



Institut für Technische Informatik und Ingenieurinformatik  
Fakultät für Informatik und Automatisierung  
Fachgebiet System- und Software-Engineering

Bachelorarbeit

# **Modellgetriebene Entwicklung von Microservices**

vorgelegt von

Patrik Löffler  
Matrikelnummer 56207

Betreuer:

Prof. Dr.-Ing. habil. Armin Zimmermann  
Dipl.-Inf. Tino Jungebloud

Ilmenau, den 30. Januar 2024



# **Selbstständigkeitserklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Ilmenau, den 30. Januar 2024

---

Patrik Löffler



*„Ich weiß, dass ich nichts weiß“*  
– Sokrates



# Inhaltsverzeichnis

<b>1 Abstract</b>	<b>1</b>
<b>2 Einleitung</b>	<b>3</b>
2.1 Motivation . . . . .	3
2.2 Zielstellung . . . . .	3
<b>3 Grundlagen</b>	<b>5</b>
3.1 Domain-Driven Design . . . . .	5
3.1.1 Definition . . . . .	5
3.1.2 Bausteine des Model-Driven Design . . . . .	6
3.1.3 Strategic Design . . . . .	9
3.2 Microservices . . . . .	12
3.2.1 Geschichte und Definition . . . . .	12
3.2.2 Eigenschaften von Microservices . . . . .	13
3.2.3 Architektur von Microservice-Systemen . . . . .	15
3.2.4 Kommunikation . . . . .	18
3.2.5 Betrieb . . . . .	19
3.2.6 Microservices mit Spring Boot . . . . .	19
3.3 Cloud-Computing . . . . .	20
3.3.1 Docker . . . . .	20
3.3.2 Kubernetes . . . . .	21
3.3.3 Google Cloud Plattform . . . . .	22
3.4 Modellgetriebene Entwicklung von Microservices . . . . .	23
3.4.1 Verfahren . . . . .	23
3.4.2 Werkzeuge . . . . .	24
<b>4 Konzeption des Metamodells</b>	<b>25</b>
4.1 Konzeptionelle Rahmenbedingungen . . . . .	25
4.1.1 Einleitende Überlegungen . . . . .	25
4.1.2 Einbindung aktueller Technologien . . . . .	26
4.1.3 Auswahl der Technologie . . . . .	30
4.2 Entwicklung der eigenen DSL . . . . .	31
4.2.1 Modellierung des Model-Driven Designs . . . . .	32
4.2.2 Modellierung des Strategic Designs . . . . .	35
4.2.3 Modellierung der technischen Modellebene . . . . .	36

4.2.4	Modellierung der Infrastrukturellen Umgebung	37
<b>5</b>	<b>Entwicklung einer konkreten Syntax</b>	<b>39</b>
5.1	Entwurf des Editors	40
5.2	Beschreibung der Diagramme	42
5.2.1	ContextRelationshipDescription	42
5.2.2	DomainModelDescription	43
5.2.3	AggregateStructureDescription	44
5.2.4	MicroserviceCommunicationDescription	45
5.2.5	MicroserviceInterfaceDescription	46
5.2.6	InfrastructureOverviewDescription	47
5.3	Konkretisierung des Modellierungsworkflows	48
<b>6</b>	<b>Konzeption und Umsetzung der Codegenerierung</b>	<b>49</b>
6.1	Generierungstemplate	49
6.2	Migration von Diensten durch Generierung	57
6.2.1	Acceleo Protected Areas	57
6.2.2	Open Rewrite	59
<b>7</b>	<b>Anwendung</b>	<b>61</b>
7.1	Modellierung einer Microservice-Architektur für einen Anwendungsfall	61
7.1.1	Vorstellung des Anwendungsfalles	61
7.1.2	Ablauf der Modellierung mit Editor	62
7.2	Inbetriebnahme	66
<b>8</b>	<b>Fazit und Ausblick</b>	<b>69</b>

# Abbildungsverzeichnis

3.1 Bausteine des Model-Driven Design . . . . .	6
3.2 Darstellung von Bounded Contexts und ihre Rollen in der Context Map . . . . .	11
3.3 Eigenschaften von Microservices . . . . .	13
3.4 Darstellung einer Broker-Topologie in der EDA . . . . .	16
3.5 Darstellung einer Mediator-Topologie in der EDA . . . . .	17
4.1 Das AjiL Metamodell . . . . .	26
4.2 Das Context Mapper Metamodell . . . . .	27
4.3 Das OCCI Metamodell . . . . .	28
4.4 OCCLware Anwendungsumgebung . . . . .	29
4.5 Architekturkonzept von ARGON . . . . .	30
4.6 Erkennbare Schichten in einem Modell einer Microservice-Architektur . . . . .	31
4.7 Metamodell - Fokus: Fachliche Schicht, Domänenmodelle und Pakete . . . . .	32
4.8 Metamodell - Fokus: Entity, Value Object und Aggregate . . . . .	33
4.9 Metamodell - Fokus: Service, Repository, Factory und Behaviour . . . . .	34
4.10 Metamodell - Fokus: Strategic Design . . . . .	35
4.11 Metamodell - Fokus: Technische Schicht . . . . .	36
4.12 Metamodell - Fokus: Infrastrukturschicht . . . . .	37
5.1 Ein abstrakter Modellierungsworkflow für das Microservice-Metamodell . . . . .	39
5.2 Umsetzen von rekursiven Aufrufen durch das importieren von Nodes mit Sirius . . . . .	41
5.3 Eine kognitiv anspruchsvolle Aggregate Visualisierung . . . . .	41
5.4 Editoransicht der ContextRelationshipDescription . . . . .	42
5.5 Editoransicht der DomainModelDescription . . . . .	43
5.6 Editoransicht der AggregateStructureDescription . . . . .	44
5.7 Editoransicht der MicroserviceCommunicationDescription . . . . .	45
5.8 Editoransicht der MicroserviceInterfaceDescription . . . . .	46
5.9 Editoransicht der InfrastructureOverviewDescription . . . . .	47
5.10 Abbildung des abstrakten Modellierungsworkflows auf einem für diesen Editor angepassten Workflow . . . . .	48
6.1 Generierter Code mit geschützten Bereichen . . . . .	58
7.1 Use-Case Diagramm für den Webshop-Anwendungsfall . . . . .	61
7.2 Modellierung der Bounded Contexts für den Anwendungsfall . . . . .	62

7.3	Modellierung des OrderDomainModel für den Anwendungsfall . . . . .	63
7.4	Konzeptionell symetrische Modellierung des CustomerDomainModel . . . . .	63
7.5	Modellierung der Kommunikation zwischen Microservices für den Anwendungsfall	64
7.6	Modellierung der Schnittstellen auf Serviceebene für den Anwendungsfall . . . . .	64
7.7	Modellierung der Infrastruktur für den Anwendungsfall . . . . .	65
7.8	Implementieren von fehlenden Methoden vor der Inbetriebnahme . . . . .	66
7.9	Implementieren von fehlenden Klassenattributen vor der Inbetriebnahme . . . . .	66
7.10	Abschließende Code-Formatierung für bessere Lesbarkeit . . . . .	67

# 1 Abstract

Diese Arbeit erforscht die modellgetriebene Softwareentwicklung von Microservice-Architekturen. Im Fokus steht die Konzeption eines Metamodells für diese. Ziel ist es, lauffähige und in einer Cloud-Infrastruktur ausführbare Anwendungen zu modellieren und zu generieren. Hierbei sollen die Designprinzipien des Domain-Driven Designs angewendet werden. Die zentralen Beiträge dieser Arbeit umfassen die Analyse der Vor- und Nachteile verschiedener Entwurfsentscheidungen sowie die Überprüfung der Ausdrucksfähigkeit des Metamodells mittels konkreter Syntax und Visualisierung. Ein besonderer Schwerpunkt liegt auf der Untersuchung, wie sich mit diesem Modell Anwendungen für reale Problemdomänen abbilden lassen.

Ein wesentliches Ergebnis ist die erfolgreiche Generierung von Java-Spring-Applikationen, womit die grundlegende Machbarkeit bewiesen wird. Weiterhin werden dabei auftretende Probleme und mögliche Lösungsansätze im Kontext der Codegenerierung beleuchtet. Zudem wird erforscht, inwiefern sich dieser modellgetriebene Ansatz für Refaktorisierungen einsetzen lässt. Theoretische Grundlagen und eine umfassende State-of-the-Art-Recherche bilden das Fundament dieser Arbeit. Methodisch wurde ein schichtbasiertes Konzept für die Abstraktion entwickelt, das iterativ verfeinert und durch anwendungsfokussierte Entwurfsentscheidungen ergänzt wurde.

Neben der Realisierbarkeit der Metamodellierung verschiedener Aspekte von Microservice-Architekturen konnte auch die variierende Komplexität der einzelnen Aspekte und Ansätze zu deren Lösung aufgezeigt werden. Darüber hinaus werfen die Herausforderungen und Grenzen des aktuellen Ansatzes neue Fragestellungen auf. Insbesondere wird diskutiert, inwiefern eine vertiefende Modellierung dieser Domäne, ergänzend zur breiten Perspektive, nützlich sein könnte. Diese Erkenntnisse bieten wichtige Impulse für zukünftige Forschungen in der modellgetriebenen Entwicklung von Microservices.

This work explores the model-driven software development of microservice architectures. The focus is on the conception of a metamodel for these. The goal is to model and generate applications that are runnable and executable in a cloud infrastructure. In this process, the design principles of Domain-Driven Design are to be applied. The central contributions of this work include the analysis of the advantages and disadvantages of various design decisions, as well as the examination of the expressiveness of the metamodel through concrete syntax and visualization. A particular emphasis is placed on investigating how this model can represent applications for real problem domains.

A key result is the successful generation of Java-Spring applications, proving the fundamental feasibility. Furthermore, problems that arise and potential solutions in the context of code generation are illuminated. Additionally, it explores to what extent this model-driven approach can be used for refactorings. Theoretical foundations and a comprehensive state-of-the-art research form the basis of this work. Methodologically, a layer-based concept for abstraction was developed, which was iteratively refined and supplemented by application-focused design decisions.

Besides the realizability of the metamodeling of various aspects of microservice architectures, the varying complexity of the individual aspects and approaches to their solutions was also demonstrated. Moreover, the challenges and limitations of the current approach raise new questions. In particular, it discusses to what extent a more in-depth modeling of this domain, in addition to the broad perspective, could be useful. These insights provide important impulses for future research in the model-driven development of microservices.

## 2 Einleitung

### 2.1 Motivation

In einer dieser vorangegangenen Seminararbeit wurde bereits aufgezeigt, dass Webdienste mittels modellgetriebener Softwareentwicklung umgesetzt werden können. Auf dieser Grundlage strebt diese Arbeit an, einen verfeinerten Ansatz zu entwickeln, der sich spezifisch auf die Microservice-Architektur von Webdiensten fokussiert. Die Entwicklung von Microservices – kleinen, skalierbaren Anwendungen, die oft in verteilten Systemen Verwendung finden – könnte von Vorteilen wie gesteigerter Wiederverwendbarkeit, vereinfachter Wartung und generell beschleunigter Entwicklungszeit profitieren. Besonders interessant ist dabei das mögliche Potenzial welches im Kontext von Abhängigkeitsbezogenen Migrationen im Softwarelebenszyklus existieren könnte. Es stellt sich jedoch die Frage, ob Microservice-Anwendungen in ihrer gesamten konzeptionellen Breite durch ein Metamodell adäquat erfasst werden können. Zudem sind die domänenspezifischen Herausforderungen, die sich hierbei ergeben, von besonderem Interesse [Lö23].

### 2.2 Zielstellung

Ziel der Arbeit ist es, Anwendungen, die das Microservice-Paradigma verwenden, mit Werkzeugen und Methoden der modellgetriebenen Softwareentwicklung zu entwickeln. Dazu wird eine Domänenspezifische Sprache in Form eines Metamodells konzipiert, welches die grundlegenden Eigenschaften des Paradigmas abstrahiert. Dies wird ergänzt durch die Berücksichtigung von Konzepten des Domain-Driven Design zur Modellierung der Geschäftslogik.

Anwendungen, die mit diesem Metamodell entwickelt werden, sollen Abhängigkeiten, wie Frameworks und Bibliotheken, integrieren. Weiterhin soll untersucht werden, wie diese Anwendungen effektiv migriert werden können. Dabei soll erforscht werden, ob sich ein modellgetriebener Ansatz für die Durchführung von Refaktorisierungen eignet, insbesondere im Kontext von Versionsaktualisierungen bei Abhängigkeiten.

Außerdem wird untersucht, inwiefern die der Anwendung zugrundeliegende IT-Infrastruktur in einem solchen Metamodell berücksichtigt werden kann und ob es möglich ist, Deployment-fähige Anwendungen mithilfe eines Metamodells zu generieren.



# 3 Grundlagen

## 3.1 Domain-Driven Design

### 3.1.1 Definition

Die Domäne einer Software bezeichnet das Themengebiet oder den Anwendungsbereich, in dem das Programm eingesetzt wird [Eva07, S.2] [PBG11, S.307]. „Domain-Driven Design“ (Abk.: DDD) ist eine Herangehensweise, Problemdomänen in ein Modell für die Softwareentwicklung zu überführen. Modellieren ist eine grundlegende Methode der Wissenschaft um mit komplexen Systemen umzugehen. Eine Abstraktion eines solchen Systems, die es ermöglicht, Fragestellungen dazu zu beantworten, wird als Modell bezeichnet [BD10].

Geprägt wurde das DDD durch Eric Evans in seinem Buch „Domain-Driven Design: Tackling Complexity in the Heart of Software“ [Eva07]. Kern des domänengebundenen Entwurfs ist es, die reale, oft unübersichtliche Fachlichkeit fortschreitend in feiner geschnittene Teilbereiche zu kapseln. Diese sollen möglichst wenige Abhängigkeiten zueinander haben [Dow18, S.63, 70]. Dieser auf das Domänenmodell fokussierte Ansatz versucht, Kernaspekte einer Anwendung in das Modell zu integrieren, anstatt sie über externe Dienste einzubinden [Das17, S.74].

Evans stellt drei Grundeigenschaften fest, welche bei der Wahl eines Modells im DDD zu berücksichtigen sind [Eva07, S. 3-4]:

- Es existiert immer eine Abhängigkeit zwischen Modell und Implementierung.
- Die Terminologie während der Softwareentwicklung ist eine Implikation des zugrunde liegenden Modells.
- Nützliches und aufgearbeitetes Wissen kann diesem Modell entnommen werden.

Ein weiteres Kernkonzept des DDD ist die „Ubiquitous Language“. Sie verbindet Entwickler und Domänen-Experten durch eine auf dem Modell basierende Sprache. Dabei definiert sie eindeutig die verwendeten Begriffe und die ihnen zugeordneten Fachlichkeiten [Eva07, S. 25-27]. Ein weiterer Kern ist das „Model-Driven Design“ (Abk.: MDD). MDD betont und bezeichnet das notwendige Ineinandergreifen der Konzeption eines Domänenmodells mit dem Entwurf der Anwendungssoftware. Um eine enge Übereinstimmung zwischen Modell und Software sicherzustellen, ist es von Vorteil, innerhalb eines durch Software-Tools unterstützten Modellierungsansatzes zu arbeiten. Objektorientierte Programmiersprachen eignen sich besonders gut, da sich die objekthaften Eigenschaften von Modellen gut abbilden lassen [Eva07, S. 49-51].

### 3.1.2 Bausteine des Model-Driven Design

Das DDD definiert verschiedene Bausteine des MDD [Eva07, S. 65]:

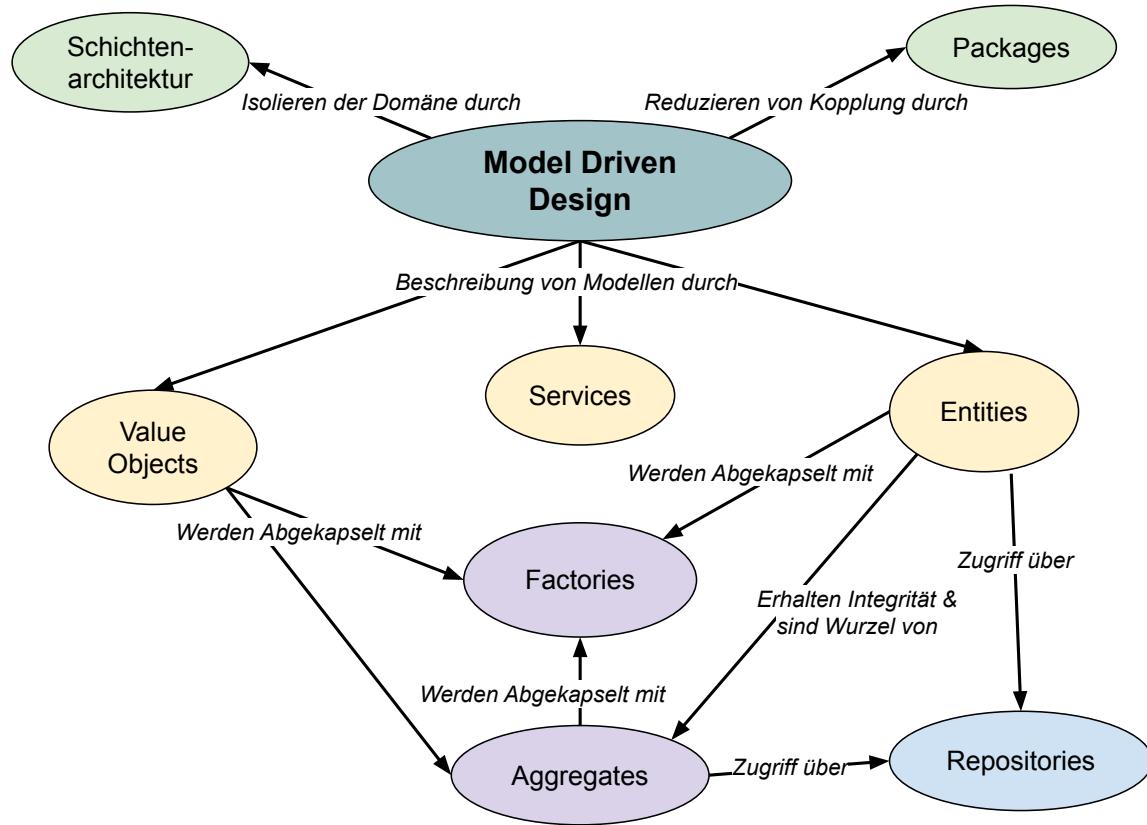


Abbildung 3.1: Die Bausteine des Model-Driven Design und wie Sie im Kontext zu anderen Bausteinen einzuordnen sind

### Schichtenarchitektur

Der schichtbasierte Architekturstil teilt Software in verschiedene Schichten auf. Jede Schicht fasst Komponenten zusammen, die ähnliche Aufgaben haben. Das macht die Software einfacher zu verstehen und zu verwalten. Die Architektur ist nach technischen Funktionen gegliedert. Gängige Schichten sind die Präsentationsschicht, Geschäftslogikschicht, Persistenzschicht, Datenbankschicht oder die Dienstschicht. Jedes Element einer Schicht hängt nur von Elementen derselben oder einer tieferen Schicht ab. DDD fordert, die Geschäftslogik in eine eigene Schicht, den „Domain Layer“, zu legen. Dies sorgt für eine enge Verbindung zum Modell, wie es das MDD vorsieht [Eva07, S. 69 - 73] [RF21, S.139-141].

## Packages

„Packages“, auch als „Modules“ bezeichnet, sorgen im Modell für starke Kohäsion innerhalb und schwache Kopplung zwischen ihnen, indem sie zusammengehörige Elemente bündeln [Eva07, S. 109 - 116].

## Entities

„Entities“ sind einzigartige, identifizierbare Objekte eines speziellen Bereichs. Dies kann z.B. mit automatisch erzeugten Identifikationswerten (Abk.: ID) umgesetzt werden. Ihre spezifischen Merkmale sind sekundär. Sie symbolisieren die durchgehende Existenz eines Objekts über dessen Lebenszyklus. Beim Erstellen eines Modells muss diese Kontinuität gewährleistet werden. Klassendetails, Aufgaben und Assoziationen sollten ebenso die Identität der Entity zeigen. Entities sollen sparsam verwendet werden. Nur notwendiges Verhalten soll ihnen hinzugefügt werden. Zusätzliches Verhalten soll zu anderen zugehörigen Objekten verschoben werden [Eva07, S. 89-96] [Das17, S.75].

## Value Objects

„Value Objects“ sind Typen ohne eigene Identität, die spezifische Werte repräsentieren. Sie können aus anderen Value Objects bestehen und Entities referenzieren. Oft werden sie als Parameter zwischen Objekten oder als Eigenschaften von Entities genutzt. Value Objects sollten eine hohe Kohäsion besitzen und sind unveränderlich, um sicher geteilt werden zu können. Bidirektionale Verbindungen bei Value Objects sollten aufgrund ihrer fehlenden Identität vermieden werden [Eva07, S. 97-103] [Das17, S.75].

## Services

„Services“ bieten eine spezielle Funktion im Modell an, ohne dabei Daten oder Zustände zu speichern. Sie werden dabei nach ihrer Aufgabe benannt. Services starten Vorgänge und arbeiten mit den Objekten des Modells. Sie kombinieren Schritte in Geschäftsprozessen und sollten bedacht eingesetzt werden. Außerdem sollen sie nicht alle Aufgaben von Entities und Value Objects abnehmen. Es existiert eine klare Schnittstelle im Bezug auf andere Elemente des Modells. Da Services auch außerhalb der Domain Layer auftreten können ist eine präzise fachliche Betrachtung nötig [Eva07, S. 104 - 108] [Das17, S.74].

## Aggregates

„Aggregates“ sind Ansammlungen von Objekten, die zusammen als Einheit betrachtet werden. Ihr Zweck ist es, Veränderungen als Ganzes zu handhaben, um Konsistenz zu gewährleisten.

Ein direkter Zugriff auf einzelne Bestandteile ist nicht möglich. Deswegen gibt es in einem Aggregate ein Hauptobjekt (Wurzel), das für alle Aktionen verwendet wird. Andere Objekte können nur auf dieses Hauptobjekt verweisen, wobei die enthaltenen Objekte sich untereinander gegenseitig verweisen können. Objekte innerhalb des Aggregates, die nicht die Hauptfunktion ausüben, sind nur in ihrem Kontext eindeutig identifizierbar [Eva07, S. 123 - 135] [Das17, S.76].

## Factories

Eine „Factory“ erstellt Objekte. Dies ist nützlich, wenn das Erstellen eines Objekts mehr als nur das Ausführen eines Konstruktors beinhaltet, da manchmal besondere Regeln oder Logiken benötigt werden. Sie stellt sicher, dass jedes erstellte Objekt korrekt und vollständig ist. Auch wenn eine Factory dafür sorgt, dass ein Objekt richtig erstellt wird, kann sie Überprüfungen an andere Teile auslagern. Es gibt weiterhin Factories, die bestehende Objekte wiederherstellen. Dabei gibt es funktionale Unterschiede. Beispielsweise weisen diese keine neuen IDs zu und müssen Verletzungen von Regeln anders behandeln [Eva07, S. 136 - 146] [Das17, S.77].

## Repositories

„Repositories“ sind dafür zuständig, Daten von Entities oder Aggregates, sicher zu speichern und zu verwalten. Die Hauptidee von Repositories ist, einen zentralen Ort zu haben, der dafür sorgt, dass alles korrekt und konsequent gespeichert wird. Ein Repository ist mit einer Sammlung vergleichbar, aus der man Daten anhand von bestimmten Kriterien abfragen kann. Dabei fügt das Repository neue Objekte der Datenbank hinzu oder löscht sie. Wenn man Daten aus einem Repository möchte, nutzt man spezielle Suchmethoden. Dabei kann die Suche einfach oder auch sehr detailliert sein [Eva07, S. 147 - 161] [Das17, S.76].

## Domain Event

„Domain Events“ sind Ereignisse, die für einen Geschäftsbereich relevant sind und zu einem Domänenmodell gehören. Sie entstehen aus Geschäftsvorfällen und haben eine spezifische Domänensemantik. In verteilten Systemen, wie zum Beispiel in Microservice-Architekturen, ergibt sich häufig die Notwendigkeit für Domain Events. Wenn in einem Microservice ein solches Ereignis eintritt, können andere Dienste an diesen interessiert sein. Eine Lösung hierfür ist, dass der betreffende Dienst ein Domain Event veröffentlicht, bei dem sich interessierte Dienste registrieren können. Dies fördert die Entkopplung der Dienste, da ihre Kommunikation hauptsächlich über Domain Events erfolgt [Das17, S.78] [Rit23].

### **3.1.3 Strategic Design**

Als „Strategic Design“ wird im DDD beschrieben, wie verschiedene Domänenmodelle miteinander interagieren können. Dabei müssen die strategischen Designprinzipien die Designentscheidungen leiten, um Abhängigkeiten zu verringern und Klarheit zu erhöhen, ohne dabei Interoperabilität und Synergie zu gefährden [Wol16, S. 45] [Eva07, S. 334].

#### **Bounded Context**

Im Zentrum dieser Designentscheidungen steht der „Bounded Context“. Grundgedanke ist, dass jedes Domänenmodell nur in einem klar definierten Rahmen sinnvoll ist. Dieser Rahmen kann ein spezifischer Codebereich oder die Arbeit eines Teams sein. Der Bounded Context beschreibt Bedingungen, unter denen Begriffe im Modell eine bestimmte Bedeutung haben. Auch legen sie fest, in welchem Bereich ein Modell und somit auch die dazugehörige Ubiquitous Language gültig ist. Eine Domäne kann aus mehreren solchen Bounded Contexts bestehen. Bounded Contexts können auch Informationen mit anderen Bounded Contexts teilen. Jeder hat weiterhin eine Schnittstelle, die bestimmt, welche Modelle für andere Bounded Contexts verfügbar sind. Im Rahmen des Bounded Contexts sollte das Modell konsistent gehalten werden, um sich keine Sorgen um die Relevanz außerhalb dieser Grenzen machen zu müssen [Wol16, S. 45] [New15, S.57] [Eva07, S. 335 - 340] [Dow18, S.64-65].

#### **Shared Kernel**

Ein Bereich des Domänenmodells kann dazu dienen, zwischen Teams geteilt zu werden. Bei einem solchen „Shared Kernel“ teilen sich die Domänenmodelle einige Elemente, können sich jedoch in anderen Aspekten unterscheiden. Dies kann entweder implizit oder explizit durch den gemeinsamen Codegebrauch bzw. durch den Zugriff auf dieselben Tabellen in der Datenbank geschehen. Oft bezeichnet der Shared Kernel die Kern-Domäne, eine Sammlung generischer Teilbereiche oder beides. Ziel dabei ist es, Wiederholungen zu vermeiden [Eva07, S. 354 - 355] [Wol16, S. 45] [Dow18, S.65].

#### **Customer/Supplier**

Zwischen zwei verschiedenen Kontexten kann eine Beziehung entstehen, die aus einer nachgelagerten und einer vorgelagerten Komponente besteht. Das Domain-Driven Design definiert deshalb eine klare Kunden-/Lieferantenbeziehung für solche Kontexte. Die „Customer/Supplier“-Beziehung zeigt, dass ein Untersystem dem Aufrufer ein Domänenmodell zur Verfügung stellt. Der Aufrufer, also der Kunde, legt die Struktur des Modells fest. Der Anbieter der Schnittstelle richtet sich beim Design nach den Wünschen des Kunden und gibt ihm oft ein Mitspracherecht [Eva07, S. 356 - 360] [Wol16, S. 45,47] [Dow18, S.66].

## **Conformist**

Wenn die Anforderungen eines nachgelagerten, abhängigen Subsystems durch den vorgelagerten Teil gedeckt werden können, bietet es sich an, dem nachgelagerten Teil die Rolle des „Conformist“ zuzuweisen. Dies vereinfacht den Datenaustausch zwischen den beiden Kontexten, indem beide dasselbe Modell verwenden. Der Aufrufer nutzt also das gleiche Modell wie das aufgerufene System und passt sich somit dem anderen Modell an [Eva07, S. 361 - 363] [Wol16, S. 47] [Dow18, S.66].

## **Anticorruption Layer**

Wenn man Systeme kombiniert, die auf verschiedenen Modellen basieren, kann es passieren, dass das Modell des einen Systems durch das andere beeinflusst wird. Ein „Anticorruption Layer“ hilft, indem es eine Schutzschicht bildet. Diese Schicht ermöglicht es den Nutzern, mit ihrem gewohnten Modell zu arbeiten, ohne das andere System zu beeinflussen. Sie dient als Übersetzer zwischen den beiden Modellen und sorgt dafür, dass sie unabhängig voneinander bleiben. Über eine Schnittstelle kommuniziert sie mit dem anderen System, ohne dass große Änderungen an diesem notwendig sind. Sie kann bei Bedarf in beide Richtungen übersetzen [Eva07, S. 364 - 370] [Wol16, S. 47] [Dow18, S.66].

## **Separate Ways**

Wenn der Nutzen im Vergleich zu den Integrationskosten zu gering ist, kann es sinnvoll sein, festzulegen, dass ein bestimmter Kontext keine Verbindung zu anderen hat. Im DDD spricht man hier von „Separate Ways“, was bedeutet, dass die beiden Systeme nicht verbunden werden und unabhängig voneinander bleiben. Anstatt Lösungen für Probleme gemeinsam zu nutzen, müssen diese auf einem klaren, speziellen Weg innerhalb dieses Rahmens umgesetzt werden. [Eva07, S. 371 - 373] [Wol16, S. 47] [Dow18, S.66].

