellt wurden. Auszuschließen ist eine absichtliche Verfälschung der Ergebnisse, da die Teilnehmer als Arbeitnehmer bei dem Industriepartner ein hohes Interesse am Erfolg der Migration haben, da hieran ein eigenes finanzielles Interesse geknüpft ist.

Externe Validität Durch den Industriekontext der Studie mit einem realen monolithischen Softwaresystem, Interessenvertretern und Rahmenbedingungen ist eine hohe externe Validität anzunehmen. Weiterhin begünstigt die Wahl des Migrationsauslösers anhand eines tatsächlichen Bedarfs die externe Validität. Lediglich die Durchführung einer einzelnen Migrationsiteration und die Betrachtung nur einer Geschäftsdomäne stellen eine Bedrohung dar.

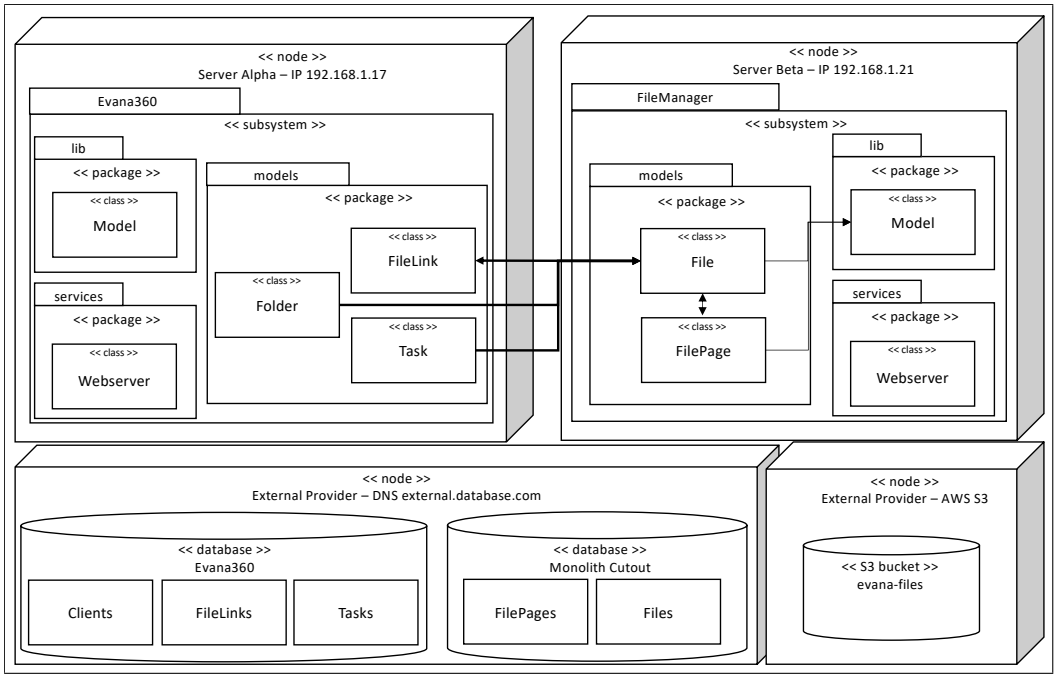

Abbildung 8.25: Logische und physische Soll-Architektur des Monolithenausschnitts

8.6.4 Zusammenfassung

Die Forschungsbeiträge dieser Arbeit (vgl. Abschnitt 1.5) wurden anhand einer industriellen Studie validiert. Die industrielle Studie wurde durch den Industriepartner Evana AG unterstützt. Zur Verfügung gestellt wurden ein reales monolithisches Softwaresystem, eine Geschäftsdomäne, ein Migrationsteam und ein Migrationsauslöser. Vor und nach der Studie wurden über einen Bewertungsbogen empirisch Daten erhoben und ausgewertet. Weiterhin durften Ergebnisse der Extraktionsphase, die während der Studie entstanden, vorgestellt werden. Die restlichen Migrationsphasen wurden von der Datenerhebung aufgrund einer unterzeichneten NDA explizit durch die Evana AG ausgeschlossen.

Es wurde geprüft, ob die Migrationsaktivitäten und Migrationsartefakte der Systematik ein Migrationsteam ganzheitlich bei der Durchführung einer Migrationsiteration unterstützen. Ausgehend von Bewertungsbögen wurden keine schwerwiegenden Aussagen bezüglich des entworfenen systematischen und nachvollziehbaren Vorgehens getroffen. Mehrfach angemerkt wurde die fehlende Unterstützung während der Implementierung des Monolithenausschnitts und der Microservices. Die Migrationsaktivitäten bezüglich der Implementierung wurden jedoch bewusst nicht im Rahmen dieser Forschungsarbeit betrachtet. Ansonsten wurden für die Extraktions- und Modernisierungsphase keine fehlenden Aspekte in der Systematik angemerkt. Lediglich bei der Inbetriebnahmephase wurde dies bemängelt, Aussagen über konkrete fehlende Aspekte blieben jedoch aus. Letztendlich kann durch das erfolgreiche Abschließen der Migrationsiteration festgehalten werden, dass die entworfenen

Migrationsaktivitäten und -artefakte sich für den Entwurf des Monolithenausschnitts und der domänengetriebenen Microservice-Architektur geeignet haben. Daraus folgt auch, dass die Systematik ganzheitlich das Migrationsteam unterstützt. Gerade im Hinblick auf die Vielzahl an Entwurfsentscheidungen, die das Migrationsteam treffen muss, ist die Entscheidungsunterstützung von großer Bedeutung. Durch die Einführung der formalen Migrationsartefakte hat sich gezeigt, dass nicht nur das Migrationsteam alleine Entscheidungen ableiten konnte, sondern auch Interessenvertreter in die Entscheidungsfindung miteinbezogen werden konnten. So konnte die Qualität der Entscheidungen noch einmal gesteigert werden. Das gesamte Migrationsrahmenwerk legt einen großen Fokus auf die Betrachtung der Geschäftsdomäne und legt einigen Entscheidungen das dort vorzufindende Domänenwissen zugrunde. Aus den Studiendaten geht hervor, dass die Geschäftsdomäne zufriedenstellend im Migrationsrahmenwerk berücksichtigt wurde, denn dem Entwurf der Microservice-Architektur wurde eine starke Orientierung an der Geschäftsdomäne zugesprochen. Weiterführend wurde bei dem Migrationsteam ein tieferes Verständnis über die Geschäftsdomäne etabliert.

8.7 Zusammenfassung der Validierung

Zur Validierung der Forschungsbeiträge dieser Arbeit wurden verschiedene empirische Studien durchgeführt. Die Studien richteten sich nach den von Durdik [Du16] vorgestellten Validierungsarten der Typ 0- (Machbarkeit), Typ I- (Eignung) und Typ II-Validierung (Anwendbarkeit). Bei der Durchführung der Validierungsarten wurde auf das Vorgehen von Giessler [Gi18] zurückgegriffen, das sich für die Anwendung auf Software Engineering-Prozesse eignet. Auf die Typ III-Validierung (Kosten/-Nutzen) wurde aufgrund des zeitlichen und finanziellen Aufwandes verzichtet. Die Ergebnisse der Studien wurden hinsichtlich von Bedrohungen und Einschränkungen der Validität bewertet.

Die Typ 0-Validierung sah einen Vergleich bestehender Arbeiten und der hier vorliegenden Forschungsarbeit vor. Der Vergleich wurde anhand des Anforderungskatalogs durchgeführt, der auch zur Bewertung der Arbeiten im Stand der Forschung verwendet wurde. Es stellte sich heraus, dass der im Stand der Forschung identifizierte Handlungsbedarf durch das Migrationsrahmenwerk geeignet aufgegriffen wird und Lücken schließt. Lediglich das Treffen von Vorbereitungen zur Bewältigung des komplexeren Betriebs der Microservices sollte noch intensiver betrachtet werden. Diese Validierung betrachtete alle vier Forschungsbeiträge gleichermaßen.

Für die Typ I-Validierung wurden drei studentische Projektteams aufgestellt. In Kooperation mit dem Industriepartner Evana AG führten zwei der Projektteams voneinander unabhängige Migrationsiterationen durch. Das dritte Projektteam schuf ein fiktives monolithisches Softwaresystem als Grundlage für die Migrationsiterationen. Es wurde die Entscheidung getroffen, das Migrationsrahmenwerk zunächst durch studentische Projektteams zu erproben, um bereits im Vorfeld einer Industriestudie

Schwachstellen finden und verbessern zu können. Bei der Durchführung der Migrationsiterationen bewerteten Domänenexperten, welche von der Evana AG zur Verfügung gestellt wurden, kontinuierlich die Ergebnisse der Projektteams. Die durchgeführten Studien haben gezeigt, dass das Migrationsrahmenwerk für den Einsatz in einer iterativen Migration geeignet ist und es tendenziell die Durchführung einer Migrationsiteration ganzheitlich unterstützt. Die Ergebnisse sind trotz festgestellter Bedrohungen und Einschränkungen als hoch relevant zu einzustufen, denn in Anbetracht der Eignung konnte die zuvor genannte Tendenz festgestellt werden. Auch die Typ I-Validierung betrachtete alle vier Forschungsbeiträge.

Zum Abschluss der Validierung wurde mittels einer Studie die Typ II-Validierung angestrebt. Im Kontext des Industriepartners Evana AG wurde eine unter realen Bedingungen durchgeführte iterative Migration gestartet. Der Industriepartner stellte dabei das monolithische Softwaresystem, das Migrationsteam, die Rahmenbedingungen und den Migrationsauslöser. Bei der Durchführung der Migrationsiteration wurde auf eine strenge Einhaltung des Migrationsrahmenwerks geachtet. Das Ergebnis beziehungsweise die statistischen Daten zur durchgeführten Migrationsiteration wurden über einen Bewertungsbogen erhoben und ausgewertet. Weiterhin durften die Ergebnisse der Extraktionsphase in dieser Arbeit dargestellt werden, um die Systematik und Nachvollziehbarkeit zu verdeutlichen. Es zeigte sich, dass das Migrationsrahmenwerk im Kontext einer realen Migrationsiteration das Migrationsteam ganzheitlich unterstützen konnte, auch wenn Schwachstellen bei der Implementierungsunterstützung festgestellt wurden. Die explizite Betrachtung der Implementierung ist jedoch im Rahmen dieser Forschungsarbeit nicht erfolgt und sollte aufgrund der aus der Studie hervorgehenden Ergebnisse perspektivisch erfolgen. Auch gilt es einen höheren Fokus auf die Inbetriebnahmephase zu legen, um fehlende Aspekte im Migrationsrahmenwerk abzudecken. Zusammenfassend kann die Anwendbarkeit des Migrationsrahmenwerks in einem Industriekontext positiv bewertet werden. Es konnten zwar auch Bedrohungen und Einschränkungen der Validität festgestellt werden, doch gerade die externe Validität ist in dieser Studie als besonders hoch zu bewerten. Die präsentierten Ergebnisse der Typ II-Validierung beziehen sich unterschiedlich stark auf die Forschungsbeiträge. Während der Bewertungsbogen das gesamte Migrationsvorhaben abdeckt und damit Daten über alle Forschungsbeiträge erhoben hat, konnte mit den vorgestellten Migrationsartefakten nur die Extraktionsphase und damit der Forschungsbeitrag (B2) “Umfang einer Migrationsiteration bestimmen” (vgl. Abschnitt 1.5.3) validiert werden.

9 Fazit und Ausblick

In diesem abschließenden Kapitel werden im Rahmen eines Fazits die geleisteten Forschungsbeiträge und deren Validierung ergebnisorientiert zusammengefasst. Zuletzt wird ein Ausblick gegeben, der weitere Fragestellungen, die im Kontext der geleisteten Forschungsbeiträge aufgetreten sind, aufzeigt, um damit eine Fortführung der Forschung am Migrationsrahmenwerk zu motivieren.

