Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Automatisierte Aggregation von Musterimplementierungen

Marcel Graf

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Karoline Wild, M.Sc. Michael Falkenthal, M.Sc.
Beginn am:	22. April 2020
Beendet am:	17. Dezember 2020

Kurzfassung

Christopher Alexander publizierte 1977 erste Muster (Patterns) für die Stadt- und Gebäudearchitektur. Mittlerweile sind Muster in der Softwarearchitektur und -entwicklung etabliert und weitverbreitet. Muster beschreiben abstrakte und bewährte Lösungen für wiederkehrende Problemstellungen. Im Fall von Architekturentwürfen von Softwaresystemen, stehen Muster in verschiedenen Beziehungen zueinander. Um dies abbilden zu können, wird ein Entwurfsmodell entwickelt, welches aus Musterinstanzen und ihren Beziehungen besteht. Werden auf diese Weise Softwaresysteme modelliert, so ist oft eine manuelle Umsetzung in eine konkrete Programmiersprache oder Technologie erforderlich. Zur besseren Wiederverwendbarkeit wird ein Konzept vorgestellt um die Verbindung zwischen Muster und Musterimplementierung sowie die Verbindung zwischen den Musterimplementierungen abzubilden. Basierend auf einer benutzerdefinierten Selektion, die eine Abbildung von Musterinstanzen auf Musterimplementierungen darstellt, sollen die Musterimplementierungen automatisiert aggregiert werden. Hierfür werden die Eigenschaften von Musterimplementierungen und Aggregationsoperatoren betrachtet. Insbesondere wird die Eigenschaft berücksichtigt, dass Entwurfsmodelle einem Graphen entsprechen. Daraus wird ein Konzept für die Aggregation entwickelt. Um die Anforderungen zu erarbeiten und abschließend den Prototyp zu validieren, werden drei Referenzszenarien aus den Cloud Computing Patterns und den Enterprise Integration Patterns definiert. Das Konzept wird in einem Prototypen umgesetzt, basierend auf Pattern Atlas. Dazu wird ein Editor für Entwurfsmodelle und Aggregationsoperatoren zur automatisierten Aggregation von Musterimplementierungen implementiert. Dies umfasst eine Implementierung für die Aggregation von Musterimplementierungen zu Amazon Web Services Cloud Formation Templates und Implementierungen für die Aggregation von Apache ActiveMQ Konfigurationen. Anhand des Prototyps wird demonstriert, dass eine automatisierte Aggregation von Musterimplementierungen von Mustern mehrerer Mustersprachen möglich ist.

Abstract

Christopher Alexander published the first patterns for urban and building architecture in 1977. In the meantime, patterns have become established and widespread in software architecture and development. Patterns describe abstract and proven solutions for recurring problems. In the case of architectural designs of software systems, patterns are related to each other in different ways. In order to be able to represent this, a design model is developed, which consists of pattern instances and their relations. When software systems are modeled in this way, a manual conversion into a concrete programming language or technology is often necessary. For better reusability, a concept is presented to map the connection between pattern and concrete solution as well as the connection between concrete solutions. Based on a user-defined selection, which is a mapping from pattern instances to concrete solutions, the concrete solutions are to be aggregated in an automated way. For this purpose, the properties of concrete solutions and aggregation operators are considered. In particular, the fact that design models correspond to a graph is considered. From this, a concept for aggregation is developed. To elaborate the requirements and finally validate the prototype, three reference scenarios from the Cloud Computing Patterns and the Enterprise Integration Patterns are defined. The concept is implemented in a prototype based on Pattern Atlas. For this purpose, a design model editor and aggregation operators for automated aggregation of concrete solutions are implemented. This includes an implementation for aggregating concrete solutions to Amazon Web Services Cloud Formation Templates and implementations for aggregating Apache ActiveMQ configurations. The prototype is used to demonstrate that automated aggregation of concrete solutions of patterns from multiple pattern languages is possible.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Listings

Verzeichnis der Algorithmen

Abkürzungsverzeichnis

API Programmierschnittstelle. 29

AWS Amazon Web Services. 16

CCP Cloud Computing Patterns. 15, 20

DSL Domänenspezifische Sprache. 32

EIP Enterprise Integration Patterns. 15, 21

GCP Google Cloud Platform. 20

HTTP Hypertext Transfer Protocol. 52

IaaS Infrastructure as a Service. 20

JSON JavaScript Object Notation. 29

JVM Java Virtuelle Maschine. 20

MIA Musterimplementierungsartefakt. 22, 40

MID Musterimplementierungsdeskriptor. 22, 40

MIME Multipurpose Internet Mail Extensions. 56

MOM Message-oriented Middleware. 21

PaaS Platform as a Service. 20

REST Representational State Transfer. 51

SaaS Software as a Service. 20

SQL Structured Query Language. 51

URI Uniform Resource Identifier. 40

URL Uniform Resource Locator. 41

VM Virtuelle Maschine. 29

XML Extensible Markup Language. 32

YAML YAML Ain't Markup Language. 29

1. Einleitung

Muster (Patterns) beschreiben abstrakte und bewährte Lösungen für wiederkehrende Problemstellungen. Ursprünglich entwickelte Christopher Alexander erste Muster für die Architektur von Städten und Gebäuden [AIS+77]. Beispielsweise hat Alexander beschrieben, welche Aspekte bezüglich der Positionierung beachtet werden sollten, wenn man plant ein Ladengeschäft zu eröffnen. Mit dem Aufkommen der objektorientierten Softwareentwicklung wurde dieses Konzept auch im Bereich der Softwareentwicklung übernommen. Unter anderem publizierten Gamma et al. [GHJV95] das viel beachtete Buch „Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software“. Heute sind Muster in der Softwarearchitektur und Softwareentwicklung weitverbreitet. Beispielsweise publizierten Hohpe et al. [HW04] die Enterprise Integration Patterns (EIP), eine Sammlung von Mustern zur Anwendungsintegration mittels Nachrichtenübermittlung. Fehling et al. [FLR+14] beschreiben mit den Cloud Computing Patterns (CCP) bewährte Konzepte, um ein grundlegendes Verständnis der Domäne des Cloud Computing und insbesondere der Architektur von cloudnativen Applikationen zu vermitteln.