## **Open Host Service**

Wenn ein Teilsystem mit vielen anderen verbunden werden muss, kann es problematisch sein, für jedes einen Übersetzer anzupassen. Hier bietet es sich an, ein offenes Protokoll von Diensten zu definieren, das Zugang zum Teilsystem ermöglicht. Dieser Kontext, als „Open Host Service“ bezeichnet, stellt also Schnittstellen bereit, die jeder nutzen kann. So kann jeder Kontext seine eigene Anbindung erstellen. [Eva07, S. 374] [Wol16, S. 48] [Dow18, S.66].

## Published Language

Das direkte Übersetzen von Modellsprachen kann zu sehr komplexen Strukturen führen. Selbst wenn eine Sprache für den Datenaustausch genutzt wird, ist sie einmal festgelegt und lässt sich nicht leicht an Entwicklungsbedürfnisse anpassen. Deshalb kann eine gut dokumentierte, gemeinsame Sprache als allgemeines Kommunikationsmittel bereitgestellt werden. Diese „Published Language“ stellt eine Modellierung der Domäne als universelle Sprache zwischen den Bounded Contexts dar [Eva07, S. 375-378] [Wol16, S. 48] [Dow18, S.65].

## Context Map

Eine „Context Map“ liegt an der Schnittstelle zwischen Projektmanagement und Softwareentwurf. Sie ist eine grafische Übersicht über bestehende Modelle, wobei der Fokus auf den Berührungs punkten liegt. Diese macht explizit die Kommunikationswege und gemeinsam genutzten Bereiche deutlich. Context Maps werden verwendet, um die Beziehungen verschiedener Bounded Contexts darzustellen [Eva07, S. 344 - 353] [Wol16, S. 45].

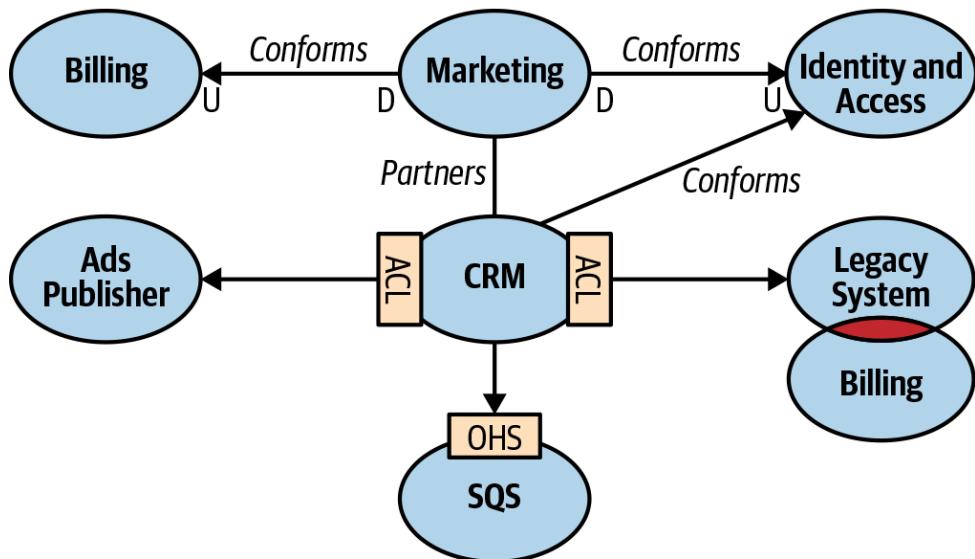


Abbildung 3.2: Darstellung von Bounded Contexts und ihre Rollen zueinander in einer Context Map [Kho]

## 3.2 Microservices

### 3.2.1 Geschichte und Definition

Im Mai 2011 wurde der Begriff „Microservice“ während eines Architekten-Workshops in Venedig eingeführt, um einen neu entstehenden Architekturstil zu beschreiben. Dieser Stil entwickelte sich aus der „Service-orientierten Architektur“ (Abk.: SOA). Ein Software-Service bezeichnet in der SOA eine Softwarekomponente, die über das Internet von entfernten Computern aus zugänglich ist. Microservice-Architekturen ähneln der SOA, aber SOA wird oft unterschiedlich verstanden und unterscheidet sich in der Regel von Microservices. Merkmale von SOA-Architekturen sind unter anderem ein zentraler Zugriffsbuss (ESB) oder die Abbildung ganzer Geschäftsabläufe in einem Service.

Viele SOA-Projekte stoßen wegen ihrer Komplexität oder hohen Kosten auf Schwierigkeiten. Aufgrund von Herausforderungen bei Skalierung und Verschiebung der Services entstand ein neuer Ansatz: Ein völlig unabhängiger Dienst mit eigener Datenbank, eigener Benutzeroberfläche und der Fähigkeit, einen Dienst zu ersetzen oder zu replizieren, ohne andere Systemdienste ändern zu müssen. Ein Jahr später wurde der Name Microservices als am besten passend für dieses Konzept gewählt. Der Microservice-Architekturstil wurde dann von Martin Fowler und James Lewis im März 2014 in einem Blogbeitrag erstmals ausführlicher vorgestellt [LF14] [RF21, S.251]. [Som20, S.150,151].

Microservices wurden nicht im Voraus geplant, sondern entstanden durch beobachtete Trends und Muster in der Praxis. Dazu gehören Continuous Delivery, bedarfsorientierte Virtualisierung, automatisierte Infrastrukturen, skalierbare Systeme und unabhängige Entwicklerteams. Die Hauptinspiration stammt jedoch aus dem DDD, insbesondere dem Konzept des Bounded Context [New15, S.21] [RF21, S.251].

Microservices können als kleine, leicht ersetzbare, unabhängige Komponenten definiert werden, die sich auf eng begrenzte Geschäftsvorgänge konzentrieren. Ihr Hauptziel ist eine hohe Entkopplung [RF21, S.252] [Fow17, S.5,6] [Som20, S.152] [Erl17, S.113] [New15, S.22-23].

### 3.2.2 Eigenschaften von Microservices

Verschiedene Eigenschaften zeichnen einen Microservice aus:

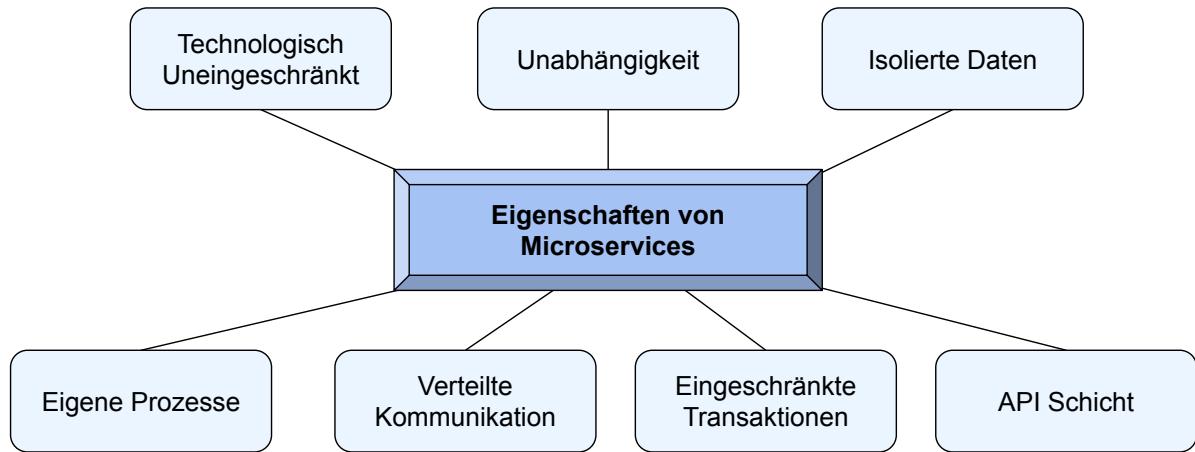


Abbildung 3.3: Die verschiedenen Eigenschaften von Microservices

#### Technologisch Uneingeschränkt

In ihrer Implementierung können Microservices unabhängig voneinander mit verschiedenen Technologien umgesetzt werden. Es existieren keine technologischen Einschränkungen [Wol16, S. 2, 31-37] [Som20, S.154-157] [New15, S.24].

#### Unabhängigkeit

Der Ansatz von Microservices zielt darauf ab, Services für eng begrenzte Geschäftsvorgänge zu erstellen, die unabhängig voneinander geändert und bereitgestellt werden können. Der Fokus liegt auf Ersetzbarkeit statt Wiederverwendung. Automatisierte Builds und Deployments vereinfachen dies. [Wol16, S. 2, 31-37] [New15, S.22,23,28] [Som20, S.154-157] [Dow18, S.128].

#### Isolierte Daten

Microservices, die auf dem Konzept des Bounded Contexts basieren, legen Wert auf Datensolierung. Sie vermeiden jegliche Formen der Kopplung, einschließlich gemeinsam genutzter Schemata und Datenbanken als Integrationspunkte, und verfügen über einen eigenen Datenhaushalt auf ihrer ausführenden Infrastruktureinheit [Wol16, S. 2, 31-37] [RF21, S.255].

## **Eigene Prozesse**

Ein Microservice ist wie ein eigenständiges Programm, das entweder als eigener Dienst auf einer Plattform oder als eigener Prozess auf einem Betriebssystem läuft. Jeder dieser Dienste arbeitet unabhängig und kann automatisch und einzeln bereitgestellt werden [New15, S.23] [RF21, S.253] [Wol16, S. 2, 31-37] [Som20, S.154-157].

## **Verteilte Netzwerkkommunikation**

Microservices repräsentieren eine verteilte Architektur und nutzen verteilte Netzwerkkommunikation für die Interaktion mit anderen Diensten. Die Integration der Services erfolgt hauptsächlich über REST oder Lightweight Messaging. Durch die Kommunikation über leichte Protokolle wird der Zusatzaufwand minimiert. Die Netzwerkkommunikation betont die Isolierung der Services und vermeidet die Gefahren einer engen Kopplung [RF21, S.253] [Wol16, S. 2, 31-37] [Som20, S.154-157] [Dow18, S.128] [New15, S.23].

## **Einschränkungen bei Transaktionen**

Aufgrund dieser Architektur, sollten Transaktionen nicht über mehrere Services hinweg durchgeführt werden. Transaktionen sind durch die ACID-Eigenschaften (Atomizität, Konsistenz, Isolation, Dauerhaftigkeit) gekennzeichnet [RF21, S.253] [Wol16, S. 2, 31-37].

## **API-Schicht**

In vielen Darstellungen von Microservices ist eine API(Application Programming Interface)-Schicht erkennbar, die zwischen den Nutzern und dem System liegt. Obwohl diese Schicht optional ist, wird sie häufig verwendet, da sie in der Architektur gut positioniert ist, um nützliche Aufgaben zu erfüllen. Sie sollte jedoch nicht als Vermittler oder Orchestrierungswerkzeug fungieren, da Microservices streng domänen spezifisch sind. Jeder Service stellt eine API bereit, über die andere Services mit ihm kommunizieren können. Es ist wichtig, dass durch die API keine Kopplung an die Nutzer entsteht, was bei der Technologieauswahl zu berücksichtigen ist [RF21, S.255,256] [New15, S.24].

### **3.2.3 Architektur von Microservice-Systemen**

Microservices sind kleine Dienste, die zusammengesetzt Anwendungen bilden. Sie unterscheiden sich von traditionellen, geschichteten Architekturen. Statt einer festgelegten Menge von Komponenten, setzen sie auf flexiblere, unabhängige Dienste. In der Softwareentwicklung gibt es den Trend, Systeme modular zu gestalten. Microservice-Architekturen tun dies, indem sie Software in Dienste statt in klassische Bibliotheken gliedern. Vorteile sind unter anderem die unabhängige Bereitstellung und klare Schnittstellen, die eine enge Verknüpfung vermindern [Som20, S.154-157] [LF14].

#### **DDD-Prinzipien**

Der Kerngedanke hinter Microservices ist der Bounded Context. Jeder Dienst repräsentiert eine Domäne oder einen bestimmten Arbeitsprozess. Ein Microservice sollte eine eigenständige fachliche Einheit darstellen, die so gestaltet ist, dass bei Änderungen oder neuen Funktionen nur dieser eine Microservice angepasst werden muss. Ein Bounded Context kann in mehrere Microservices unterteilt werden, falls das zweckmäßig ist. Zudem sollten die aus dem DDD bekannten Konzepte des Strategic Design berücksichtigt werden. Die Zuständigkeiten der Anwendungen müssen daher klar definiert und von anderen Anwendungen abgegrenzt sein [RF21, S.253,254] [Wol16, S. 48-51] [Das17, S.289].

#### **Kopplung und Kohäsion**

Kohäsion beschreibt, wie eng Teile eines Moduls zusammengehören sollten [RF21, S.40]. Kopplung hingegen zeigt, wie stark eine Komponente von einer anderen abhängt [RF21, S.44]. Das Ziel bei Microservices ist es, einen Dienst mit starker Kohäsion und schwacher Kopplung zu schaffen. Dies basiert auf dem „Single Responsibility Principle“ [Som20, S.154-157].

Innerhalb eines Dienstes sollten die Bestandteile eng zusammenarbeiten (hohe Kohäsion). Zwischen den Microservices sollte eine schwache Kopplung herrschen, sodass sie unabhängig und leicht änderbar sind. Wenn Dienste schwach gekoppelt sind, bedeutet das, dass eine Änderung in einem Dienst nicht automatisch Änderungen in anderen erfordert. Ein schwach gekoppelter Dienst kennt nur das Nötigste über andere Dienste. Deshalb ist es ratsam, die Kommunikation zwischen den Diensten zu beschränken, um Geschwindigkeitsprobleme und enge Kopplungen zu vermeiden. Zyklische Abhängigkeiten in der Architektur sind problematisch, da sie die Unabhängigkeit der Änderungen beeinträchtigen [Wol16, S. 101-105] [New15, S.56].

Das Ausrichten von Microservice-Schnittstellen auf spezifische Kontexte schafft optimale Voraussetzungen für schwache Kopplung und starke Kohäsion [New15, S.59,60].

## Ereignisgetriebene Architektur in Microservices

Um gemeinsame Logik zu implementieren, müssen Microservices miteinander kommunizieren. Dadurch werden größere Transaktionen in kleinere aufgeteilt, die zwar für sich genommen inkonsistent sein können, aber im Gesamtkontext für Konsistenz sorgen. Eine ereignisgetriebene Architektur (Event-Driven Architecture, Abk.: EDA) kann hierbei helfen. Bei dieser Architektur sendet ein Dienst, bei einem bestimmten Ereignis, eine Nachricht aus. Dieser Dienst, der „Event-Emitter“, informiert so alle beteiligten Microservices darüber, dass eine Aktion stattgefunden hat. Der eventbasierte Architekturstil arbeitet verteilt und asynchron. In dieser Architektur erfolgt die Kommunikation über asynchrone Nachrichten, die zuverlässig gesendet und empfangen werden. Die Komponenten sind entkoppelt und verarbeiten diese Nachrichten asynchron [Wol16, S. 137 - 138] [RF21, S.183] [Das17, S.301].

Für die Ereignisgetriebene Architektur existieren zwei grundlegende Topologien: Die „Broker-Topologie“ und die „Mediator-Topologie“.

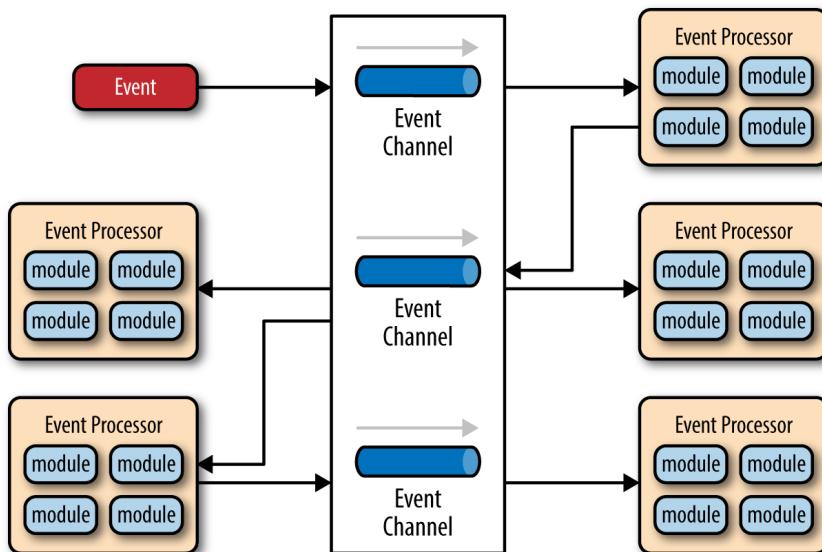


Abbildung 3.4: Eine Broker Topologie mit einem Message Broker und verschiedenen Event-Prozessoren [RF21]

In der Broker-Topologie verteilt ein „Message Broker“ Nachrichten über die Event-Prozessoren, ähnlich einem verketteten Broadcasting. Diese Topologie ist sinnvoll, wenn eine hohe Reaktionsfähigkeit und dynamische Kontrolle über die Eventverarbeitung gefordert sind. Sie ist besonders nützlich, wenn der Event-Verarbeitungsfluss einfach ist und keine zentrale Orchestrierung oder Koordination benötigt.

Die Broker-Topologie besteht aus vier Hauptkomponenten: Einem auslösenden Event, dem Event-Broker, einem Event-Prozessor und einem verarbeitenden Event.

Das auslösende Event startet den Event-Fluss und wird zur Verarbeitung an einen Kanal des Event-Brokers weitergeleitet. Ein einzelner Event-Prozessor übernimmt und verarbeitet das Event. Nach der Verarbeitung informiert er das System über ein sogenanntes Verarbeitungs-

Event über seine Aktionen. Dieses wird asynchron an den Event Broker gesendet, um eventuell weitere Verarbeitungsschritte zu initiieren. Andere Event-Prozessoren reagieren darauf, führen Aktionen aus und informieren das System erneut über ein Verarbeitungs-Event. Dieser Prozess wiederholt sich, bis kein weiteres Interesse an den Aktionen des letzten Event-Prozessors besteht.

Ein Event-Broker ist oft in Domänen unterteilt, wobei jeder Broker alle für eine Domäne notwendigen Kanäle enthält. In der Broker-Topologie wird ein entkoppeltes, asynchrones „fire and forget“ Broadcasting mit „Topics“ verwendet, die nach dem „Publish-Subscribe“-Modell funktionieren. Jeder Event-Prozessor teilt dabei dem System mit, was er getan hat, unabhängig davon, ob dies für andere Event-Prozessoren relevant ist oder nicht [RF21, S.184-186].

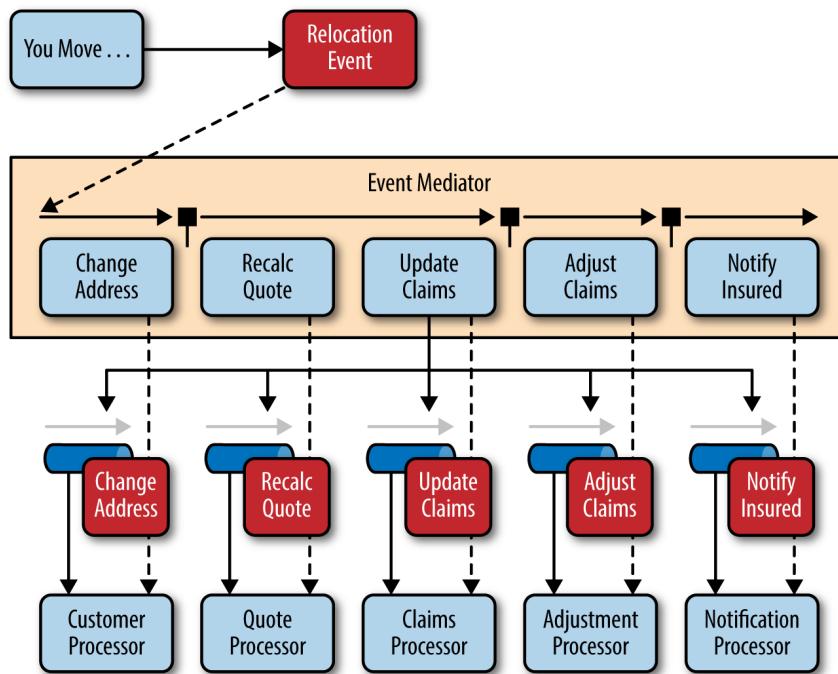


Abbildung 3.5: Eine Mediator Topologie mit einem Message Broker und verschiedenen Event-Prozessoren [RF21]

Die Mediator-Topologie verfügt über einen „Event-Mediatoren“, der den Arbeitsablauf und die beteiligten Event-Prozessoren für initiale Events steuert. Sie setzt sich zusammen aus einem initialen Event, einer „Event-Queue“, dem Event-Mediatoren, Event-Kanälen und Event-Prozessoren.

In dieser Topologie wird das initiale Event zuerst an die Event-Queue gesendet, die vom Event-Mediatoren verwaltet wird. Der Event-Mediatoren kennt nur die für die Event-Verarbeitung erforderlichen Schritte und erzeugt daher Verarbeitungs-Events, die zu den jeweiligen Event-Kanälen weitergeleitet werden. Die Event-Prozessoren hören auf bestimmten Event-Kanäle, bearbeiten das Event und informieren in der Regel den Event-Mediatoren nach Abschluss ihrer Aufgabe. Anders als bei einer Broker-Topologie teilen die Event-Prozessoren nicht mit, was sie verarbeitet haben. Häufig gibt es verschiedene Mediatoren, die jeweils für bestimmte Bereiche oder Event-Gruppen verantwortlich sind [RF21, S.189-190].

### **3.2.4 Kommunikation**

In einer Microservice-Architektur ist es möglich, die Kommunikation entweder synchron oder asynchron zu gestalten. Bei synchroner Kommunikation wartet der aufrufende Dienst auf eine Rückmeldung des aufgerufenen Dienstes. Diese Wartezeit blockiert den Aufrufer, bis die Transaktion abgeschlossen ist. Synchroner Datenaustausch gilt generell als weniger komplex im Vergleich zur asynchronen. Bei asynchroner Kommunikation hingegen muss der aufrufende Dienst nicht auf die Fertigstellung der Transaktion warten. Für den Aufrufer kann es unter Umständen unerheblich sein, ob die Anfrage vollständig bearbeitet wird oder nicht. Asynchrone Kommunikation ist oftmals effizienter, da die beteiligten Services nicht untätig während dem Warten sind. Zudem erleichtert die lose Kopplung bei ausschließlich asynchron kommunizierenden Services deren Modifikation [RF21, S.260,261] [Som20, S.161-163] [New15, S.70-71].

### **REST**

Der REST(Representational State Transfer)-Architekturstil ermöglicht die Übertragung von Ressourcen von einem Server zu einem Client. In diesem Zusammenhang bezeichnet eine Ressource ein Objekt, das dem Service bekannt ist. Die Art und Weise, wie eine Ressource nach außen präsentiert wird, ist unabhängig von ihrer internen Darstellung. Beziehungen zwischen Ressourcen können durch Links repräsentiert werden, was dem Prinzip von „HATEOAS“(Hypermedia as the Engine of Application State) entspricht. REST selbst macht keine spezifischen Angaben zum verwendeten Protokoll. Allerdings wird am häufigsten HTTP eingesetzt. Ein „RESTful-Service“ nutzt die im HTTP-Protokoll definierten Methoden [FR14] und arbeitet zustandslos. Zu den Operationen eines RESTful-Services gehören das Erstellen (umgesetzt mit „POST“), das Lesen (umgesetzt mit „GET“), das Aktualisieren (umgesetzt mit „PUT“) und das Löschen (umgesetzt mit „DELETE“) von Ressourcen. RESTful HTTP arbeitet synchron. Durch die Verwendung von Zeitüberschreitungen können Probleme wie Serverausfälle adressiert werden [New15, S.79-80] [Wol16, S. 182 - 183] [Som20, S.173-175].

### **Messaging**

Messaging-Systeme ermöglichen eine asynchrone Kommunikation zwischen Microservices. Ihr Hauptmerkmal ist das Versenden von Nachrichten, die sogar bei einem Netzwerkausfall übermittelt werden können. Dies wird durch das Zwischenspeichern der Nachrichten in den Messaging-Systemen erreicht. Bei Auftreten eines Fehlers sorgt die Möglichkeit, eine Übertragung erneut zu initiieren, für eine korrekte Verarbeitung. Ein wichtiger Vorteil ist, dass der Aufruf eines anderen Services die weitere Verarbeitung nicht blockiert, wodurch ein Microservice ohne Verzögerung auf eine Antwort weiterarbeiten kann. Im Messaging-System ist der Sender der Nachrichten typischerweise anonym, da diese auf einer Queue oder einem Topic landen, auf die sich interessierte Empfänger registrieren können. Apache Kafka ist eine Technologie mit der Messaging implementiert werden kann [Wol16, S. 183 - 187] [New15, S.87-88].

### **3.2.5 Betrieb**

In der Vergangenheit waren Betrieb und Entwicklung eines Systems klar voneinander getrennte Bereiche, die jeweils spezifische Aufgaben für Systemadministratoren und Entwickler mit sich brachten. Heute jedoch erfordern komplexe, aus vielen Microservices bestehende und über die Cloud verteilte Systeme eine konsistente und einheitliche Verwaltung. Diese verteilten Systeme basieren auf dem Einsatz von virtuellen Maschinen und Containern. Tools wie Kubernetes leisten wertvolle Dienste bei der Planung, der Verteilung von Arbeitslasten und im Management von Docker-Containern. Nachdem ein System entwickelt und in Betrieb genommen wurde, sind kontinuierliches Monitoring und regelmäßige Updates unerlässlich. Im Bereich der Softwaretechnik bezeichnet Monitoring das Überwachen von Metriken, um Zustand und Leistung einer Software zu analysieren und zu verstehen. Zu den wichtigsten Metriken zählen dabei Host- und Infrastrukturdaten wie CPU- und RAM-Auslastung, Thread-Anzahl, offene Dateideskriptoren und Datenbankverbindungen jedes einzelnen Microservices. Aufgrund der dynamischen Natur von Microservices, deren Instanzen jederzeit gestartet oder beendet werden können, ist es entscheidend, dass das Monitoring sowohl systemübergreifend als auch unabhängig vom jeweiligen Dienst funktioniert. Technologien wie Graphite, Grafana oder Nagios sind hierfür geeignete Werkzeuge. Auch Cloud-Umgebungen bieten spezielle Tools, um ein leistungsfähiges und effizientes Monitoring zu gewährleisten [Som20, S.158,179,181] [RF21, S.253] [Wol16, S. 80-81, S.250-256, S.263-269] [Fow17, S.105-108].

### **3.2.6 Microservices mit Spring Boot**

Microservices lassen sich effektiv mit dem Java-Framework Spring Boot realisieren, welches auf dem Spring Framework basiert. Zur Verwaltung von Abhängigkeiten in einem Spring Boot-Projekt werden Build-Management-Tools wie Maven oder Gradle verwendet. So werden bei Gradle in der build.gradle-Datei sämtliche Abhängigkeiten sowie spezifische Besonderheiten für die Kompilierung definiert. Innerhalb von Gradle werden verschiedene Abschnitte des Build-Prozesses, wie das Kompilieren oder Testen, als Tasks bezeichnet.

Ein Schlüsselkonzept des Spring Frameworks ist die Dependency Injection. Hierbei instanziieren oder referenzieren Objekte ihre erforderlichen Kollaboratoren nicht selbst. Stattdessen stellt das Framework diese von außen bereit. Um auf diese Weise instanziert werden zu können, müssen Klassen als Komponenten gekennzeichnet sein. Das Spring Framework bietet zudem diverse Module für zusätzliche Funktionalitäten. So ermöglicht das Spring MVC-Modul durch Annotationen die Zuordnung von URLs zu Methoden, was besonders bei der Implementierung von REST-Schnittstellen nützlich ist. Spring Data JPA erleichtert das Mapping von Java-Klassen auf Datenbankobjekte. Das Spring Webflux-Modul stellt mit dem „WebClient“ eine Implementierung für das Stellen von HTTP-Anfragen bereit [Sim18, S.30-39,S.54-58, S.130-139, S.220-227, S.279-281]. Die breite aller Funktionalitäten von Spring und Spring Boot lässt sich detailliert in der offiziellen Dokumentation einsehen [VMwb] [VMwa].

## 3.3 Cloud-Computing

Cloud-Computing bedeutet, Rechnerressourcen über ein Netzwerk zu nutzen. Mithilfe von Virtualisierung lassen sich die Ressourcen eines von Cloud-Anbietern betriebenen Rechenzentrums abgrenzen. Das National Institute of Standards and Technology (NIST) beschreibt Cloud-Computing als ein Modell für einen allgegenwärtigen, bequemen und bedarfsgerechten Netzwerkzugang zu einem Pool von konfigurierbaren Rechenressourcen, die schnell und mit minimalem Verwaltungsaufwand oder ohne große Interaktion mit dem Dienstanbieter bereitgestellt und freigegeben werden können [Ste20, S. 20] [Rit18, S.19-22] [Nat].