9.1 Fazit

Mit der Digitalisierung sind Softwaresysteme bei der Ausführung geschäftlicher Prozesse weiter in das Blickfeld von Organisationen gerutscht. Unabhängig davon, ob Organisationen diese Softwaresysteme einkaufen oder selbst (für den Verkauf) entwickeln, werden durch die zentrale Rolle der Softwaresysteme in der Digitalisierung spezielle Anforderungen an diese Softwaresysteme gestellt. Insbesondere fordert die Digitalisierung eine hohe Flexibilität von den Softwaresystemen, um auf veränderte Gegebenheiten einfach und schnell reagieren zu können. Die Bewältigung von Lasten stellt eine weitere Anforderung dar, denn Softwaresysteme werden immer mehr durch Softwarehäuser als Cloud-Lösungen zur Verfügung gestellt, um globaler agieren zu können und die Nachteile der On-Premise-Installationen bei Organisationen aufzulösen. In den letzten Jahren hat sich im Rahmen der Softwareentwicklung ein Architekturstil herauskristallisiert, der den Anforderungen der Digitalisierung gerecht werden soll. Dabei handelt es sich um die Microservice-Architektur, die durch ihre vielversprechenden Eigenschaften nun in der Softwareentwicklung der Organisationen adaptiert werden soll. Um dies zu erreichen, müssen bestehende Softwaresysteme, die überwiegend über eine monolithische Architektur verfügen, in die Microservice-Architektur migriert werden. Es existieren verschiedene Vorgehensweisen zur Migration eines Softwaresystems. In dieser Forschungsarbeit wurde eine iterative Migration betrachtet mit dem Ziel, ein auf dem iterativen Vorgehen basierendes Migrationsvorhaben zu unterstützen. Zur Gewährleistung dieser Unterstützung wurden bestimmte Forschungsbeiträge angestrebt, die zudem anhand verschiedener Studien validiert wurden. Aus den Ergebnissen dieser Forschungsbeiträge, die in den folgenden Abschnitten nochmal kurz zusammenfassend dargestellt werden, können verschiedene Schlussfolgerungen gezogen werden, die ebenfalls unter den folgenden Abschnitten erläutert werden.

9.1.1 Forschungsbeiträge

Nachfolgend werden die Forschungsbeiträge B1 bis B4, die detailliert den Kapiteln 4 bis 7 entnommen werden können, kurz zusammengefasst. Die Forschungsbeiträge wurden aus dem Handlungsbedarf (vgl. Abschnitt 3.3) abgeleitet, um die identifizierten Lücken bestehender Arbeiten zu schließen.

B1 - Entwurf des Migrationsrahmenwerks

Der erste Beitrag dieser Forschungsarbeit befasst sich mit dem allgemeinen Entwurf eines Migrationsrahmenwerks zur Gewährleistung einer iterativen Migration monolithischer Softwaresysteme in eine Microservice-Architektur. Für den Entwurf wurde zu Beginn das Ziel gesetzt, dass dieser eine Generalisierbarkeit für verschiedene Organisationen mit monolithischen Softwaresystemen aufweisen soll und damit den unterschiedlichen Praktiken in der Softwareentwicklung gerecht wird. Aus diesem Grund wurde bei dem Entwurf des Migrationsrahmenwerks auf die Verwendung konkreter Softwareentwicklungsansätze verzichtet. Durch diesen Verzicht wurden die Migrationsaktivitäten der Systematik innerhalb des Migrationsrahmenwerks so abstrakt gefasst, dass die Organisationen beim Einsatz in der Lage sind, die abstrakten Migrationsaktivitäten durch konkrete Softwareentwicklungsansätze zu verfeinern. Um den Entwurf des Migrationsrahmenwerks möglichst verständlich zu gestalten, wurden *allgemeine Definitionen* von Begrifflichkeiten im Kontext einer iterativen Migration aufgestellt. Hierfür wurden bestehende Arbeiten gesichtet und konsolidiert. Aus den Definitionen gingen zudem die zentralen Bestandteile des Migrationsrahmenwerks hervor: (1) *Systematik*, (2) *Migrationsphase*, (3) *Migrationsaktivität*, (4) *Migrationsartefakt*, (5) *Migrationsteam*, (6) *Migrationsingenieur* und (7) *Interessenvertreter*. Das Vorgehen bei der Migration wird durch die Systematik des Migrationsrahmenwerks vorgegeben. Diese Systematik besteht aus Migrationsphasen, Migrationsaktivitäten und Migrationsartefakten. Entworfen wurden die drei Migrationsphasen (1) *Extraktion*, *Modernisierung* und *Inbetriebnahme*. Die Extraktionsphase umfasst die Identifikation des Inkrements in Form eines Monolithenausschnitts. Die Modernisierungsphase überführt den Monolithenausschnitt in domänengetriebene Microservices. Zuletzt bereitet die Inbetriebnahmephase die entwickelten Microservices auf den Betrieb vor.

Das entworfene Migrationsrahmenwerk ist ein erster wichtiger Schritt eine systematische und nachvollziehbare iterative Migration monolithischer Softwaresysteme in eine Microservice-Architektur zu gewährleisten und knüpft damit an Vorarbeiten zu Migrationsrahmen wie [AF+18] an. Durch das Einführen von einheitlichen Definitionen steigt das Verständnis über das entworfene Migrationsrahmenwerk an. Hinzu kommt, dass auch zukünftige Arbeiten an diesen Definitionen anknüpfen können, um weitere Beiträge im Kontext des Migrationsrahmenwerks zu leisten. Weiterhin kann festgehalten werden, dass die im Migrationsrahmenwerk vorgesehenen Migrationsaktivitäten und

-artefakte die Analyse- und Entwurfsphasen während der Durchführung einer Migrationsiteration ganzheitlich abdecken.

B2 - Bestimmen des Umfangs eines Inkrements

Im zweiten Beitrag dieser Forschungsarbeit wird die *Extraktionsphase* des Migrationsrahmenwerks im Hinblick auf das Bestimmen des Umfangs des Inkrements einer Migrationsiteration detailliert betrachtet. Die abstrakten Migrationsaktivitäten werden anhand *konkreter Migrationsaktivitäten* und Softwareentwicklungsansätze konkretisiert. Eingesetzt werden die *verhaltensgetriebene Entwicklung* (engl. Behavior-Driven Development, BDD) [Sm14], der *domänengetriebene Entwurf* (engl. Domain-Driven Design, DDD) [Ev04] und das *Design Recovery* [CC90]. Aufgrund der Annahme, dass keine Architekturbeschreibung für das Softwaresystem vorhanden ist, muss über das Design Recovery diese wiederhergestellt werden. Die Extraktionsphase verfolgt dabei das Ziel, den Aufwand des Design Recoverys zu minimieren, indem nur die für die Migrationsiteration relevanten Softwarebausteine des Monolithen dokumentiert werden. Zunächst wird der Migrationsauslöser formalisiert, um diesen als Grundlage für die Identifikation kohäsiver Funktionalitäten nutzen zu können. Unterschieden wird dabei zwischen funktionalen und nicht-funktionalen Anforderungen als Auslöser. Eine funktionale Anforderung wird mittels BDD in ein *Gherkin Feature* formalisiert. Dagegen wird eine nicht-funktionale Anforderung in einen *Control Case* überführt. Eine weitere Quelle für die Identifikation der kohäsiven Funktionalität wird mit der *strategischen Modellierung*, einer DDD-Praktik, geschaffen. Denn nach der Formalisierung des Migrationsauslösers wird die grobgranulare Struktur der Geschäftsdomäne modelliert. Modelliert wird eine *Context Map* mit den dazugehörigen Subdomänen. Über ein vorgestelltes *Unified Modeling Language-Profil* (UML-Profil) wird die Context Map als formales Migrationsartefakt verstanden. Nach der Modellierung erfolgt die Identifikation der kohäsiven Funktionalitäten, die ebenfalls als Gherkin Features festgehalten werden. Ausgehend von den Gherkin Features wird die *Ist-Architekturbeschreibung* wiederhergestellt. In die Ist-Architekturbeschreibung werden von den Gherkin Features *unmittelbare* und *mittelbare Softwarebausteine* sämtlicher Schichten der *Schichtenarchitektur* [Ev04] aufgenommen, die letztendlich den Umfang des Inkrements und damit auch den Monolithenausschnitt darstellen. Abschließend wird die *Soll-Architekturbeschreibung* abgeleitet, die den Monolithenausschnitt als eigenständigen Softwarebaustein neben dem Monolithen beschreibt. Wichtige Teile der Soll-Architekturbeschreibung sind dabei die entworfenen Web-Schnittstellen, die das unabhängige Verteilen des Monolithenausschnitts auf die physische Architektur ermöglichen.

Die für die Extraktionsphase vorgesehenen konkreten Migrationsaktivitäten decken die Analyse und den Entwurf des Monolithenausschnitts innerhalb einer Migrationsiteration ganzheitlich ab. Die vorgestellten konkreten Migrationsartefakte unterstützen das Migrationsteam dabei, Entscheidungen nachvollziehbar zu treffen und festzuhalten. Zudem sind die Migrationsartefakte durch die formalen Modellierungssprachen konsistent in der Darstellung, was zu einer besseren Verständlichkeit bei dem

Migrationsteam führt. Bei der Identifikation der Funktionalität für den Monolithenausschnitt wird durch die Betrachtung der Geschäftsdomäne eine hohe Kohäsion forciert, sodass der Monolithenausschnitt sich im weiteren Verlauf der Migrationsiteration für die Modernisierung in Microservices besser eignet.

B3 - Domänengetriebener Entwurf der Microservice-Architektur

Platziert wurde der domänengetriebene Entwurf in der *Modernisierungsphase* des Migrationsrahmenwerks. Grundlegend wird die Orientierung an der Geschäftsdomäne durch den Einsatz von *DDD* [Ev04] erreicht. Zu Beginn der Modernisierungsphase werden jedoch zunächst die Anforderungen überarbeitet, sodass diese an veränderte Bedürfnisse der Organisation oder den Bedürfnissen der Kunden angepasst werden können. Denn während in der Extraktionsphase nur der Ist-Zustand der Anforderungen dokumentiert wurde, zielt die Modernisierungsphase auf die Verbesserung der Situation ab. Die Anforderungen liegen weiterhin nach der Überarbeitung in Form von *Gherkin Features* vor. Die Orientierung an der Geschäftsdomäne beginnt mit der Identifikation von Domänenobjekten in den Gherkin Features. Die Domänenobjekte werden in einer *Relationensicht*–ein für die Zwecke angepasstes Klassendiagramm–miteinander verbunden. Im Kontext von DDD wird die Modellierung der Domänenobjekte als *taktische Modellierung* bezeichnet. Aus der Relationensicht lassen sich die *Bounded Contexts* ableiten. Ein Bounded Context entspricht dabei einem Microservice [Ne15]. Die Unterteilung der Relationensicht in die Bounded Contexts erfolgt anhand der Kohäsion von Domänenobjekten. Die Bewertung der Kohäsion beruht auf der Betrachtung von den Gherkin Features und Domänenereignissen, die spezialisierte Domänenobjekte darstellen. Wichtig für den effizienten Betrieb des daraus resultierenden domänengetriebenen Entwurfs ist weiterführend die Betrachtung nicht-funktionaler Anforderungen. Der Entwurf wird somit nachträglich anhand nicht-funktionaler Anforderungen überarbeitet, sodass gegebenenfalls Microservices zu einem Microservice zusammengeführt oder ein Microservice in mehrere aufgeteilt wird. Zum Abschluss der Modernisierungsphase wird das Entwicklungsteam der Organisation auf den Architekturstil angepasst. Jedem Microservice wird ein autonomes *Service-Team* zugewiesen, welches aus Softwareentwicklern des ehemals monolithischen Entwicklungsteams besteht. Die Softwareentwickler werden einem Service-Team unter Abwägung von Interessenskonflikten und der möglichst heterogenen Verteilung von Fähigkeiten und Erfahrungsstufen der Softwareentwickler zugewiesen.

Die vorgestellten konkreten Migrationsaktivitäten der Modernisierungsphase bilden die Analyse und den Entwurf der Microservices ganzheitlich ab, sodass aus dem Monolithenausschnitt domänengetriebene Microservices abgeleitet werden können. Infolge der Anpassung der Anforderungsspezifikation zu Beginn der Modernisierungsphase entsprechen die Microservices auch den eigentlichen Bedürfnissen der Organisation und der Kunden. Das Berücksichtigen der Reorganisation des monolithischen Entwicklungsteam hilft der Organisation bei der Wartung und dem Betrieb der neuen Microservices.

B4 - Vorbereitungen zur Inbetriebnahme der Microservices

Im letzten Beitrag dieser Forschungsarbeit werden Vorbereitungen zur I*nbetriebnahme* der Microservices aufgezeigt. Dies ist, bedingt durch die höhere Komplexität des Betriebs von Microservices in Relation zu monolithischen Softwaresystemen, notwendig. Die Ergebnisse der Inbetriebnahmephase stellen *Konfigurationsdateien* für spezielle *Werkzeuge* dar. Die Werkzeuge sind motiviert durch die *Development & Operation-Prinzipien* (DevOps) [BW+15]. Zunächst wird eine *Containerisierung* der geschaffenen Microservices mittels Docker [Doc-Ref] vorgesehen. Durch die Containerisierung können die Microservices reproduzierbar auf verschiedenen Hardwarebausteinen bereitgestellt werden. Zudem stellt die Containerisierung die Grundlage für die Integration der Microservices in eine *Container-Orchestrierung* dar, welche einen hohen Stellenwert bei dem Betrieb der Microservices hat. In der Inbetriebnahmephase wird als Container-Orchestrierung das weit verbreitete Kubernetes [Lin-Kub] verwendet. Zum Schluss werden die Microservices in die *Continuous Integration/Continuous Delivery-Pipeline* (CI/CD-Pipeline) Jenkins [Jen-Jen] integriert, um eine automatisierte Bereitstellung neuer Entwicklungsversionen der Microservices zu gewährleisten.