1.1. Motivation

Die Eigenschaft von Mustern, abstrakte Lösungen auf einer konzeptionellen Ebene zu sein, bietet direkte Vorteile. Die Wiederverwendung und die Austauschbarkeit von konkret genutzten Systemen wird durch die Unabhängigkeit von den eingesetzten Softwareprodukten, Anbietern und Technologien gestützt. Gleichzeitig ist diese Eigenschaft ein Nachteil. Werden Softwaresysteme mit Mustern modelliert, so müssen diese oft manuell in Konfigurationen oder Implementierungen übersetzt werden [FBB+14b]. Diese manuelle Anwendung erfordert wiederum Wissen über die Muster, ihrer Konkretisierung und die Implementierung [FBB+14b]. Dies erfordert einen hohen personellen Aufwand für die Umsetzung [FBB+14b]. Gleichzeitig stellen manuelle Umsetzungen eine mögliche Fehlerquelle dar. Fehler können beispielsweise durch fehlendes Wissen über das Muster oder der zu implementierenden Technologie entstehen. Ebenso durch nicht bedachte Seiteneffekte oder übersehene Randbedingungen.

Falkenthal et al. schlagen deshalb Musterimplementierungen vor, die mit den Mustern verknüpft sind [FBB+14a]. Dies sind konkrete Implementierungen von Mustern, umgesetzt für spezifische Technologien, Programmiersprachen oder Softwaresysteme. Musterimplementierungen ermöglichen die Wiederverwendung existierender Implementierungen, wodurch der Aufwand für die Anwendung von Mustern reduziert wird. Dieser Ansatz adressiert die genannten Nachteile. Der Aufwand für die Umsetzung wird reduziert, da die Implementierung bereits vorhanden ist. Die Wiederverwendung der Musterimplementierungen vermeidet Fehler bei der manuellen Implementierung.

Um Musterimplementierungen zu integrieren, kommen Aggregationsoperatoren zum Einsatz. Diese spezifizieren wie bestimmte Musterimplementierungen zu einer Lösung zusammengefügt werden können. Falkenthal et al. [FBBL19] beschreiben die mathematischen Eigenschaften von Aggregationsoperatoren für Musterimplementierungen und zeigen auf, dass diese Operatoren domänenspezifisch sind. Bislang fehlt eine generische Modellierungssprache für musterbasierte Entwurfsmodelle, welche als Basis für die Auswahl von Musterimplementierungen und deren Aggregation dienen. Ein besondere Interesse fällt, im Kontext eines Mustergraphen, dabei auf die Auswahl der Aggregatoren und die Aggregation dieser Musterimplementierungen.

1.2. Ziele

Das Ziel dieser Masterarbeit ist der Entwurf einer generischen Modellierungssprache um musterbasierte Entwurfsmodelle ausdrücken zu können. Neben den Entwurfsmodellen soll die Verbindung zwischen Muster und Musterimplementierung sowie die Verbindung zwischen den Musterimplementierungen abgebildet werden können. Um die Anforderungen an die Aggregation von Musterimplementierungen zu veranschaulichen und anschließend zu validieren, sollen Referenzszenarien mit den CCP und den EIP definiert werden. Darauf aufbauend sollen Aggregatoren zur Aggregation von Musterimplementierungen der CCP und der EIP anhand jeweils einer konkreten Technologie konzipiert und implementiert werden. Beispielsweise soll eine Implementierung für die Aggregation von CCP zu einem Amazon Web Services (AWS) Cloud Formation Template und eine Implementierung für die Aggregation von Musterimplementierungen der EIP zu einer Apache ActiveMQ Konfiguration entwickelt werden. Das Zusammenspiel der Modellierungssprache und der Operatoren soll anhand eines auf Pattern Atlas [Pata] basierenden Prototyps gezeigt werden. Pattern Atlas ist eine open-source Software für die Dokumentation von Mustern und Mustersprachen.

1.3. Gliederung

Die Masterarbeit ist in sechs Kapitel gegliedert. Die weiteren Kapitel sind dabei wie folgt aufgebaut und strukturiert:

Kapitel 2 - Grundlagen und verwandte Arbeiten beschreibt und erläutert die thematischen Grundlagen für diese Arbeit. Dafür bietet das Kapitel einen Überblick über Muster, die zwei Mustersprachen CCP und EIP sowie Musterimplementierungen. Außerdem wird die Konkretisierung von Mustern und die Aggregation von Musterimplementierungen betrachtet. Dabei werden Arbeiten aus demselben Themengebiet zu Grunde gelegt.

Kapitel 3 - Anwendungsfälle und Anforderungen zeigt Anwendungsfälle auf und beschreibt Referenzszenarien für die Aggregation von Musterimplementierungen für CCP und EIP. Diese Referenzszenarien sind die Grundlage für das Konzept und werden als Anwendung und zur Validierung des Prototyps zugrunde gelegt. Dafür werden aus den Referenzszenarien die Anforderungen herausgearbeitet und beschrieben.

Kapitel 4 - Konzept zeigt das Konzept einer Lösung für die automatisierte Aggregation von Musterimplementierungen auf. Hierfür wird ein Entwurfsmodell eingeführt sowie die Selektion von Musterimplementierung und deren automatisierte Aggregation beschrieben. Dabei werden die Anforderungen aus Kapitel 3 berücksichtigt.