### 3.3.1 Docker

Softwarecontainer sind eine Methode, Anwendungen in standardisierte und portable Einheiten zu verpacken, die autonom funktionieren. Das Besondere an diesen Umgebungen ist ihre Isolation vom restlichen System, wobei sie als abgegrenzte Prozesskapseln innerhalb definierter Grenzen operieren. Diese Isolation ermöglicht den Betrieb unterschiedlicher Anwendungen und sogar verschiedener Komponenten einer Anwendung in separaten Containern, die in der Cloud individuell skaliert werden können. Dieses Prinzip findet im Bereich der Microservices sowohl in der Entwicklung als auch in der Applikationsarchitektur intensive Anwendung.

Die Containerisierung kann als eine Evolution der Virtualisierung betrachtet werden, bei der ein Betriebssystem auf einem Hostrechner simuliert wird. Es existieren hauptsächlich zwei Arten von Containern: System-Container, die virtuelle Maschinen nachahmen und oft einen vollständigen Boot-Prozess durchlaufen, sowie Anwendungscontainer, die in der Regel nur ein einzelnes Programm ausführen. Docker, als die wohl bekannteste Container-Software, bietet eine Virtualisierung auf Betriebssystemebene, die mit Containerisierung vergleichbar ist. Diese Technik der Isolation ermöglicht es, mehrere Betriebssysteme innerhalb eines anderen Betriebssystems zu betreiben [Ste20, S. 54-59] [BBHE23, S.18] [Rit18, S.63].

„Images“ bilden das Kernstück der Docker-Architektur und sind essentiell für das Starten von Containern. Man kann eigene Images erzeugen, indem man von einem Basismodell ausgeht. Jedes Container-Image stellt ein Binärpaket dar, welches sämtliche für die Ausführung eines Programms in einem Betriebssystemcontainer erforderlichen Dateien beinhaltet. Das Docker-Image-Format hat sich mittlerweile als Industriestandard durchgesetzt. Die Erstellung eines Docker-Container-Images lässt sich durch ein Dockerfile automatisieren. In der Praxis ist ein Container ein in Betrieb genommenes Image, das je nach Konfiguration und Aufbau des Images mehrere Prozesse beherbergen kann. Durch die Kombination mehrerer Container können komplexe Anwendungen realisiert werden. Als „Registry“ bezeichnet man den Speicherort für Images, wobei zwischen privaten und öffentlichen Registries unterschieden wird [BBHE23, S.16-18] [Rit18, S.64-65].

### 3.3.2 Kubernetes

Für die Orchestrierung einer komplexen, containerbasierten Anwendung in einem „Cluster“ ist ein Cluster-Manager notwendig. Ein Cluster besteht aus einer Gruppe von Servern, die Container enthalten und als Knoten bezeichnet werden. Beispiele für Cluster-Manager sind Docker Swarm oder Kubernetes, wobei letzteres nachfolgend detaillierter betrachtet wird:

Kubernetes, ein von Google entwickeltes und von der Cloud Native Computing Foundation verwaltetes Open-Source-System, dient dem Deployment containerbasierter Anwendungen. Es ermöglicht die Verwaltung und Skalierung solcher Anwendungen innerhalb eines Clusters und hat sich als moderner Standard für den verteilten Betrieb von Applikationen etabliert, insbesondere in bereitgestellten Cloud-Umgebungen. Dabei verwendet Kubernetes standardmäßig Docker als Container-Engine. Über eine eigene API, die vor allem in der Entwicklung spezifischer Werkzeuge Anwendung findet, bietet Kubernetes umfassende Steuerungsmöglichkeiten. Ein Kubernetes-Cluster setzt sich aus Knoten (Nodes) zusammen, die in einen Master-Knoten und beliebig viele Worker-Knoten untergliedert sind. Ein Cluster kann dabei ausschließlich aus einem Master-Knoten bestehen, wobei jeder Knoten je nach Konfiguration als Master oder Worker fungieren kann. Kubernetes basiert auf dem Prinzip deklarativer Konfigurationsobjekte, die den angestrebten Zustand des Systems beschreiben. Der Ansatz, deklarative Konfigurationen unter Versionskontrolle zu halten, ist auch als „Infrastructure as Code“ bekannt. Mit dem Werkzeug kubectl interagieren Benutzer über die auf dem Master-Knoten laufende API mit dem Cluster. Der Master-Knoten steuert die Worker-Knoten und setzt die vom Benutzer gewünschten Änderungen um [BBHE23, S.1,4] [BGO<sup>+</sup>16] [Rit18, S.79,81] [Ste20, S. 60-63,166].

Kubernetes definiert eine Reihe grundlegender Objekte, die zur Erstellung und Verwaltung von Ressourcen dienen. Diese Komponenten sind flexibel konzipiert, sodass sie miteinander kombiniert werden können. Innerhalb von Kubernetes repräsentieren Objekte verschiedene Elemente, die im Cluster genutzt werden. Jedes dieser Objekte verfügt über einen einzigartigen HTTP-Pfad, da sie in Kubernetes alle als REST-Ressourcen abgebildet werden [BBHE23, S.40, 42] [Rit18, S.80-81] [Ste20, S. 166-192].

## Deployment

Ein Deployment ist ein Objekt, welches etwas im Cluster installiert und es anschließend dort ausführt. Es wird genutzt, um zuverlässig neue Softwareversionen zu releases. Es kann als deklaratives YAML-Objekt beschrieben werden, in dem Details wie das verwendete Image näher spezifiziert werden [Ste20, S. 175] [BBHE23, S.117-127].

## Pods

Ein Pod ist eine Sammlung von Anwendungscontainern. Diese operieren in einer gemeinsamen Ausführungsumgebung innerhalb eines Kubernetes-Clusters. Er stellt das kleinste deploybare

Bauelement in diesem Cluster dar. Pods lassen sich replizieren, um die Skalierung der in ihnen enthaltenen Applikationen zu ermöglichen. Ein typischer Pod umfasst eine Gruppe von Docker-Containern, die alle dieselbe IP-Adresse nutzen. Dieser kann aus einem oder mehreren Containern bestehen, die auf dem gleichen Host laufen und gemeinsam Ressourcen nutzen. Jeder Pod verfügt über eine einzigartige IP-Adresse innerhalb des Clusters und kann manuell, über die API oder mittels eines Controllers verwaltet werden. Anwendungen in einem Pod teilen sich nicht nur dieselbe IP-Adresse und denselben Port-Bereich, sondern auch den gleichen Hostnamen. Zudem können sie über native Interprozesskommunikationskanäle miteinander interagieren [BBHE23, S.48,49][Rit18, S.80-81] [Ste20, S. 176].

## Service

Um Pods auffindbar zu machen, ist der Einsatz von Service-Discovery erforderlich. Dies lässt sich mit dem Kubernetes-Service-Objekt umsetzen. In Kubernetes repräsentiert ein Service eine Gruppe von Pods, die gemeinsam funktionieren. Die Arbeitsweise eines solchen Services ist vergleichbar mit der eines DNS. Mithilfe eines Service-Objekts können zudem Pods für den Zugriff von außerhalb des Clusters freigegeben werden. Dafür lässt sich ein „NodePort“ aus dem Bereich 30000 bis 32767 einem Pod-Port zuordnen [BBHE23, S.77-80] [Rit18, S.80-81] [Ste20, S. 177-178].

## Replica Set

Ein Replica Set überwacht über ein definiertes Label designierte Pods daraufhin, dass immer die gewünschte Anzahl davon im Cluster vorhanden ist. Das Replica Set stellt die gewünschte Anzahl der Pods automatisch sicher [Ste20, S.179].

### 3.3.3 Google Cloud Plattform

Die Google Cloud Platform (Abk.: GCP) ist eine von Google angebotene Cloud-Umgebung. Diese bietet verschiedenste Services für Nutzer an. So ermöglicht die Google Compute Engine beispielsweise das Erzeugen von virtuellen Maschinen in der Cloud. Die Google Kubernetes Engine wiederum ist ein Orchestrierungsmanager für Kubernetes-Cluster. Die Artifact Registry ist ein sich in der GCP befindendes Docker-Repository. Die GCP Dokumentation bietet weiterhin einen breiten Überblick über alle Produkte an [Goob].

Einstiegspunkt zur Verwendung der GCP ist das Google Cloud Projekt. Dieses besitzt einen Namen sowie eine global einzigartige Projekt-ID. Alternativ zur Verwendung des UI kann mit der Google Cloud SDK die Cloud effektiv in der Konsole verwaltet werden [Rit18, S.23-35].

## 3.4 Modellgetriebene Entwicklung von Microservices

### 3.4.1 Verfahren

Bei der modellgetriebenen Entwicklung von Microservice-Anwendungen ist es möglich, Verfahren zu definieren, welche eine Modellierungsmethodik vorgeben. Das allgemein für serviceorientierte Architekturen ausgelegte Verfahren „Model Driven SOA“, eine Variante der MDD-Modellierungsmethodik M<sup>3</sup>, ist ein Beispiel dafür. In diesem werden iterativ verschiedene Phasen wiederholt:

In einer Initiationsphase wird die Fachlichkeit aufgenommen, deren Anforderungen identifiziert und fachliche Services werden ermittelt. Anschließend folgt die Systemevaluierungsphase, in der die technischen Services spezifiziert werden, zusammen mit dem Fachdatenmodell und den Datenflüssen. In einer Architekturprojektionsphase wird dann die gewählte Architektur abgebildet und die in der Initiations- und Evaluierungsphase spezifizierten Inhalte werden auf konkrete Elemente der Architektur übertragen. Schließlich umfasst eine Softwarekonstruktionsphase die Generierung von Artefakten und die Integration aller Ergebnisse zu einer lauffähigen Anwendung auf einer spezifischen Plattform [rem11, S.62-66].

Ein Verfahren welches im Zusammenhang der Modellierungstechnologie VxBPMN4MS verwendet wird, besteht aus drei Phasen: Modellierung, Servicezusammensetzung und -bereitstellung sowie Ausführung.

In der Modellierungsphase werden Geschäftsprozesse mit der Technologie erstellt. Dabei entstehen variable Modelle und Konfigurationsdateien. Diese Modelle helfen dabei, Microservices zu koordinieren, um spezifische Geschäftsprozesse zu entwickeln. Ähnliche Prozesse werden zusammengefasst, um Variationspunkte und Varianten zu identifizieren. In der Servicezusammensetzungs- und -bereitstellungsphase werden Microservice-Frameworks entwickelt, die in die Infrastruktur integriert werden. Die erstellten Modelle werden in diese Frameworks eingefügt, um die Geschäftslogik und spezifische Prozessvarianten umzusetzen. In der Ausführungsphase wird die Microservice-Architektur aktiviert. Prozessinstanzen werden in Echtzeit basierend auf den Konfigurationsdateien erstellt. Wenn im Prozess ein Variationspunkt erreicht wird, werden die entsprechenden Varianten dynamisch ausgewählt und der Prozess setzt sich fort, bis er vollständig abgeschlossen ist. [SWLH21].

### 3.4.2 Werkzeuge

Verschiedene wissenschaftliche Artikel und Arbeiten thematisieren Aspekte der modellgetriebenen Microservice-Entwicklung und stellen dabei Werkzeuge mit verschiedenen Funktionalitäten vor. Im Folgenden werden einige aufgelistet und ihre jeweiligen Kernfunktionen aufgezählt.

Werkzeug	Funktionalität
AjiL [RSS18]	<ul style="list-style-type: none"><li>• Eclipse-basierte grafische Modellierung und Bearbeitung für Microservices und Domänenmodelle</li><li>• Unterstützung für Schnittstellenmanagement, von Entwurf bis hin zu Zwischenmodellen</li><li>• Generierung von Java-Code, Service-Stubs und Konfigurationen, basierend auf Spring Boot und Spring Cloud</li></ul>
Context Mapper [KZ20]	<ul style="list-style-type: none"><li>• Darstellung von strategischen Domain-Driven Design Mustern durch eine DSL</li><li>• Bearbeiten, Validieren und Transformieren von strategischen DDD-Mustern</li><li>• Erstellen von grafischen Context Maps und Microservice-Beschreibungen, was das Design und die Integration von Diensten unterstützt</li></ul>
OCCI [ZCM19]	<ul style="list-style-type: none"><li>• Implementierung offener Standards für das durchgängige Management von Cloud-Ressourcen über alle Service-Modelle (IaaS, PaaS, SaaS) hinweg</li><li>• OCCIware Studio und Metamodell als Kernstücke für eine MDA, die Entwurf, Entwicklung und Verwaltung von Cloud-Diensten ermöglichen</li></ul>
ARGON [SIA19]	<ul style="list-style-type: none"><li>• Modellgesteuerten Ansatz für die Bereitstellung von Infrastruktur, was die Automatisierung und Vereinfachung des Prozesses ermöglicht</li><li>• Nahtlose Integration in eine Vielzahl von Cloud-Plattformen</li><li>• Unterstützung für die Automatisierung der Ressourcenzuweisung</li><li>• Vereinfacht das Management komplexer Bereitstellungen in Cloud-Umgebungen</li></ul>

Tabelle 3.1: Werkzeuge für MDSD von Microservices

# 4 Konzeption des Metamodells

## 4.1 Konzeptionelle Rahmenbedingungen

### 4.1.1 Einleitende Überlegungen

Folgende Voraussetzungen werden für diese modellgetriebene Entwicklung festgelegt: Das zu entwerfende Metamodell soll die Konzepte von Microservice-Architekturen aufgreifen. Dabei soll die noch zu definierende Auswahl der Technologien, die für eine Umsetzung benötigt werden, berücksichtigt werden. Auch die Möglichkeit, andere Technologien verwenden zu können, soll bedacht werden. Weiterhin wird davon ausgegangen, dass zusätzliche Software zum Betrieb einer Microservice-Architektur vorhanden ist. Kandidaten für solche Software wären die Cloud-Umgebung, die Datenbanktechnologie oder Software Development Kits. Die vorausgesetzte Software muss daher definiert und berücksichtigt werden. Ebenso wird die Einbindung von grafischen Oberflächen für die entwickelten Microservices in der Konzeption nicht betrachtet. Zu konzipieren ist hingegen eine Umsetzung durch APIs, welche ein von dieser Modellierung unabhängiges Frontend beliefern könnten.

Methodisch soll zu Beginn der State of the Art erfasst und in einen Kontext zur Zielstellung dieser Arbeit gesetzt werden. Dabei sollen Möglichkeiten zum Einbezug bestehender Forschungsergebnisse geprüft werden. Die Abstraktion soll in ihrem Vorgehen auf den theoretischen Grundlagen von Microservice-Architekturen beruhen. Abweichungen von diesen sollen nach Möglichkeit vermieden werden. Die zu entwerfende abstrakte Syntax soll iterativ im Kontext der Darstellung und Generierung verbessert werden. Grund dafür ist die frühe Erkennung von konzeptionellen Fehlern und Problemen. Diese sollen dabei gelöst werden und bei der Erklärung des schließlich entstehenden Metamodells angesprochen werden. Auf eine Visualisierung dieser Zwischenlösungen wird verzichtet. Der anwendungsorientierte Teil der Entwicklung soll einerseits lokal und in einer Cloud-Umgebung durchgeführt werden.

Für die Entscheidungsfindung und die abschließende Bewertung der Konzeption werden folgende Kriterien festgelegt: Das Hauptkriterium ist der Mehrwert, den die Generierung von Anwendungen für Cloud-Umgebungen bietet, insbesondere in Bezug auf die Lösung realer Probleme. Zusätzlich soll die Entwicklung kritisch auf die Einbindung von Eigenschaften und Konzepten von Microservices geprüft werden. Es gilt bei dem Entwurf der abstrakten Syntax auch, die Erweiterbarkeit dieser zu ermöglichen. Weitere Kriterien sind das mögliche Potential für Code-Wiederverwendung, die Möglichkeit, durch Abstraktion und Visualisierung Komplexität greifbarer zu machen, und die Nutzung im Rahmen von Refaktorisierungen.

## 4.1.2 Einbindung aktueller Technologien

### AjiL

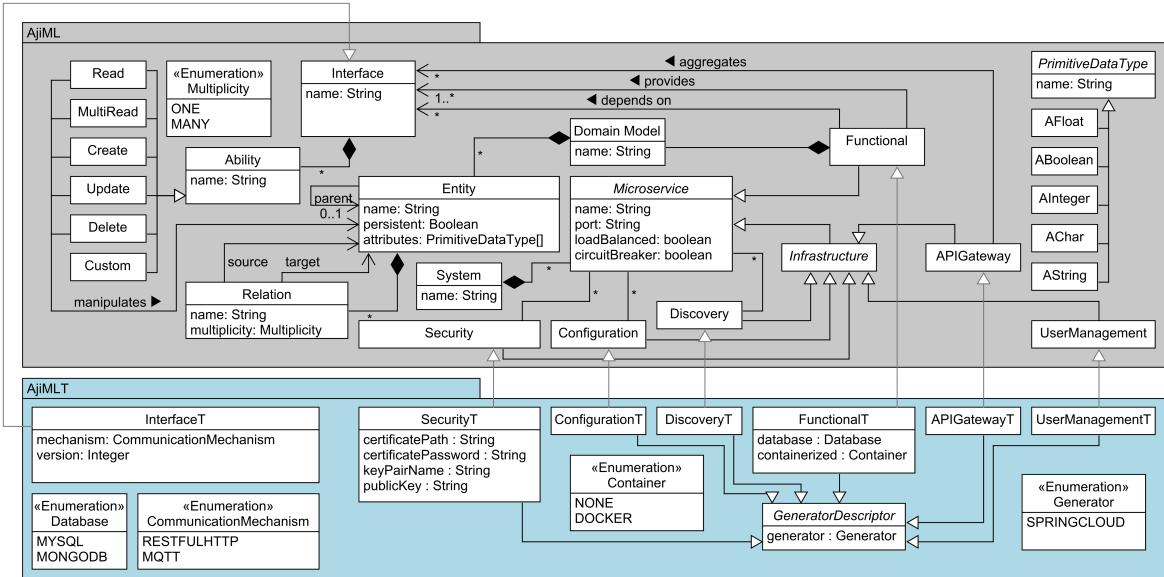


Abbildung 4.1: Das AjiL Metamodell [RSS]

AjiL bildet in seinem Metamodell verschiedene Aspekte von Microservice-Architekturen ab: So existieren Domain Models und Entities, welche Fachlichkeiten abbilden. Klassen wie Microservice, Security, Configuration oder Discovery bilden die technischen Metaklassen ab. Ebenso existieren Infrastruktur-Klassen wie Database oder Container. Das AjiMLT-Paket des Metamodells bildet konkrete Typen ab. So existiert beispielsweise ein Enum, welcher die Kommunikation einer Schnittstelle spezifiziert, konkret REST oder das Messaging-Protokoll MQTT.

Das AjiL-Metamodell bildet einen recht breiten Bereich der Microservice-Entwicklung ab, jedoch gibt es auch einige fehlende Abstraktionen, wie solche, die das Domain-Driven Design abbilden oder konkretere Cloud-Architekturen. AjiL setzt vieles davon implizit in den Generierungstemplates um, ohne explizite Modellklassen. Dadurch entsteht eine hohe Inflexibilität durch starre Teile des Templates.

Auch wenn AjiL einige interessante Ansätze bietet, auf denen man aufbauen kann, wie dessen Ansatz zum Schnittstellenmanagement und den ähnlichen Anforderungen an die zu generierenden Anwendungen, ist AjiL keine hinreichende Lösung, um DDD abzubilden und deploybare Anwendungen zu generieren.

## Context Mapper

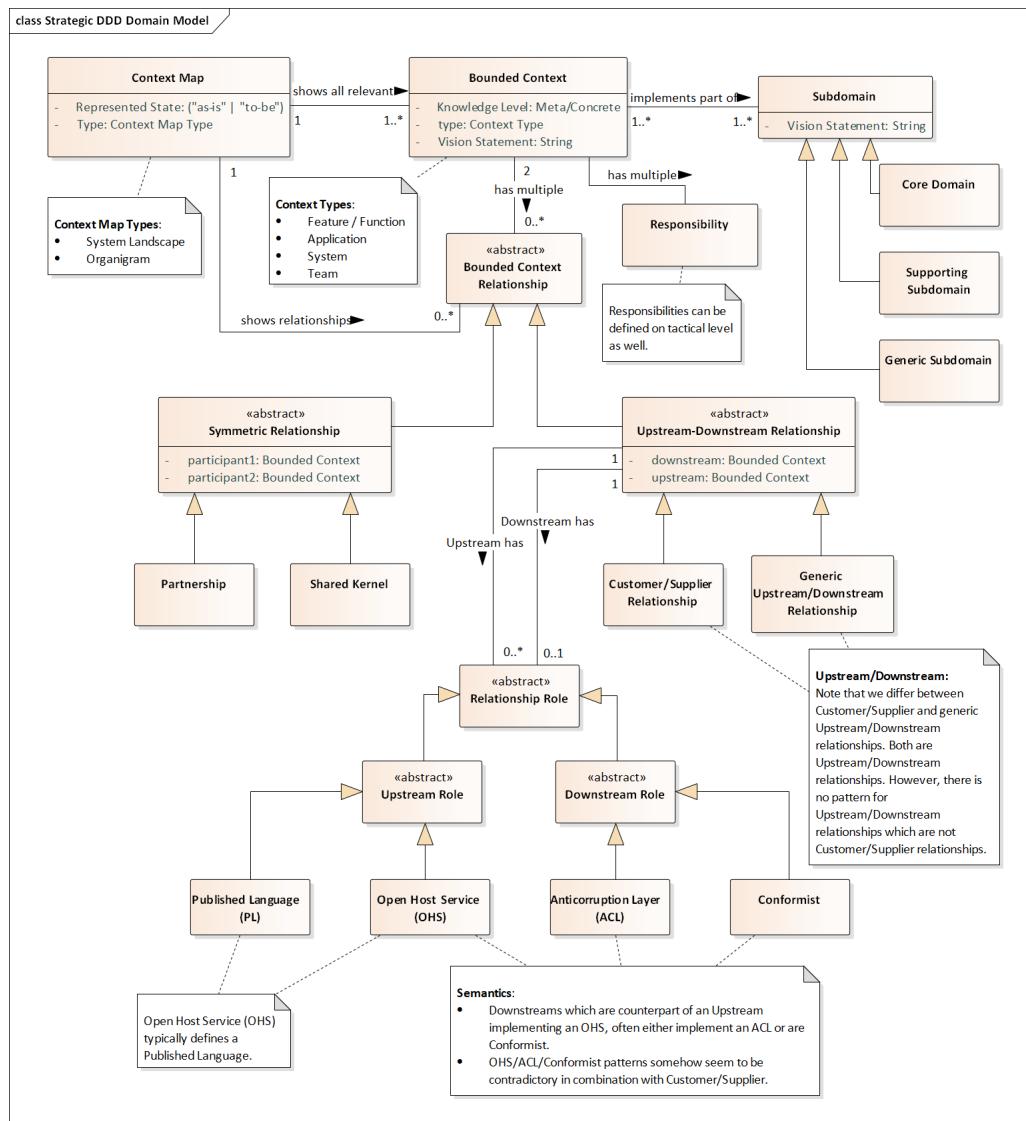


Abbildung 4.2: Das Context Mapper Metamodell [KZ20, S.3]

Context Mapper definiert ein sehr umfassendes Metamodell, welches die Konzepte des DDD in höchster Ausführlichkeit modelliert. Context Mapper nutzt dieses, um Context Maps von Domänen zu generieren. Dementsprechend ist die Technologie keine Lösung, um Microservice-Anwendungen zu generieren. Jedoch ist die Modellierung eine umfassende Grundlage für eine Metamodellschicht, die das Strategic Design des DDD abbildet. Eine exakte Wiederverwendung bietet sich aus Gründen der Übersichtlichkeit nicht an, da Context Mapper so umfassend modelliert, dass eine auf generierende Anwendungen fokussierte Modellierung auch mit weniger Metamodell-Klassen auskommen kann. So ist beispielsweise die explizite Modellierung von Responsibilities oder Subdomains für zu generierende Microservices nicht notwendig. Aspekte der Visualisierung hingegen eignen sich sehr gut, um in einem dem Ziel dieser Arbeit angepassten Rahmen aufgegriffen zu werden.

## OCCI

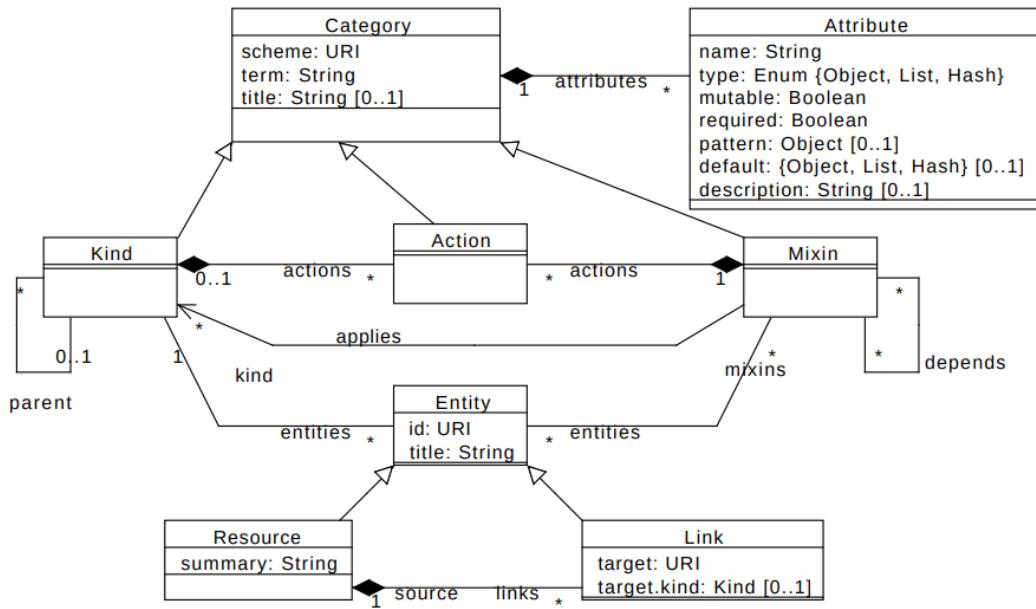


Abbildung 4.3: Das OCCI Metamodell [EPM16, S.5]

Das Metamodell des Open Cloud Computing Interface (OCCI) stellt eine kompakte Abstraktion dar, um Cloud-Ressourcen und -Dienste effektiv zu modellieren. Es vermeidet eine direkte Abstraktion von spezifischen Konzepten wie Container oder Deployments. Stattdessen verwendet OCCI allgemeinere Klassen wie Entity, Resource und Link, um eine breite Palette von Cloud-Konzepten und Beziehungen darzustellen.

Beispielsweise kann eine Entity in diesem Modell einen konkreten Cloud-Service oder eine virtuelle Maschine repräsentieren, während Resource für spezifische Dienste wie Speicherplatz oder Rechenkapazitäten steht. Link kann genutzt werden, um die Verbindungen zwischen verschiedenen Ressourcen oder Diensten abzubilden, wie etwa die Zuordnung eines Netzwerkdienstes zu einem bestimmten Container. Dieser abstrakte Ansatz ermöglicht es, das Modell flexibel und anpassbar für eine Vielzahl von spezifischen Cloud-Umgebungen und Anwendungsfällen zu gestalten.

Trotz der Fähigkeit der Technologie, Cloud-Infrastrukturen umfassend zu modellieren und zu generieren, ist sie für die Anforderungen dieser Arbeit aufgrund ihrer komplexen Handhabung und der Fokussierung auf Infrastruktur anstatt auf Anwendungssoftware weniger geeignet.

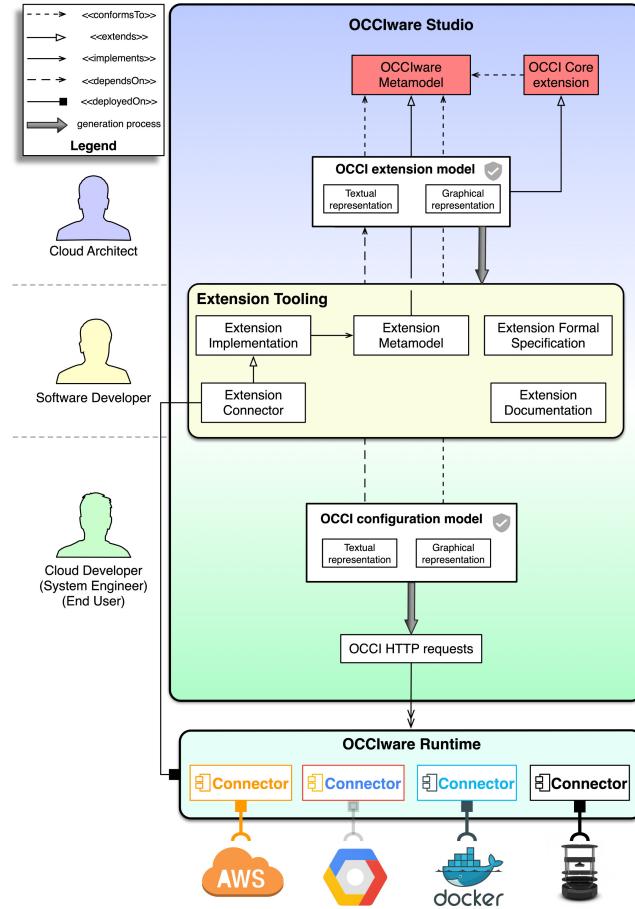


Abbildung 4.4: Das OCCI Metamodell als Teil der komplexen OCCIware Anwendungsumgebung [ZCM19]

## ARGON

Die Technologie ARGON strukturiert den Ansatz, Cloud-Infrastrukturen modellbasiert zu entwickeln, in mehrere Ebenen. Im Gegensatz zu OCCCI, das ein umfangreiches Anwendungs-framework bietet, konzentriert sich ARGON auf die Definition eines plattformunabhängigen Metamodells (PIM) sowie spezifischer Metamodelle (PSM) für AWS und Microsoft Azure. Mittels M2M-Transformationen ist eine gegenseitige Umwandlung möglich. Zudem sind M2T-Transformationen vorgesehen, um Infrastructure-as-Code mittels Ansible oder Terraform zu erzeugen.