Die entworfene Inbetriebnahmephase ist ein erster Schritt zur Unterstützung des Migrationsteams und der Service-Teams bei dem Betrieb der Microservices. Betrachtet werden grundlegende Konzepte, die im Rahmen der Microservice-Architekturen notwendig sind.

9.1.2 Validierung der Forschungsbeiräge

Die zuvor vorgestellten Forschungsbeiträge wurden im Rahmen dieser Arbeit in universitären und industriellen Kontexten validiert. Die Validierung erfolgte anhand dreier Validierungstypen: (1) *Typ 0-Validierung* (Machbarkeit), (2) *Typ I-Validierung* (Eignung) und *Typ II-Validierung* (Anwendbarkeit). Für jeden der Validierungstypen wurden geeignete Studien festgelegt. Die Ergebnisse jeder Studie wurden auf *Bedrohungen* und *Einschränkungen* der Validität geprüft. Die Typ 0-Validierung wurde durch den Autor dieser Arbeit durchgeführt, indem bestehende Arbeiten mit der hier vorliegenden Arbeit verglichen wurden. Der Vergleich wurde anhand eines zuvor festgelegten *Anforderungskatalogs* durchgeführt. Zur Demonstration der Eignung der Forschungsbeiträge B1 bis B4 wurde eine Studie im universitären Bereich durchgeführt. Im Rahmen von Praktika bildeten Studierende Projektteams, die beispielhaft eine iterative Migration mit Hilfe des Migrationsrahmenwerks durchführten. Begleitet wurde die Studie von einem industriellen Kooperationspartner, um ein realistisches Migrationsszenario beschreiben zu können. Insbesondere sollte diese Studie zeigen, ob die Forschungsbeiträge sich für eine Demonstration im industriellen Kontext eignen würden. Die Validierung im industriellen Kontext wurde anhand der Typ II-Validierung durchgeführt. Der industrielle Kooperationspartner führte anhand des entworfenen Migrationsrahmenwerks eine Migrationsiteration durch. Hierfür wurden die Ressourcen durch den Kooperationspartner zur Verfügung gestellt. Abschließend wurden die teilnehmenden

Softwareentwickler gebeten, mittels eines Bewertungsbogens das Migrationsrahmenwerk und den Verlauf der Migrationsiteration zu bewerten. Diese Daten ergaben, dass das Migrationsrahmenwerk für die Durchführung einer iterativen Migration geeignet ist. In der Studie konnten die Softwareentwickler einen kohäsiven Monolithenausschnitt erfolgreich extrahieren, in domänengetriebene Microservices modernisieren und automatisiert in Betrieb nehmen. Die Anwendbarkeit der getätigten Forschungsbeiträge konnte mit einer hohen *externen Validität* aufgrund der Daten als positiv bewertet werden.

9.2 Ausblick

Aus der Validierung der Forschungsbeiträge und aus denen im Vorfeld getätigten Einschränkungen ergeben sich bedeutende Fragestellungen, die sich zur Weiterführung dieser Forschungsarbeit anbieten.

9.2.1 Integration von Implementierungsaspekten

Bislang wurden Migrationsaktivitäten, welche die Implementierung betreffen, explizit aus dem Migrationsrahmenwerk ausgeschlossen, um Einschränkungen des Migrationsteams zu vermeiden. Aus der Validierung bezüglich Anwendbarkeit der Forschungsbeiträge ging als Ergebnis hervor, dass sich das durchführende Migrationsteam eine Unterstützung bei den Implementierungsaktivitäten gewünscht hätte. Dieser Wunsch betrifft sowohl die Extraktionsphase mit dem Monolithenausschnitt als auch die Modernisierungsphase mit den Microservices. Es gilt an dieser Stelle zu erarbeiten, wie das Migrationsteam unterstützt werden kann, ohne dabei die Freiheit maßgeblich einzuschränken. Das Migrationsteam sollte beispielsweise weiterhin frei über die verwendete Programmiersprache entscheiden können. Bei der Implementierung der Microservices bestünde die Möglichkeit das Migrationsteam durch eine systematische Abbildung der Mikrosoftwarebausteine auf die Schichten der Zwiebelarchitektur [Pa08] zu unterstützten. Dies wäre unabhängig von der Wahl der Programmiersprache, auch wenn das Migrationsteam dadurch zwangsläufig die Zwiebelarchitektur einsetzen muss.

9.2.2 Partielle Automatisierung von Migrationsaktivitäten

Die Migrationsaktivitäten dieser Forschungsarbeit müssen manuell von dem Migrationsteam durchlaufen werden. Folglich müssen auch die Migrationsartefakte manuell modelliert werden. Es wurde bewusst die Entscheidung getroffen, zunächst auf eine Automatisierung der Migrationsaktivitäten zu

verzichten. Gerade die Ansätze zur statischen Quellcodeanalyse beziehen sich auf den gesamten Monolithen und erstellen damit eine Architekturbeschreibung, welche den gesamten Ist-Zustand umfasst. Der Architekturbeschreibung fehlt damit der Fokus auf die für die Migrationsiteration relevanten Softwarebausteine. Zudem verliert die Architekturbeschreibung je nach Größe des Monolithen an Übersichtlichkeit. Doch bestehende Arbeiten wie [GK+16] oder [MC+17] zeigen den Vorteil automatisierter Verfahren auf. Besonders im Hinblick auf die finanziellen und zeitlichen Ressourcen ist eine Automatisierung empfehlenswert. Es bietet sich daher an, in zukünftigen Arbeiten eine partielle Automatisierung der Migrationsaktivitäten anzustreben. Es gilt, einen Ansatz für das Migrationsrahmenwerk zu finden oder zu entwerfen, der ausgehend von geeigneten Informationsquellen (Gherkin Features, Context Map) nur die relevanten unmittelbaren und mittelbaren Softwarebausteine findet.

9.2.3 Automatische Quellcodegenerierung aus Migrationsartefakten

Eine weitere Automatisierung zur Einsparung wichtiger Ressourcen kann auch durch eine automatisierte Quellcodegenerierung erreicht werden. Als Quelle können die formalen Migrationsartefakte herangezogen werden. Bestehende Ansätze wie die *modellgetriebene Entwicklung* [VS+13], die *Model-Driven Architecture* (MDA) [OMG20] und die *Architecture-Driven Modernization* (ADM) [ISO-19506] können hierfür im Migrationsrahmenwerk platziert werden. Hierfür müssen die formalen Migrationsartefakte den *Modelltransformationsprozess* der MDA durchlaufen. Der aktuelle Informationsgehalt und die Darstellung der Migrationsartefakte entspricht der Modellkategorie *Platform Independent Model* (PIM). Für die automatische Quellcodegenerierung müssen die Migrationsartefakte mit plattformspezifischen Informationen angereichert werden, damit die Migrationsartefakte den Anforderungen eines *Platform Specific Model* (PSM) gerecht werden. Aus diesen Modellen können dann die Quellcodeartefakte automatisiert generiert werden. Abzuwägen ist jedoch, ob der Nutzen der Quellcodegenerierung den zeitlichen Mehraufwand der Modelltransformation übersteigt.

9.2.4 Berücksichtigen Cloud-nativer Prinzipien im Entwurf

Die durch das Migrationsrahmenwerk abgeleitete Microservice-Architektur weist eine hohe Orientierung an der Geschäftsdomäne vor. Dieser Entwurf wird, zur Steigerung der Effizienz im Betrieb, anhand nicht-funktionaler Anforderungen an die Microservices überarbeitet. Doch unabhängig von den nicht-funktionalen Anforderungen, müssen die Microservices für einen effizienten Betrieb weitere wichtige Prinzipien erfüllen. Diese Prinzipien werden als *Cloud-nativen Prinzipien* [LA+18, Da19] bezeichnet. Von großer Bekanntheit sind beispielsweise die *Twelve Factors* [Wi17]. Einige dieser Cloud-nativen Prinzipien werden bereits in der Inbetriebnahmephase des Migrationsrahmenwerks berücksichtigt. Doch manche Prinzipien müssen bereits im Entwurf der Microservice-Architektur vorgesehen werden, damit diese den Cloud-Reifegrad des Softwaresystems positiv beeinflussen.

Folglich müssen die Cloud-nativen Prinzipien in der Modernisierungsphase vorgesehen werden. Möglich ist eine Erweiterung der Migrationsaktivität, welche den Entwurf anhand nicht-funktionaler Anforderungen überarbeitet.

10 Anhang

A Ergänzungen

A.1 Bewertungsbogen der Typ II-Validierung

Karlsruher Institut für Technologie

Bewertungsbogen – Iterative Migration monolithischer Softwaresysteme in eine domänengetriebene Microservice-Architektur

Vorwort zum Bewertungsbogen

Sehr geehrte(r) Teilnehmer/in,

im Rahmen meiner Forschungsarbeit zur Erlangung der Promotion am Karlsruher Institut für Technologie (Fakultät für Informatik, Forschungsgruppe Cooperation & Management, Prof. Abeck) beschäftige ich mich mit der Migration von monolithischen Softwaresystemen in eine domänengetriebene Microservice-Architektur. Das Ziel ist der Entwurf eines ganzheitlichen Migrationsrahmenwerkes, welches maßgeblich das Softwareentwicklungsteam bei der Durchführung einer iterativen Migration unterstützt. Ein solches Migrationsrahmenwerk stellt sequentiell ausgeführte Migrationsaktivitäten zur Verfügung, um das Softwareentwicklungsteam systematisch durch die Migration zu führen. Zusätzlich werden zur Gewährleistung der Nachvollziehbarkeit von getroffenen Entscheidungen, formale Migrationsartefakte definiert.

Zur industrienahen Anwendung des von mir erarbeiteten Migrationsrahmenwerks führe ich ein Experiment durch, dessen Daten zur Validierung genutzt werden. Hierfür hat Ihr Arbeitgeber **Evana AG** einer Industriekooperation zugestimmt und Sie als Teammitglied für das Experiment vorgesehen. Der Zeitraum des Experiments beträgt **zwei Kalendermonate** (nicht zu verwechseln mit Personenmonaten!). Auch die Beantwortung dieses Bewertungsbogens ist als Teil des Experiments vorgesehen und wird durch Ihren Arbeitgeber gestützt.

Die Erhebung der Daten erfolgt **anonym** und wird in zwei Phasen des Experiments stattfinden. Vor Beginn der iterativen Migration werden allgemeine Informationen über die Teammitglieder erhoben. Dies stellt die erste Phase da. Nach Abschluss der iterativen Migration erfolgt eine Bewertung des Migrationsrahmenwerks. Dies ist die zweite Phase. Den gesamten Bewertungsbogen erhalten Sie bereits vor Abschluss der iterativen Migration. Damit Sie bei der Ausfüllung des Bewertungsbogens möglichst präzise antworten können, empfehle ich Ihnen, vor Beginn der iterativen Migration die Fragestellungen der zweiten Phase einmalig zu betrachten. Insgesamt beträgt die Dauer der Bewertung circa 20 Minuten, wobei die erste Phase voraussichtlich 5 Minuten einnehmen wird.

Ich bedanke mich bereits im Voraus bei Ihnen für die Teilnahme an dem Experiment und die gewissenhafte Bearbeitung des Bewertungsbogens!

Mit freundlichen Grüßen
Benjamin Hippchen

Erste Phase – Allgemeine Informationen

Welche Rolle haben Sie aktuell inne?

- ☐ Junior Softwareentwickler
- ☐ Senior Softwareentwickler
- ☐ Leitender Softwareentwickler
- ☐ Softwarearchitekt
- ☐ sonstige:

Bitte beschreiben Sie kurz (2 - 3 Sätze) Ihre Aufgabengebiete in dieser Rolle:

Wie viele Jahre arbeiten Sie bereits in der Softwareentwicklung?

- ☐ weniger als 1 Jahr
- ☐ seit 1 – 3 Jahren
- ☐ seit 3 – 5 Jahren
- ☐ seit 5 – 8 Jahren
- ☐ seit 8 – 10 Jahren
- ☐ seit über 10 Jahren

In welchen der "klassischen" Softwareentwicklungsbereichen waren Sie bereits tätig?

- ☐ Frontend-Entwicklung
- ☐ Backend-Entwicklung
- ☐ Softwarearchitektur
- ☐ Operations
- ☐ Infrastrukturen
- ☐ IT-Security

Wie sehr sind Sie mit dem Softwareentwicklungsansatz des domänengetriebenen Entwurfs (engl. domain-driven design, DDD) vertraut?