Kapitel 5 - Prototyp stellt den für diese Arbeit entwickelten Prototyp vor. Hierbei wird auch auf die Implementierungsdetails und spezielle Herausforderungen eingegangen. Abschließend wird der Prototyp anhand der Referenzszenarien und hinsichtlich der Anforderungen evaluiert.

Kapitel 6 - Fazit und Ausblick fasst die Erkenntnisse der Arbeit zusammen und stellt mögliche Anknüpfungspunkte für zukünftige Forschungsarbeit in diesem Themenfeld vor.

2. Grundlagen und verwandte Arbeiten

Zur Vorstellung der thematischen Grundlagen und verwandter Arbeiten werden in diesem Abschnitt zunächst Muster und Mustersprachen erläutert. Im Anschluss werden Musterimplementierungen und Lösungssprachen und ihre Beziehung zu den Mustern und Mustersprachen dargelegt. Auf dieser Grundlage werden die Aggregationsoperatoren und die Lösungsalgebra betrachtet. Abschließend wird die open-source Software Pattern Atlas vorgestellt, welche die Grundlage für die prototypische Implementierung dieser Arbeit bietet.

2.1. Muster und Mustersprachen

Christopher Alexander entwickelte ursprünglich die ersten Muster für die Architektur von Städten und Gebäuden. Ein Muster beschreibt ein Problem, welches wiederholt auftritt. Dazu beschreibt es den Kern einer Lösung für dieses Problem auf eine abstrakte Art und Weise. Durch die Abstraktion kann die Lösung immer wieder angewandt werden, ohne dass die konkreten Lösungen identisch sein müssen. Alexander definierte ein Schema für seine Muster, die somit alle demselben Format entsprechen. Das Muster beginnt mit einem Bild, gefolgt von einem Absatz, der den Kontext des Musters wiedergibt. Anschließend folgt der Titel des Musters. Das Problem wird im nächsten und meist längsten Abschnitt beschrieben. Hierbei geht er auch auf den empirischen Hintergrund ein, belegt die Gültigkeit und stellt unterschiedliche Formen der Problemstellung vor. Darauf folgt die Lösung für das Problem, welche so abstrakt ist, dass sie wiederholt angewandt werden kann und trotzdem nicht zu zwei gleichen Umsetzungen führen muss. Abgeschlossen wird das Muster durch Referenzen auf andere Muster. [AIS+77]

Während Christopher Alexander das vorgestellte Format wählte, haben sich seitdem viele weitere Formen etabliert [Fow06]. Diese unterscheiden sich in ihrem Umfang und den einzelnen Abschnitten. Dabei spielt es unter anderem eine Rolle, aus welchem Wissensgebiet die Muster stammen. Gamma et al. [GHJV95] publizierten 1995 die Design Patterns für den Entwurf von objektorientierter Software. Die Muster wurden von Gamma et al. um Abschnitte erweitert, welche für die Softwareentwicklung relevant sind, darunter auch Quellcodebeispiele. So unterschiedlich die Form von Mustern sein kann, generell lässt sich festhalten: Muster sind von Menschen lesbare Artefakte, welche Problemwissen mit abstraktem Lösungswissen kombinieren [FBB+14b]. Muster sind somit eine technologieneutrale Beschreibung und Vorgehensweise für die Problemlösung. Diese Abstraktheit von Mustern ermöglicht, dass bewährte Lösungen, die von erfahrenen Experten erarbeitet wurden, weitergegeben und von Dritten verstanden und angewendet werden können [HW04].

Gemeinsam haben alle Formen auch Verweise auf verwandte Muster. Dafür enthält das Muster, meist in einem separaten Abschnitt, Referenzen auf andere Muster und beschreibt das Verhältnis zu diesen. Diese Information bildet die Mustersprache und stellt den Zusammenhang zwischen einzelnen Mustern her. Die Referenzen ermöglichen die Navigation des Lesers zwischen den Mustern und führen ihn bei der Auswahl der passenden Muster.

2.1.1. Cloud Computing Patterns

Cloud Computing bedeutet die Möglichkeit jederzeit, selbstständig und bedarfsgerecht auf konfigurierbare Computing-Ressourcen, aus einem gemeinsamen Ressourcenpool, zugreifen zu können. Diese Computing-Ressourcen können Netzwerke, Server, Speicher, Anwendungen, Plattformen oder ähnliche Dienste sein. Die Ressourcen können mit minimaler Interaktion mit dem Anbieter und geringem Verwaltungsaufwand bereitgestellt und genutzt werden. Wesentlich für Cloud Computing sind dabei die folgenden fünf Merkmale. (i) Das Selbstbedienungsprinzip, damit bei Bedarf ohne Kontakt zu Mitarbeitern des Anbieters Ressourcen gemietet oder gekündigt werden können. (ii) Breite Zugriffsmöglichkeiten via Netzwerk, damit unterschiedlichste (Client-)Plattformen angebunden werden können. (iii) Ein gemeinsam genutzter Ressourcenpool aus dem sich viele Nutzer die Ressourcen innerhalb einer Cloud teilen. (iv) Schnelle Elastizität bietet die Möglichkeit Ressourcen zu nutzen oder freizugeben und deshalb eine dynamische Anpassung an veränderte Anforderungen. (v) Die Nutzung wird zur Kontrolle, Analyse und Abrechnung gemessen. [MG11]