In diesem Kontext erweist sich ARGON als deutlich geeigneter als OCCCI, da es sich auf die praxisnahe Erstellung von Software-Komponenten konzentriert, die durch ein konkreteres Metamodell definiert werden, anstatt auf einen abstrakten, plattformübergreifenden Cloud-Baukasten. Auch der hierarchische Ansatz und das in Relation setzen von Cloud-spezifischen Modellen sind sinnvoll aufgreifbare Konzepte. Die Problematik der fehlenden Anwendungsebene liegt jedoch weiterhin vor. Die Entwicklung einer Schnittstelle zu ARGON wäre eine Möglichkeit, diese zu lösen.

Stattdessen soll im Rahmen dieser Arbeit, gemäß dem Fokus auf das lauffähige Deployment von Microservice-Architekturen, eine Lösung entwickelt werden, die Anwendungsmodellierung und Deployment für eine konkrete Zielplattform in einem Metamodell vereint. Dabei kann der ARGON-Ansatz plattformunabhängige Modellelementen und spezifischer Modellelemente zu modellieren integriert werden.

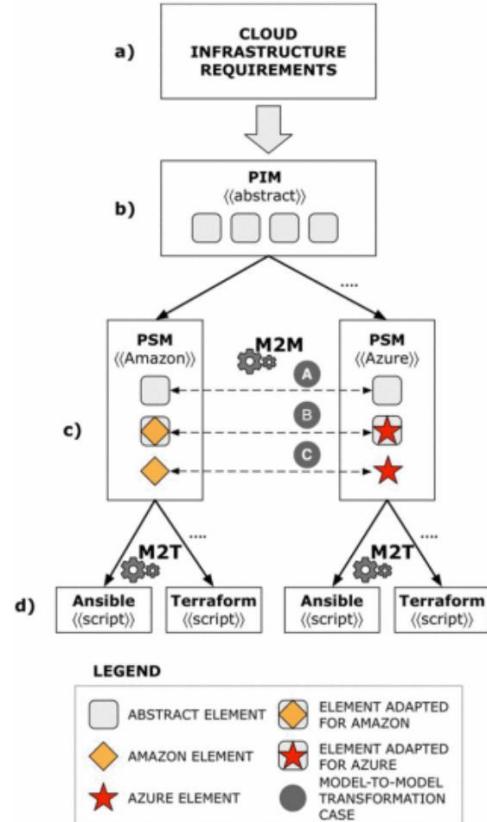


Abbildung 4.5: Architekturkonzept von ARGON [SIA19]

#### 4.1.3 Auswahl der Technologie

Die Umsetzung des Metamodells erfolgt mit dem Eclipse Modeling Framework (Abk.: EMF), einem quelloffenen Java-Framework. Weiterhin kommen die ebenfalls quelloffenen Frameworks Eclipse Sirius, das die Visualisierung von Modellen ermöglicht, und Acceleo, das aus EMF-Modellen Code generieren kann, zum Einsatz. Diese stehen Eclipse als Plugins zur Verfügung. Die generierten Anwendungen sollen die Programmiersprache Java verwenden. Zudem wird das Spring Framework eingesetzt, welches die Implementierung verschiedenster Strukturen vereinfacht. Als Technologie zur Umsetzung von eventgetriebener Architektur wird Apache Kafka verwendet. Die generierten Applikationen sollen mittels Gradle gebaut und als Docker Image ausgeführt werden können. Als Cloud-Infrastruktur wird die Google Cloud gewählt, in welcher die Anwendungen auf einem Kubernetes-Cluster laufen sollen. Mit der Unix-Shell Bash werden dazu nötige Skripte umgesetzt. Als Datenbank Technologie wird die in-memory Datenbank H2 verwendet.

## 4.2 Entwicklung der eigenen DSL

### Grundkonzept - Definition verschiedener Modellschichten

Um ein Metamodell für Microservice-Architekturen zu konzipieren, wird zunächst ein plausibles, konkretes Modell betrachtet. Basierend auf einer Analyse dieses Modells soll beginnend ein grundlegender Ansatz gewählt werden, der im Kontext einer Abstraktion sinnvoll erscheint.

Hierzu wird ein Beispielmodell (Abbildung 4.6) eingeführt, das eine Domäne umfasst. Diese besteht aus den Kontexten Customer und Marketing, die eine Customer/Supplier-Beziehung haben. Weiterhin gibt es Microservices zu diesen Kontexten, die über eine Schnittstelle miteinander kommunizieren. Die Ausführungsumgebung wird mit spezifischer Servicekonfiguration und einem Modellelement, das die Cloud-Umgebung beschreibt – hier die Google Cloud-Infrastruktur – ebenfalls modelliert.

Bei der Betrachtung wird folgendes festgestellt: Die existierenden Modellelemente lassen sich in fachliche Elemente, technische Elemente oder Infrastrukturelemente gruppieren. Diese Gruppierung wird als gemeinsame Schicht definiert. Innerhalb ihrer Schicht haben die Elemente Beziehungen zu anderen Modellelementen derselben Schicht. Weiterhin können Beziehungen existieren, die über diese Schichten hinausgehen, das heißt, es gibt Abbildungen von Elementen einer Schicht auf Elemente anderer Schichten.

Da durch Schichten die mögliche Komplexität besser gegliedert und somit einfacher handhabbar gemacht wird, wird das Metamodell ebenfalls schichtbasiert konzipiert. Neben der Zuordnung der Konzepte in die passende Schicht muss dabei erforscht werden, wie die Beziehungen innerhalb und zwischen den Schichten umgesetzt werden können. Nach abschließender Betrachtung aller Schichten muss das Metamodell im Kontext der Konzeption von Codegenerierung und Migrationsfähigkeiten weiterhin iterativ überprüft und gegebenenfalls nachgebessert werden. Hierbei werden die Bewertungskriterien aus den einleitenden Überlegungen berücksichtigt.

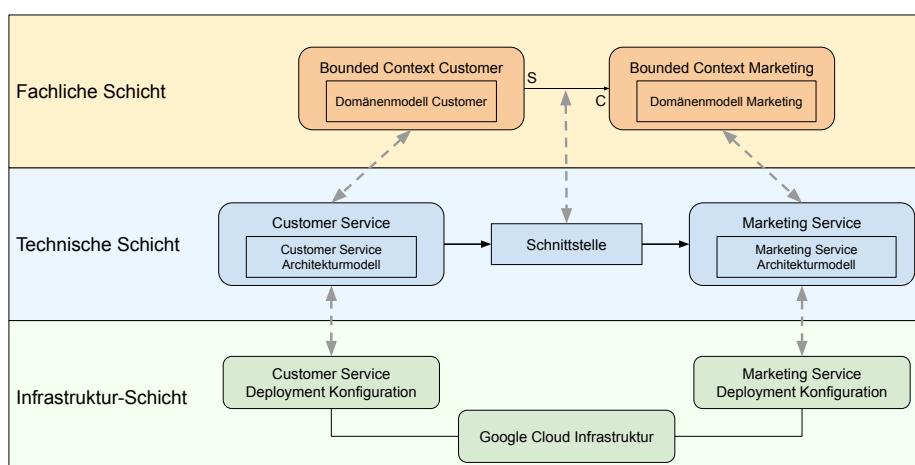


Abbildung 4.6: Erkennbare Schichten in einem Modell einer Microservice-Architektur

#### 4.2.1 Modellierung des Model-Driven Designs

Beginnend wird das DDD in das Metamodell integriert. Dieses findet sich in einer Klasse für die fachlichen Schicht, der „DomainModelLayer“, wieder. Die Schicht besteht aus drei Abstraktionsebenen: dem Domänenmodell, dem MDD und dem Strategic Design. Im Folgenden werden diese farblich unterschieden und die detailliertere Konzeption ausgeführt. Zur besseren Übersicht werden dabei bereits eingeführte Klassen und Relationen nach Möglichkeit ausgeblendet.

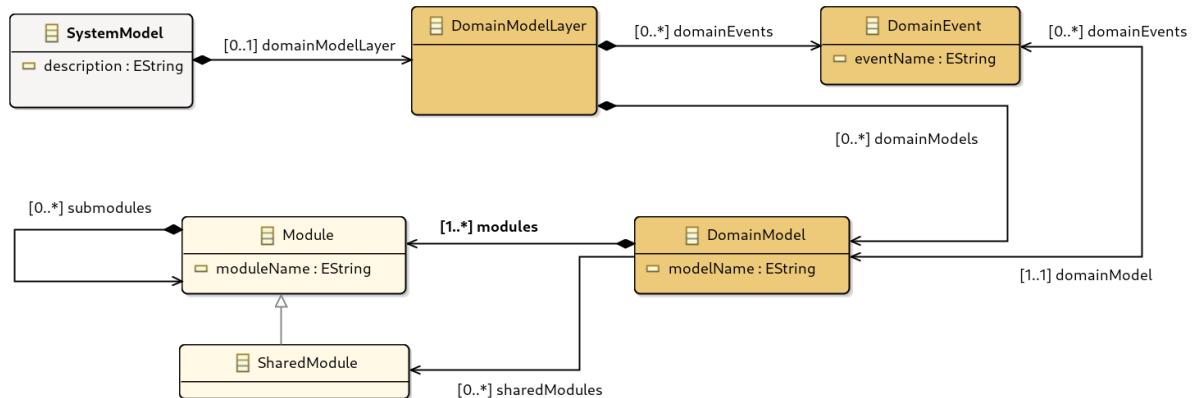


Abbildung 4.7: Metamodell - Fokus: Fachliche Schicht, Domänenmodelle und Pakete

Die fachliche Schicht besteht aus beliebig vielen benannten „DomainModels“. Weiterhin enthält diese ebenfalls beliebig viele „DomainEvents“. Durch eine bidirektionale Assoziation kann die Referenz von einem DomainModel zu verschiedenen Events beschrieben werden. Die Events selbst sind dabei nur einem DomainModel zugeordnet. Ein hypothetisches Event, das in mehreren Domänenmodellen vorkommt, wäre in dem Analyseprozess einer Problemdomäne ein Indiz für ein nicht erkanntes Domänenmodell.

Das DomainModel besitzt weiterhin Module, umgesetzt mit einer Klasse „Module“, welche wiederum Submodule enthalten können. Es zeigte sich während der Konzeption, dass es vorteilhaft ist, dass jedes DomainModel mindestens ein Modul besitzen muss. Diese Entscheidung erleichtert einerseits eine strukturiertere Codegenerierung. Andererseits wird dadurch eine deutlich einfachere Besitzbeziehung für das noch einzuführende „ModelElement“ realisiert. Auch der Aufwand für die Konzeption der dazugehörigen konkreten Syntax sinkt bedeutend. Die von Module abgeleitete Klasse „SharedModule“ beschreibt Module, die in mehreren DomainModels vorkommen können. Diese Modellierung ist notwendig, um eine korrekte Umsetzung des noch einzuführenden Konzepts Shared Kernel zu ermöglichen.

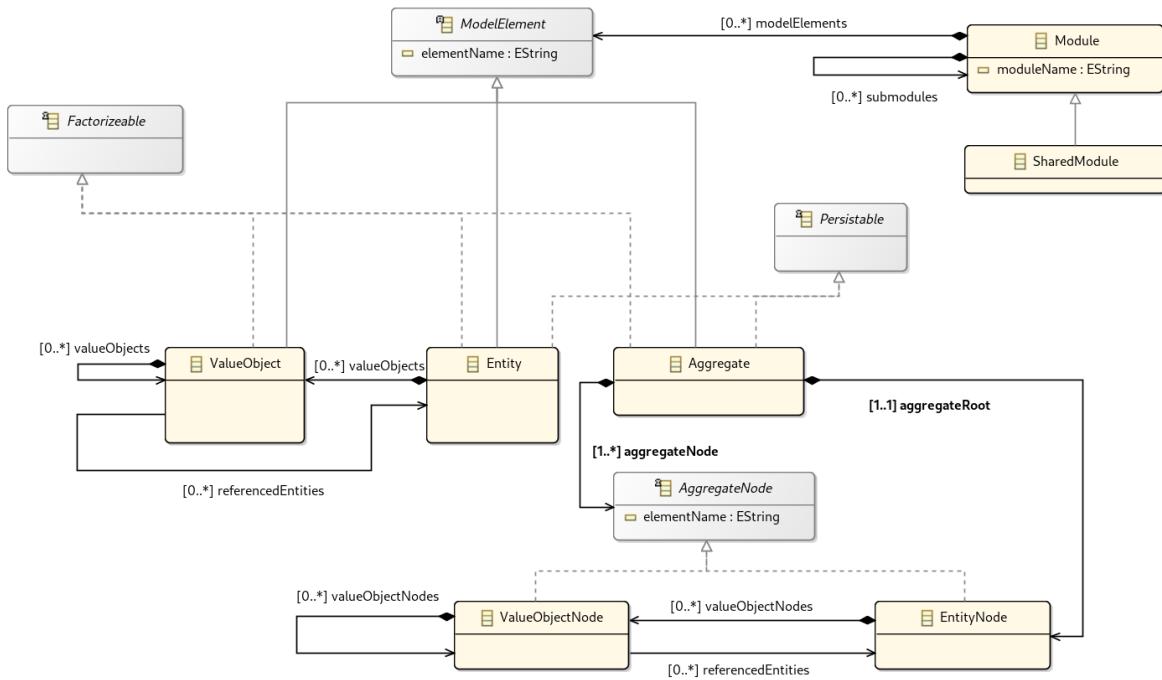


Abbildung 4.8: Metamodell - Fokus: Entityy, Value Object und Aggregate

Als abstrakte Basisklasse des MDD wird das `ModelElement` entworfen, welches in `Module` beliebig oft enthalten sein kann. Davon lassen sich die aus dem MDD bekannten Konzepte der `Entities`, `Value Objects` und `Aggregates` als Klassen ableiten. Nach dem DDD sind diese von Factories erzeugbare Objekte, umgesetzt durch die Einführung einer Schnittstelle namens „`Factorizable`“. Dies wurde symmetrisch für `Entities` und `Aggregates` mit der Schnittstelle „`Persistable`“ umgesetzt. Die Besitz- und Referenzrelation für `Value Objects` und `Entities` ist ebenso eine direkte Umsetzung der Definition des DDD.

Die modellgetreue Definition des `Aggregates` gestaltet sich schwierig. Die Umsetzung einer von außen zugreifbaren `Entity`-Wurzel mit von außen nicht zugreifbaren Blättern, welche `Entities` oder `Value Objects` sind, mit denselben Relationen, die aber nur untereinander gelten, hat sich als nicht elegant umsetzbar erwiesen. Der Versuch, eine einzige `Entity`-Klasse im Modell abzubilden, führte dazu, dass Elemente, die eigentlich vom `Aggregate` abgekapselt sein sollten, Zugriff darauf erhalten könnten. Diese Kapselung von Beziehungen muss durch eigene Klassen für `Entities` und `Value Objects` in einem `Aggregate` realisiert werden. Dementsprechend wird eine Klasse „`EntityNode`“, die einmalig als Wurzel existieren muss, modelliert. Weiterhin wird dazu eine abstrakte Klasse „`AggregateNode`“, und die diese ableitenden Klassen „`EntityNode`“ und „`ValueObjectNode`“ eingeführt.

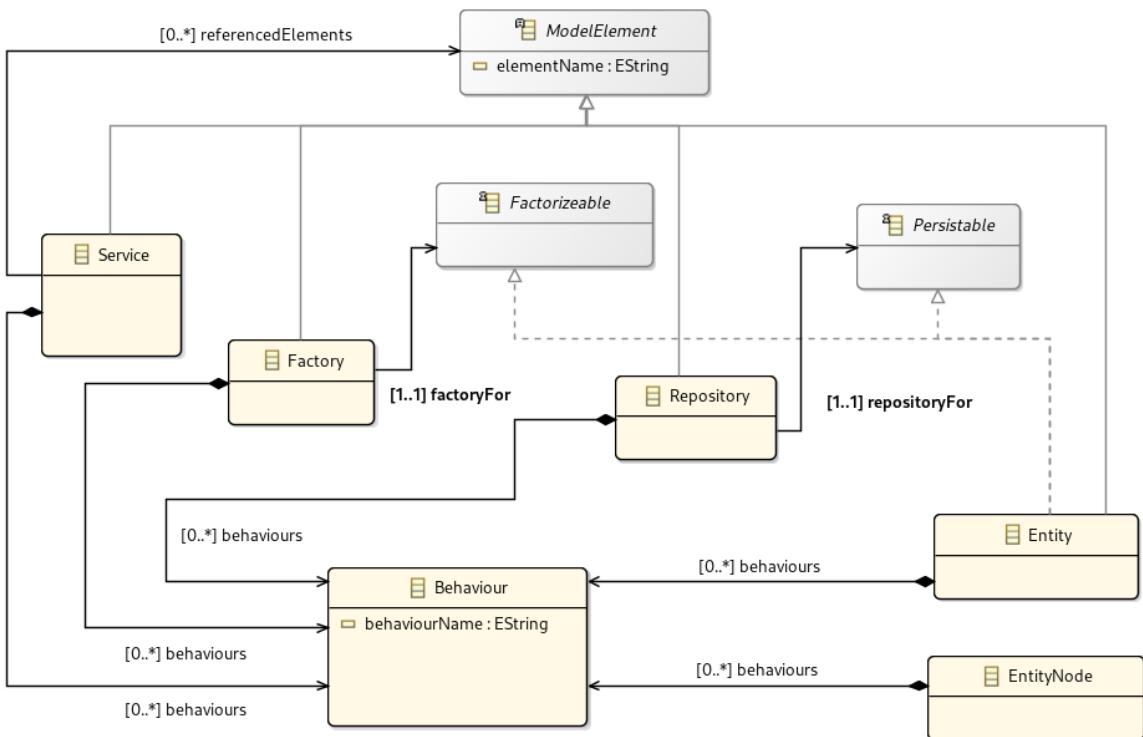


Abbildung 4.9: Metamodell - Fokus: Service, Repository, Factory und Behaviour

Es werden zu den MDD-Konzepten passenden Klassen „Service“, „Factory“ und „Repository“, welche ebenfalls Ableitungen von **ModelElement** sind, konzipiert. Die bereits eingeführten Schnittstellen **Factorizable** und **Persistable** werden von **Factory** und **Repository** passend referenziert. Dabei sind sowohl **Factory** als auch **Repository** je Objekt ihrer Klasse nur maximal einem, die Schnittstelle implementierenden, Objekt zugewiesen. Die Klasse **Service** hat weiterhin eine Assoziation zu beliebig vielen **ModelElements**, mit dem Zweck, Anwendungslogik umsetzen zu können. Eine Klasse **Behaviour** wird weiterhin zusätzlich eingeführt. Dies zeigte sich als notwendig, um das Verhalten in der Generierung und Modellierung präzise beschreiben zu können.

## 4.2.2 Modellierung des Strategic Designs

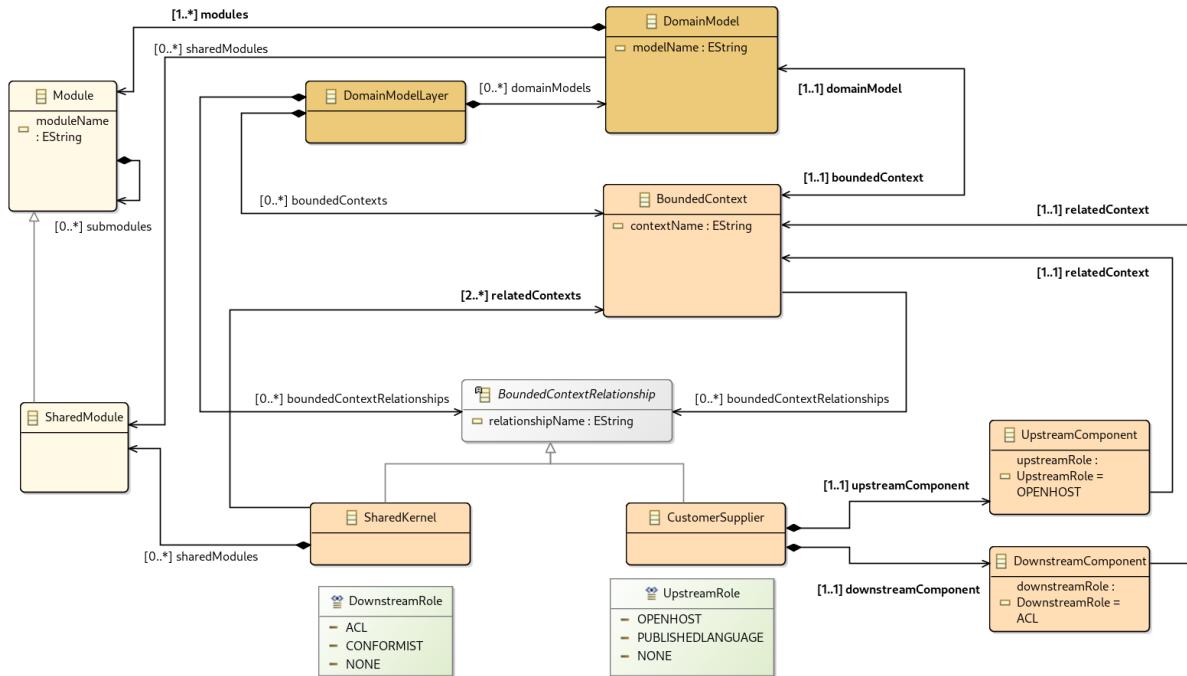


Abbildung 4.10: Metamodell - Fokus: Strategic Design

Die Modellierung des Bounded Context im Metamodell beginnt mit der Zuweisung der zugehörigen Klasse „BoundedContext“ als Teil des DomainModelLayer. Weiterhin wird nach dem DDD eine Ist-Beziehung zwischen DomainModel und BoundedContext eingeführt. In Betrachtung des Strategic Design soll der Bounded Context explizit vom DomainModel unterscheidbar sein, da im Folgenden die Beziehungen dieser untereinander im Vordergrund stehen, während das DomainModel den Blick in den Bounded Context symbolisiert. Ebenso Teil des DomainModelLayer ist die abstrakte Klasse „BoundedContextRelationship“, welche beliebig oft von einem Bounded Context referenziert werden kann. Diese beschreibt Beziehungen als eigene Objekte, analog zu dem Konzept von ContextMapper. „SharedKernel“ und „CustomerSupplier“ sind die zwei davon abgeleiteten Klassen. CustomerSupplier besitzt weiterhin jeweils eine sendende und empfangende Komponente, welche auf genau einen Bounded Context verweist. Im Gegensatz zu ContextMapper reicht hier eine reduzierte Modellierung der Konzepte des Strategic Design. Dies zeigte sich bei der Konzeption von Visualisierung und Generierung. Dabei ergab sich, dass die Modellierung der übrigen Strategic Design Konzepte als statische Enumerationen hinreichend ist. SharedKernel enthält außerdem noch die bereits erwähnte Referenz auf das SharedModule. Dies ist notwendig, um bei der Beschreibung des DomainModels dieses korrekt zu referenzieren. Das Fehlen kann somit zu einer unvollständigen Beschreibung führen, wodurch möglicherweise wichtige Aspekte zum Verständnis des Modells fehlen würden.

### 4.2.3 Modellierung der technischen Modellebene

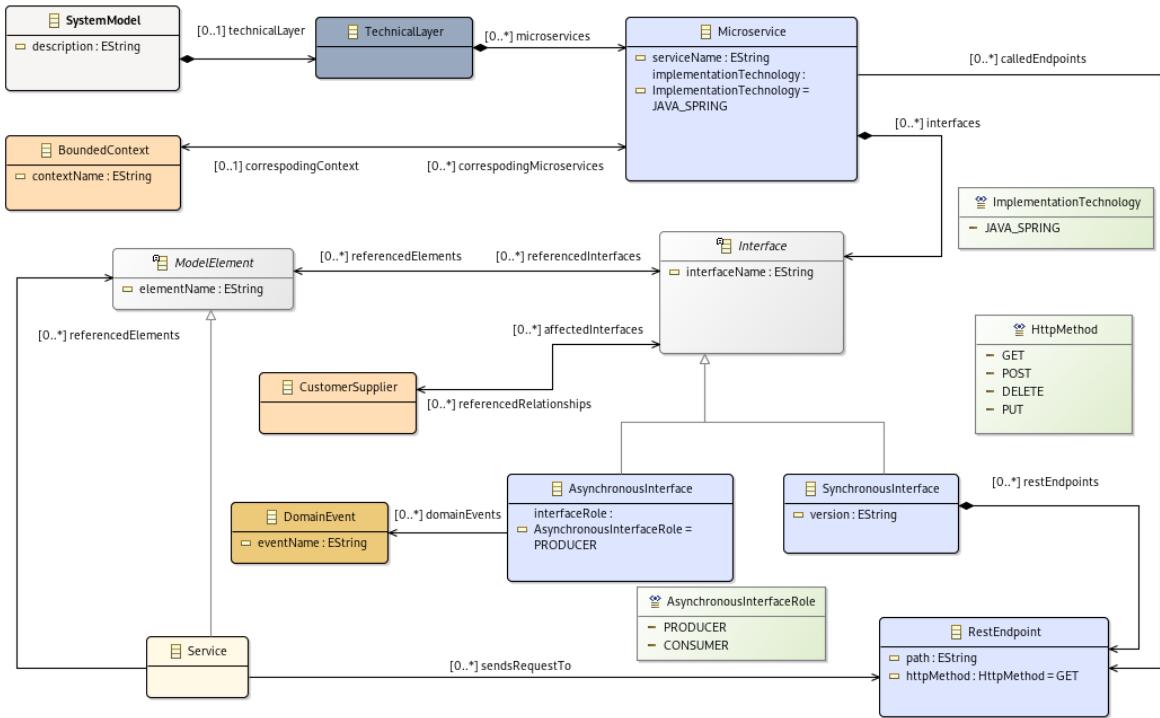


Abbildung 4.11: Metamodell - Fokus: Technische Schicht

Die technische Modellebene wird mit einer Klasse „TechnicalLayer“ umgesetzt, welche die Microservices enthält. Die dazugehörige Klasse „Microservice“ ist benannt und weiterhin mit einem Attribut vom Typ „ImplementationTechnology“ versehen. Mögliche Implementierungstechnologien können so zukünftig hinzugefügt werden. Ein Objekt der Klasse Microservice besitzt beliebig viele „Interfaces“. Von dieser benannten abstrakten Klasse existieren weiterhin die Ableitungen „AsynchronousInterface“ und „SynchronousInterface“. Asynchrone Schnittstellen sind dabei entweder Produzenten oder Konsumenten. Sychrone Schnittstellen sind demgegenüber versioniert und implementieren HTTP-Schnittstellen. Hier wird sich bewusst für eine Eingrenzung auf die Protokolle HTTP und das REST-Paradigma entschieden, da diese für synchrone Schnittstellen die relevantesten im Microservice-Kontext sind. Die zur synchronen Schnittstelle gehörigen REST-Endpunkte, modelliert durch eine Klasse „RestEndpoint“, besitzen einen Pfad sowie eine HTTP-Methode. Diese wird im dazugehörigen Enum spezifiziert. Bei der Konzeption der Implementierungsumsetzung der MDD-Bausteine wurde anfangs versucht, Gegenstücke zu diesen als eigenständige Klassen der technischen Schicht zu definieren. Jedoch stellte sich heraus, dass aufgrund des grundlegenden Konzepts des MDD, dem Erzeugen einer hohen Kohäsion zwischen Modellbausteinen und Umsetzung dieser, diese nicht zwingend benötigt werden. Dadurch hat eine Abwägung stattgefunden: Es wurde erwogen, ob aus Gründen der kognitiven Komplexität des Metamodells bestimmte Elemente explizit als Teil der Schicht definiert werden sollen.

Die alternative Überlegung war, ob auf diese Definition verzichtet werden sollte, obwohl Modelllemente in der technischen Abstraktionsebene implizit existieren. Entschieden wurde, auf die explizite Definition zu verzichten.

Die Klassen der technischen Schicht besitzen weiterhin verschiedene Assoziationen zu Klassen der fachlichen Schicht. So kann ein BoundedContext durch beliebig viele Microservices umgesetzt werden. Zwischen ModelElements und Interfaces können bidirektional beliebig viele Assoziationen existieren. Eine CustomerRelationship-Beziehung kann beliebig viele Schnittstellen betreffen. Ebenso kann eine Schnittstelle beliebig viele referenzieren. Dem AsynchronousInterface können verschiedene DomainEvents zugewiesen werden. Dies ist insbesondere im Prozess der Analyse der Problemdomäne eine wertvolle Assoziation.