- ☐ nicht vertraut
- ☐ wenig vertraut
- ☐ bedingt vertraut
- ☐ vertraut
- ☐ sehr vertraut

Wie sehr sind Sie mit dem Softwareentwicklungsansatz der verhaltensgetriebenen Entwicklung (engl. behavior-driven development, BDD) vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Wie sehr sind Sie mit dem Architekturstil der Microservices vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Wie sehr sind Sie mit der Thematik *Migration von Softwaresystemen* vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Waren Sie bereits an einer Migration beteiligt?

☐ Ja ☐ Nein

Wenn ja, bitte beschreiben Sie kurz das Projekt und die darin gelebte Vorgehensweise:

Wie sehr sind Sie mit Modellierungssprachen wie der Unified Modeling Language vertraut?

☐ nicht vertraut ☐ wenig vertraut ☐ bedingt vertraut
☐ vertraut ☐ sehr vertraut

Zweite Phase – Bewertung der durchgeführten Migrationsiteration

Allgemeine Fragen zum Migrationsablauf
Nachfolgend werden Ihnen Fragen zum Ablauf der Migrationsiteration gestellt.

Wie viele Personen waren an der Migration beteiligt (inklusive Stakeholder)?

☐ 5 – 7 Personen ☐ 7 – 10 Personen
☐ 10 – 13 Personen ☐ 13 – 15 Personen
☐ über 15 Personen

Wie oft wurden Sie bei der Durchführung der Migrationsiteration durch andere Themen unterbrochen?

Anzahl an Unterbrechungen:

Konnten Sie den Zeitrahmen für die Migrationsiteration von 2 Monaten einhalten?

☐ Ja ☐ Nein

Falls nicht, bitte nennen Sie den Grund hierfür:

Fragen zur Extraktionsphase
Die nachfolgenden Fragestellungen widmen sich der Extraktionsphase des Migrationsrahmenwerks und thematisieren den Verlauf der Spezifikation des Migrationsauslösers bis hin zum Entwurf des Monolithenausschnitts.

Konnte mithilfe der Migrationsaktivitäten der Extraktionsphase ein Monolithenausschnitt entworfen werden?

☐ Ja ☐ Nein

Falls nicht, bitte begründen Sie:

Waren durch die vorgegebenen Migrationsaktivitäten alle Aspekte des Monolithen abgebildet?

☐ Ja ☐ Nein

Falls nicht, bitte zeigen Sie die fehlenden Aspekte auf:

Welcher Typ Migrationsauslöser wurde als Start der Migrationsiteration betrachtet?

☐ Funktionale Anforderung ☐ Nicht-funktionale Anforderung

Wie viele Softwarebausteine der Mikroarchitektur (Klassen) des Monolithen waren dem Monolithenausschnitt zugeordnet?

Anzahl der Softwarebausteine:

Wie bewerten Sie den Umfang des Monolithenausschnitts?

zu umfangreich	passend	zu klein
☐	☐	☐

Wurde auf eine strenge Einhaltung der ubiquitären Sprache geachtet?

☐ Ja ☐ Nein

Falls nicht, bitte begründen Sie:

Wie bewerten Sie das systematische Vorgehen der Extraktionsphase und die darin auftretenden Migrationsartefakte?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Verständlichkeit	☐	☐	☐	☐	☐
Nachvollziehbarkeit	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐
Zeitlicher Aufwand	☐	☐	☐	☐	☐
Korrektheit der Ergebnisse	☐	☐	☐	☐	☐

Welche Verbesserungsvorschläge sehen Sie für das systematische Vorgehen der Extraktionsphase und für dessen Migrationsartefakte?

Durchführung der Modernisierungsphase
Nachfolgend werden Fragen zur Modernisierungsphase und zu deren Ablauf formuliert.

Konnte mithilfe der Migrationsaktivitäten der Modernisierungsphase eine domänengetriebene Microservice-Architektur entworfen werden?

☐ Ja ☐ Nein

Falls nicht, bitte begründen Sie:

Wurden Anforderungen während der Modernisierung überarbeitet?

☐ Ja ☐ Nein

Falls ja, bitte bewerten Sie wie hilfreich die Spezifikation der Geschäftsziele bei der Überarbeitung der Anforderungsspezifikation waren:

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

Wie schätzen Sie ihr Verständnis bezüglich der Geschäftsdomäne nach der Durchführung der Modernisierungsphase ein?

sehr niedrig	niedrig	moderat	hoch	sehr hoch
☐	☐	☐	☐	☐

Wie beurteilen Sie den abgeleiteten Entwurf der Microservice-Architektur?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Orientierung an der Geschäftsdomäne	☐	☐	☐	☐	☐
Effizienz im Betrieb der Microservices	☐	☐	☐	☐	☐
Wiederverwendbarkeit	☐	☐	☐	☐	☐
Effizienz der Web-Schnittstellen	☐	☐	☐	☐	☐
Umfang der Dokumentation	☐	☐	☐	☐	☐

Wie bewerten Sie das systematische Vorgehen der Modernisierungsphase und die darin auftretenden Migrationsartefakte?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Verständlichkeit	☐	☐	☐	☐	☐
Nachvollziehbarkeit	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐
Zeitlicher Aufwand	☐	☐	☐	☐	☐
Korrektheit der Ergebnisse	☐	☐	☐	☐	☐

Welche Verbesserungsvorschläge sehen Sie für das systematische Vorgehen der Modernisierungsphase und für dessen Migrationsartefakte?

Durchführung der Inbetriebnahmephase
Nachfolgend werden Fragen zur Inbetriebnahme und zu deren Ablauf formuliert.

Wie bewerten Sie das systematische Vorgehen der Modernisierungsphase und die darin auftretenden Migrationsartefakte?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Verständlichkeit	☐	☐	☐	☐	☐
Nachvollziehbarkeit	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐
Zeitlicher Aufwand	☐	☐	☐	☐	☐
Korrektheit der Ergebnisse	☐	☐	☐	☐	☐

Welche Verbesserungsvorschläge sehen Sie für das systematische Vorgehen der Modernisierungsphase und für dessen Migrationsartefakte?

Bewertung des Migrationsrahmenwerks
Zum Abschluss des Bewertungsbogen wird noch einmal allgemein auf das Migrationsrahmenwerk eingegangen.

Wie bewerten sie das Migrationsrahmenwerk in seiner Gesamtheit?

	sehr niedrig	niedrig	moderat	hoch	sehr hoch
Systematik	☐	☐	☐	☐	☐
Nachvollziehbarkeit	☐	☐	☐	☐	☐
Verständlichkeit	☐	☐	☐	☐	☐
Vollständigkeit	☐	☐	☐	☐	☐
Technologie-Agnostizität	☐	☐	☐	☐	☐
Entscheidungsunterstützung	☐	☐	☐	☐	☐
Praktische Anwendbarkeit	☐	☐	☐	☐	☐

Wie hilfreich war das Migrationsrahmenwerk bei der Durchführung einer Migrationsiteration?

nicht hilfreich	wenig hilfreich	moderat	hilfreich	sehr hilfreich
☐	☐	☐	☐	☐

A.2 Gherkin Features der Extraktionsphase

Im Rahmen der Typ II-Validierung in Kapitel 8 wurden verschiedene Gherkin Features erstellt. Der Quellcode 10.1 zeigt den Migrationsauslöser, der in der Typ II-Validierung an das Migrationsteam herangetragen wurde. Die *Versionierung von Dateien* wird anhand verschiedener Szenarien genauer beschrieben.

```
1  Feature: Versioning of Files
2                As an asset manager
3                I would like to create a new version of my file
4                And restore them as needed
5                So that I can display a history of the file in Evana360
6                And change them in an audit-proof way
7
8          Scenario: Open a dialog to replace the file
9              Given I am on the page for uploading files
10              When I mark a single file
11              Then it opens a context menu where I can upload a new
                    version of the file
12
13         Scenario: Replace a file
14             Given I am in the context menu of a single file
15              When I upload a new version of the file
16              Then the existing file is replaced by the new file
17               And assign a version to the new file
18               And flag the new file as active
19               And flag the old file as inactive
20
21         Scenario: View the versions of a file
22             Given I am in the context menu of a single file
23              When I view all versions of the file
24              Then it opens a dialog with a history of the versions of
                    this file
25
26         Scenario: Restore old version
27             Given I am in the dialog of all versions of a single file
28              When I restore a specific version of this file
29              Then the current file is replaced by the file of the
                    version
30               And the former current file is flagged as inactive
```

Quelltext 10.1: Gherkin Feature zur Versionierung und Historisierung der Dateien im Dokumentenmanagementsystem

Neben dem Migrationsauslöser wurden von dem Migrationsteam die bestehenden Funktionalitäten des

zentralen Domänenobjekts “*File*” als Gherkin Features erhoben. Diese werden in den nachfolgenden Quellcodeausschnitten detailliert dargestellt.

```
1  Feature: Uploading Files
2                As an asset manager
3                I would like to upload new files
4                And assign it to my building
5                So that I can work with these files within Evana360
6
7          Background:
8              Given I am in the context of a specific fund or building
9                And I am on the page for managing documents
10               And I am logged in as an authorized user
11
12         Scenario: Open an upload dialog
13             Given I am on the page for uploading files
14              When I click on the button for uploading new files
15              Then a upload dialog opens
16               And I am able to select my files from my local disk
17
18         Scenario: Uploading via drag'n'drop
19             Given I am on the page for uploading files
20              When I drag my files from my local disk on a specific spot
                    of Evana360
21              Then a confirmations dialog opens
22               And I am able to decline or confirm the upload
```

Quelltext 10.2: Gherkin Feature zum Hochladen einer oder mehrerer Dateien

```
1  Feature: Downloading Files
2                As an asset manager
3                I would like to download files from my building
4                So that I can work with these files on my local disk
5
6          Background:
7              Given I am in the context of a specific fund or building
8                And I am on the page for managing documents
9                And I am logged in as an authorized user
10
11         Scenario: Download files
12             Given I am on the page for uploading files
13               And I selected one or more files
14              When I click on the button for downloading files
15              Then these files are downloaded to the local disk
```

Quelltext 10.3: Gherkin Feature zum Herunterladen einer oder mehrerer Dateien

```
Feature: Deleting Files
               As an asset manager
               I would like to delete existing files
               So that I can clean up my buildings on Evana360

        Background:
            Given I am in the context of a specific fund or building
              And I am on the page for managing documents
              And I am logged in as an authorized user

        Scenario: Delete a single file
            Given I select a single file
             When I click on the button for deleting this file
             Then a confirmation dialog opens
              And I am able to decline or confirm the deletion

        Scenario: Delete multiple files
            Given I select multiple files in Evana360
             When I click on the button for deleting this file
             Then a confirmation dialog opens
              And I am able to decline or confirm the deletion
```

Quelltext 10.4: Gherkin Feature zum Löschen von Dateien

```
Feature: Open a File
               As an asset manager
               I would like to open a existing files
               And browse its content
               So that I can read the content of this file

        Background:
            Given I am in the context of a specific fund or building
              And I am logged in as an authorized user

        Scenario: Open a single file
            Given I on the page to see all files belonging to my one
                specific building
              And I select a single file
             When I click on the button for opening this file
             Then the file and each page of it is shown in a central
                spot in Evana360
              And I am able to scroll the file
```

Quelltext 10.5: Gherkin Feature zum Öffnen einer Datei

```
Feature: Editing a single File
               As an asset manager
               I would like to edit my file
               So that I can adapt the content of this file within
                   Evana360

        Background:
            Given I am in the context of a specific fund or building
              And I am on the page for managing documents
              And I am logged in as an authorized user

        Scenario: Edit an Microsoft Office related file
            Given I select a existing file of my building
              And it is a Microsoft Office document like Workd
             When I click on the button for editing this file
             Then a web-based version of Microsoft Office opens
              And the file is editable within this context
```

Quelltext 10.6: Gherkin Feature zum Editieren einer Datei

```
Feature: Cutting a File into multiple Files
               As an asset manager
               I would like to cut a single file into multiple files
               And save them directly on Evana360
               So that I can separate a file

        Background:
            Given I am in the context of a specific fund or building
              And I am on the page for managing documents
              And I am logged in as an authorized user

        Scenario: Open the cutting dialog
            Given I selected a single file
             When I click on the button for cutting this file
             Then a cutting dialog opens
              And all pages of this file are shown

        Scenario: Cut a open file
            Given I selected a file
              And opened the cutting dialog
             When I click on the button for placing a cut between two
                 pages
              And I confirm this cut
             Then the cutting dialog closes
              And the new files are saved accordingly to the configured
```

```
                    cuts
```

Quelltext 10.7: Gherkin Feature zum Schneiden einer Datei

```
1  Feature: Merging Files
2                As an asset manager
3                I would like to merge two or more files
4                And assign it to my building
5                So that I can combine multiple files to one file
6
7          Background:
8              Given I am in the context of a specific fund or building
9                And I am on the page for managing documents
10               And I am logged in as an authorized user
11
12         Scenario: Open the merging dialog
13             Given I am on the page for uploading files
14               And I selecting two or more files
15              When I click on the button for merging these files
16              Then a merge dialog opens
17
18         Scenario: Arrange files
19             Given I selected two or more files
20               And I opened the merging dialog
21              When I drag'n'drop a single file within this dialog
22              Then the position of this specific file is changed
                    accordingly
23
24         Scenario: Merging files
25             Given I selected two or more files
26               And I opened the merging dialog
27               And I arranged the files
28              When I click on the button to merge
29              Then the merge dialog closes
30               And a new file is saved in Evana360
```

Quelltext 10.8: Gherkin Feature zum Zusammenfassen mehrerer Dateien

A.3 REST-API-Spezifikation der Extraktionsphase

Für die Kommunikation zwischen dem Monolithenausschnitt und dem weiterhin bestehenden Monolithen wurde eine REST-API spezifiziert. Die Endpunkte können der Abbildung 10.1 entnommen werden. Bei dem Entwurf der API wurden die Effect Sketches, die Datenbankschemata und die Soll-Architekturbeschreibung zugrunde gelegt.