Beim Cloud Computing gibt es drei grundlegende Dienstmodelle: Software as a Service (SaaS), Platform as a Service (PaaS) und Infrastructure as a Service (IaaS). (i) SaaS bedeutet die Software wird vollständig vom Anbieter verwaltet. Ein Beispiel hierfür ist Google Maps [Good], eine vollständig von Google gemanagte Karten- und Navigationssoftware, bei der die Nutzer keine Karten- oder Softwareupdates installieren müssen. (ii) PaaS bezeichnet eine bereitgestellte Laufzeitumgebung für Software. Beispielsweise kann ein Unternehmen ihre Java-Software auf einer vollständig verwalteten Java Virtuelle Maschine (JVM) betreiben. Die Verwaltung und Verantwortung für das darunterliegende Betriebssystem samt JVM liegt hingegen beim Plattformbetreiber. (iii) IaaS umfasst bereitgestellte Infrastruktur-Ressourcen, wie Server, Netzwerke und ähnliches. Je nach Definition sind auch weitere Dienstmodelle möglich. Dabei ist wesentlich, dass die Dienste immer und, aus Nutzersicht, „unendlich“ verfügbar sind. Vergleichbar mit der Versorgung von Strom oder Wasser. [MG11]

Diese Merkmale und Dienstmodelle machen Cloud Computing attraktiv. Denn die dynamische Nutzungsmöglichkeit erlaubt schnell auf neue Anforderungen oder ein verändertes Nutzerverhalten zu reagieren. Außerdem ermöglicht es eine Ressourcennutzung ohne Kapitalbindung. Ein weiterer Aspekt ist die Nutzung von Services zur bestmöglichen Ergänzung der eigenen Kompetenzen. So können eigenes Wissen und Fähigkeiten genutzt und fehlende Ressourcen vom Cloud-Anbieter zugekauft werden. Während dies aus technischer Perspektive kein Novum ist, so ist es ein fundamentaler Umbruch im Geschäftsmodell, denn für Cloud-Ressourcen fallen lediglich für die tatsächliche Nutzung Kosten an [FLR+14]. Zu den Anbietern solcher Cloud-Ressourcen mit der größten Marktdurchdringung gehören AWS [Amab], Microsoft Azure [Mic] und Google Cloud Platform (GCP) [Gooc].

Fehling et al. [FLR+14] entwickelten die Cloud Computing Patterns (CCP). Dies sind Muster aus dem Cloud Computing. Die 74 Muster der Mustersprache CCP sind unterteilt in fünf Bereiche. (i) Cloud Computing Grundlagen mit verschiedenen Workload-Szenarien, Dienstmodellen und Bereitstellungsmodellen wie Private Cloud, Community Cloud und Public Cloud sowie Hybrid Cloud. (ii) Cloud Angebote beschreiben anbieterneutral Dienste wie beispielsweise Speicher, Datenbanken, Messaging sowie elastische Infrastruktur oder Plattformen. (iii) Der dritte Bereich besteht aus Cloud Anwendungsarchitekturen und umfasst architektonische Grundlagen für eigene Anwendungen um die Cloud Angebote gut nutzen zu können. Dazu gehören unter anderem lose

Kopplung oder *Stateless Component*, welche direkt die Skalierbarkeit von Anwendungen beeinflussen. (iv) Cloud Anwendungsmanagement umfasst Muster für die automatisierte Skalierung von Anwendungen, Updatevorgängen und weitere Managementaufgaben. (v) Im fünften Bereich der zusammengesetzten Muster werden die Muster zu weiterführenden Lösungskombinationen zusammengefügt. Insgesamt liegt der Fokus der CCP auf den Prinzipien von Cloud Computing, dem Verständnis davon und der Anleitung, wie dies gewinnbringend in der Infrastruktur und für Anwendungen genutzt werden kann. Dafür zeigen sie verschiedene Arten von Cloud Computing und deren Angebote sowie die Grundlagen und weiterführenden Themen auf, um darauf basierend Anwendungen entwickeln zu können. Die CCP unterstützen somit bei der Evaluation von Cloud-Angeboten, der Entwicklung von Cloud-Anwendungen oder eigenen Cloud-Angeboten und bei der Bewertung ob Anwendungslandschaften für die Cloud geeignet sind. [FLR+14]

2.1.2. Enterprise Integration Patterns

Im Unternehmensumfeld müssen eine wachsende Anzahl an Softwaresystemen zusammenarbeiten. Um diesen Informations- und Datenaustausch zu ermöglichen, werden Lösungen und Techniken benötigt. Erschwerend kommt oft hinzu, dass bei der Entwicklung einzelner Softwaresysteme diese Integration nicht berücksichtigt wurde. Hierfür stellt Messaging eine vielversprechende Lösung dar. Messaging ist eine Technologie zur Kommunikation mittels Nachrichten (Messages). Sie ermöglicht schnelle, asynchrone und zuverlässige Kommunikation zwischen Anwendungen oder Softwarekomponenten, welche durch den Austausch von Nachrichten stattfindet. Eine Nachricht ist eine einfache Datenstruktur und besteht aus einem Nachrichtenkopf und einem Nachrichteninhalt. Der Nachrichtenkopf beinhaltet Metainformationen für die Middleware, beispielsweise wann und von wem die Nachricht gesendet wurde oder wer ihr Empfänger ist. Der Nachrichteninhalt enthält die für den Empfänger bestimmten Daten. Diese Daten können die Form eines Dokuments, eines Befehls oder eines, beim Sender aufgetretenen, Ereignisses haben. [HW04; Ora14]

Zwei zentrale Aspekte beim Messaging sind „send and forget“ und „store and forward“. Eine Anwendung kann eine Nachricht senden und diese unmittelbar nach der Akzeptanz durch die Middleware wieder vergessen. Die Middleware stellt eine Übertragung an den Empfänger sicher. Hierdurch müssen in der sendenden Anwendung keine Übertragungsprobleme, Netzwerklatenzen oder vorübergehende Nichterreichbarkeit des Empfängers berücksichtigt werden. „Store and forward“ sorgt dafür, dass eine Nachricht von der Middleware zuerst gespeichert und anschließend weitergereicht wird. Dieses Verfahren wird beim Sender und in jeder Zwischenstation ausgeführt. Somit ist sichergestellt, dass Nachrichten nicht verloren gehen und schließlich den endgültigen Empfänger erreichen. Diese und weitere Funktionalitäten werden typischerweise von einem Softwaresystem bereitgestellt, das Message-oriented Middleware (MOM) genannt wird. [HW04]