Um den Zugriff auf REST-Endpunkte zu modellieren, wird einerseits eine Assoziation vom Microservice zum RestEndpoint eingeführt. Diese ist notwendig, um auf der Intermicroservice-Kommunikationsebene Architekturen zu entwerfen. Zusätzlich kann spezifiziert werden, wo im Service selbst ein Aufruf ausgelöst wird. Dafür wurde basierend auf dem MDD und somit der Richtlinie, Logik durch Services primär umzusetzen, dieses Aufrufen in der Klasse Service verortet.

#### 4.2.4 Modellierung der Infrastrukturellen Umgebung

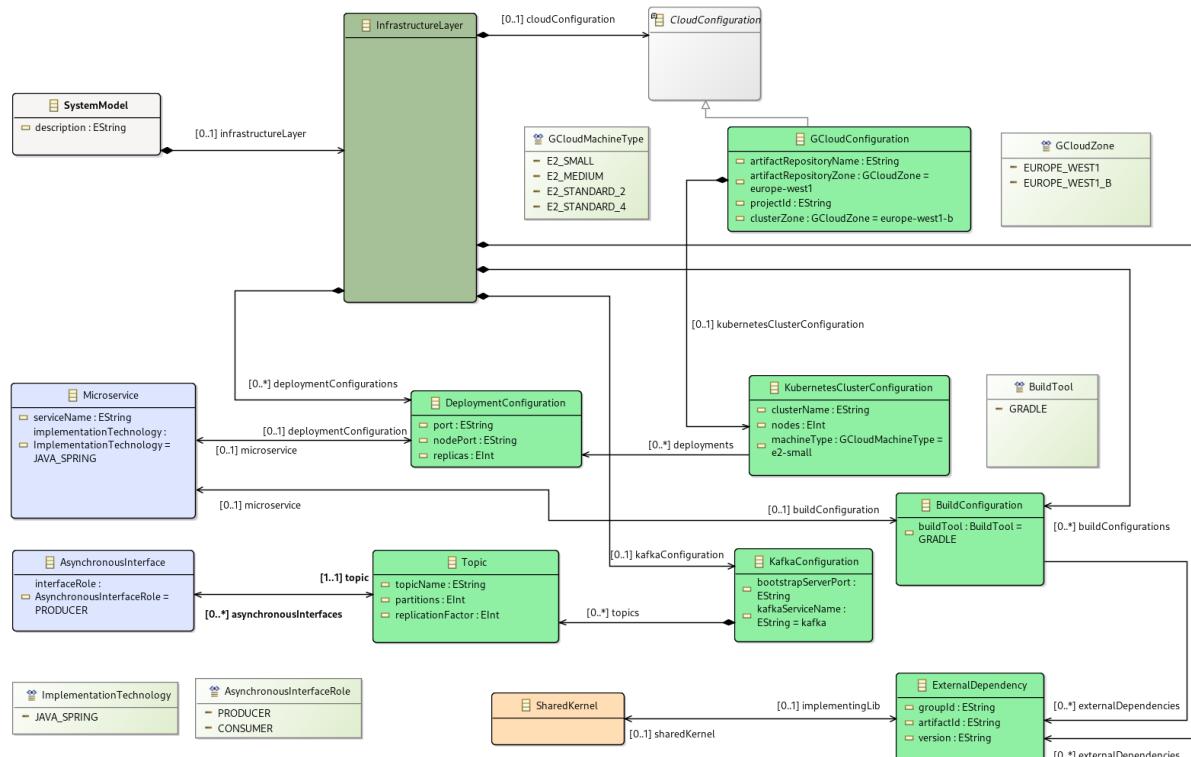


Abbildung 4.12: Metamodell - Fokus: Infrastrukturschicht

Die Infrastrukturebene, umgesetzt durch die Klasse „InfrastructureLayer“, besitzt eine Konfiguration der Cloud-Umgebung. Diese wird als abstrakte Klasse „CloudConfiguration“ entworfen, von der die „GCloudConfiguration“ abgeleitet wird. Ergänzungen für AWS oder Azure würden dort ansetzen. Die GCloudConfiguration definiert GCP-spezifische, notwendige Eigenschaften. So definiert „artifactRepositoryName“ und „artifactRepositoryZone“ ein Ziel-Repository für Docker Images.

Das Attribut „clusterZone“ definiert die Google-spezifische Region des Kubernetes-Clusters, welches durch die Klasse „KubernetesClusterConfiguration“ modelliert wird. Diese wird außerdem als Teil der GCloudConfiguration modelliert. Bei der Kubernetes Konfiguration muss weiterhin bedacht werden, dass ein Kubernetes-Cluster Attribute besitzt, die unabhängig von dem Cloud-Anbieter sind. Jedoch gibt es auch welche die cloudspezifisch sind. Konkret wird hier noch ein Google-spezifischer Maschinentyp verwendet, welcher bei einer Erweiterung der Cloud Umgebungen abstrakter konzipiert werden müsste. Ebenso wird aktuell nur ein existierendes Cluster in der Cloud modelliert, da dies hinreichend für das Deployment einer Microservice-Architektur ist. Grundsätzlich möglich wären aber auch mehrere Cluster in einer Cloud-Umgebung.

Das Cluster referenziert weiterhin die Klasse „DeploymentConfiguration“. Diese beschreibt Deployment-Objekte in einem Kubernetes-Cluster. Modelliert werden diese als Teil der Infrastrukturschicht. Sie haben weiterhin eine bidirektionale Assoziation, welche Objekte der Klasse zu einem Microservice zuweisen können. Hier wurde darauf verzichtet das Konzept der Pods in das Metamodell mit aufzunehmen um die Komplexität zu reduzieren. Es wird daher festgelegt, dass jeder Service auf genau einem Pod, der genau einem Deployment zugewiesen ist, ausgeführt wird.

Symmetrisch zu der DeploymentConfiguration wurde die bidirektionale Assoziation für die Klasse „BuildConfiguration“ und Microservice modelliert. Diese sind Teile der Infrastrukturschicht und gehen von einer Verwendung des Build-Tools Gradle aus. Andere Build-Tools wie z.B. Maven können in der dazugehörigen Enumeration „BuildTool“ bei möglicher Erweiterung des Metamodells ergänzt werden. Die BuildConfiguration verweist auf die Klasse „ExternalDependency“, ebenfalls Teil des InfrastructureLayers. Diese besitzt eine auf das öffentliche Bibliotheksverzeichnis „mvnrepository“ abgestimmte Attributierung. Eine weitere Relation zwischen ExternalDependency und SharedLib modelliert den Zusammenhang dieser.

Eine „KafkaConfiguration“ definiert Port und Service-Name eines Kafka-Containers in der Cloud. Hier wurde sich gegen eine Abstraktion der asynchronen Kommunikationstechnologie entschieden, um die Komplexität des Metamodells nicht weiter zu erhöhen. Inwiefern eine Abstraktion verschiedener solcher Technologien überhaupt möglich wäre, wurde nicht näher betrachtet. Die KafkaConfiguration besitzt benannte Topics, welche von asynchronen Schnittstellen beschrieben oder konsumiert werden. Eine asynchrone Schnittstelle ist so modelliert, dass sie nur mit einem Topic interagiert. Weiterhin besitzt es die Attribute „partitions“, welches das Topic mit der angegebenen Zahl partitioniert, und „replicationFactor“, welches die Anzahl der Replikationen für Ausfälle definiert.

## 5 Entwicklung einer konkreten Syntax

Mit Hilfe des Eclipse Modeling Frameworks und der Sirius-Erweiterung ist es möglich, spezifische Modelle zu konzeptionieren und zu visualisieren, die auf dem entwickelten Microservice-Metamodell basieren. Im Folgenden wird eine konkrete Syntax präsentiert und erörtert, wie diese erstellt wurde. Dabei wurde versucht, die Komplexität von Microservice-Architekturen durch verschiedene Modellsichten greifbarer zu machen, welche sich gegenseitig ergänzen.

Beginnend soll ein grundlegender, abstrakter Workflow skizziert werden, welcher die Modellierung einer Microservice-Architektur mit der DSL beschreibt. Dazu wird festgelegt, dass ein initiales Grundmodell vorhanden sein muss. Dieses muss alle notwendigen Wurzelklassen, wie zum Beispiel die Klasse „System Model“, beinhalten. Danach wird die Problemdomäne konzeptionell erfasst. Dies würde durch ein Zuweisen der DDD-Konzepte, bzw. der Metamodellklassen der DomainModelLayer, umgesetzt werden. Dazu sollen im folgenden notwendige Darstellungen und Editoren entworfen werden. In einem nächsten Schritt, kann ein Modell mit einer technischen Schicht erweitert werden, welche die konkreten Dienste dazu modelliert. Abschließend soll eine dazu passende Infrastrukturschicht modelliert werden, die wiederum das bis zu diesem Schritt modellierte Modell ergänzt. Auch hierfür werden im folgenden Darstellungen und Editoren entworfen.

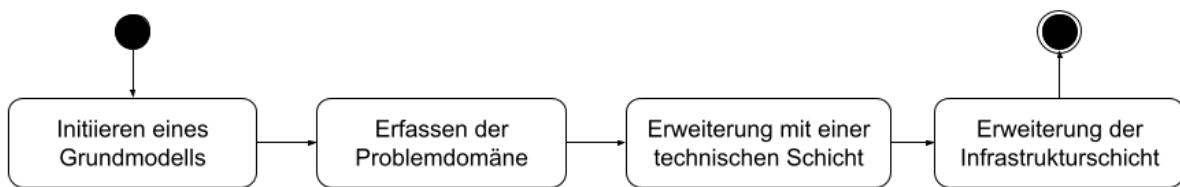


Abbildung 5.1: Ein abstrakter Modellierungsworkflow für das Microservice-Metamodell

## 5.1 Entwurf des Editors

Die verschiedenen Visualisierungen und Werkzeuge werden mit einer „Sirius Viewpoint Specification“ umgesetzt. Grundlegend funktioniert dies analog zu den Methoden, die in der dieser Arbeit vorausgehenden Seminararbeit vorgestellt wurden [LÖ23, S.11-13].

Aufgrund der Komplexität des Metamodells ergeben sich grundlegende konzeptionelle Fragen, die vor der eigentlichen Umsetzung zu klären sind. Es hat sich gezeigt, dass der Versuch, allumfassende Modelldiagramme zu erstellen, kognitiv schwer nachvollziehbar war. Der ursprüngliche Ansatz, pro Schicht ein separates Diagramm zu verwenden, wurde daher verworfen. Stattdessen zielt die Neukonzeption der Diagramme darauf ab, einen effektiven Workflow zu unterstützen. Zudem soll der Einsatz verschachtelter Darstellungen, bei denen einzelne Grafiken als Elemente in übergeordneten Strukturen dienen, das Verständnis erleichtern und eine strukturierte Präsentation ermöglichen.

Zentral für den Entwurf sind verschiedene AQL-Ausdrücke, die genutzt werden, um Logik in der grafischen Oberfläche umzusetzen. So erweist sich die Umsetzung des Werkzeugs, das Microservices in der „ContextRelationshipDescription“, einem Diagramm welches die Beziehungen von BoundedContexts zueinander zeigt, erzeugt, als verhältnisweise komplex:

```
1 //Precondition
2 aql:self.oclIsTypeOf(microserviceMetamodell::BoundedContext)
3
4 //Change Context to Right Container
5 aql:self.eContainer().eContainer().technicalLayer
6
7 //Setting the service name after creating the instance for the layer
8 aql:self.correspondingContext.contextName.toLowerCase().concat('-service')
9
10 //Changing back to the container, the bounded context,
11 var:container
12 //and setting container.correspondingMicroservices to the instance which was
13     created before
14 var:instance
```

Quellcode 5.1: Erzeugen eines Microservices als Border Node an einem Bounded Context

Dieser Ansatz musste öfters ähnlich angewandt werden. Grund hierfür ist die Entscheidung gegen Diagramme, welche nur die Schichtebene visualisieren. Dadurch muss an einigen Stellen auf hierarchisch höher liegende Container zugegriffen werden, um die passenden Objekte zu referenzieren. Weiterhin wird der Ansatz der typisierten Vorbedingung häufig genutzt.

Eine andere Art von Problematik ergibt sich bei der Darstellung durch rekursive Beziehungen von Elementen zu sich selbst. Dies soll am Beispiel der „AggregateStructureDescription“, welche die Struktur eines Aggregate visualisiert, verdeutlicht werden.

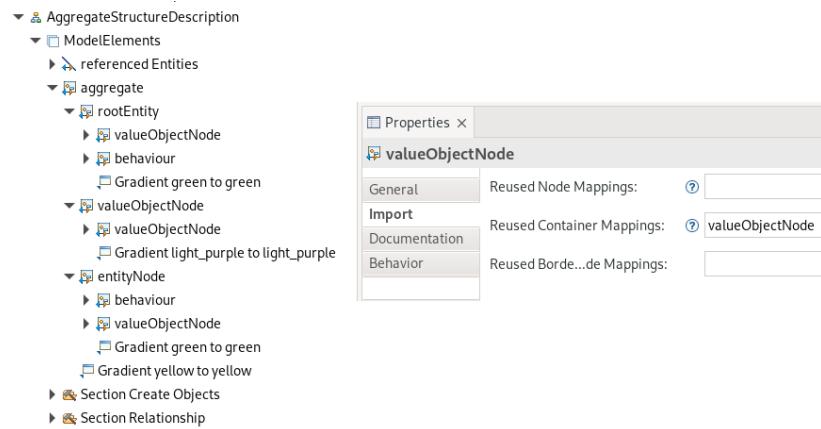


Abbildung 5.2: Umsetzen von rekursiven Aufrufen durch das importieren von Nodes mit Sirius

Hier ist zu beachten, dass eine rekursive Beziehung, wie sie beispielsweise der ValueObjectNode aufweist, korrekt umgesetzt wird. Dazu muss der Node, der sich selbst enthält, sich selbst importieren, wie in der Abbildung zu sehen ist. Weiterhin stellt sich nun die Frage, ob eine Darstellung solcher rekursiven Beziehungen grundsätzlich sinnvoll ist.

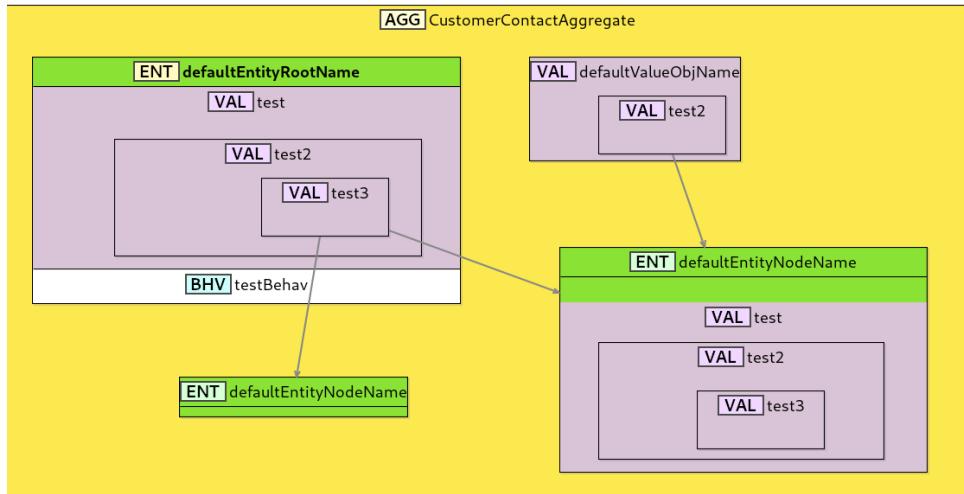


Abbildung 5.3: Eine kognitiv anspruchsvolle Aggregate Visualisierung - Hat ein Betrachter hier noch einen Mehrwert ?

Eine Modellierung von Klassenattributen in der Implementierung wurde bisher nicht berücksichtigt. Dies sollte im Rahmen der notwendigen Feinimplementierung händisch geschehen. Da außerdem auch andere EMF-Beschreibungsmodelle existieren, die für genau diese Aufgabe konzipiert wurden, wird die Entscheidung getroffen, keine Anpassung vorzunehmen und die Attributierung nicht konzeptionell in das Metamodell aufzunehmen. Fortschreitend werden für die Entity und das ValueObject keine feineren Strukturdiagramme entworfen. Konzeptionell würden diese jedoch symmetrisch zur AggregateStructureDescription umgesetzt werden.

## 5.2 Beschreibung der Diagramme

### 5.2.1 ContextRelationshipDescription

Diese Darstellung zeigt die in einer DomainModelLayer vorhandenen BoundedContexts und ihre Beziehungen zueinander. Zweck des Diagramms ist die initiale Feststellung von Kontexten bei der Analyse einer Problemdomäne. Weiterhin können hier DomainModels und Microservices für einen Bounded Context erzeugt und implizit zugewiesen werden. Für eine Shared Kernel Beziehung, welche durch einen eigenen Container dargestellt wird, kann eine ExternalDependency ebenso erzeugt und implizit zugewiesen werden. Andere Beziehungstypen werden durch eine Auswahl im Eigenschaftsfenster mittels Radio-Buttons realisiert. Module können außerdem hier schon erzeugt werden, sofern einem BoundedContext ein DomainModel zugewiesen ist. Das Erzeugen eines Moduls in einem SharedKernel führt dabei zur Erstellung eines SharedModule.

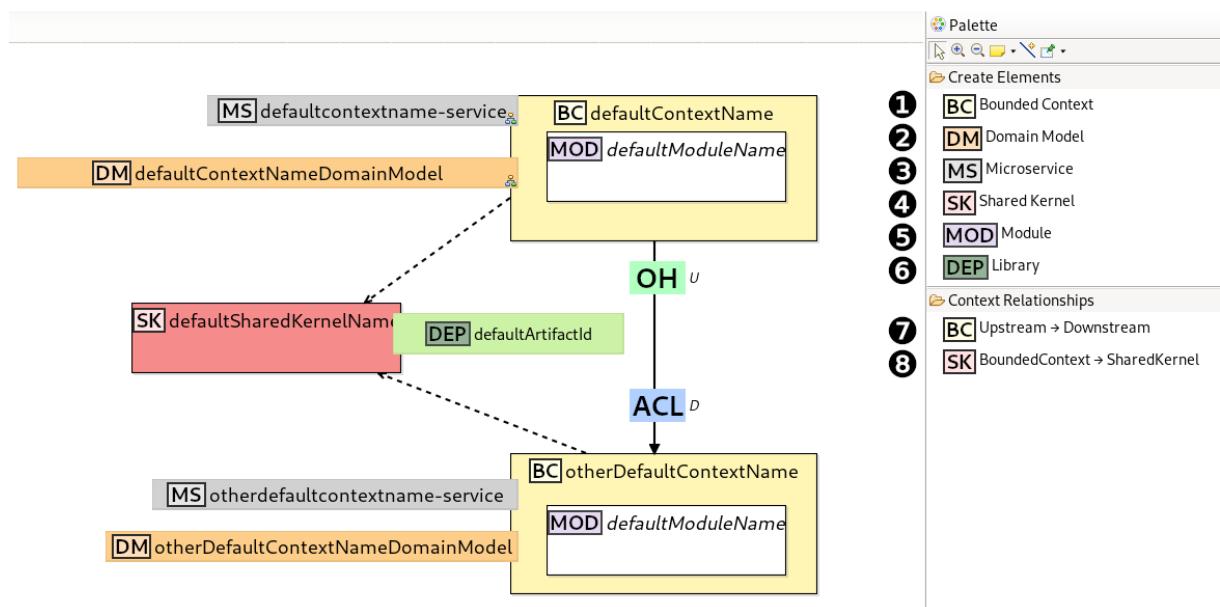


Abbildung 5.4: Editoransicht der ContextRelationshipDescription

1	Erzeugt ein BoundedContext. Namenskonvention: camelCase
2	Erzeugt auf einem BoundedContext ein zugewiesenes DomainModel. Namenskonvention: camelCase
3	Erzeugt auf einem BoundedContext ein zugewiesenen Microservice. Namenskonvention: kebab-case
4	Erzeugt ein SharedKernel. Namenskonvention: camelCase
5	Erzeugt auf einem BoundedContext oder SharedKernel ein (Shared)Module. Vorbedingung: Existierendes DomainModel. Namenskonvention: camelCase
6	Erzeugt auf einem SharedKernel eine ExternalDependency. Namenskonvention: Abh. zu der Bezeichnung in mvnrepository
7	Erzeugt Beziehung zwischen Quellkontext (Upstream) und Zielkontext (Downstream). Nähere Typisierung im Eigenschaftsfenster
8	Verbindet ein BoundedContext mit einem SharedKernel

Tabelle 5.1: Legende - ContextRelationshipDescription

## 5.2.2 DomainModelDescription

Die „DomainModelDescription“ visualisiert ein DomainModel. Sie zeigt Module, die es enthält, sowie DomainEvents, die mit dem Modell assoziiert sind. Zudem stellt sie die verwendeten ModelElements dar und ermöglicht es dem Bearbeiter, diese nach Bedarf mit anderen Elementen des DomainModels zu verknüpfen. Außerdem visualisiert sie die in Entities und Aggregates enthaltenen Objekte.

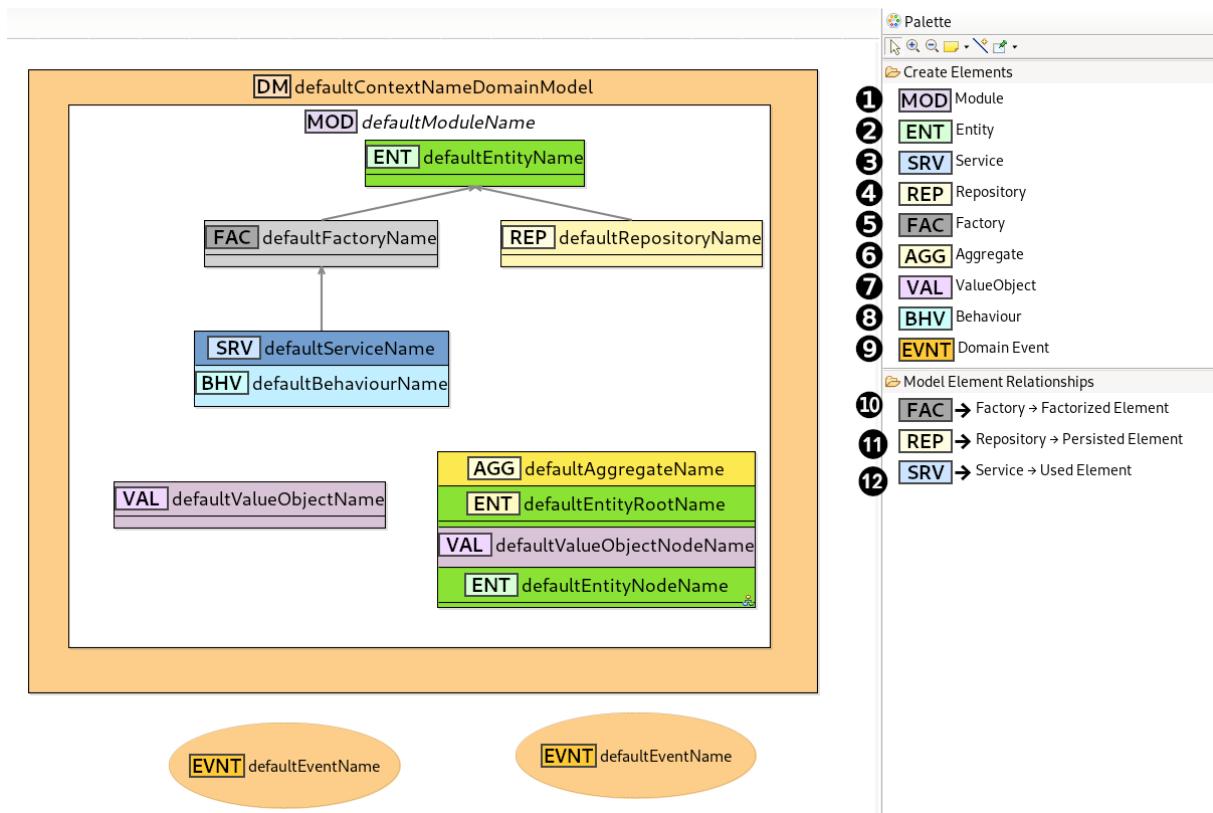


Abbildung 5.5: Editoransicht der DomainModelDescription

1	Erzeugt auf einem DomainModel ein enthaltenes Module. Namenskonvention: camelCase
2	Erzeugt auf einem Module eine enthaltene Entity. Namenskonvention: camelCase
3	Erzeugt auf einem Module einen enthaltenen Service. Namenskonvention: camelCase
4	Erzeugt auf einem Module ein enthaltenes Repository. Namenskonvention: camelCase
5	Erzeugt auf einem Module eine enthaltene Factory. Namenskonvention: camelCase
6	Erzeugt auf einem Module ein enthaltenes Aggregate. Namenskonvention: camelCase
7	Erzeugt auf einem Module ein enthaltenes ValueObject. Namenskonvention: camelCase
8	Erzeugt auf geeigneten ModelElements ein Behaviour. Namenskonvention: camelCase Beim Repository zusätzlich: JPA konformer Funktionsname
9	Erzeugt außerhalb des DomainModels ein zu diesem assoziiertes Event. Namenskonvention: keine
10	Weist einer Factory ein Factorizable Element zu.
11	Weist einem Repository ein Persistable Element zu.
12	Weist einem Service ein Element zu.

Tabelle 5.2: Legende - DomainModelDescription

### 5.2.3 AggregateStructureDescription

Um die kognitive Last, die durch komplexe Aggregates in der DomainModelDescription entstehen kann, zu reduzieren, wird eine AggregateStructureDescription genutzt. Diese spezialisiert sich auf die Darstellung der nach dem Metamodell verschachtelbaren Struktur. Diese wurde, wie bereits erläutert, zwar konzipiert, jedoch ist ihr Mehrwert diskutabel.

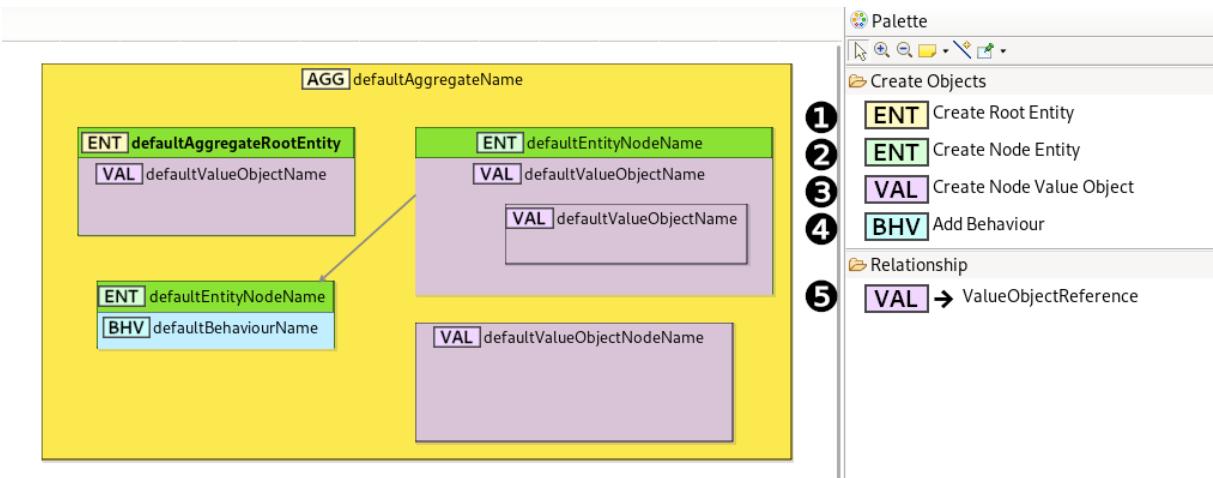


Abbildung 5.6: Editoransicht der AggregateStructureDescription

1	Erzeugt (eimalig) auf einem Aggregate eine dazugehörige EntityNode als Wurzel. Namenskonvention: camelCase
2	Erzeugt auf einem Aggregate eine EntityNode als Blatt. Namenskonvention: camelCase
3	Erzeugt auf einem Aggregate, einer Entity oder einem ValueObject ein enthaltenes ValueObject. Namenskonvention: camelCase
4	Erzeugt auf einer Entity ein Behaviour. Namenskonvention: camelCase
5	Weist einem ValueObject eine referenzierte Entity zu

Tabelle 5.3: Legende - AggregateStructureDescription

## 5.2.4 MicroserviceCommunicationDescription

Die „MicroserviceCommunicationDescription“ zeigt die Microservices in der technischen Schicht. Interfaces können hier betrachtet und verwaltet werden. Durch die zusätzliche Visualisierung der DomainEvents, die dem Microservice zugehörigen DomainModel zugewiesen sind, kann der Betrachter sich bei der Modellierung an diesen orientieren, um AsynchronousInterface Objekte zu verorten. SynchronousInterfaces können hier RestEndpoints zugewiesen werden. AsynchronousInterfaces können weiterhin Topics zugewiesen werden. Mit Werkzeugen zum Verbinden der Kommunikationsteilnehmer kann hier somit eine vollständige Übersicht der Kommunikationsabläufe in einer Microservice-Architektur umgesetzt werden.