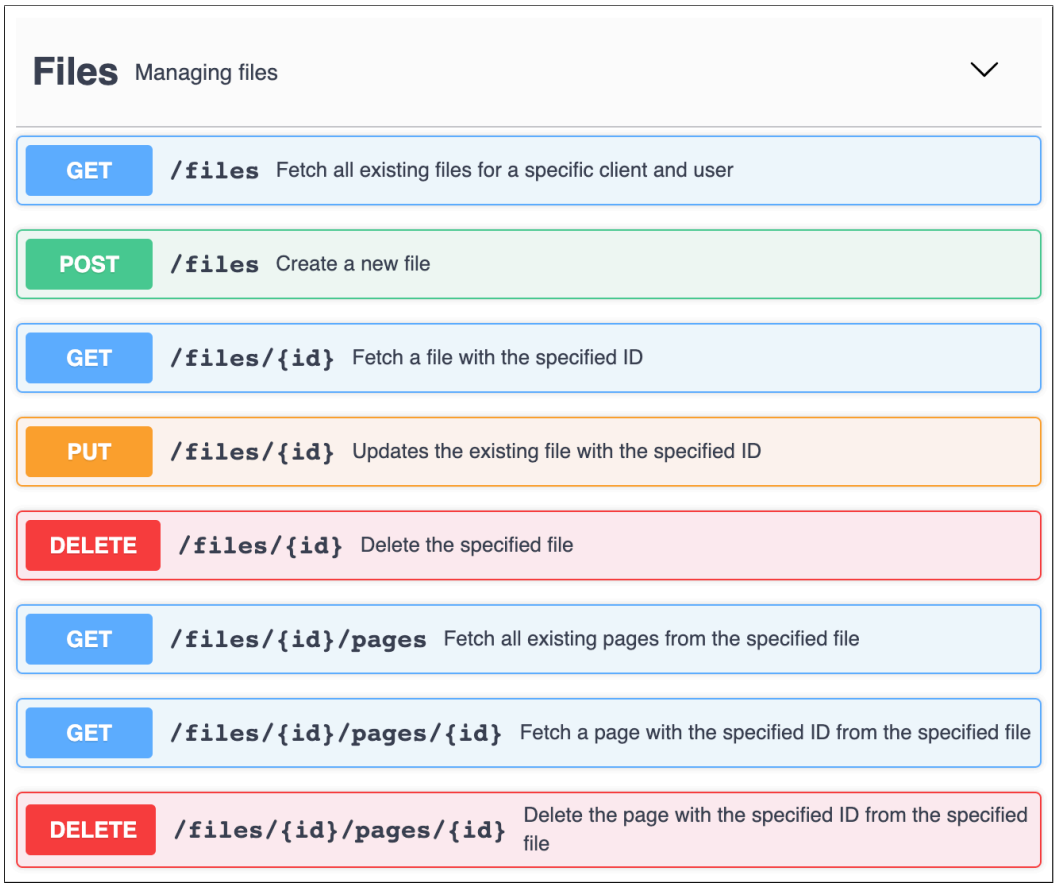

Abbildung 10.1: REST-API-Spezifikation des Monolithenausschnitts

A.4 Verwendete Icons von Dritten

Im Rahmen dieser Forschungsarbeit wurden verschiedene Icons von der Webseite *Flaticon* in den Abbildungen verwendet. Um den Lizenzbestimmungen dieser Icons gerecht zu werden, folgt nun in Tabelle 10.1 eine Auflistung der Icons inklusive Nennung der Lizenzinhaber.

Tabelle 10.1: Auflistung der Flaticon Icons und Autoren

	Eucalyp	https://www.flaticon.com/free-icon/connection_1401917
	geotatah	https://www.flaticon.com/free-icon/enterprise_993891
	DinosoftLabs	https://www.flaticon.com/free-icon/manager_641695
	Eucalyp	https://www.flaticon.com/free-icon/property_1924782
	dDara	https://www.flaticon.com/free-icon/assets_1907617
	Eucalyp	https://www.flaticon.com/free-icon/sublease_1464210
	Pause08	https://www.flaticon.com/free-icon/presentation_858174
	catkuro	https://www.flaticon.com/free-icon/report_3029319
	Freepik	https://www.flaticon.com/free-icon/personal-information_3076343
	Freepik	https://www.flaticon.com/free-icon/folders_2206426
	Freepik	https://www.flaticon.com/free-icon/computer_1197446
	surang	https://www.flaticon.com/free-icon/learn_2071993
	phatplus	https://www.flaticon.com/free-icon/database_901509
	Pixel perfect	https://www.flaticon.com/free-icon/folder_2572652

B Abkürzungsverzeichnis

ADL	Architecture Description Language
ADM	Architecture-Driven Modernization
API	Application Programming Interface
BDD	Behavior-Driven Development
CI	Continuous Integration
CD	Continuous Delivery
C&M	Cooperation & Management
DDD	Domain-Driven Design
Dev	Development
DevOps	Development & Operations
DMS	Dokumentenmanagementsystem
DNS	Domain Name Service
EARS	Easy Approach Requirement Syntax
ELP	Epidemic Label Propagation
ER	Entity Relationship
gRPC	gRemote Procedure Call
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
KDM	Knowledge Discovery Meta-Model
KI	Künstliche Intelligenz
LoC	Lines of Code
LP	Leistungspunkt
MBaaS	Mobile-Backend-as-a-Service
MDA	Model-Driven Architecture
MDD	Model-Driven Design

MOF	Meta Object Facility
MVP	Minimum Viable Product
NDA	Non-Disclosure Agreement
OMG	Object Management Group
OOP	Objektorientierte Programmierung
Ops	Operations
PIM	Platform Independent Model
PSM	Platform Specific Model
RAM	Random-Access Memory
REST	Representational State Transfer
RUP	Rationale Unified Process
SaaS	Software-as-a-Service
sDMS	simple Document Management System
SOA	Service-Oriented Architecture
SoSe	Sommersemester
SRP	Single Responsibility Principle
TCP	Transmission Control Protocol
TDD	Test-Driven Development
TF-IDF	Term-Frequency Inverse Document Frequency
UI	User Interface
UML	Unified Modeling Language
WiSe	Wintersemester
YAML	YAML Ain't Markup Language

C Abbildungsverzeichnis

D Tabellenverzeichnis

E Quelltextverzeichnis

F Literaturverzeichnis

[Sem-Sem] Semantic Versioning 2.0, URL https://semver.org/ [Zuletzt aufgerufen: 30.10.2020], 2020.

[AS+19] Sebastian Abeck, Michael Schneider, Jan-Philip Quirmbach, Heiko Klarl, Christof Urbaczek, Shkodran Zogaj: A Context Map as the Basis for a Microservice Architecture for the Connected Car Domain, INFORMATIK 2019: 50 Jahre Gesellschaft für Informatik–Informatik für Gesellschaft, 2019.

[AH+95] Serge Abiteboul, Richard Hull, Victor Vianu: Foundations of Databases, vol. 8, Addison-Wesley Reading, 1995.

[AG+05] Ellen Ackermann, R Gimnich, A Winter: Ein Referenz-Prozess der Software-Migration, Softwaretechnik-Trends, vol. 25, no. 4, pages 20–22, 2005.

[Ad11] Gojko Adzic: Specification by Example: How Successful Teams Deliver the Right Software, 2011.

[Al15] V Alagarasan: Seven Microservices Anti-Patterns, InfoQ, vol. 24, pages 35–50, 2015.

[ANSI-729] ANSI/IEEE: Standard Glossary of Software Engineering Terminology, 1991.

[IEEE-1471] Standards Association: 1471-2000 - IEEE Recommended Practice for Architectural Description for Software-Intensive Systems, URL [Zuletzt aufgerufen: 30.10.2020], 2000.

[IEEE-14764] Standards Association: 14764-2006 - ISO/IEC/IEEE International Standard for Software Engineering - Software Life Cycle Processes - Maintenance, URL https://standards.ieee.org/standard/14764-2006.html [Zuletzt aufgerufen: 30.10.2020], 2006.

[IEEE-42010] Standards Association: 42010-2011 - ISO/IEC/IEEE Systems and software engineering – Architecture description, URL https://standards.ieee.org/standard/42010-2011.html [Zuletzt aufgerufen: 30.10.2020], 2011.

[AF+18] Florian Auer, Michael Felderer, Valentina Lenarduzzi: Towards Defining a Microservice Migration Framework, in Proceedings of the 19th International Conference on Agile Software Development: Companion, pages 1–2, 2018.

[AW05] Aybüke Aurum, Claes Wohlin: Engineering and Managing Software Requirements, Springer Science & Business Media, 2005.

[BH+16] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi: Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture, Ieee Software, vol. 33, no. 3, pages 42–52, 2016.

[BH+18] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A Tamburri, Theo Lynn: Microservices Migration Patterns, Software: Practice and Experience, vol. 48, no. 11, pages 2019–2042, 2018.

[Ba17] Len Bass: The Software Architect and DevOps, IEEE Software, vol. 35, no. 1, pages 8–10, 2017.

[BC+03] Len Bass, Paul Clements, Rick Kazman: Software Architecture in Practice, Addison-Wesley Professional, 2003.

[BW+15] Len Bass, Ingo Weber, Liming Zhu: DevOps: A Software Architect's Perspective, Addison-Wesley Professional, 2015.

[BM97] Ira D Baxter, Michael Mehlich: Reverse Engineering is Reverse Forward Engineering, in Proceedings of the Fourth Working Conference on Reverse Engineering, IEEE, pages 104–113, 1997.

[BT76] Thomas E Bell, Thomas A Thayer: Software Requirements: Are They Really a Problem?, in Proceedings of the 2nd International Conference on Software Engineering, IEEE Computer Society Press, pages 61–68, 1976.

[Be20] Adam Bellemare: Building Event-Driven Microservices, O'Reilly Media, 2020.

[BR00] Keith H Bennett, Václav T Rajlich: Software Maintenance and Evolution: A Roadmap, in Proceedings of the Conference on the Future of Software Engineering, pages 73–87, 2000.

[BR99] KH Bennett, VT Rajlich: A new Perspective on Software Evolution: The Staged Model, submitted for publication to IEEE, 1999.

[BB+13] Alexander Bergmayr, Hugo Bruneliere, Javier Luis Canovas Izquierdo, Jesus Gorronogoitia, George Kousiouris, Dimosthenis Kyriazis, Philip Langer, Andreas Menychtas, Leire Orue-Echevarria, Clara Pezuela, et al.: Migrating Legacy Software to the Cloud with ARTIST, in 2013 17th European Conference on Software Maintenance and Reengineering, IEEE, pages 465–468, 2013.

[Be12] Saul J Berman: Digital Transformation: Opportunities to Create new Business Models, Strategy & Leadership, vol. 40, no. 2, pages 16–24, 2012.

[BM+94] Tim Berners-Lee, Larry Masinter, Mark McCahill, et al.: Uniform Resource Locators (URL), 1994.

[BC+03b] Alessandro Bianchi, Danilo Caivano, Vittorio Marengo, Giuseppe Visaggio: Iterative Reengineering of Legacy Systems, IEEE Transactions on Software Engineering, vol. 29, no. 3, pages 225–241, 2003.

[Bi89] Ted J Biggerstaff: Design Recovery for Maintenance and Reuse, Computer, vol. 22, no. 7, pages 36–49, 1989.