In diesem Kontext entwickelten Hohpe und Woolf die Mustersprache Enterprise Integration Patterns (EIP). Die EIP umfassen 65 Muster mit Fokus auf der Integration von Softwaresystemen mittels Messaging, in und zwischen Konzernen. Die Muster sind aufgeteilt in die Abschnitte Integrationsstile, Messaging-Systeme, Messaging-Channels, Nachrichtenkonstruktion, Nachrichten-Routing, Nachrichtentransformation, Messaging-Endpunkte und Systemmanagement. So wird ein breiter Bereich an Anwendungsmöglichkeiten von Messaging abgedeckt. Wie bei Mustern einer Mustersprache üblich, folgen die EIP einem festgelegten Schema. Im Rahmen von Musterimplementierungen ist der Abschnitt „Examples“ zu erwähnen, der im Buch „Enterprise Integration Patterns“ sehr ausführlich behandelt wird und oft den kompletten Quellcode einer Beispielimplementierung enthält. [HW04]

2.2. Musterimplementierungen

Die abstrakte Eigenschaft von Mustern ermöglicht eine Vielzahl unterschiedlicher Implementierungen der konkreten Problemlösung. Prinzipiell fehlt die Möglichkeit, Muster mit konkreten Implementierungen zu verknüpfen. Muster enthalten zwar oft einen Abschnitt „Known uses“ um konkrete Implementierungen zu beschreiben, allerdings ist an dieser Stelle nur textuelle Beschreibung möglich [FBB+14b]. Die EIP enthalten zusätzlich einen Abschnitt „Examples“, teilweise mit kompletten Beispielimplementierungen [HW04]. Dennoch sind diese Informationen aufgrund der Charakteristik eines Musters in Umfang und Anzahl begrenzt und können nicht sämtliche Implementierungen für verschiedene Programmiersprachen oder Technologien enthalten. Außerdem sind diese Beispielimplementierungen in textueller Form aus dem Muster nicht direkt nutzbar, sondern müssen erst noch in Quellcodedateien übernommen und ergänzt werden. Des Weiteren geht das Wissen verloren, aus welchen Implementierungen die Muster abstrahiert wurden, weil es an dieser Stelle schlicht nicht weitergegeben werden kann [FBB+14b]. Werden neue Technologien entwickelt, ist es zudem nicht möglich die „langlebigen“ Muster anzupassen und die Bücher zeitnah und fortwährend zu ergänzen. Nachträglich erstellte Implementierungen können auf diese Art nicht mit dem zugrundeliegenden Muster verknüpft werden. Dies erschwert die Wiederverwendung der Implementierung.

Falkenthal et al. [FBB+14b] schlagen deshalb Musterimplementierungen vor. Musterimplementierungen sind ein Vorschlag, welcher die drei folgenden Eigenschaften erfüllt, um die genannten Probleme zu lösen. (i) Sie definieren konkretes und implementiertes Lösungswissen als wiederverwendbare Bausteine. (ii) Muster sind mit den Musterimplementierungen verknüpft, sowohl mit den ursprünglichen Umsetzungen, aus denen die Muster abstrahiert wurden, als auch mit neuen Implementierungen, die für ein Muster entwickelt wurden. (iii) Musterimplementierungen sind Bausteine und ermöglichen die Zusammensetzung der konkreten Implementierungen. Je nach Domäne können Musterimplementierungen sehr unterschiedlich sein. In der Domäne der Softwareentwicklung bestehen Musterimplementierungen meist aus Quellcode, der direkt für die Entwicklung einer eigenen Anwendung verwendet werden kann. Die Verknüpfung zwischen Muster und Musterimplementierung ermöglicht eine einfachere Wiederverwendung der Implementierung und erfordert weniger Wissen für die Entwicklung, da das Design und die Entwicklung der Implementierung nicht erneut erfolgen muss. So können Nutzer existierende Implementierungen für ihr Nutzungsszenario anwenden und den manuellen Aufwand signifikant reduzieren. Außerdem können Musterimplementierungen für die zukünftige Anwendung ergänzt werden, was bei dem Abschnitt „Known uses“ von Mustern nicht immer möglich ist.

Im Detail besteht eine Musterimplementierung aus zwei Teilen, einem Musterimplementierungsartefakt (MIA) und einem Musterimplementierungsdeskriptor (MID). Das MIA ist eine mögliche Implementierung des Musters. Diese Implementierung kann beispielsweise ein Quellcodefragment, eine Konfigurationsdatei oder eine ausführbare Datei sein. In nicht informationstechnischen Bereichen kann das MIA ein konkreter Gegenstand sein, welcher an einem bestimmten Ort gelagert ist. Krieger [Kri18] entwarf den MID um Muster mit MIA zu verknüpfen. Mehrere MIAs die jeweils für einen speziellen Anwendungsfall entwickelt wurden, können so miteinander verlinkt werden. Die Anwendungsfälle können beispielsweise unterschiedliche Programmiersprachen oder Technologien sein. Der MID enthält über die reine Implementierung der Problemlösung hinausgehende Informationen, welche für die Musterimplementierung relevant sind. Diese Metainformationen enthalten eine Referenz zum MIA und eine Referenz zum implementierten Muster und stellen so

die Verbindung zwischen Muster und konkreter Implementierung her. Weitere Informationen wie Name, Beschreibung, Referenzen zu anderen Musterimplementierungen oder Eigenschaften, wie die Programmiersprache, können im MID enthalten sein. Die im MID enthaltenen Informationen dienen der Verbindung zwischen Muster und Musterimplementierung sowie der Auswahl der passenden Musterimplementierung bei der Umsetzung eines Musters [Kri18].