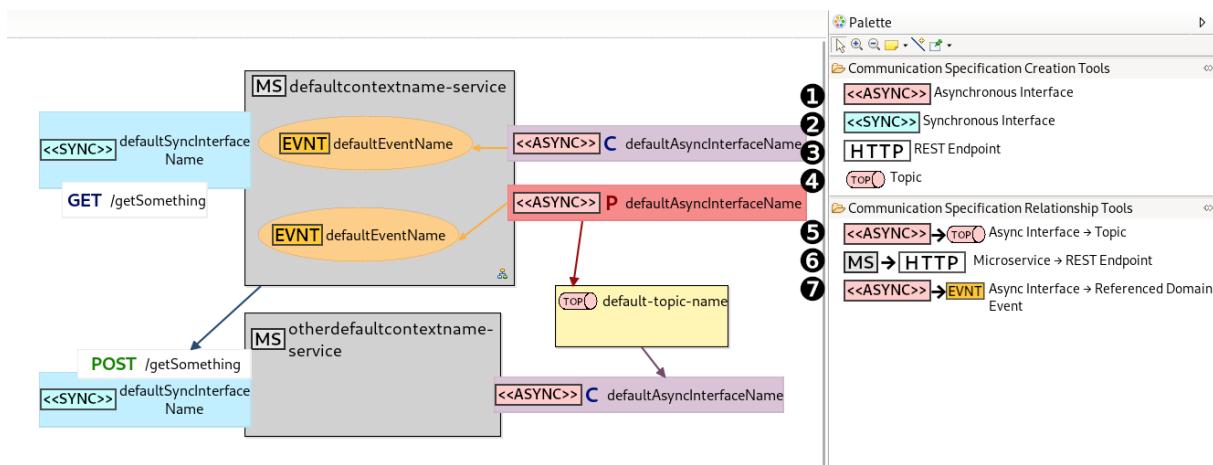


Abbildung 5.7: Editoransicht der MicroserviceCommunicationDescription

1	Erzeugt auf einem Microservice ein zugewiesenes AsynchronousInterface. Namenskonvention: camelCase
2	Erzeugt auf einem Microservice ein zugewiesenes SynchronousInterface. Namenskonvention: camelCase
3	Erzeugt auf einem SynchronousInterface einen zugewiesenen RestEndpoint. Namenskonvention: "/" + camelCase
4	Erzeugt ein neues Topic. Setzt existierende KafkaConfiguration voraus. Namenskonvention: kebab-case
5	Weist einem AsynchronousInterface ein Topic zu. Datenflussrichtung wird implizit aus Rolle der Schnittstelle bestimmt.
6	Weist einem Microservice einen zu aufrufenden RestEndpoint zu.
7	Weist einem AsynchronousInterface ein Event zu.

Tabelle 5.4: Legende - MicroserviceCommunicationDescription

## 5.2.5 MicroserviceInterfaceDescription

Die folgende „MicroserviceInterfaceDescription“ dient der Darstellung und Verwaltung der Kommunikation und deren Ursprung auf Service-Ebene. Daher werden Relationen aus dem Domain-Model ausgeblendet. Zusätzlich werden Interfaces des Dienstes und RestEndpoints anderer Dienste, die dem Microservice zugewiesen sind, visualisiert. Das Anlegen und Zuweisen dieser externen Schnittstellen wird nur in der MicroserviceCommunicationDescription umgesetzt, um die kognitive Last dieser Beschreibung zu reduzieren.

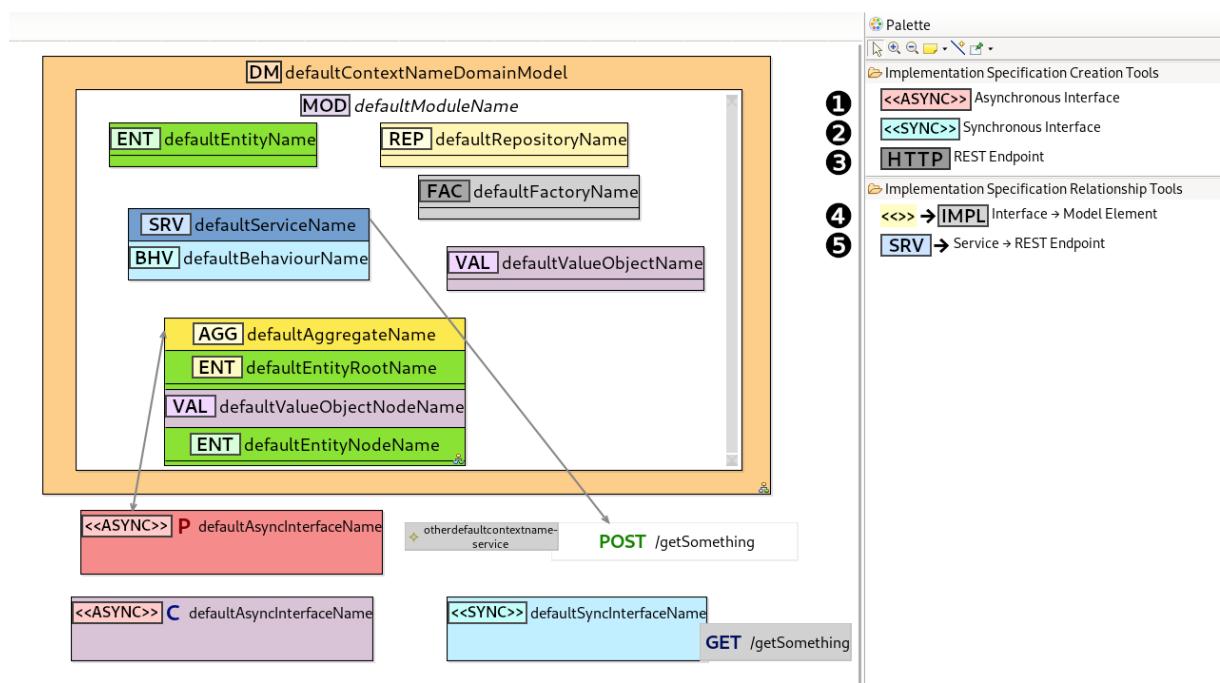


Abbildung 5.8: Editoransicht der MicroserviceInterfaceDescription

1	Erzeugt ein AsynchronousInterface. Namenskonvention: camelCase
2	Erzeugt ein SynchronousInterface. Namenskonvention: camelCase
3	Erzeugt auf einem SynchronousInterface einen RestEndpoint. Namenskonvention: "/" + camelCase
4	Weist einem Interface referenzierte ModelElements zu.
5	Weist einem Service einen externen, zu aufrufenden RestEndpoint zu.

Tabelle 5.5: Legende - MicroserviceInterfaceDescription

## 5.2.6 InfrastructureOverviewDescription

Schließlich wird in der „InfrastructureOverviewDescription“ die Infrastrukturschicht dargestellt. Hier werden vor allem Konfigurationen angelegt und diese den existierenden Microservices zugewiesen. Weiterhin werden zusätzliche modellierte Abhängigkeiten dargestellt.

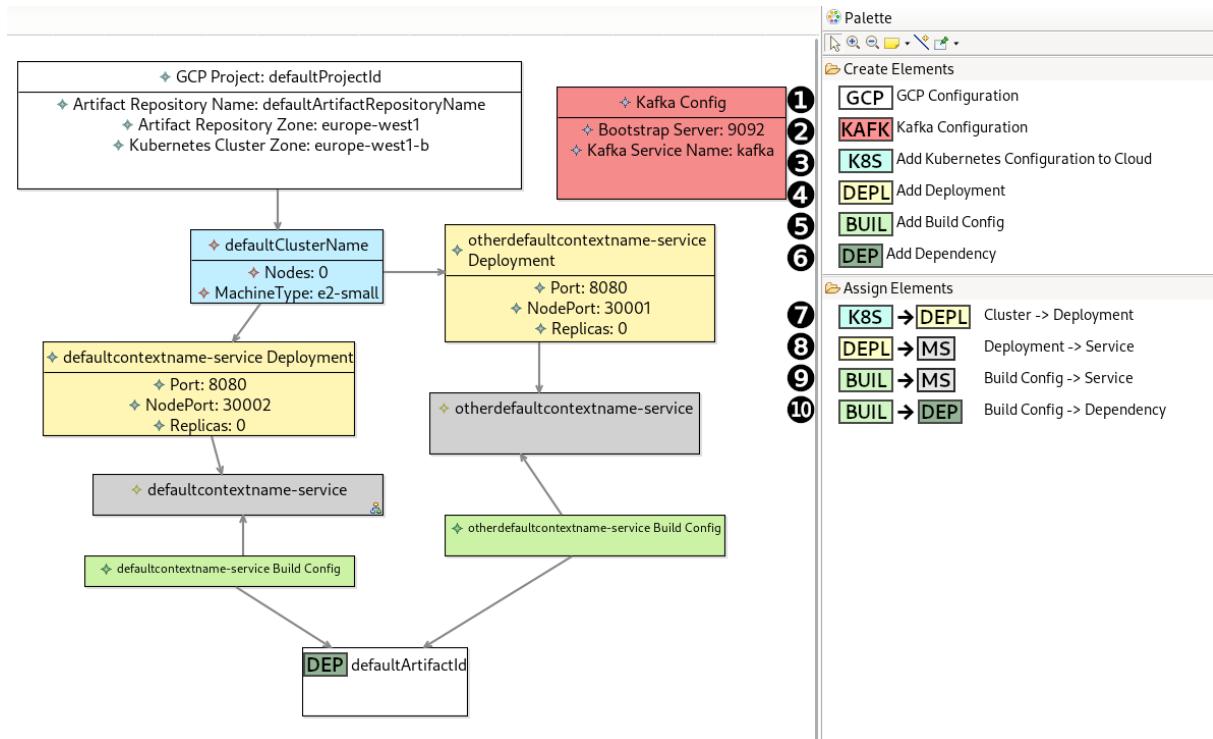


Abbildung 5.9: Editoransicht der InfrastructureOverviewDescription

1	Erzeugt eine GCloudConfiguration. Namenskonvention: Der GCP zu entnehmen, kebab-case
2	Erzeugt eine KafkaConfiguration. Namenskonvention: kebab-case
3	Erzeugt eine KubernetesClusterConfiguration. Namenskonvention: kebab-case
4	Erzeugt eine DeploymentConfiguration. Namenskonvention: Generiert
5	Erzeugt eine BuildConfiguration. Namenskonvention: Generiert
6	Erzeugt eine ExternalDependency. Namenskonvention: Äquivalent zur Benennung in mvnrepository
7	Weist einer KubernetesClusterConfiguration DeploymentConfigurations zu.
8	Weist einer DeploymentConfiguration einen Service zu.
9	Weist einer BuildConfiguration einen Service zu.
10	Weist einer BuildConfiguration eine Dependency zu.

Tabelle 5.6: Legende - InfrastructureOverviewDescription

## 5.3 Konkretisierung des Modellierungsworkflows

Auf Basis des abstrakten Modellierungsworkflows wird nun ein konkreter Workflow, der den konzipierten Editor und die dazugehörige konkrete Syntax verwendet, definiert. Dazu kann beginnend eine dem Quellcodeverzeichnis beiliegende Modellvorlage [Löb] verwendet werden. Alternativ können die notwendigen Klassen, die in Abbildung 5.10 aufgezählt werden, händisch in ein leeres EMF-Modell eingefügt werden. Der Schritt des Erfassens der Problemdomäne erfolgt durch das sequenzielle Anlegen der ContextRelationshipDescription, der DomainModelDescriptions (je DomainModel) und der AggregateStructureDescriptions. Darauf folgt die Konzeption der technischen Schicht mittels MicroserviceCommunicationDescription und MicroserviceInterfaceDescription (je Microservice). Abschließend wird mit der InfrastructureOverviewDescription die nötige Infrastruktur modelliert.

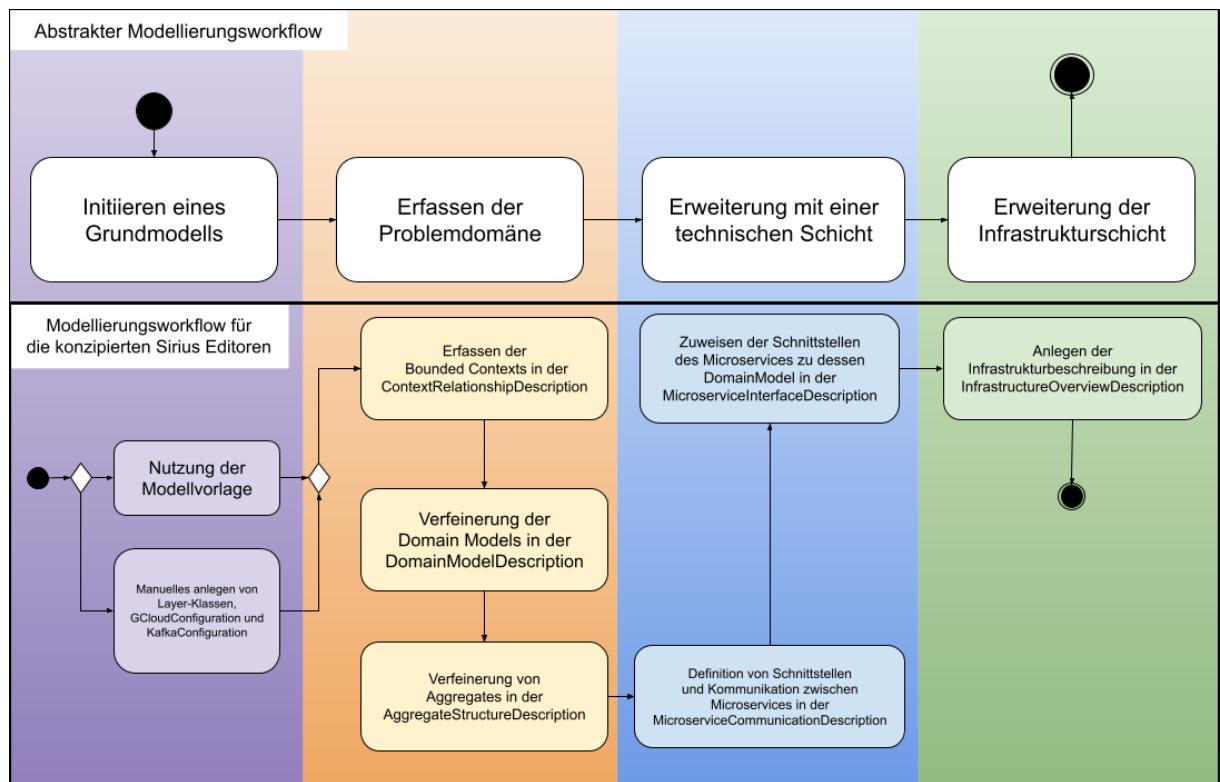


Abbildung 5.10: Abbildung des abstrakten Modellierungsworkflows auf einem für diesen Editor angepassten Workflow

# 6 Konzeption und Umsetzung der Codegenerierung

Basierend auf dem Metamodell soll eine Codegenerierung mit dem Acceleo-Codegenerator konzipiert werden. Dabei soll durch ein Generierungstemplate Code erzeugt werden, der Java/Spring-Applikationen umsetzt, welche mit Docker als Images ausgeführt werden können. Nutzer sollen weiterhin einfach Deployments in eine existierende Google-Cloud-Umgebung ausführen können. Abgegrenzt von der Generierung wird die konkrete Implementierung von Funktionen, die Datenstruktur von Entities und ValueObjects sowie detailliertere Inhalte der Kommunikation (wie z.B.: HTTP-Headers, Request Bodies oder Kafka-Nachrichtenstruktur).

Dafür werden folgende Anforderungen an die lokale Ausführungsumgebung gestellt:

- Ein Linux-Betriebssystem mit einer Bash-Shell
- Installierte Google Cloud CLI [[Gooa](#)]
- Installiertes Kubernetes Befehlszeilenprogramm kubectl [[Clo](#)]
- Authentifizierung von kubectl durch das gke-gcloud-auth-plugin [[Gooc](#)]
- Installiertes Docker Desktop [[Doc](#)]
- Installiertes Gradle [[Gra](#)]

## 6.1 Generierungstemplate

Das Generierungstemplate erzeugt zunächst Dateien, die nur einmal für die gesamte Architektur existieren müssen. Diese werden genutzt, um die generierten Microservices zu bauen und Deployments auszuführen. Sie befinden sich in einem Verzeichnis namens „bash\_scripts“. Anschließend durchläuft das Template drei Hauptiterationen: Eine über alle Microservices, eine über alle Module und Modellemente und abschließend eine über alle Schnittstellen. Hierbei wird je Service ein eigenes Verzeichnis angelegt, in dem sich nach der Generierung die servicespezifischen Dateien in einer typischen Projektstruktur wiederfinden. Im Folgenden wird die Struktur des Generierungstemplates tabellarisch vorgestellt und kurz auf die erzeugten Klassen eingegangen.

Dateiname	S	M	I	Beschreibung
setup_kubernetes_cluster.sh				Erzeugt ein neues Kubernetes Cluster
setup_docker_registry.sh				Erzeugt eine neue Artifact Repository
check_gcp_and_cluster.sh				Überprüft Verfügbarkeit von GCP und Kubernetes Cluster
setup_kafka.sh				Startet Kafka Container und Kubernetes Service
delete_kafka.sh				Entfernt Kafka Container und Service im Cluster
create_topics.sh				Erzeugt bei laufendem Kafka Container neue Topics
deployment.yml				Kubernetes Deployment für Kafka
whitelist_ip.sh				Setzt IP des Users als einzige erlaubte IP für das Cluster
deploy_services.sh				Führt Deployment aus und konfiguriert Firewall
undeploy_services.sh				Macht Deployment und Firewall rückgängig
add_wrapper.sh				Fügt Gradle Wrapper zu jedem Service hinzu
build_images.sh				Baut Images für Services und pusht diese
healthcheck.sh				Schickt einen GET Request gegen alle Health Endpunkte
get_cluster_info.sh				Gibt Informationen zum Cluster aus
scale_down_cluster.sh				Setzt die Cluster Nodes auf 0 und damit das Cluster inaktiv
scale_up_cluster.sh				Setzt die Cluster Nodes auf 1
cleanup_registry.sh				Entfernt alle Images im Artifact Repository
delete_cluster.sh				Löscht das Kubernetes Cluster
delete_registry.sh				Löscht die Artifact Registry
execute_local.sh				Führt Services lokal aus
cancel_execute_local.sh				Beendet lokal gestartete Microservices
kafka_local.sh				Führt Kafka lokal aus
ApplicationStarter.java	•			Einstiegspunkt für die Spring Boot Anwendung
application.yml	•			Anwendungseigenschaften
build.gradle	•			Build Konfiguration und Abhängigkeitsverwaltung
deployment.yml	•			Servicespezifisches Kubernetes Deployment
build_and_push.sh	•			Baut Anwendungsimage und pusht dieses in die Cloud
Dockerfile	•			Beschreibt Docker Image
HealthCheckController.java	•			Schnittstelle für Healthcheck
WebSecurityConfiguration.java	•			Setzt Schnittstellensicherheit. Definiert Endpunkte public.
WebClientConfig.java	•			Benötigt für die Bean Injection des WebClient.
EntityName.java	•	•		Erzeugt JPA konforme Entity mit ID
ValueObjectName.java	•	•		Erzeugt Klassenrumpf
AggregateName.java	•	•		Erzeugt JPA konformes Aggregate mit ID und Nodes
EntityNodeName.java	•	•		Erzeugt JPA konforme Entity mit ID
ValueObjectNodeName.java	•	•		Erzeugt Klassenrumpf
ServiceName.java	•	•		Erzeugt Service Komponente
FactoryName.java	•	•		Erzeugt Factory Komponente für zugewiesene Klasse
RepositoryName.java	•	•		Erzeugt JPA Repository für zugewiesene Klasse
SyncInterfaceName.java	•		•	Erzeugt einen Spring Controller mit Endpunkten
AsyncInterfaceConsumerName.java	•		•	Erzeugt eine Kafka Consumer als Komponente
AsyncInterfaceProducerName.java	•		•	Erzeugt eine Kafka Producer als Komponente

Tabelle 6.1: Übersicht der generierten Dateien

### Legende:

**S** - Iteration über alle Microservices

**M** - Iteration über alle Module und Modellelemente

**I** - Iteration über alle Schnittstellen

Detailierter soll nun auf eine konkrete Implementierung einer Metamodellklasse eingegangen werden. Dafür wird die Klasse Service näher betrachtet. Diese erwies sich in ihrer Umsetzung als am komplexesten. Im Folgenden wird auf die verwendeten Lösungen näher eingegangen und die Funktionsweise der resultierenden Java Datei angeschnitten.

```

1 [if modelElement.oclIsTypeOf(microserviceMetamodell::Service)]
2 [file ('/' + microservice.serviceName + '/src/main/java/app/' + module.
    moduleName + '/' + modelElement.elementName + '.java', false, 'UTF-8')]
3 package app.[module.moduleName];
4
5 import org.slf4j.Logger;
6 import org.slf4j.LoggerFactory;
7 import org.springframework.stereotype.Service;
8
9 [for( element: ModelElement | modelElement.oclAsType(microserviceMetamodell::
    Service).referencedElements)]
10 import app.[element.eContainer().oclAsType(microserviceMetamodell::Module).
    moduleName/].[element.elementName/];
11 [/for]
12
13 [if modelElement.oclAsType(microserviceMetamodell::Service).sendsRequestTo ->
    notEmpty()]
14 import org.springframework.web.reactive.function.client.WebClient;
15 import reactor.core.publisher.Mono;
16 import org.springframework.beans.factory.annotation.Value;
17 [/if]
18
19 [for( interface: Interface | modelElement.oclAsType(microserviceMetamodell::
    Service).referencedInterfaces)]
20 [if interface.oclIsKindOf(microserviceMetamodell::AsynchronousInterface)]
21     [if interface.oclAsType(microserviceMetamodell::AsynchronousInterface).
        interfaceRole = (microserviceMetamodell::AsynchronousInterfaceRole::PRODUCER
        )]
22 import app.kafka.[interface.interfaceName/];
23     [/if]
24 [/if]
25 [/for]
```

Quellcode 6.1: Importieren anderer Klassen

Der in Quellcode 6.1 dargestellte Code überprüft die Iteration aller Modellelemente auf die Klasse Service und erzeugt diese gegebenenfalls mit der passenden Paketdefinition. Mithilfe von Java-Importausdrücken werden dann die benötigten Klassen eingebunden. Dies wird durch eine Iteration über alle referenzierten Objekte umgesetzt. Weiterhin gibt es Imports, die nicht unbedingt vorhanden sein müssen. Die erforderlichen WebClient-Imports werden beispielsweise nur dann erzeugt, wenn entsprechende Zielschnittstellen existieren. Auch für asynchrone Schnittstellen erfolgt eine bedingte Erzeugung, die davon abhängig ist, ob sie in der erzeugten Klasse verwendet wird. Zusätzlich wird mit einer bedingten Abfrage eingefordert, dass die Schnittstelle vom Typ Producer ist.

Dies liegt daran, dass trotz der allgemeinen Assoziation „referencedInterfaces“ ein Service nur von Producern sinnvoll verwendet werden kann. Dies resultiert aus der Verwendung dieser als sendende Schnittstelle. Empfangende Schnittstellen hingegen werden beim Empfang erst selbst aktiv und müssten dementsprechend den Service referenzieren.

```

1 @Service
2 public class [modelElement.elementName] {
3
4     Logger logger = LoggerFactory.getLogger([modelElement.elementName].class);
5
6     [for( element: ModelElement | modelElement.oclAsType(microserviceMetamodell::Service).referencedElements)]
7     [if not element.oclIsKindOf(microserviceMetamodell::Factorizable)]
8         private final [element.elementName] [element.elementName.toLowerCase()];
9     [/if]
10    [/for]
11
12    [for( interface: Interface | modelElement.oclAsType(microserviceMetamodell::Service).referencedInterfaces)]
13    [if interface.oclIsKindOf(microserviceMetamodell::AsynchronousInterface)]
14        [if interface.oclAsType(microserviceMetamodell::AsynchronousInterface).
15            interfaceRole = (microserviceMetamodell::AsynchronousInterfaceRole::PRODUCER)
16        ]
17        private final [interface.interfaceName] [interface.interfaceName.toLowerCase()];
18    [/if]
19    [/if]
20    [/for]
21
22    [comment]Using the CorrectUrls - As communication targets must be distinct, a
23        more complex approach is needed here[/comment]
24    [if (modelElement.oclAsType(microserviceMetamodell::Service).sendsRequestTo->
25        notEmpty())]
26        private final WebClient webClient;
27
28        [let allServiceNames : Set(String) = modelElement.oclAsType(
29            microserviceMetamodell::Service).sendsRequestTo->collect(restEndpoint |
30                restEndpoint.ancestors(microserviceMetamodell::Microservice)->first().
31                serviceName.toString())->asSet()]
32            [for (service : String | allServiceNames)]
33                @Value("${external-services.[service/.base-url}")
34                    private String [service.replace('-service', '')] baseUrl;
35            [/for]
36        [/let]
37
38    [/if]
```

Quellcode 6.2: Klassenattribute

Quellcode 6.2 zeigt die korrekte Erzeugung der Klassenattribute. Aufgrund des Dependency-Injection-Konzepts von Spring-Anwendungen gibt es keine Klassenattribute für Klassen, die das Factorizable-Interface implementieren. Diese würden bedarfsgesteuert erzeugt werden. Wie bei den Importen werden auch hier Producer nur generiert, wenn sie verwendet werden. Falls ein Service andere Schnittstellen über REST aufrufen möchte, muss ein Attribut für den WebClient gesetzt werden. Ein WebClient sendet Anfragen an die URLs, die in den Anwendungsvariablen, also der application.yml, referenziert sind. Um dies korrekt umzusetzen, ist es notwendig, eine disjunkte Menge zu erzeugen, in der alle Ursprungsservices der referenzierten Endpunkte enthalten sind. Dies wird durch die Filterung der Vorfahrenliste, der Umwandlung in ein einzelnes Element und der Konvertierung in den Typen Set erreicht. Anschließend kann über die resultierende disjunkte Menge aller Servicenamen iteriert werden.

```

1 [comment]The Constructor Parameters for a Service Class - The correct placement
2     of the comma in all cases makes this complex[/comment]
3 [modelElement.elementName]()
4 [for( element: ModelElement | modelElement.oclAsType(microserviceMetamodell::
5     Service).referencedElements)]
6   [if not element.oclIsKindOf(microserviceMetamodell::Factorizable)]
7     [element.elementName/] [element.elementName.toLowerCase()]
8   [if not modelElement.oclAsType(microserviceMetamodell::Service).
9     referencedElements->last().elementName.equalsIgnoreCase(element.elementName)
10    ]
11  ,
12  [/if]
13 [/if]
14 [/for]
15
16 [for( interface: Interface | modelElement.oclAsType(microserviceMetamodell::
17     Service).referencedInterfaces)]
18   [if interface.oclIsKindOf(microserviceMetamodell::AsynchronousInterface)]
19     [if interface.oclAsType(microserviceMetamodell::AsynchronousInterface).
20       interfaceRole = (microserviceMetamodell::AsynchronousInterfaceRole::PRODUCER
21     )]
22     [if modelElement.oclAsType(microserviceMetamodell::Service).
23       referencedElements->notEmpty()]
24       [if not modelElement.oclAsType(microserviceMetamodell::Service).
25         referencedElements->exists(e | e.oclIsKindOf(microserviceMetamodell::
26           Factorizable))]
27         [if modelElement.oclAsType(microserviceMetamodell::Service).
28           referencedInterfaces->first().interfaceName.equalsIgnoreCase(interface.
29             interfaceName)]
30       ,
31       [/if]
32     [/if]
33     [/if]
34   [interface.interfaceName/] [interface.interfaceName.toLowerCase()]
35   [if not modelElement.oclAsType(microserviceMetamodell::Service).
36     referencedInterfaces->last().interfaceName.equalsIgnoreCase(interface.
37       interfaceName)]

```

```

24     ,
25     [/if]
26     [/if]
27     [/if]
28 [/for]
29
30 [if modelElementoclAsType(microserviceMetamodell::Service).sendsRequestTo ->
   notEmpty()]
31 [if modelElementoclAsType(microserviceMetamodell::Service).referencedElements
   ->notEmpty()]
32   [if modelElementoclAsType(microserviceMetamodell::Service).
   referencedInterfaces->select(interface | interfaceoclIsKindOf(
   microserviceMetamodell::AsynchronousInterface))->notEmpty()]
33   ,
34   [/if]
35   [/if]
36   WebClient webClient
37 [/if]
38 )

```

Quellcode 6.3: Konstruktorparameter

Der Code aus den Quellen 8.3 und 8.4 zeigt die korrekte Erzeugung des Klassenkonstruktors. Dabei werden die notwendigen Klassenattribute als Konstruktorparameter übergeben und im Konstruktorrumpf gesetzt. Insbesondere die korrekte Setzung der Kommas erweist sich als äußerst kompliziert. Hierbei muss zunächst bedacht werden, dass ein Komma explizit nicht gesetzt wird, wenn das letzte Element der „referencedElements“ im Konstruktor aufgelistet wurde. Die darauf folgenden Schnittstellen-Konstruktorparameter müssen neben der bereits geschilderten Differenzierung zwischen Producern und Consumern ebenfalls eine besondere Kommasetzung berücksichtigen. Hier darf kein Komma gesetzt werden, wenn der Service ModelElements referenziert oder nur solche referenziert werden, die nicht als Attribute existieren und somit das Factorizable-Interface implementieren. Außerdem muss für den Fall, dass mehrere Producer existieren, eine gesonderte Kommasetzung beachtet werden.

```

1 [comment]Constructor Values [/comment]
2 {
3 [for( element: ModelElement | modelElement.oclAsType(microserviceMetamodell::
4     Service).referencedElements)]
5   this.[element.elementName.toLowerFirst()/] = [element.elementName.
6     toLowerFirst()/];
7 [/for]
8
9 [for( interface: Interface | modelElement.oclAsType(microserviceMetamodell::
10    Service).referencedInterfaces)]
11   [if interface.oclIsKindOf(microserviceMetamodell::AsynchronousInterface)]
12     [if interface.oclAsType(microserviceMetamodell::AsynchronousInterface).
13      interfaceRole = (microserviceMetamodell::AsynchronousInterfaceRole::PRODUCER
14      )]
15     this.[interface.interfaceName.toLowerFirst()/] = [interface.interfaceName.
16       toLowerFirst()/];
17   [/if]
18 [/if]
19 [/for]
20
21 [if modelElement.oclAsType(microserviceMetamodell::Service).sendsRequestTo ->
22   notEmpty()]
23   this.webClient = webClient;
24 [/if]
25 }
```

Quellcode 6.4: Konstruktorrumpf

Der Konstruktorrumpf muss nur die bisher schon beschriebenen Sonderfälle berücksichtigen. Das korrekte Setzen des Semikolons zeigt sich für die Sprache Java in Relation zu den Kommas der Konstruktorparameter als einfacher.

```

1 [comment]Behaviours [/comment]
2 [for ( behaviour: Behaviour | modelElement.oclAsType(microserviceMetamodell::
3     Service).behaviours)]
4   public void [behaviour.behaviourName/](){
5     logger.info("[behaviour.behaviourName/] executed");
6     //TODO: Implement Behaviour
7   }
8 [/for]
```

Quellcode 6.5: Methoden

Für die modellierten Objekte der Klasse Behaviour werden Methodenköpfe generiert. Diese werden standardmäßig mit einem Logging, einem „TODO“ und dem Rückgabetypen „void“ generiert.

```

1 [comment]Calls to other Microservices[/comment]
2 [for (restEndpoint : RestEndpoint | modelElement.oclAsType(
3     microserviceMetamodell::Service).sendsRequestTo)]
4     [let serviceName : String = restEndpoint.ancestors(microserviceMetamodell::
5         Microservice)->first().serviceName.toString()]
6         private Mono<String> request[restEndpoint.path.toString().substring(2) .
7             toUpperFirst() /] () {
8             logger.info("request[restEndpoint.path.toString().substring(2) .
9                 toUpperFirst() /] executed");
10            return webClient.[restEndpoint.httpMethod.toString().toLowerCase() )
11                () (
12                    .uri([serviceName.replace('-service', '')]/BaseUrl + "[
13                        restEndpoint.path/]")
14                    .retrieve()
15                    .bodyToMono(String.class);
16                }
17            [/let]
18        [/for]
19    }
20 [/file]
21 [/if]
```

Quellcode 6.6: WebClient Implementierung

Die Generierung des WebClient-Aufrufs erstellt eine passende Implementierung basierend auf dem Zielservice und dem HTTP-Typ. Es ist zu beachten, dass einmalig der Zielpfad angepasst wird, um den führenden Schrägstrich zu entfernen. Konkretere Aufrufseigenschaften wie Header oder Body wurden nicht in der Generierung berücksichtigt. Abschließend wird mit dem schließenden File-Tag das Ende der generierten Klasse markiert.