[BL+99] Jesús Bisbal, Deirdre Lawless, Bing Wu, Jane Grimson: Legacy Information System Migration: A Brief Review of Problems, Solutions and Research Issues, IEEE Software, vol. 16, no. 5, pages 103–111, 1999.

[Bj06] Dines Bjørner: Software Engineering 3: Domains, Requirements, and Software Design, Springer Science & Business Media, 2006.

[BR08] Rainer Böhme, Ralf Reussner: Validation of pPredictions With Measurements, in Dependability metrics, Springer, pages 14–18, 2008.

[Bo06] Grady Booch: Object-Oriented Analysis & Design with Application, Pearson Education India, 2006.

[BR+05] Grady Booch, James Rumbaugh, Ivar Jacobson: The Unified Modeling Language - User Guide, 2005.

[BB+16] Martin Böschen, Ralf Bogusch, Anabel Fraga, Christian Rudat: Bridging the Gap between Natural Language Requirements and Formal Specifications., in REFSQ Workshops, 2016.

[BF+14] Pierre Bourque, Richard E Fairley, et al.: Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0, IEEE Computer Society Press, 2014.

[BH98] Ivan T Bowman, Richard C Holt: Software Architecture Recovery Using Conway's Law, in Proceedings of the 1998 Conference of the Centre for Advanced Studies on Collaborative Research, IBM Press, page 6, 1998.

[Br13b] Alberto Brandolini: Introducing Event Storming, 2013.

[BS95] M Brodie, M Stonebraker: Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach, 1995.

[Br13] Simon Brown: Agile Software Architecture Sketches and NoUML, URL `https://www.infoq.com/articles/agile-software-architecture-sketches-NoUML/` [Zuletzt aufgerufen: 30.10.2020], 2013.

[Br20] Simon Brown: The C4 Model for Visualising Software Architecture - Context, Containers, Components and Code, URL `https://c4model.com/` [Zuletzt aufgerufen: 30.10.2020], 2020.

[BM+98] William H Brown, Raphael C Malveau, Hays W McCormick, Thomas J Mowbray: Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis, John Wiley & Sons, Inc., 1998.

[BP19] Morgan Bruce, Paulo A. Pereira: Microservices in Action, Manning Publications, 2019.

[BD09] Bernd Bruegge, Allen H Dutoit: Object-Oriented Software Engineering. Using UML, Patterns, and Java, Learning, vol. 5, no. 6, page 7, 2009.

[BD+18] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan T Larsen, Manuel Mazzara: From Monolithic to Microservices: An Experience Report from the Banking Domain, IEEE Software, vol. 35, no. 3, pages 50–55, 2018.

[CP+82] Bobby J Calder, Lynn W Phillips, Alice M Tybout: The concept of external validity, Journal of consumer research, vol. 9, no. 3, pages 240–244, 1982.

[CC79] Donald Thomas Campbell, Thomas D Cook: Quasi-Experimentation: Design & Analysis Issues for Field Settings, Rand McNally College Publishing Company Chicago, 1979.

[CP07] Gerardo CanforaHarman, Massimiliano Di Penta: New Frontiers of Reverse Engineering, in Future of Software Engineering (FOSE'07), IEEE, pages 326–341, 2007.

[Ch05] Robert N Charette: Why Software Fails [Software Failure], IEEE spectrum, vol. 42, no. 9, pages 42–49, 2005.

[CC90] Elliot J. Chikofsky, James H Cross: Reverse Engineering and Design Recovery: A Taxonomy, IEEE software, vol. 7, no. 1, pages 13–17, 1990.

[CB+10] Bee Bee Chua, Danilo Valeros Bernardo, June Verner: Understanding the Use of Elicitation Approaches for Effective Requirements Gathering, in 2010 Fifth International Conference on Software Engineering Advances, IEEE, pages 325–330, 2010.

[CV10] Bee Bee Chua, June Verner: Examining Requirements Change Rework Effort: A Study, arXiv preprint arXiv:1007.5126, 2010.

[CN+12] Lawrence Chung, Brian A Nixon, Eric Yu, John Mylopoulos: Non-Functional Requirements in Software Engineering, vol. 5, Springer Science & Business Media, 2012.

[CG+03] Paul Clements, David Garlan, Reed Little, Robert Nord, Judith Stafford: Documenting Software Architectures: Views and Beyond, in 25th International Conference on Software Engineering, 2003. Proceedings., IEEE, pages 740–741, 2003.

[CI+03] Paul Clements, James Ivers, Reed Little, Robert Nord, Judith Stafford: Documenting Software Architectures in an Agile World, Tech. rep., Carnegie-Mellon University Pittsburgh PA Software Engineering Institute, 2003.

[Co05] Alistair Cockburn: Hexagonal Architecture, URL `https://alistair.cockburn.us/hexagonal-architecture/` [Zuletzt aufgerufen: 30.10.2020], 2005.

[Co68] Melvin E Conway: How do Committees Invent, Datamation, vol. 14, no. 4, pages 28–31, 1968.

[CK+92] Bill Curtis, Marc I Kellner, Jim Over: Process Modeling, Communications of the ACM, vol. 35, no. 9, pages 75–90, 1992.

[Da19] Cornelia Davis: Cloud-Native Patterns, O'Reilly Media, 2019.

[DF+07] Andrea De Lucia, Fausto Fasano, Giuseppe Scanniello, Genny Tortora: Enhancing Collaborative Synchronous UML Modelling with Fine-Grained Versioning of Software Artefacts, Journal of Visual Languages & Computing, vol. 18, no. 5, pages 492–503, 2007.

[DF+16] Sandro De Santis, Luis Florez, Duy V Nguyen, Eduardo Rosa, et al.: Evolve the Monolith to Microservices with Java and Node, IBM Redbooks, 2016.

[SA+05] Sergio Cozzetti B de Souza, Nicolas Anquetil, Káthia M de Oliveira: A Study of the Documentation Essential to Software Maintenance, in Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information, pages 68–75, 2005.

[DL03] Tom DeMarco, Tim Lister: Risk Management During Requirements, IEEE Software, vol. 20, no. 5, pages 99–101, 2003.

[FL+18] Paolo Di Francesco, Patricia Lago, Ivano Malavolta: Migrating Towards Microservice Architectures: An Industrial Survey, in 2018 IEEE international conference on software architecture (ICSA), IEEE, pages 29–2909, 2018.

[DD+17] Nicola Dragoni, Schahram Dustdar, Stephan T Larsen, Manuel Mazzara: Microservices: Migration of a Mission Critical System, arXiv preprint arXiv:1704.04173, 2017.

[DG+17] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, Larisa Safina: Microservices: Yesterday, Today, and Tomorrow, in Present and Ulterior Software Engineering, Springer, pages 195–216, 2017.

[DL+17] Nicola Dragoni, Ivan Lanese, Stephan Thordal Larsen, Manuel Mazzara, Ruslan Mustafin, Larisa Safina: Microservices: How to make your Application Scale, in International Andrei Ershov Memorial Conference on Perspectives of System Informatics, Springer, pages 95–104, 2017.

[DP09] Stephane Ducasse, Damien Pollet: Software Architecture Reconstruction: A Process-Oriented Taxonomy, IEEE Transactions on Software Engineering, vol. 35, no. 4, pages 573–591, 2009.

[DR+00] Alastair Dunsmore, Marc Roper, Murray Wood: Object-Oriented Inspection in the Face of Delocalisation, in Proceedings of the 22nd international conference on Software engineering, pages 467–476, 2000.

[Du16] Zoya Durdik: Architectural Design Decision Documentation Through Reuse of Design Patterns, vol. 14, KIT Scientific Publishing, 2016.

[Eb97] Christof Ebert: Dealing with Non-Functional Requirements in Large Software Systems, Annals of Software Engineering, vol. 3, no. 1, pages 367–395, 1997.

[EC+16] Christian Esposito, Aniello Castiglione, Kim-Kwang Raymond Choo: Challenges in Delivering Software in the Cloud as Microservices, IEEE Cloud Computing, vol. 3, no. 5, pages 10–14, 2016.

[EF+03] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, Anne-Marie Kermarrec: The Many Faces of Publish/Subscribe, ACM Computing Surveys (CSUR), vol. 35, no. 2, pages 114–131, 2003.

[Ev04] Eric Evans: Domain-Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley Professional, 2004.

[Ev14] Eric Evans: Domain-Driven Design Reference: Definitions and Pattern Summaries, Dog Ear Publishing, 2014.

[Fa10] George Fairbanks: Just Enough Software Architecture: A Risk-Driven Approach, Marshall & Brainerd, 2010.

[Fa05] Jean-Marie Favre: Foundations of Model (Driven)(Reverse) Engineering: Models, in Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2005.

[Fe05] Michael Feathers: Working Effectively with Legacy Code, Prentice Hall Professional, 2005.

[Hel-Doc] Cloud Native Computing Foundation: Helm Documentation, URL https://helm.sh/docs/ [Zuletzt aufgerufen: 30.10.2020], 2020.

[Gra-Gra] The GraphQL Foundation: A query language for your API, URL https://graphql.org/ [Zuletzt aufgerufen: 30.10.2020], 2020.

[Lin-Kub] The Linux Foundation: Kubernetes, URL https://kubernetes.io/de/ [Zuletzt aufgerufen: 30.10.2020], 2020.

[Fo04] Martin Fowler: StranglerFigApplication, Tech. rep., URL https://martinfowler.com/bliki/StranglerFigApplication.html [Zuletzt aufgerufen: 30.10.2020], 2004.

[Fo15] Martin Fowler: MonolithFirst, Tech. rep., URL https://martinfowler.com/bliki/MonolithFirst.html [Zuletzt aufgerufen: 30.10.2020], 2015.

[FL14] Martin Fowler, James Lewis: Microservices a Definition of This new Architectural Term, URL http://martinfowler.com/articles/microservices.html [Zuletzt aufgerufen: 30.10.2020], 2014.

[FB+19] Jonas Fritzsch, Justus Bogner, Stefan Wagner, Alfred Zimmermann: Microservices Migration in Industry: Intentions, Strategies, and Challenges, in 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pages 481–490, 2019.

[Ga95] Erich Gamma: Design Patterns: Elements of Reusable Object-Oriented Software, Pearson Education India, 1995.

[GB+17] Dennis Gannon, Roger Barga, Neel Sundaresan: Cloud-Native Applications, IEEE Cloud Computing, vol. 4, no. 5, pages 16–21, 2017.

[Ga00] David Garlan: Software Architecture: A Roadmap, in Proceedings of the Conference on the Future of Software Engineering, pages 91–101, 2000.

[GG+16] Michael Gebhart, Pascal Giessler, Sebastian Abeck: Challenges of the Digital Transformation in Software Engineering, ICSEA 2016, page 149, 2016.

[Gi18] Pascal Giessler: Domänengetriebener Entwurf von ressourcenorientierten Microservices., Ph.D. thesis, Karlsruhe Institute of Technology, Germany, 2018.

[Gi05] Tom Gilb: Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, Elsevier, 2005.

[GW05] Rainer Gimnich, Andreas Winter: Workflows der Software-Migration, Softwaretechnik-Trends, vol. 25, no. 2, pages 22–24, 2005.

[GA05] Rainer Gimnich, Andreas Winter: Workflows der Software-Migration, Softwaretechnik-Trends, vol. 25, no. 2, pages 22–24, 2005.

[Git-Doc] git: Documentation, URL `https://git-scm.com` [Zuletzt aufgerufen: 30.10.2020].

[Gl02] Robert L Glass: Software Engineering: Facts and Fallacies, 2002.

[Gl07] Martin Glinz: On Non-Functional Requirements, in 15th IEEE International Requirements Engineering Conference (RE 2007), IEEE, pages 21–26, 2007.

[Go11] Hassan Gomaa: Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures, Cambridge University Press, 2011.

[Ka17] Walter Goralski: The Illustrated Network: How TCP/IP Works in a Modern Network, Morgan Kaufmann, 2017.

[Go13] Lev Gorodinski: Sub-Domains and Bounded Contexts in Domain-Driven Design, URL `http://gorodinski.com/blog/2013/04/29/sub-domains-and-bounded-contexts-in-domain-driven-design-ddd/` [Zuletzt aufgerufen: 30.10.2020], 2019.

[OMG20b] The Object Management Group: Architecture-Driven Modernization, URL `https://www.omg.org/adm/` [Zuletzt aufgerufen: 30.10.2020], 2020.

[OMG20] The Object Management Group: Model-Driven Architecture, URL `https://www.omg.org/mda/` [Zuletzt aufgerufen: 30.10.2020], 2020.

[GK+16] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, Olaf Zimmermann: Service Cutter: A Systematic Approach to Service Decomposition, in European Conference on Service-Oriented and Cloud Computing, Springer, pages 185–200, 2016.