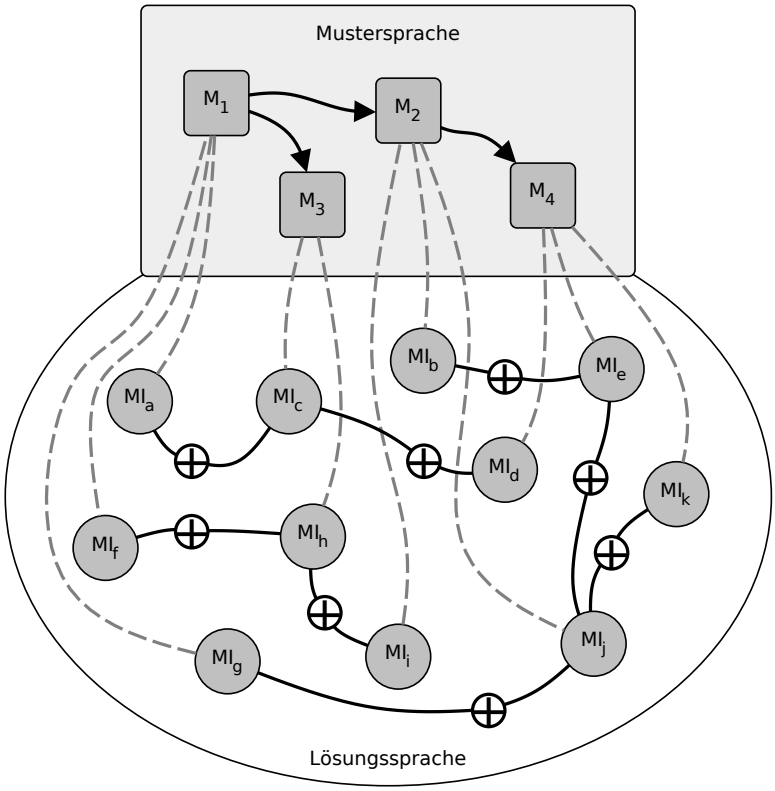

Abbildung 2.1.: Eine Mustersprache besteht aus verbundenen Mustern, welche von Musterimplementierungen implementiert werden. Eine Lösungssprache beinhaltet die Musterimplementierungen und ihre Beziehungen. [FL17]

Abbildung 2.1 zeigt den Zusammenhang zwischen Mustern und Musterimplementierungen. Die gestrichelte Linie stellt die Beziehung zwischen Mustern und Musterimplementierungen dar, welche durch den MID hergestellt wird. Ebenfalls gibt es Beziehungen zwischen M_1, M_2, M_3 und M_4. Diese Beziehungen unterscheiden eine Menge von Mustern von einer Mustersprache, in der die Muster einen Zusammenhang haben. Dasselbe Prinzip gilt auch für die Lösungssprache, die von Falkenthal et al. [FL17] aufbauend auf einen Lösungsraum entwickelt wurde. Hier existieren Beziehungen zwischen den einzelnen Musterimplementierungen, welche beispielsweise eine Kombinierbarkeit, Alternative oder Aggregation ausdrücken können [FL17]. Diese Beziehungen unterscheiden eine Lösungssprache von einem Lösungsraum. Der Lösungsraum ist eine Menge von Musterimplementierungen, welche die Muster einer Mustersprache implementieren, allerdings ohne explizite Beziehungen zwischen den Musterimplementierungen [FBB+16].

2.2.1. Konkretisierung von Mustern

Muster können unterschiedlich abstrakt sein. Sehr abstrakte Muster decken mehr Anwendungsfälle ab. Konkretere Muster können dafür einfacher in Implementierungen übersetzt werden [FBB+16]. Ein *Point-to-Point Channel* aus den EIP kann in vielen MOMs direkt konfiguriert werden. Das Muster *SaaS* aus den CCP ist sehr viel unspezifischer, lässt sich aber durch weitere Muster detaillierter beschreiben. Beispielsweise kann intern eine *Message-oriented Middleware* mit mehreren *Point-to-Point Channels* genutzt werden. Auf diese Art kann eine Konkretisierung von Mustern vorgenommen werden, bis schließlich eine Musterimplementierung gewählt wird [FBB+16]. Je abstrakter und allgemeiner ein Muster verfasst ist, desto mehr Anwendungsfälle können damit gelöst werden und desto größer ist der Lösungsraum [FBB+16].

Harzenetter et al. [HBF+18] zeigten anhand musterbasierenden Deploymentmodellen ein Vorgehen zur automatisierten Konkretisierung. Mit diesem Vorgehen lassen sich die Vorteile von Mustern, nicht auf spezifische Anbieter oder Technologien festgelegt zu sein, zeitsparender und weniger fehleranfällig konkretisieren und als Softwaresystem umsetzen. Auch der Austausch einzelner Technologien, wie beispielsweise der Wechsel der *relationalen Datenbank* von MariaDB zu PostgeSQL, wird damit erleichtert. [HBF+18]