Andere zu generierende Klassen wurden ähnlich, jedoch mit geringerer Komplexität umgesetzt. Wiederkehrende AQL-Funktionalitäten umfassen das Prüfen und Casten von Klassen, die Iteration, bedingte Ausdrücke und die Weiterverarbeitung von Ergebnissen, wie Listen, mithilfe von Funktionen wie „notEmpty()“ oder „first()“.

## 6.2 Migration von Diensten durch Generierung

Aufgrund von veränderten Anforderungen oder Aktualisierungen von Bibliotheken, kann es notwendig sein, dass der generierte Quellcode migriert werden muss. Dieses im Software-Lebenszyklus wiederkehrende Problem wirft die Frage auf, wie der bisherige modellgetriebene Entwicklungsansatz damit umgehen könnte. Dabei muss bedacht werden, was mit Code geschieht, der auf Basis der bisher konzipierten Codegenerierung ergänzend hinzugefügt wurde.

### 6.2.1 Acceleo Protected Areas

Ein Ansatz wären geschützte Bereiche, die mit der AQL definiert werden können. Dies lässt sich im folgenden Quellcode betrachten. Mittels des Schlüsselwortes „protected“ werden diese beispielsweise innerhalb von Methoden definiert. Die dadurch markierten Codeblöcke werden nach der initialen Generierung nicht mehr neu erzeugt.

```
1 [comment]Behaviours [/comment]
2 [for ( behaviour: Behaviour | modelElement.oclAsType(microserviceMetamodell::
   Service).behaviours)]
3 // [protected (behaviour.behaviourName)]
4   public void [behaviour.behaviourName/](){
5     logger.info("[behaviour.behaviourName/] executed");
6     //TODO: Implement Behaviour
7   }
8
9 // [/protected]
10 [/for]
11
12 [comment]Calls to other Microservices [/comment]
13 [for (restEndpoint : RestEndpoint | modelElement.oclAsType(
   microserviceMetamodell::Service).sendsRequestTo)]
14   [let serviceName : String = restEndpoint.ancestors(microserviceMetamodell::
   Microservice)->first().serviceName.toString()]
15 // [protected (restEndpoint.path)]
16   private Mono<String> request[restEndpoint.path.toString().substring(2).
   toUpperFirst()]{{
17     logger.info("request[restEndpoint.path.toString().substring(2).
   toUpperFirst()] executed");
18     return webClient.[restEndpoint.httpMethod.toString().toLowerCase()
   /]()
19       .uri([serviceName.replace('-service', '')]/BaseUrl + "[
   restEndpoint.path/]")
20       .retrieve()
21       .bodyToMono(String.class);
22   }}
23 // [/protected]
24 [/let]
25 [/for]
```

```

27 // [protected ('additional protected methods')]
28 // [/protected]
29 }
30 [/file]
31 [/if]

```

Quellcode 6.7: Einsatz von geschützen Bereichen im Generierungstemplate

So würde eine konkrete Service-Klasse, die diese Strategie konsequent umsetzt, nach der Generierung und detaillierteren Implementierung wie folgt aussehen:

```

// Start of user code Imports
import ...
// End of user code
@Service
public class OrderProcessService {
    // Start of user code Members
    private final ObjectMapper objectMapper;
    private final CustomerRepository customerRepository;
    private final OrderRepository orderRepository;
    private final OrderFactory orderFactory;
    private final OrderProducer orderProducer;
    private final WebClient webClient;
    Logger logger = LoggerFactory.getLogger(OrderProcessService.class);
    @Value("${external-services.checkout-service.base-url}")
    private String checkoutBaseUrl;
    // End of user code

    // Start of user code Constructor
    OrderProcessService(ObjectMapper objectMapper, CustomerRepository customerRepository, OrderRepository orderRepository, OrderFactory orderFactory, OrderProducer orderProducer, WebClient webClient, Logger logger) {
        this.objectMapper = objectMapper;
        this.customerRepository = customerRepository;
        this.orderRepository = orderRepository;
        this.orderFactory = orderFactory;
        this.orderProducer = orderProducer;
        this.webClient = webClient;
        this.logger = logger;
    }
    // End of user code

    // Start of user code createOrder
    @Scheduled(cron = "*/20 * * * *")
    private void createBetterOrder() {...}
    // End of user code

    // Start of user code coolNewFeature
    public void coolNewFeature() {...}
    // End of user code

    // Start of user code /processCheckout
    private void requestProcessCheckout(UUID customerUuid) {...}
    // End of user code

    // Start of user code additional protected methods
    private void handleResponse(String response) {...}
    private void handleError(Throwable error, OrderEntity orderEntity) {...}
    private void completeProcessing(OrderEntity orderEntity) {...}
    // End of user code
}

```

Abbildung 6.1: Generierter Code mit geschützten Bereichen

Hierbei ist hervorzuheben, dass die Methode „coolNewFeature()“ erst nach der initialen Codegenerierung dem Modell als Verhalten hinzugefügt wurde. Dabei blieb bestehender Code erhalten. Der durch das Verhalten neu generierte wurde problemlos hinzugefügt. Das Löschen dieses Verhaltens aus dem Modell führt ebenfalls zu einer funktionierenden Entfernung der Methode und deren Implementierung. Das Umbenennen würde hingegen zum Verlust der Implementierung führen.

Alternativ zu diesem feingranulareren Ansatz ist es auch möglich, ganze Klassen als einen einzigen geschützten Bereich zu kennzeichnen. Exemplarisch wurde dies in einem dem Quellcode anhängenden Beispielprojekt [[Löc](#)], welches einen Microservice enthält, für Entity und Repository umgesetzt. In diesem ist ebenso die in Abbildung 6.1 abgebildete Service-Klasse zu finden.

### 6.2.2 Open Rewrite

Eine weitere Möglichkeit, solche Migrationen durchzuführen, ist die Verwendung der Software OpenRewrite [[Mod](#)]. Sie kann über eine Gradle- oder Maven-Build-Konfiguration in ein Projekt eingebunden werden und ermöglicht es, bereits existierende Migrationsrezepte zu nutzen. Diese erlauben einfache Refactorings, wie das Umbenennen eines Pakets, einer Methode oder das automatische Formatieren von Code. Auch komplexere Rezepte, die eine Codebasis vollständig auf eine aktuelle Java-Version migrieren können, existieren. Zudem können eigene Rezepte in Java implementiert werden. Ergänzend zur Implementierung mit AQL-geschützten Bereichen sollen nun exemplarisch einige Rezepte vorgestellt werden. Diese wurden ebenfalls dem Beispielprojekt hinzugefügt.

```
1 ---
2 type: specs.openrewrite.org/v1beta/recipe
3 name: app.ChangeMethod
4 displayName: Change method name example
5 recipeList:
6   - org.openrewrite.java.ChangeMethodName:
7     methodPattern: app..* createOrder()
8     newMethodName: createBetterOrder
9
10 ---
11 type: specs.openrewrite.org/v1beta/recipe
12 name: app.order
13 recipeList:
14   - org.openrewrite.java.ChangePackage:
15     oldPackageName: app.order
16     newPackageName: app.ordernew
```

Quellcode 6.8: OpenRewrite Rezepte

In der rewrite.yml werden neue Namen für das Ändern von Methodennamen und das Umstrukturieren von Paketen innerhalb der Rezepte definiert.

```
1 rewrite {
2     activeRecipe(
3         'app.order',
4         'app.ChangeMethod',
5         'org.openrewrite.java.format.AutoFormat'
6     )
7 }
```

Quellcode 6.9: Anwenden von OpenRewrite Rezepten

In der build.gradle können diese dann mit dem vorher definierten Rezeptnamen referenziert werden. Auch Rezepte wie das automatische Formatieren von Code, die keine weitere Konfiguration verlangen, können hier genutzt werden. Durch das Anwenden des Gradle-Tasks „rewriteRun“ werden diese nun ausgeführt.

Somit können auch speziellere Migrationen, die sich nicht durch Änderungen des Modells umsetzen lassen, durchgeführt werden.

# 7 Anwendung

## 7.1 Modellierung einer Microservice-Architektur für einen Anwendungsfall

### 7.1.1 Vorstellung des Anwendungsfalles

In diesem Kapitel wird eine Softwarelösung für einen Webshop modelliert. Diese Lösung soll es Kunden ermöglichen, sich zu registrieren, Bestellungen durchzuführen, diese abzuwickeln und abschließend an ein Backoffice-System zu übertragen. Zusätzlich sollen bei der Registrierung betrügerische Kunden automatisch erkannt werden. Um das Einkaufsverhalten exemplarisch zu zeigen, soll das Kaufverhalten so implementiert werden, dass Kunden so lange Käufe tätigen, bis sie zwei Bestellungen getätigt haben. Weiterhin soll der Checkout-Prozess mit einer fünfprozentigen Wahrscheinlichkeit fehlschlagen. Das Registrieren eines Kunden, die Erzeugung einer Bestellung und der erfolgreiche Checkout-Prozess sollen in Form eines Events prozessiert werden, und dieses soll dem Backoffice-System mitgeteilt werden.

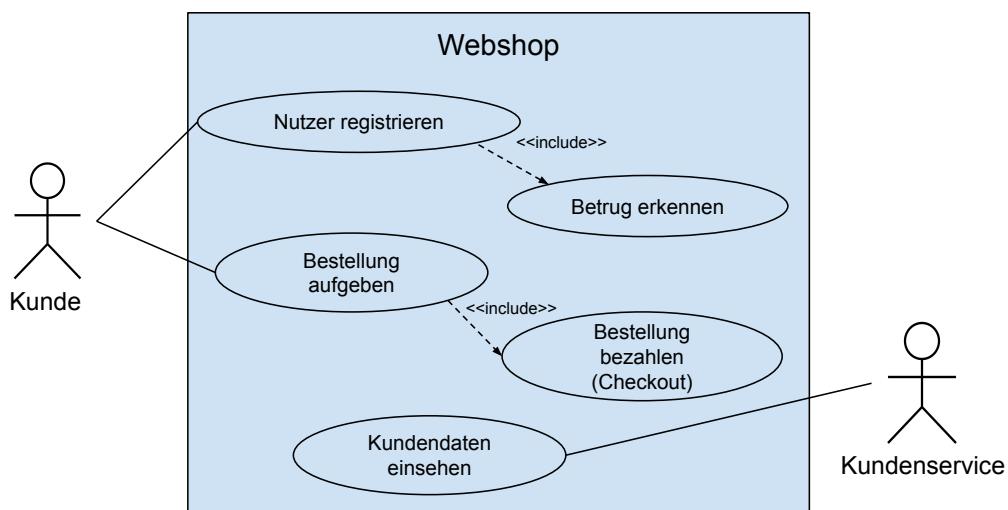


Abbildung 7.1: Use-Case Diagramm für den Webshop-Anwendungsfall

### 7.1.2 Ablauf der Modellierung mit Editor

Nach der Initialisierung des Grundmodells, wie in Kapitel 5.3 definiert, werden mit einer ContextRelationshipDescription erkannte BoundedContexts modelliert. Diesen werden Microservices und DomainModels zugewiesen. Ebenso wird ein SharedKernel „Pricing“ identifiziert, welcher Teil der Kontexte „Order“ und „Checkout“ ist. Eine Abhängigkeit zu einer passenden Bibliothek wird ebenso modelliert. Zusätzlich werden Beziehungen zwischen den BoundedContexts entworfen. Der „Backoffice“-Kontext besitzt als Datensenke ein auf seinen Anwendungszweck als Datenarchiv spezialisiertes Verständnis von Kunden-, Bestellungs- und Checkout-Informationen. Dies kann in der Realität durch ein ERP-System wie SAP oder Salesforce notwendig sein. Der Kundenkontext wird als Published Language gegenüber dem Order Kontext modelliert, was durch das Veröffentlichen einer Beschreibung der Datenstruktur für Kunden umgesetzt werden könnte. Da der Order Kontext aus Gründen der Datensparsamkeit nur ein begrenztes Verständnis von den Daten hat, wird in Form einer Anticorruption Layer der Kunde in ein eigenes Modell übersetzt. Anders hingegen wird der Checkout modelliert, welcher das Modell der Bestellung übernehmen soll.

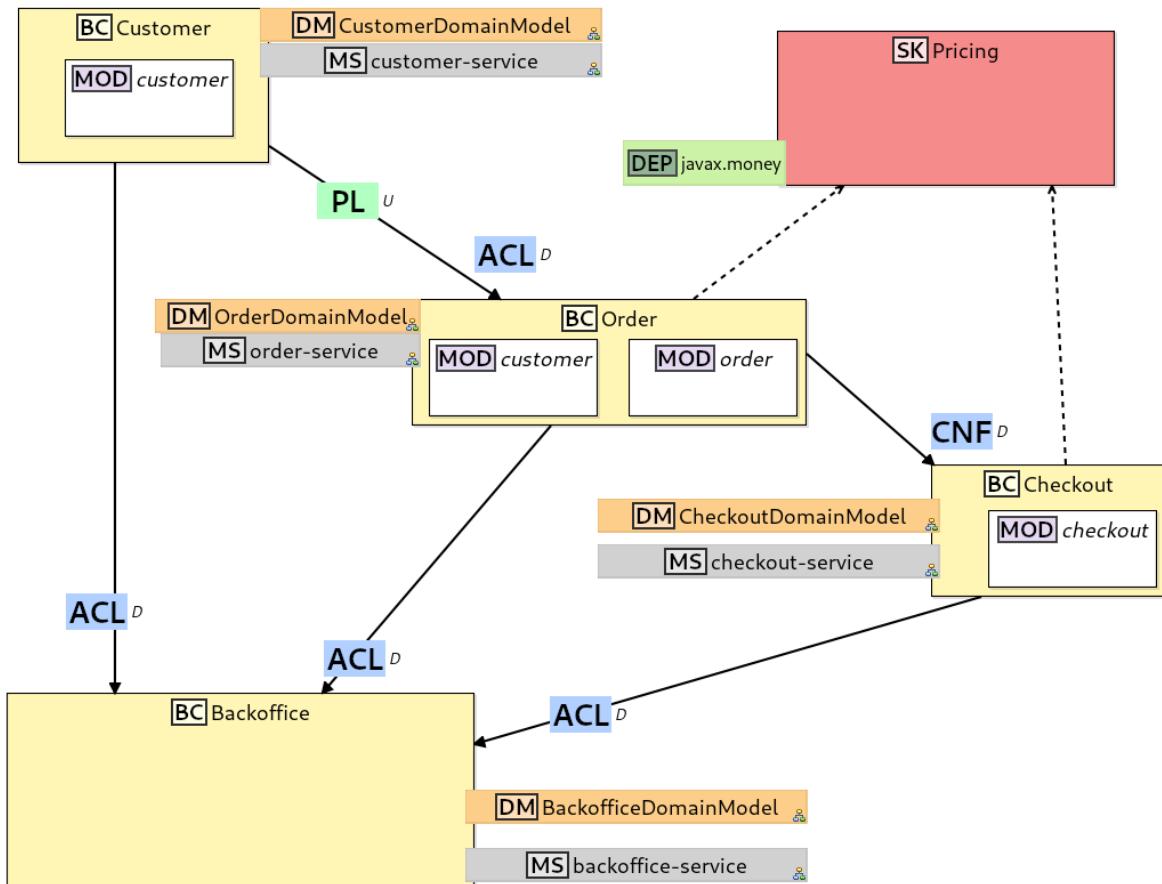


Abbildung 7.2: Modellierung der Bounded Contexts für den Anwendungsfall

Nun werden die einzelnen DomainModels weiter verfeinert. Dem DomainModel des Bounded-Context Order werden zwei Module, „Customer“ und „Order“, hinzugefügt. Eine „CustomerEntity“ beinhaltet die notwendigen Daten eines Kunden und wird von einer „CustomerFactory“ erzeugt sowie von einem „CustomerRepository“ persistiert. Dieses kann Kunden mit weniger als einer variablen Anzahl an Bestellungen finden. Ähnlich wird eine „OrderEntity“ mit einem Repository und einer Factory modelliert, welche bei der Erzeugung einen Bestellwert berechnet. Ein „Order-ProcessService“ erzeugt Bestellungen und fordert einen Checkout-Prozess an. Zusätzlich wird das Event, das bei der Erzeugung einer Bestellung auftritt, angelegt. Das Modellieren anderer Domänen funktioniert nun konzeptionell symmetrisch zu diesem Vorgehen.

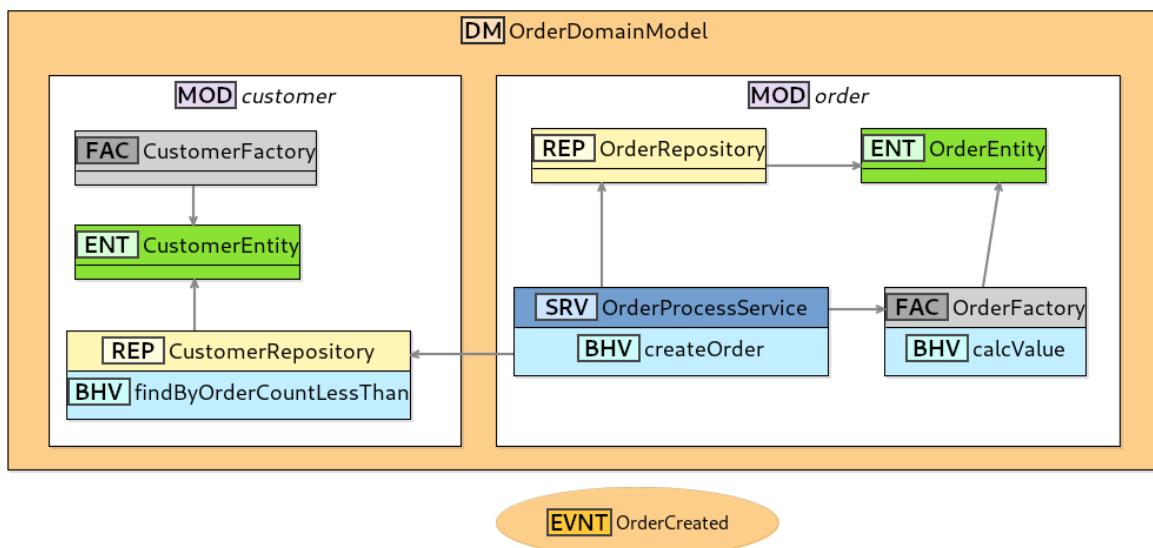


Abbildung 7.3: Modellierung des OrderDomainModel für den Anwendungsfall

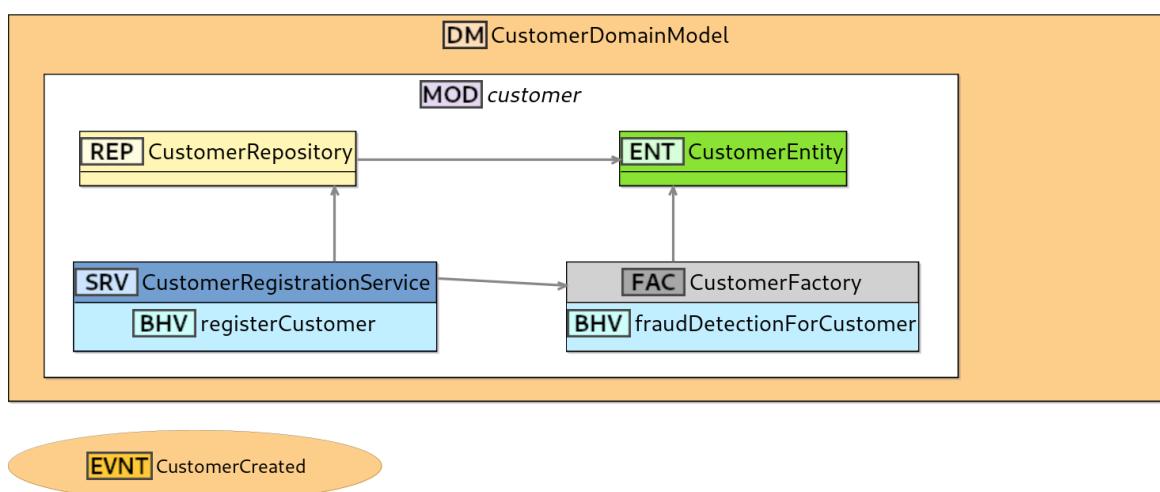


Abbildung 7.4: Konzeptionell symmetrische Modellierung des CustomerDomainModel

In der Übersicht über die Microservice-Kommunikation werden nun passende Schnittstellen für die Microservices angelegt. Mittels asynchroner Schnittstellen werden über Topics Daten übertragen. Die Kommunikation zwischen Order und Checkout hingegen läuft synchron, da eine Bestellung erst abgeschlossen wird, sobald ein Checkout erfolgreich durchgeführt wurde.

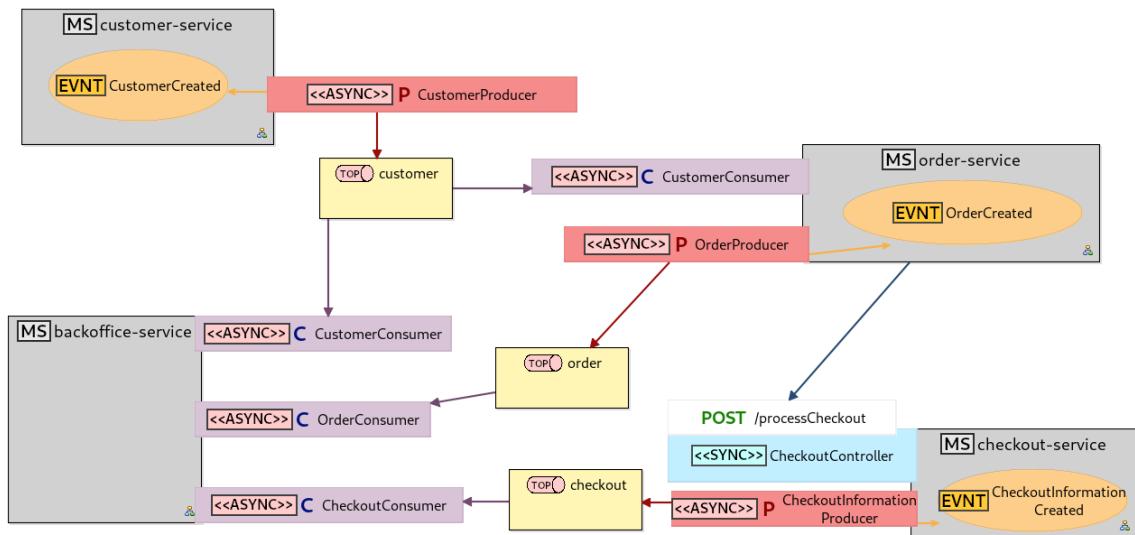


Abbildung 7.5: Modellierung der Kommunikation zwischen Microservices für den Anwendungsfall

In den servicespezifischen Schnittstellenbeschreibungen werden die im vorherigen Modellierungsschritt definierten Schnittstellen den Modellelementen zugewiesen. Hier wird beispielsweise durch den „OrderProcessService“ der Checkout-Service angefragt und mit dem „OrderProducer“ das erfolgreiche Erzeugen einer Bestellung auf das im vorherigen Schritt angelegte Topic geschrieben. Weiterhin werden dem „CustomerConsumer“ die passende Factory zum Erzeugen der Entities und das Repository zum Persistieren zugewiesen.

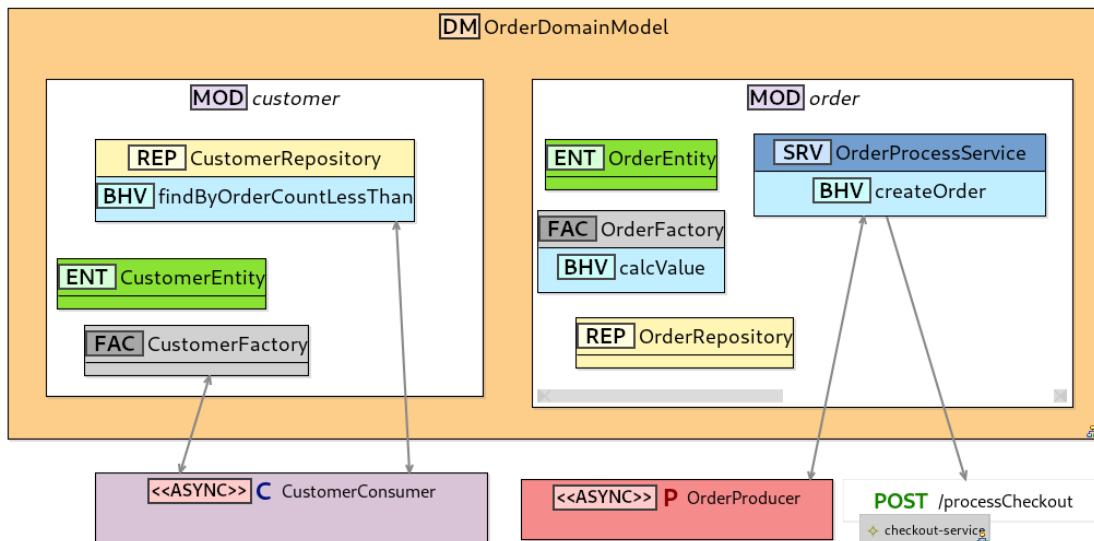


Abbildung 7.6: Modellierung des Schnittstellen auf Serviceebene für den Anwendungsfall

Nun kann abschließend der Cloud-Konfiguration ein Cluster hinzugefügt werden, und diesem wiederum je Service ein Deployment. Abschließend werden die Build-Konfigurationen angelegt und diesen die bereits bei der Analyse der Domäne modellierte Bibliothek „javax.money“ zugewiesen.