[Ha15] Einas Haddad: Service-Oriented Architecture: Scaling the Uber Engineering Codebase as we Grow, URL `https://eng.uber.com/service-oriented-architecture/` [Zuletzt aufgerufen: 30.10.2020], 2015.

[HE+93] Susan DP Harker, Ken D Eason, John E Dobson: The Change and Evolution of Requirements as a Challenge to the Practice of Software Engineering, in [1993] Proceedings of the IEEE International Symposium on Requirements Engineering, IEEE, pages 266–272, 1993.

[HB+94] Anton Frank Harmsen, Jacobus Nicolaas Brinkkemper, JL Han Oei: Situational Method Engineering for Information System Project Approaches, Citeseer, 1994.

[HS17] Wilhelm Hasselbring, Guido Steinacker: Microservice Architectures for Scalability, Agility and Reliability in E-Commerce, in 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), IEEE, pages 243–246, 2017.

[HR20] Alexis Henry, Youssef Ridene: Migrating to Microservices, in Microservices, Springer, pages 45–72, 2020.

[HG+17] Benjamin Hippchen, Pascal Giessler, R Steinegger, Michael Schneider, Sebastian Abeck: Designing Microservice-Based Applications by Using a Domain-Driven Design Approach, International Journal on Advances in Software, vol. 10, no. 3&4, pages 432–445, 2017.

[HS+19] Benjamin Hippchen, Michael Schneider, Pascal Giessler, Sebastian Abeck: Systematic Application of Domain-Driven Design for a Business-Driven Microservice Architecture, International Journal on Advances in Software, vol. 12, no. 3&4, pages 343–355, 2019.

[Hm19] Reda Hmeid: Airbnb's Migration from Monolith to Services, URL `https://www.infoq.com/news/2019/02/airbnb-monolith-migration-soa/` [Zuletzt aufgerufen: 30.10.2020], 2019.

[HM+14] Jez Humble, Joanne Molesky, Barry O'Reilly: Lean Enterprise: How High Performance Organizations Innovate at Scale, O'Reilly Media, Inc., 2014.

[Doc-Ref] Docker Inc.: Dockerfile Reference, URL `https://docs.docker.com/engine/reference/builder/` [Zuletzt aufgerufen: 30.10.2020], 2020.

[Doc-Use] Docker Inc.: Use Multi-Stage Builds, URL `https://docs.docker.com/develop/develop-images/multistage-build/` [Zuletzt aufgerufen: 30.10.2020], 2020.

[IK20] Kasun und Danesh Kuruppu Indrasiri: gRPC: Up and Running, O'Reilly Media, 2020.

[In18] Joseph Ingeno: Software Architect's Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts, Packt Publishing Ltd, 2018.

[Ope-Ope] OpenAPI Initiative: OpenAPI Specification, URL `http://spec.openapis.org/oas/v3.0.3` [Zuletzt aufgerufen: 30.10.2020], 2020.

[Iz16] Yury Izrailevsky: Completing the Netflix Cloud Migration, URL `https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration` [Zuletzt aufgerufen: 30.10.2020], 2016.

[JA+13] Pooyan Jamshidi, Aakash Ahmad, Claus Pahl: Cloud Migration Research: A Systematic Review, IEEE Transactions on Cloud Computing, vol. 1, no. 2, pages 142–157, 2013.

[JB05] Anton Jansen, Jan Bosch: Software Architecture as a Set of Architectural Design Decisions, in 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05), IEEE, pages 109–120, 2005.

[JN+16] David Jaramillo, Duy V Nguyen, Robert Smart: Leveraging Microservices Architecture by Using Docker Technology, in SoutheastCon 2016, IEEE, pages 1–5, 2016.

[Jen-Jen] Jenkins: Jenkins, URL `https://www.jenkins.io/` [Zuletzt aufgerufen: 30.10.2020], 2020.

[KM+17] Miika Kalske, Niko Mäkitalo, Tommi Mikkonen: Challenges when Moving from Monolith to Microservice Architecture, in International Conference on Web Engineering, Springer, pages 32–47, 2017.

[KW+98] Rick Kazman, Steven G Woods, S Jeromy Carrière: Requirements for Integrating Software Architecture and Reengineering Models: CORUM II, in Proceedings Fifth Working Conference on Reverse Engineering (Cat. No. 98TB100261), IEEE, pages 154–163, 1998.

[Ke11] Liz Keogh: ATDD vs. BDD, and a Potted History of some Related Stuff, URL `https://lizkeogh.com/2011/06/27/atdd-vs-bdd-and-a-potted-history-of-some-related-stuff/` [Zuletzt aufgerufen: 30.10.2020], 2011.

[Kh17] Asif Khan: Key Characteristics of a Container Orchestration Platform to Enable a Modern Application, IEEE cloud Computing, vol. 4, no. 5, pages 42–48, 2017.

[KH18] Holger Knoche, Wilhelm Hasselbring: Using Microservices for Legacy Software Modernization, IEEE Software, vol. 35, no. 3, pages 44–49, 2018.

[KR+11] Jens Knodel, H Dieter Rombach, Frank Bomarius, Peter Liggesmeyer: Sustainable Structures in Software Implementations by Live Compliance Checking, Fraunhofer Verlag, 2011.

[Ko08] Heiko Koziolek: Parameter Dependencies for Reusable Performance Specifications of Software Components, Ph.D. thesis, Universität Oldenburg, 2008.

[Kr95] Philippe Kruchten: The 4+1 View Model of Architecture, IEEE Softw., vol. 12, no. 6, page 42–50, 1995.

[Kr04] Philippe Kruchten: The Rational Unified Process: An Introduction, Addison-Wesley Professional, 2004.

[KN+12] Philippe Kruchten, Robert L Nord, Ipek Ozkaya: Technical Debt: From Metaphor to Theory and Practice, IEEE Software, vol. 29, no. 6, pages 18–21, 2012.

[KO+06] Philippe Kruchten, Henk Obbink, Judith Stafford: The Past, Present, and Future for Software Architecture, IEEE software, vol. 23, no. 2, pages 22–30, 2006.

[LA+18] Tom Laszewski, Kamal Arora, Erik Farr, Piyum Zonooz: Cloud Native Architectures: Design High-Availability and Cost-Effective Applications for the Cloud, Packt Publishing Ltd, 2018.

[LB85] Manny M Lehman, Laszlo A Belady: Program Evolution: Processes of Software Change, Academic Press Professional, Inc., 1985.

[Le80] Meir M Lehman: Programs, Life Cycles, and Laws of Software Evolution, Proceedings of the IEEE, vol. 68, no. 9, pages 1060–1076, 1980.

[Le95] Dorothy Leonard-Barton: Wellspring of Knowledge, Harvard Business School Press, Boston, MA, 1995.

[LS+03] Timothy C Lethbridge, Janice Singer, Andrew Forward: How Software Engineers use Documentation: The State of the Practice, IEEE Software, vol. 20, no. 6, pages 35–39, 2003.

[LX+13] Jun Li, Yingfei Xiong, Xuanzhe Liu, Lu Zhang: How does Web Service API Evolution Affect Clients?, in 2013 IEEE 20th International Conference on Web Services, IEEE, pages 300–307, 2013.

[LC+16] Ronald Loui, Lucinda Caughey, Mohammad Ghasemisharif, Rogelio Salvador: Virtualized Dynamic Port Assignment and Windowed Whitelisting for Securing Infrastructure Servers, in 2016 IEEE International Conference on Electro Information Technology (EIT), IEEE, pages 0516–0521, 2016.

[ML+12] Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, Antony Tang: What Industry Needs from Architectural Languages: A Survey, IEEE Transactions on Software Engineering, vol. 39, no. 6, pages 869–891, 2012.

[Ma02] Robert C Martin: Agile Software Development: Principles, Patterns, and Practices, Prentice Hall, 2002.

[Ma09] Robert C Martin: Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2009.

[MB+15] Antonio Martini, Jan Bosch, Michel Chaudron: Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study, Information and Software Technology, vol. 67, pages 237–253, 2015.

[Ma19] Cyrille Martraire: Living Documentation: Continuous Knowledge Sharing by Design, Pearson Education, ISBN 9780134689418, 2019.

[MA11] Chris Matts, Gojko Adzic: Feature Injection: Three Steps to Success, URL `https://www.infoq.com/articles/feature-injection-success/` [Zuletzt aufgerufen: 30.10.2020], 2011.

[MW+09] Alistair Mavin, Philip Wilkinson, Adrian Harwood, Mark Novak: Easy Approach to Requirements Syntax (EARS), in 2009 17th IEEE International Requirements Engineering Conference, IEEE, pages 317–322, 2009.

[MC+17] Genc Mazlami, Jürgen Cito, Philipp Leitner: Extraction of Microservices From Monolithic Software Architectures, in 2017 IEEE International Conference on Web Services (ICWS), IEEE, pages 524–531, 2017.

[MB+20] Manuel Mazzara, Antonio Bucchiarone, Nicola Dragoni, Victor Rivera: Size Matters: Microservices Research and Applications, in Microservices, Springer, pages 29–42, 2020.

[Me12] Tom Mens: On the Complexity of Software Systems, Computer, vol. 45, no. 8, pages 79–81, 2012.

[MT15] Scott Millett, Nick Tune: Patterns, Principles, and Practices of Domain-Driven Design, John Wiley & Sons, 2015.

[ISO-19506] Object Management Group Architecture-Driven Modernization: Knowledge Discovery Meta-Model (KDM), URL `https://www.iso.org/standard/32625.html` [Zuletzt aufgerufen: 30.10.2020], 2012.

[MW16] Fabrizio Montesi, Janine Weber: Circuit Breakers, Discovery, and API Gateways in Microservices, arXiv preprint arXiv:1609.05830, 2016.

[NM+16] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, Mike Amundsen: Microservice Architecture: Aligning Principles, Practices, and Culture, O'Reilly Media, Inc., 2016.

[NG04] Mark EJ Newman, Michelle Girvan: Finding and Evaluating Community Structure in Networks, Physical review E, vol. 69, no. 2, page 026113, 2004.

[Ne15] Sam Newman: Building Microservices: Designing Fine-Grained Systems, O'Reilly Media, Inc., 2015.

[Ne19] Sam Newman: Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith, O'Reilly Media, 2019.

[No06] Dan North: Introducing BDD, URL https://dannorth.net/introducing-bdd/ [Zuletzt aufgerufen: 30.10.2020], 2006.

[No07] Dan North: What is a User Story?, URL https://dannorth.net/whats-in-a-story/ [Zuletzt aufgerufen: 30.10.2020], 2007.

[No09] Dan North: BDD & DDD, URL https://www.infoq.com/presentations/bdd-and-ddd/ [Zuletzt aufgerufen: 30.10.2020], 2009.

[NZ+04] Nur Nurmuliani, Didar Zowghi, S Powell: Analysis of Requirements Volatility During Software Development Life Cycle, in 2004 Australian Software Engineering Conference. Proceedings., IEEE, pages 28–37, 2004.

[NZ+06] Nur Nurmuliani, Didar Zowghi, S Williams: Requirements Volatility & its Impact on Change Effort: Evidence-Based Research in Software Development Projects, in Verified OK, University of South Australia, 2006.

[OS+05] Liam O'Brien, Dennis Smith, Grace Lewis: Supporting Migration to Services using Software Architecture Reconstruction, in 13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05), IEEE, pages 81–91, 2005.

[Ot15] Otto: On Monoliths and Microservices, URL https://www.otto.de/jobs/technology/techblog/artikel/on-monoliths-and-microservices_2015-09-30.php [Zuletzt aufgerufen: 30.10.2020], 2015.

[Pa08] Jeffrey Palermo: The Onion Architecture, URL https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/ [Zuletzt aufgerufen: 30.10.2020], 2008.

[PR04] Ilian Pashov, Matthias Riebisch: Using Feature Modeling for Program Comprehension and Software Architecture Recovery, in Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems, 2004., IEEE, pages 406–417, 2004.

[PB01] Daniel J Paulish, Len Bass: Architecture-Centric Software Project Management: A Practical Guide, Addison-Wesley Longman Publishing Co., Inc., 2001.

[PZ+17] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, Nicolai Josuttis: Microservices in Practice, Part 1: Reality Check and Service Design, IEEE Software, , no. 1, pages 91–98, 2017.

[PG+11] Ricardo Pérez-Castillo, Ignacio Garcia-Rodriguez De Guzman, Mario Piattini: Knowledge Discovery Metamodel-ISO/IEC 19506: A Standard to Modernize Legacy Systems, Computer Standards & Interfaces, vol. 33, no. 6, pages 519–532, 2011.

[PM06] Roland Petrasch, Oliver Meimberg: Model Driven Architecture, 2006.

[Po10] Klaus Pohl: Requirements Engineering: Fundamentals, Principles, and Techniques, Springer Publishing Company, Incorporated, 2010.