Für die Auswahl der passenden Musterimplementierungen sind oft weitere Rahmenbedingungen entscheidend. Im Unternehmenskontext sind dies häufig Einschränkungen durch bereits genutzte Cloud-Anbieter, rechtliche Vorgaben oder die Wahl einer bereits bekannten Technologie um auf bestehendes Wissen zurückzugreifen. Je nach Anwendungsfall können auch mehrere Faktoren gleichzeitig zutreffen. Um die passende Musterimplementierung aus der Gesamtmenge an Implementierungen für ein Muster zu finden, erweitern Falkenthal et al. [FBB+14b] das Konzept um Selektionskriterien. Die Selektionskriterien werden den Kanten hinzugefügt, welche die Beziehung zwischen Muster und Musterimplementierung darstellen. Diese Selektionskriterien ermöglichen die Auswahl anhand nichtfunktionaler Anforderungen, beispielsweise welcher Cloud-Anbieter genutzt werden soll [FBB+14b]. Außerdem ermöglichen die Selektionskriterien die Validierung, ob mögliche Nachbedingungen der zuvor gewählten Musterimplementierung erfüllt sind [FBB+14b]. Anhand der Selektionskriterien kann eine Sequenz aus Mustern auf eine Sequenz aus Musterimplementierungen abgebildet werden [FBB+14b]. Falazi [Fal17] untersuchte und implementierte diese Abbildung von einer Mustersequenz auf eine Musterimplementierungssequenz. Hierfür beschreibt Falazi einen Algorithmus bestehend aus zwei Phasen. Zuerst werden sämtliche mögliche Pfade durch eine Menge von Musterimplementierungen generiert. Anschließend werden die Pfade anhand vom Nutzer spezifizierter Anforderungen gefiltert. Nach Falkenthal et al. [FBB+16] ist die Selektion von Mustern ein Teilgraph der Mustersprache. Ein Graph kann, im Gegensatz zu einer Sequenz, Knoten mit mehr als zwei Kanten enthalten. Im Fall von Mustergraphen können sich die Muster auch über verschiedene Abstraktionsebenen erstrecken [FBB+16]. Einen möglichen Lösungsgraph für den Mustergraph M_1, M_2, M_3 zeigt Abbildung 2.2. Ein weiteres Beispiel für einen Lösungsgraph sind die Musterimplementierungen MI_g, MI_j und MI_k. Ein anderes Beispiel, welches nicht als Sequenz abgebildet werden kann, ist der Mustergraph aus M_1, M_2 und M_3 mit seinem korrespondierenden Lösungsgraph bestehend aus MI_f, MI_h und MI_i.

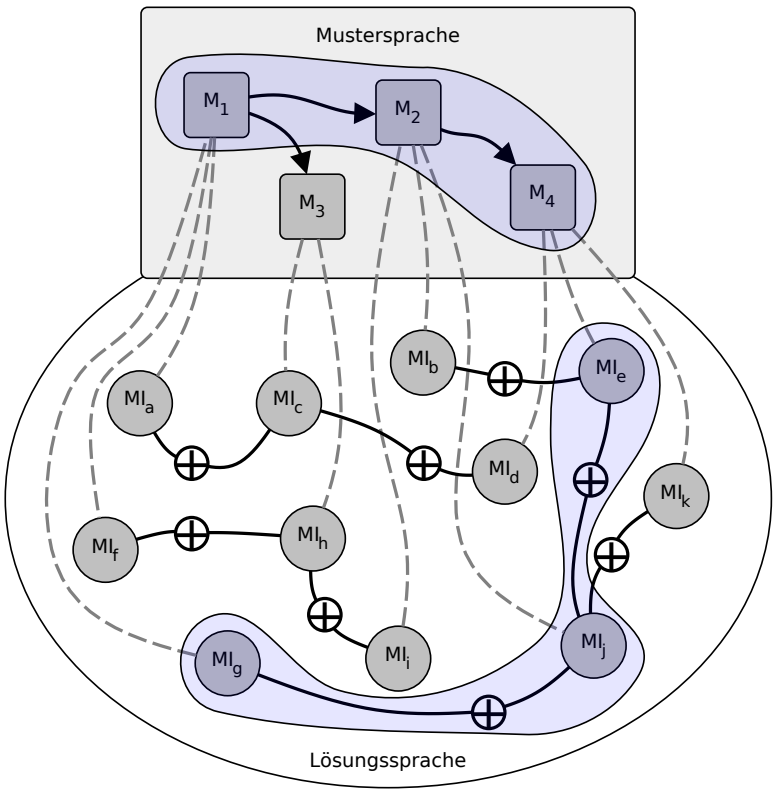

Abbildung 2.2.: Ein Mustergraph als Teil einer Mustersprache und eine Selektion eines passenden Lösungsgraphen. [FL17]

2.3. Aggregationsoperatoren und Lösungsalgebra

Um Musterimplementierungen zu aggregieren, kommen Aggregationsoperatoren zum Einsatz. Diese spezifizieren, wie bestimmte Musterimplementierungen zu einer Lösung zusammengefügt werden können. Hierfür können zwei Musterimplementierungen mit einem Aggregationsoperator verknüpft werden, in Abbildung 2.2 dargestellt durch $\oplus$. Diese Aggregationsoperatoren sind entsprechend abhängig von der Art der zu aggregierenden Musterimplementierungen. Ein Aggregationsoperator kann je nach Domäne eine textuelle Beschreibung, Quellcode oder ein ausführbares Programm sein, welche die Aggregation beschreibt beziehungsweise automatisiert ausführt. Falkenthal et al. [FBBL19] beschreiben die mathematischen Eigenschaften von Aggregationsoperatoren für Musterimplementierungen. Insbesondere wird eine Lösungsalgebra definiert, die aus einem Tupel von Musterimplementierungen und einem Tupel von Aggregationsoperatoren besteht. Wobei jeder Aggregationsoperator ein Tupel von Musterimplementierungen in eine Musterimplementierung abbilden kann. Damit kann ausgedrückt werden, wie diese Musterimplementierungen aggregiert werden können. Bislang fehlt eine generische Modellierungssprache für musterbasierte Entwurfsmodelle, welche als Basis für die Auswahl von Musterimplementierungen und deren Aggregation dienen. Ein besonderes Interesse fällt im Kontext eines Mustergraphen dabei auf die Auswahl der Aggregatoren und die Aggregation der Musterimplementierungen.