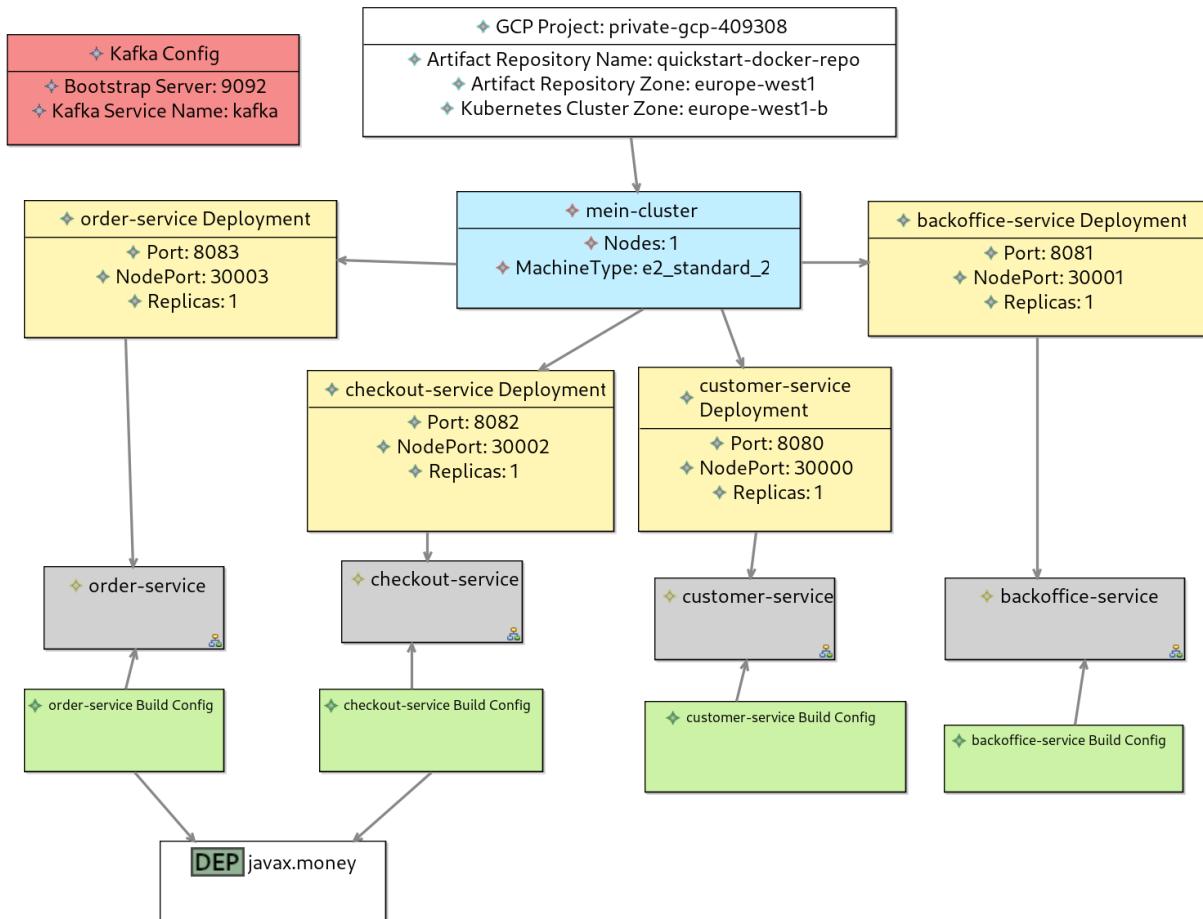


Abbildung 7.7: Modellierung der Infrastruktur für den Anwendungsfall

Somit wurde ein Anwendungsfall mit einer Microservice-Architektur modelliert, die die Konzepte des DDD einbezieht. Weiterhin wurden eine Ausführungsumgebung und die zur Ausführung benötigten Abstraktionen in dieses Architekturmodell einbezogen.

## 7.2 Inbetriebnahme

Mit diesem Modell kann nun der dem Modell entsprechende Code generiert werden. Im Folgenden sollen die abschließenden Schritte beschrieben werden, die nun notwendig sind, um die generierten Projekte in der Cloud auszuführen.

Die modellierten Funktionalitäten müssen zunächst implementiert werden. Dies wird hier exemplarisch anhand der Methode gezeigt, die Bestellungen erzeugt.

```
@Scheduled(cron = "*/20 * * * *")
private void createOrder() {
    logger.info("Starting the process to create a new order");
    var customers = customerRepository.findByOrderCountLessThan(orderCountThreshold: 2);
    if (customers.isEmpty()) {
        logger.info("No customers found with order count less than 2");
        return;
    }
    Random random = new Random();
    var customer = customers.get(random.nextInt(customers.size()));
    logger.info("Processing checkout for customer with UUID: {}", customer.getCustomerUuid());
    requestProcessCheckout(customer.getCustomerUuid());
    customer.setOrderCount(customer.getOrderCount() + 1);
    customerRepository.saveAndFlush(customer);
    logger.info("Updated order count for customer with UUID: {}", customer.getCustomerUuid());
}
```

Abbildung 7.8: Implementieren von fehlenden Methoden vor der Inbetriebnahme

Weiterhin müssen Klassen attribuiert werden. Im Folgenden wird ein Vergleich zwischen einer generierten Entity und einer leicht ausimplementierten Variante gezeigt.

<pre>@Entity public class CustomerEntity {      @Id     private UUID id;      //TODO: Implement the entities fields, getters and setters      public void setId(UUID id) { this.id = id; }      public UUID getId() { return id; } }</pre>	<pre>@Entity public class CustomerEntity {      @Id     private UUID customerUuid;     private String name;     private int orderCount;      public int getOrderCount() {return orderCount;}     public void setOrderCount(int orderCount) {this.orderCount = orderCount;}     public String getName() {return name;}     public void setName(String name) {this.name = name;}     public UUID getCustomerUuid() {return customerUuid;}     public void setCustomerUuid(UUID id) {this.customerUuid = id;} }</pre>
--	--

Abbildung 7.9: Implementieren von fehlenden Klassenattributen vor der Inbetriebnahme

Um das Codegenerierungstemplate lesbar zu gestalten, wurde es durch Verwendung von Tabstopps entsprechend formatiert. Dies führt jedoch zu dem Problem, dass diese mitgeneriert werden, was zu schlecht lesbarem Code führt. Deswegen wird in einem weiteren manuellen Schritt der Code nochmals formatiert. Die folgende Abbildung zeigt exemplarisch die dadurch eingesparte Menge an Codezeilen:

```

1 package app.order;
2
3 import org.slf4j.Logger;
4 import org.slf4j.LoggerFactory;
5 import org.springframework.stereotype.Service;
6
7 import app.customer.CustomerRepository;
8 import app.order.OrderRepository;
9 import app.order.OrderFactory;
10
11 import org.springframework.web.reactive.function.client.WebClient;
12 import reactor.core.publisher.Mono;
13 import org.springframework.beans.factory.annotation.Value;
14
15 import app.kafka.OrderProducer;
16
17 @Service
18 public class OrderProcessService {
19
20     Logger logger = LoggerFactory.getLogger(OrderProcessService.class);
21
22     private final CustomerRepository customerRepository;
23     private final OrderRepository orderRepository;
24     private final OrderFactory orderFactory;
25
26     private final OrderProducer orderProducer;
27
28     private final WebClient webClient;
29
30     @Value("http://checkout-service-k8service:8082")
31     private String checkoutBaseUrl;
32
33     OrderProcessService(
34         CustomerRepository customerRepository,
35
36         OrderRepository orderRepository,
37
38         OrderFactory orderFactory,
39
40         OrderProducer orderProducer,
41
42         WebClient webClient
43
44     ) {
45
46         this.customerRepository = customerRepository;
47         this.orderRepository = orderRepository;
48         this.orderFactory = orderFactory;
49
50         this.orderProducer = orderProducer;
51
52         this.webClient = webClient;
53
54     }
55
56     public void createOrder(){
57         logger.info("createOrder executed");
58         //TODO: Implement Behaviour
59     }
60
61     private Mono<String> requestProcessCheckout() {
62         logger.info("requestProcessCheckout executed");
63         return webClient.post()
64             .uri(checkoutBaseUrl + "/processCheckout")
65             .bodyToMono(String.class)
66             .retrieve().bodyToMono(String.class);
67     }
68 }
69
70
71
72

```

**72 Codezeilen**

```

1 package app.order;
2
3 import ...
4
5 @Service
6 public class OrderProcessService {
7
8     private final CustomerRepository customerRepository;
9     private final OrderRepository orderRepository;
10    private final OrderFactory orderFactory;
11    private final OrderProducer orderProducer;
12    private final WebClient webClient;
13    Logger logger = LoggerFactory.getLogger(OrderProcessService.class);
14
15    OrderProcessService(CustomerRepository customerRepository, OrderRepository orderRepository, OrderFactory orderFactory, OrderProducer orderProducer, WebClient webClient) {
16        this.customerRepository = customerRepository;
17        this.orderRepository = orderRepository;
18        this.orderFactory = orderFactory;
19        this.orderProducer = orderProducer;
20        this.webClient = webClient;
21    }
22
23    public void createOrder() {
24        logger.info("createOrder executed");
25        //TODO: Implement Behaviour
26    }
27
28    private Mono<String> requestProcessCheckout() {
29        logger.info("requestProcessCheckout executed");
30        return webClient.post()
31            .uri(checkoutBaseUrl + "/processCheckout")
32            .bodyToMono(String.class)
33            .retrieve().bodyToMono(String.class);
34    }
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72

```

**47 Codezeilen**

Abbildung 7.10: Abschließende Code-Formatierung für bessere Lesbarkeit

Nachdem alle notwendigen Anpassungen durchgeführt wurden, können mit den generierten Bash-Skripten für die Dienste Deployments ausgeführt werden. Zu beachten ist bei den Skripten die möglicherweise notwendige Anpassung des Wurzelverzeichnisses. Dieses sollte auf das Verzeichnis verweisen, welches die generierten Dienste enthält. Durch die folgende Ausführung können dann die Dienste ausgerollt werden.

Schritt	Beschreibung
1	Erzeugen von Cluster und Artifact Repository, wenn noch nicht vorhanden.
2	Whitelisten der eigenen IP
3	Deployment des Kafka-Images
4	Erzeugen der Kafka-Topics
5	Erzeugen der Gradle-Wrapper
6	Bauen der Images
7	Deployment der Services

Tabelle 7.1: Deployment-Schritte für die Software

Damit wäre die modellierte Microservice-Architektur ausgeliefert und kann sowohl über Logs beobachtet, als auch über direkte HTTP-Anfragen angesprochen werden. Somit wurde mittels eines modellgetriebenen Softwareentwicklungsansatzes eine lauffähige Microservice-Architektur modelliert, entwickelt und bereitgestellt.

Der vollständige Quellcode, der das in dieser Arbeit konzipierte Metamodell, die Definition der konkreten Syntax, das Generierungstemplate und verschiedene Beispielprojekte umfasst, liegt dieser Arbeit bei. Weiterhin ist dieser öffentlich in einem GitHub-Repository [[Löa](#)] einsehbar (Stand: 30. Januar 2024).

## 8 Fazit und Ausblick

Das entworfene Metamodell für Microservices kann nun genutzt werden, um basierend auf einem modellgetriebenen Entwicklungsansatz Software zu entwickeln. Dadurch wurde die grundlegende Zielsetzung erfolgreich erfüllt. Weiterhin sind diese Anwendungen auch mit relativ geringem Mehraufwand in eine Cloud-Umgebung auslieferbar. Dementsprechend konnte gezeigt werden, dass die einer Microservice-Architektur zugrunde liegende Infrastruktur, in diesem Fall eine Kubernetes-Architektur in der Google Cloud, berücksichtigt werden kann. Es wurde außerdem gezeigt, dass Microservices auch in einem Deployment-fähigen Zustand modellgetrieben entwickelt werden können.

Trotzdem muss berücksichtigt werden, dass einige vereinfachende Annahmen und Architekturentscheidungen zugunsten dieser Ergebnisse getroffen wurden. Einige davon lassen sich dabei durchaus mit den selbst definierten Entscheidungs- und Bewertungskriterien als sinnvoll und nützlich bewerten. So ist die pragmatische Lösung, ein Microservice-Anwendungsimage auf maximal einen Pod laufen zu lassen, eine Vereinfachung, die für das Lösen von Anwendungsfällen hinreichend ist. Auch die sehr einfache Modellierung des Messaging-Systems Kafka hat sich als ausreichend für den erfolgreichen Betrieb einer Microservice-Architektur erwiesen. Kritischer sind Modellierungsentscheidungen wie das Fehlen einer abstrakten Messaging-Konfigurationsklasse oder der Google Cloud-spezifischen Maschinentyp-Modellierung innerhalb der Kubernetes-Konfigurationsklasse einzuordnen.

Der tatsächliche Mehrwert dieser konzeptionell breiten Modellierung kann ebenfalls kritisch hinterfragt werden. Es ist notwendig, jeden einzelnen Service in einer der Generierung folgenden, feineren Implementierung zu vervollständigen. Dann müssen diese über die erzeugten Skripte manuell ausgerollt werden. Es stellt sich die Frage, inwiefern sich der Aufwand durch den generierten Code reduziert bzw. sogar erhöht. Wie würde sich dies bei einer hochskalierenden Anzahl an Diensten entwickeln? Durch die Erfassung von Software-Qualitätsmetriken ließe sich dies beispielsweise messen. Auch die Möglichkeiten von erhöhter Automatisierung, wie beispielsweise das Modellieren von Deployment-Pipelines und Infrastructure as Code, könnten in diesem Zusammenhang vielversprechende Erweiterungen des Metamodells sein.

Weiterhin wurde versucht, die Eigenschaften von Microservice-Architekturen in das Metamodell einfließen zu lassen. So konnte erfolgreich das Konzept des Domain-Driven Designs in ein Metamodell integriert werden. Es zeigt sich, dass die Abstraktion des DDD gut durch eine konkrete Syntax ausgedrückt werden kann. Diese eignet sich insbesondere sehr gut um Problemdomänen zu analysieren. Die direkte Verbindung zu den technischen Microservice-Konzepten vereinfacht daraufhin die ersten Schritte des Grobentwurfs. Insbesondere die Veranschaulichung der ver-

teilten Systemarchitektur und der genutzten Kommunikationskanäle konnte effektiv umgesetzt werden. Weiterhin zeigt sich der Vorteil des Model-Driven Designs, da die Symmetrien zwischen Codebasis und Modell beachtlich sind. Aber auch hier können wieder verschiedene vereinfachende Abstraktionsschritte thematisiert werden. So wurden viele tiefergehende Konzepte, insbesondere aus dem Bereich des Betriebs wie Monitoring oder Metriken, aber auch Aspekte der Datenbankimplementierung, nicht in das Modell integriert. Dies zeigte sich zwar als nicht unmittelbar notwendig, um lauffähige Anwendungen zu entwickeln, die Problemstellungen lösen. Jedoch könnte man auch argumentieren, dass diese Bereiche sind, die zumindest in einem Modell, das Anspruch auf Vollständigkeit erheben würde, notwendig wären.

Bei der Modellierung des Strategic Designs aus dem DDD kann man auch den dadurch entstandenen Mehrwert kritisch diskutieren. So ist im Rahmen der Problemerfassung und -modellierung ein Verständnis und die Visualisierung der unterschiedlichen Kontexte und ihrer Beziehung zueinander sicherlich hilfreich. Aber der effektive Mehrwert bei einer Codegenerierung konnte während der Konzeption nicht festgestellt werden. Dies lag insbesondere an der Entscheidung, die Attributierung der Modellelemente nicht intensiv zu berücksichtigen. Die Fragestellung, ob eine Modellierung des Strategic Designs im Kontext der Codegenerierung gewinnbringend eingesetzt werden kann, bleibt offen.

Auch die Modellierung von Entities, Value Objects und insbesondere Aggregates ist zu hinterfragen. Die modellierten Assoziationen, die diese Klassen besitzen, wurden im Rahmen der Generierung nicht effektiv aufgegriffen. Die Entscheidung die innere Struktur dieser Klassen händisch zu implementieren zeigte sich in der Umsetzung als eingängiger. Weiterhin existieren andere Metamodelle, die sich auf die Modellierung von Datenstrukturen spezialisieren. Diese aufzugreifen, wäre möglicherweise eine bessere Lösung.

Die Entscheidung, das Metamodell in Schichten zu strukturieren, bot zwar den Vorteil einer klaren Grundstruktur. Retrospektiv zeigten sich jedoch auch Schwächen. Anfangs wurden einige konzeptionelle Gemeinsamkeiten bei der initialen Abstraktion auf Basis eines Modells erkannt. Dabei entwickelte sich die anfangs als technische Schicht abstrahierte Ebene jedoch überraschend anders als erwartet. Es zeigte sich, dass eine zwischenzeitliche explizite Modellierung der fachlichen Interpretation des MDD und der zusätzlichen Interpretation auf Implementierungsebene nicht notwendig war. Dies führte dazu, dass eine Schicht entstand, deren konzeptioneller Kern eher die Kommunikation umfasste. Es stellt sich die Frage, welche Schichteneinteilung geeigneter gewesen wäre. Insbesondere könnte eine ergänzende vertikale Schichtbildung, zusätzlich zur bisherigen horizontalen Einteilung in Fachlichkeit, technische Umsetzung und Infrastruktur, sehr aufschlussreich sein. Eine solche Abstraktion, die den einzelnen Bounded Context, dessen Service und dessen Infrastruktur als eine Komponente begreift, sowie eine weitere Abstraktion, die Gruppen solcher Komponenten beschreibt, könnte interessante Ergebnisse liefern.

Durch die konzipierte Codegenerierung wurde demonstriert, dass Microservice-Anwendungen modellgetrieben erstellt werden können, wobei viele Vorteile der modellgetriebenen Softwareentwicklung genutzt werden. Der modellzentrierte Ansatz ermöglicht insbesondere bei umfassenden

und grundlegenden Änderungen, die eine große Anzahl an Services betreffen, eine effektive Umsetzung. Dies gilt vor allem, wenn die Entwicklung von Anfang an modellgetrieben erfolgt, da auf diese Weise manuell hinzugefügter Code geschützt werden kann. Allerdings bedarf die konkrete Umsetzung der Generierung einer genaueren Betrachtung. Es bestehen signifikante Abhängigkeiten zwischen der Generierung und dem Metamodell. So hat sich gezeigt, dass die Art und Weise, wie das Spring Framework Objekte instanziert, in Kombination mit dem konzipierten Metamodell zu äußerst komplexen und kognitiv anspruchsvollen Templates führen kann.

Die Handhabung von Kafka Producern und Consumern mit Spring wirft die Frage auf, ob eine abstrakte bidirektionale Referenz zwischen Modellelementen und Interfaces auf der Generierungsebene passend ist. Dies liegt an der Dependency Injection, die dazu führt, dass Modellelemente eine klar gerichtete Assoziation zu Produzenten und eine Assoziation zu ihnen von Konsumenten haben. Hier wäre es interessant herauszufinden, wie eine Generierung auf Basis dieses Metamodells mit einer anderen Implementierungstechnologie umgesetzt werden würde. Auch könnte man retrospektiv fordern, dass die Modellierung darauf verzichtet, technologieunabhängige Dienste zu generieren. Dafür würde man die technologiespezifischen Besonderheiten, zum Beispiel die eines Spring Frameworks, bei der Abstraktion berücksichtigen.

Im speziellen Kontext von Migrationen wurde einerseits gezeigt, wie Refaktorisierungen mit diesem Metamodell prinzipiell umgesetzt werden können. Hierbei liegen die Stärken insbesondere in der Veränderung der Grundstruktur von Anwendungen. So lassen sich beispielsweise Änderungen, die sich in Konfigurationsdateien oder Java-Klassen durch das Hinzufügen bzw. Entfernen bestehenden Codes realisieren lassen, gut in einem modellgetriebenen Ansatz umsetzen. Schwieriger gestaltet sich dies bei Änderungen, die bereits feinimplementierten Quellcode betreffen. Hierfür können zwar existierende Technologien wie OpenRewrite integriert werden, jedoch ist eine Nutzung dieser auch unabhängig von einem modellgetriebenen Einsatz denkbar.

Es kann auch notwendig sein, eine Migration einer noch höheren Größenordnung durchzuführen, wie etwa wenn eine Anwendung auf eine völlig neue Technologie umgestellt werden soll. Dabei bringt der modellgetriebene Ansatz, insbesondere zur initialen Codegenerierung, einige interessante Aspekte mit sich, da vor allem das fachliche Modell erhalten bleibt. Auch hier wären Erkenntnisse durch ergänzende Konzepte, die dieses Modell mit einer anderen Zieltechnologie umsetzen, nützlich.

Zusammenfassend lässt sich festhalten, dass das in dieser Arbeit konzipierte Modell für Anwendungen in einem überschaubaren Rahmen solide Ergebnisse liefert, aber die inhaltliche Tiefe von Microservice-Architekturen, der Betrieb dieser in verschiedenen Cloud-Umgebungen und die Fragestellung der effektiven Nutzung für Migrationen nur angeschnitten werden konnten. Zukünftige wissenschaftliche Arbeiten könnten an diesen Ansätzen anknüpfen und auf den in dieser Arbeit gewonnenen Erkenntnissen und Ideen für eine potenziell effektivere Modellierung und Generierung aufbauen.



# Literaturverzeichnis

- [BBHE23] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes: eine kompakte Einführung*. dpunkt.verlag, Heidelberg, 3., aktualisierte und erweiterte auflage edition, 2023. Brendan Burns, Joe Beda, Kelsey Hightower, Lachlan Evenson.
- [BD10] Bernd Brügge and Allen Henry Dutoit. *Object-oriented software engineering: using UML, patterns, and Java*. Pearson, Boston, 3. ed., international ed. edition, 2010. Bernd Bruegge & Allen H. Dutoit.
- [BGO<sup>+</sup>16] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93, jan 2016. <https://doi.org/10.1145/2898442.2898444>.
- [Clo] Cloud Native Computing Foundation. Installieren und konfigurieren von kubectl. <https://kubernetes.io/de/docs/tasks/tools/install-kubectl/>, Zugriff am 22.01.2024.
- [Das17] Sebastian Daschner. *Architecting modern Java EE applications: designing light-weight, business-oriented enterprise applications in the age of cloud, containers, and Java EE 8*. Packt, Birmingham, 2017. Sebastian Daschner ; Foreword by: David Blevins, Founder and CEO, Tomitribe.
- [Doc] Docker, Inc. Install Docker Desktop on Linux. <https://docs.docker.com/desktop/install/linux-install/>, Zugriff am 22.01.2024.
- [Dow18] Herbert Dowalil. *Grundlagen des modularen Softwareentwurfs: der Bau langlebiger Mikro- und Makro-Architekturen wie Microservices und SOA 2.0*. Hanser, München, 2018. Herbert Dowalil.
- [EPM16] OCCI-WG Andy Edmonds, Alexander Papaspyrou, and Thijs Metsch. Open cloud computing interface-core. *Update*, page 6, 2016.
- [Erl17] Thomas Erl. *Service-oriented architecture: analysis and design for services and microservices*. The Prentice Hall service technology series from Thomas Erl. Prentice Hall ; Service Tech Press, Boston, second edition edition, 2017. Thomas Erl ; with contributions by Paulo Merson and Roger Stoffers.
- [Eva07] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*.

- Addison-Wesley, Upper Saddle River, NJ [u.a.], 11. printing edition, 2007. Eric Evans.
- [Fow17] Susan J. Fowler. *Production-ready microservices: building standardized systems across an engineering organization*. O'Reilly, Beijing, first edition edition, 2017. Susan J. Fowler.
- [FR14] Roy T. Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Authentication. RFC 7235, June 2014.
- [Gooa] Google LLC. Google Cloud CLI installieren. <https://cloud.google.com/sdk/docs/install-sdk?hl=de#linux>, Zugriff am 22.01.2024.
- [Goob] Google LLC. Google Cloud-Dokumentation. <https://cloud.google.com/docs?hl=de>, Zugriff am 22.01.2024.
- [Gooc] Google LLC. Here's what to know about changes to kubectl authentication coming in GKE v1.26. <https://cloud.google.com/blog/products/containers-kubernetes/kubectl-auth-changes-in-gke?hl=en>, Zugriff am 22.01.2024.
- [Gra] Gradle Inc. Installing Gradle. <https://docs.gradle.org/current/userguide/installation.html>, Zugriff am 22.01.2024.
- [Kho] Vladik Khononov. What Is Domain-Driven Design? <https://www.oreilly.com/library/view/what-is-domain-driven/9781492057802/ch04.html>, Zugriff am 22.01.2024.
- [KZ20] Stefan Kapferer and Olaf Zimmermann. Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. pages 299–306, 01 2020. <https://www.scitepress.org/Papers/2020/89105/89105.pdf>.
- [LF14] James Lewis and Martin Fowler. Microservices - a definition of this new architectural term, 2014. <https://martinfowler.com/articles/microservices.html>, Zugriff am 01.11.2023.
- [Löa] Patrik Löffler. GitHub-Repo. <https://github.com/Tinkerbell0807/ba/>, Zugriff am 30.01.2024.
- [Löb] Patrik Löffler. Microservice-Metamodell EMF Modellvorlage. <https://github.com/Tinkerbell0807/ba/tree/main/TemplateModel>, Zugriff am 29.01.2024.
- [Löc] Patrik Löffler. Microservice-Metamodell Migrations-Beispielprojekt. [https://github.com/Tinkerbell0807/ba/tree/main/Beispielservice\\_Protected\\_Blocks/order-service](https://github.com/Tinkerbell0807/ba/tree/main/Beispielservice_Protected_Blocks/order-service), Zugriff am 29.01.2024.
- [Lö23] Patrik Löffler. Entwicklung einer Domänenpezifischen Sprache für Webdienste. 2023.

- [Mod] Moderne. OpenRewrite Webpage. <https://docs.openrewrite.org/>, Zugriff am 29.01.2024.
- [Nat] National Institute of Standards and Technology. The NIST Definition of Cloud Computing. <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-145.pdf>, Zugriff am 22.01.2024.
- [New15] Sam Newman. *Microservices: Konzeption und Design*. mitp-Verl., [Frechen], 1. aufl. edition, 2015. Sam Newman. Übers. aus dem Amerikan. von Knut Lorenzen.
- [PBG11] Torsten Posch, Klaus Birken, and Michael Gerdom. *Basiswissen Softwarearchitektur: verstehen, entwerfen, wiederverwenden*. dpunkt-Verl., Heidelberg, 3., aktualisierte und erw. aufl. edition, 2011. Torsten Posch; Klaus Birken; Michael Gerdom.
- [rem11] *Model Driven SOA: anwendungsorientierte Methodik und Vorgehen in der Praxis*. Xpert.press. Springer, Berlin, 2011. Gerhard Rempp; Mark Akermann; Martin Löffler; Jens Lehmann. Mit Ill. von Martin Starzmann.
- [RF21] Mark Richards and Neal Ford. *Handbuch moderner Softwarearchitektur: Architekturenstile, Patterns und Best Practices*. O'Reilly, Heidelberg, 1. auflage edition, 2021. Mark Richards und Neal Ford ; deutsche Übersetzung von Jørgen W. Lang.
- [Rit18] Pierluigi Riti. *Pro DevOps with Google Cloud Platform: with Docker, Jenkins, and Kubernetes*. For professionals by professionals. Apress, [Berkeley, California], 2018. Pierluigi Riti.
- [Rit23] Rafael Ritter. Domain event: The missing building block. <https://ilegra.com/blog/domain-event-the-missing-building-block/>, 2023. Zugriff am 19.11.2023.
- [RSS] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Ajil tool suite repository. <https://github.com/SeelabFhdo/Ajil>, Zugriff am 17.01.2024.
- [RSS18] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design: A model-driven perspective. *IEEE Software*, 35(3):36–43, 2018. <https://ieeexplore.ieee.org/document/8354426>.
- [SIA19] Julio Sandobalin, Emilio Insfran, and Silvia Abrahão. Argon: A model-driven infrastructure provisioning tool. pages 738–742, 2019. <https://ieeexplore.ieee.org/abstract/document/8904494>.
- [Sim18] Michael Simons. *Spring Boot 2: moderne Softwareentwicklung mit Spring 5*. dpunkt.verlag, Heidelberg, 1. auflage edition, 2018. Michael Simons.
- [Som20] Ian Sommerville. *Engineering software products: an introduction to modern software engineering*. Pearson, Hoboken, NJ, 2020. Ian Sommerville.
- [Ste20] Daniel Stender. *Cloud-Infrastrukturen: das Handbuch für DevOps-Teams und Administratoren*. Rheinwerk computing. Rheinwerk Computing, Bonn, 1. auflage edition, 2020. Daniel Stender.

- [SWLH21] Chang-ai Sun, Jing Wang, Zhenxian Liu, and Yanbo Han. A variability-enabling and model-driven approach to adaptive microservice-based systems. pages 968–973, 2021. <https://ieeexplore.ieee.org/abstract/document/9529482>.
- [VMwa] VMware. Spring boot reference documentation. <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>, Zugriff am 22.01.2024.
- [VMwb] VMware. Spring framework documentation. <https://docs.spring.io/spring-framework/reference/index.html>, Zugriff am 22.01.2024.
- [Wol16] Eberhard Wolff. *Microservices: Grundlagen flexibler Softwarearchitekturen*. dpunkt.verlag, Heidelberg, 1., korrigierter nachdruck edition, 2016. Eberhard Wolff.
- [ZCM19] Faiez Zalila, Stéphanie Challita, and Philippe Merle. Model-driven cloud resource management with occiware. *Future Generation Computer Systems*, 99:260–277, 2019. <https://www.sciencedirect.com/science/article/pii/S0167739X18306071>.