[PM+19] Francisco Ponce, Gastón Márquez, Hernán Astudillo: Migrating from Monolithic Architecture to Microservices: A Rapid Review, in 2019 38th International Conference of the Chilean Computer Science Society (SCCC), IEEE, pages 1–7, 2019.

[Pr05] Roger S Pressman: Software Engineering: A Practitioner's Approach, Palgrave Macmillan, 2005.

[PS18] Thomas Prückler, Michael Schrefl: Reverse Engineering in der Datenmodellierung, Information Engineering: Wirtschaftsinformatik im Schnittpunkt von Wirtschafts-, Sozial-und Ingenieurwissenschaften, page 311, 2018.

[RS+17] Florian Rademacher, Sabine Sachweh, Albert Zündorf: Towards a UML Profile for Domain-Driven Design of Microservice Architectures, in International Conference on Software Engineering and Formal Methods, Springer, pages 230–245, 2017.

[RA+07] Usha Nandini Raghavan, Réka Albert, Soundar Kumara: Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks, Physical review E, vol. 76, no. 3, page 036106, 2007.

[RG15] Mazedur Rahman, Jerry Gao: A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development, in 2015 IEEE Symposium on Service-Oriented System Engineering, IEEE, pages 321–325, 2015.

[RM06] John Reekie, Rohan James Mcadam: A Software Architecture Primer, Software Architecture Primer, 2006.

[RH06] Ralf Reussner, Wilhelm Hasselbring: Handbuch der Software-Architektur, dpunkt. verlag, 2006.

[RB+16] Ralf H Reussner, Steffen Becker, Jens Happe, Robert Heinrich, Anne Koziolek, Heiko Koziolek, Max Kramer, Klaus Krogmann: Modeling and Simulating Software Architectures: The Palladio Approach, MIT Press, 2016.

[Ri16] Chris Richardson: Refactoring a Monolith into Microservices, URL https://www.nginx.com/blog/refactoring-a-monolith-into-microservices/ [Zuletzt aufgerufen: 30.10.2020], 2016.

[Ri19b] Chris Richardson: Microservices Patterns: With Examples in Java, Manning Publications, 2019.

[Ri19] Chris Richardson: Pattern: Monolithic Architecture, URL https://microservices.io/patterns/monolithic.html [Zuletzt aufgerufen: 30.10.2020], 2019.

[RN+13] Dominik Rost, Matthias Naab, Crescencio Lima, Christina von Flach Garcia Chavez: Software Architecture Documentation for Developers: A Survey, in European Conference on Software Architecture, Springer, pages 72–88, 2013.

[RW12] Nick Rozanski, Eóin Woods: Software Systems Architecture: Working with Stakeholders using Viewpoints and Perspectives, Addison-Wesley, 2012.

[Ru19] Anna Rud: Why and How Netflix, Amazon, and Uber Migrated to Microservices: Learn from Their Experience, URL https://www.hys-enterprise.com/blog/why-and-how-netflix-amazon-and-uber-migrated-to-microservices-learn-from-their-experience/, [Zuletzt aufgerufen: 30.10.2020], 2019.

[SC+11] Mehrdad Saadatmand, Antonio Cicchetti, Mikael Sjödin: UML-Based Modeling of Non-Functional Requirements in Telecommunication Systems, vol. The Sixth International Conference on Software Engineering Advances (ICSEA 2011), pages 213–220, 2011.

[SH+19] Michael Schneider, Benjamin Hippchen, Pascal Giessler, Chris Irrgang, Sebastian Abeck: Microservice Development Based on Tool-Supported Domain Modeling, in The Fifth International Conference on Advances and Trends in Software Engineering, 2019.

[SS+19] Boris Scholl, Trent Swanson, Peter Jausovec: Cloud Native: Using Containers, Functions, and Data to Build Next-Generation Applications, O'Reilly Media, 2019.

[Sc12] Hannes Schwarz: Universal Traceability. A Comprehensive, Generic, Technology-Independent, and Semantically Rich Approach, Logos Verlag Berlin GmbH, 2012.

[Se03] Bran Selic: The Pragmatics of Model-Driven Development, IEEE software, vol. 20, no. 5, pages 19–25, 2003.

[SK03] Shane Sendall, Wojtek Kozaczynski: Model Transformation: The Heart and Soul of Model-Driven Software Development, IEEE Software, vol. 20, no. 5, pages 42–45, 2003.

[Ama-Sim] Amazon Web Services: Simple Storage Service, URL https://aws.amazon.com/de/s3/ [Zuletzt aufgerufen: 30.10.2020], 2020.

[SG96] Mary Shaw, David Garlan: Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996.

[Sm14] John Ferguson Smart: BDD in Action, Manning Publications, 2014.

[Sn07] Harry Sneed: Migrating to Web Services: A Research Framework, in Proceedings of the International, 2007.

[Sn84] Harry M Sneed: Software Renewal: A Case Study, IEEE Software, , no. 3, pages 56–63, 1984.

[Sn06] Harry M Sneed: Estimating the Costs of a Reengineering Project, in 12th Working Conference on Reverse Engineering (WCRE'05), IEEE, pages 9–pp, 2005.

[SH+04] Harry M Sneed, Martin Hasitschka, Maria-Therese Teichmann: Software-Produktmanagement: Wartung und Weiterentwicklung bestehender Anwendungssysteme, dpunkt-Verlag, 2004.

[ISO-25010] ISO/IEC JTC 1/SC 7 Software, Systems Engineering: ISO/IEC 25010:2011 - Systems and Software Engineering — Systems and Software Quality Requirements and Evaluation (SQuaRE) — System and Software Quality Models, URL https://www.iso.org/standard/35733.html [Zuletzt aufgerufen: 30.10.2020], 2017.

[SA+12] Samaneh Soltani, Mohsen Asadi, Dragan Gašević, Marek Hatala, Ebrahim Bagheri: Automated Planning for Feature Model Configuration-Based on Functional and Non-Functional Requirements, in Proceedings of the 16th International Software Product Line Conference-Volume 1, pages 56–65, 2012.

[St73] Herbert Stachowiak: Allgemeine Modelltheorie, Springer, 1973.

[SL13] Kewei Sun, Ying Li: Effort Estimation in Cloud Migration Process, in 2013 IEEE seventh international symposium on service-oriented system engineering, IEEE, pages 84–91, 2013.

[Su00] Stanley M Sutton: The Role of Process in Software Start-Up, IEEE Software, vol. 17, no. 4, pages 33–39, 2000.

[Sma-Swa] Swagger: SmartBear Software, URL https://swagger.io/ [Zuletzt aufgerufen: 30.10.2020], 2020.

[TL+17] Davide Taibi, Valentina Lenarduzzi, Claus Pahl: Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation, IEEE Cloud Computing, vol. 4, no. 5, pages 22–32, 2017.

[TV+17] Davide Taibi, Valentina Lenarduzzi, Claus Pahl: Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation, IEEE Cloud Computing, vol. 4, no. 5, pages 22–32, 2017.

[TL+18] Davide Taibi, Valentina Lenarduzzi, Claus Pahl: Architectural Patterns for Microservices: A Systematic Mapping Study, SCITEPRESS, 2018.

[TL+20] Davide Taibi, Valentina Lenarduzzi, Claus Pahl: Microservices Anti-Patterns: A Taxonomy, in Microservices, Springer, pages 111–128, 2020.

[TM+09] Richard N Taylor, Nenad Medvidovic, Eric Dashofy: Software Architecture: Foundations, Theory, and Practice, John Wiley & Sons, 2009.

[HAP-HAP] HAProxy Technologies: HAProxy, URL `https://www.haproxy.com/` [Zuletzt aufgerufen: 30.10.2020], 2020.

[Ti00] Walter F Tichy: Hints for reviewing Empirical Work in Software Engineering, Empirical Software Engineering, vol. 5, no. 4, pages 309–312, 2000.

[TS+12] Saurabh Tiwari, Santosh Singh Rathore, Atul Gupta: Selecting Requirement Elicitation Techniques for Software Projects, in 2012 CSI Sixth International Conference on Software Engineering (CONSEG), IEEE, pages 1–10, 2012.

[Tu18] Nick Tune: Aligning Organisation Design & Software Architecture with Business Strategy: Business Capability Classification, Tech. rep., Medium, URL `https://medium.com/nick-tune-tech-strategy-blog/aligning-organisation-design-software-architecture-with-business-strategy-capability-43e5bca341b7` [Zuletzt aufgerufen: 30.10.2020], 2018.

[Tu19] Nick Tune: EventStorming Modelling Tips to Facilitate Microservice Design, URL `https://medium.com/nick-tune-tech-strategy-blog/eventstorming-modelling-tips-to-facilitate-microservice-design-1b1b0b838efc` [Zuletzt aufgerufen: 30.10.2020], 2019.

[Tu20b] Nick Tune: Bounded Context Canvas V3: Simplifications and Additions, URL `https://medium.com/nick-tune-tech-strategy-blog/bounded-context-canvas-v2-simplifications-and-additions-229ed35f825f` [Zuletzt aufgerufen: 30.10.2020], 2020.

[Tu20c] Nick Tune: Core Domain Patterns, Tech. rep., URL https://medium.com/nick-tune-tech-strategy-blog/core-domain-patterns-941f89446af5 [Zuletzt aufgerufen: 30.10.2020], 2020.

[Tu20] Nick Tune: Dissecting Bounded Contexts, Tech. rep., DDD Europe 2020, URL https://www.infoq.com/news/2020/02/bounded-contexts-ddd-europe/ [Zuletzt aufgerufen: 30.10.2020], 2020.

[UN+16] Takanori Ueda, Takuya Nakaike, Moriyoshi Ohara: Workload Characterization for Microservices, in 2016 IEEE International Symposium on Workload Characterization (IISWC), IEEE, pages 1–10, 2016.

[Ve13] Vaughn Vernon: Implementing Domain-Driven Design, Addison-Wesley, 2013.

[Vi15] Manish Virmani: Understanding DevOps & Bridging the Gap from Continuous Integration to Continuous Delivery, in Fifth International Conference on the Innovative Computing Technology (INTECH 2015), IEEE, pages 78–82, 2015.

[VA+11] Oliver Vogel, Ingo Arnold, Arif Chughtai, Timo Kehrer: Software Architecture: A Comprehensive Framework and Guide for Practitioners, Springer Science & Business Media, 2011.

[VS+13] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, Simon Helsen: Model-Driven Software Development: Technology, Engineering, Management, John Wiley & Sons, 2013.

[VB+17] Dragana Vukovic, Drazen Brdjanin, Slavko Maric: A UML-Based Approach to Reverse Engineering of Relational Databases, in 2017 25th Telecommunication Forum (TELFOR), IEEE, pages 1–4, 2017.

[WS+06] Hiroshi Wada, Junichi Suzuki, Katsuya Oba: Modeling Non-Functional Aspects in Service-Oriented Architecture, in 2006 IEEE International Conference on Services Computing (SCC'06), IEEE, pages 222–229, 2006.

[Wa14] Christian Wagner: Model-Driven Software Migration: A Methodology, in Model-Driven Software Migration: A Methodology, Springer, pages 67–105, 2014.

[WL01] Qing Wang, Xufang Lai: Requirements Management for the Incremental Development Model, in Proceedings Second Asia-Pacific Conference on Quality Software, IEEE, pages 295–301, 2001.

[WB13] Karl Wiegers, Joy Beatty: Software Requirements, Pearson Education, 2013.

[Wi17] Duncan CE Winn: Cloud Foundry: The Definitive Guide: Develop, Deploy, and Scale, O'Reilly Media, Inc., 2017.

[WR+12] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, Anders Wesslén: Experimentation in Software Engineering, Springer Science & Business Media, 2012.

[WH+17] Matt Wynne, Aslak Hellesoy, Steve Tooke: The Cucumber Book: Behaviour-Driven Development for Testers and Developers, Pragmatic Bookshelf, 2017.

[YL+08] Dowming Yeh, Yuwen Li, William Chu: Extracting Entity-Relationship Diagram from a Table-Based Legacy Database, Journal of Systems and Software, vol. 81, no. 5, pages 764–771, 2008.

[YD+08] Yijun Yu, Julio Cesar Sampaio do Prado Leite, Alexei Lapouchnian, John Mylopoulos: Configuring features with stakeholder goals, in Proceedings of the 2008 ACM symposium on Applied computing, pages 645–649, 2008.

[Zi17] Olaf Zimmermann: Microservices Tenets, Computer Science-Research and Development, vol. 32, no. 3-4, pages 301–310, 2017.

[ZP06] Joe Zou, Christopher J Pavlovski: Modeling Architectural Non-Functional Requirements: From Use Case to Control Case, in 2006 IEEE International Conference on e-Business Engineering (ICEBE'06), IEEE, pages 315–322, 2006.