2.4. Pattern Atlas

Pattern Atlas dient als Basis für die Entwicklung eines Prototyps für die automatisierte Aggregation von Musterimplementierungen. Ursprünglich entstand unter dem Namen „PatternPedia“ eine Software zur zentralen Dokumentation von Mustern und Mustersprachen. Die erste Version wurde aufbauend auf dem MediaWiki entwickelt. Die PatternPedia unterstützt die kollaborative Erstellung, die Suche und das Verknüpfen von Mustern und erleichtert damit den Zugang zu den Mustern. [FBFL15; Ley]

2018 wurde begonnen die PatternPedia auf der Basis einer Webanwendung mit Angular und Spring Boot vollständig neu zu entwickeln [Patb; Patc]. Seitdem wächst der Funktionsumfang der open-source Anwendung kontinuierlich. Beispielsweise können Pattern Views verwaltet werden um die Verbindung zwischen Mustern verschiedener Mustersprachen zu dokumentieren und interdisziplinäres Wissensmanagement im Bezug auf Mustern zu vereinfachen [WBB+20]. Im Jahr 2020 wurde die PatternPedia in Pattern Atlas umbenannt und wird seit dem Jahr auch im Projekt PlanQK [Kon] eingesetzt, einer Plattform und Ökosystem für quantenunterstützte künstliche Intelligenz [QuA]. Im Rahmen dieser Arbeit wird ein Prototyp für die automatisierte Aggregation von Musterimplementierungen auf der Basis von Pattern Atlas entwickelt und integriert.

3. Anwendungsfälle und Anforderungen

Um die Anforderungen an eine automatisierte Aggregation von Musterimplementierungen zu analysieren, werden in diesem Kapitel zunächst drei Anwendungsfälle beschrieben. Anhand dieser drei Referenzszenarien werden die Anforderungen an die Aggregationslösung herausgearbeitet und dargestellt. Durch eine anschließende Anforderungsanalyse bilden diese Referenzszenarien die Grundlage für die spätere Evaluation des Prototyps. Hierzu werden am Ende dieses Kapitels die Anforderungen an eine automatisierte Aggregation von Musterimplementierungen zusammengefasst.

3.1. Referenzszenario: Cloud Computing Patterns

Das erste Referenzszenario behandelt Muster aus den CCP [FLR+14]. Bei den CCP handelt es sich um eine Mustersprache für Cloud Computing. Die Muster umfassen viele relevante Themen aus dem Cloud Computing auf unterschiedlichen Abstraktionsebenen. Diese reichen von konzeptionellen Mustern zu Service- oder Deploymentmodellen bis hin zu konkreteren Arten von Cloud-Angeboten, wie *Relationale Datenbank* oder *Message-oriented Middleware* [FLR+14]. Fehling et al. haben die CCP in fünf Kategorien strukturiert: Cloud Computing-Grundlagen, Cloud-Angebote, Cloud-Anwendungsarchitekturen, Cloud-Anwendungsmanagement und zusammengesetzte Muster für Cloud-Anwendungen [FLR+14].

In diesem Referenzszenario soll eine skalierbare Anwendung entwickelt werden, welche Daten für eine Webseite und ein Mobile App bereitstellt. Es wird erwartet, dass die Anzahl der Nutzer stark schwankt und die Anwendung hauptsächlich in den Abendstunden und am Wochenende genutzt wird. Die Entwickler entscheiden sich für *IaaS*, da für die gewählte Programmiersprache Rust kein passendes *PaaS*-Angebot bei AWS existiert.

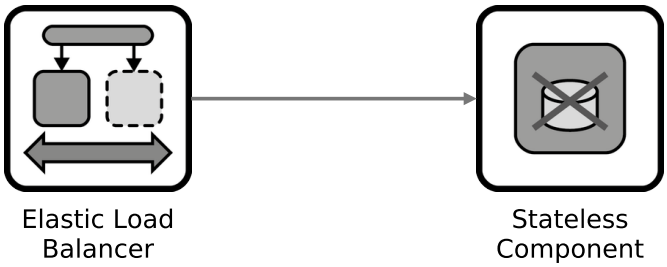

Abbildung 3.1.: Ein Referenzszenario für die Aggregation von Cloud Computing Patterns. Ein Elastic Load Balancer skaliert die Stateless Component.

Abbildung 3.1 stellt dar, wie dieses Referenzszenario mittels Muster der CCP entworfen werden kann. Der Systementwurf der Anwendung besteht aus dem Muster *Stateless Component*, welches nach Bedarf von einem *Elastic Load Balancer* skaliert wird. Je nach Nutzungsintensität der Anwendung können weitere Instanzen der *Stateless Component* gestartet oder deprovisioniert werden.

3.1.1. Implementierung

Je nach Art und Abstraktionsebene eines Musters kann eine direkte Umsetzbarkeit des Musters von trivial implementierbar bis hin zu nicht direkt implementierbar reichen. Ein Beispiel ist das Muster *Unpredictable Workload*, welches eine unvorhersagbare Intensität der Anwendungsnutzung beschreibt. Diese Art der Nutzung kann beispielsweise bei einem Onlineshop mit unregelmäßigen Sonderangeboten auftreten. Für das Muster *Unpredictable Workload* kann nicht direkt eine eigene Implementierung in Form einer Musterimplementierung bereitgestellt werden. Dieses Muster stellt eine nichtfunktionale Anforderung dar, die bei der Konkretisierung der Muster berücksichtigt werden muss und somit indirekt in die Musterimplementierung einfließt. Direkt implementierbare Muster sind wie eingangs erwähnt in den CCP hingegen genauso enthalten. Die *relationale Datenbank* kann beispielsweise durch eine Musterimplementierung wie MariaDB [Mar] oder PostgreSQL [The] implementiert werden. Diese sind bei einigen Anbietern verfügbar, wie etwa im Rahmen von

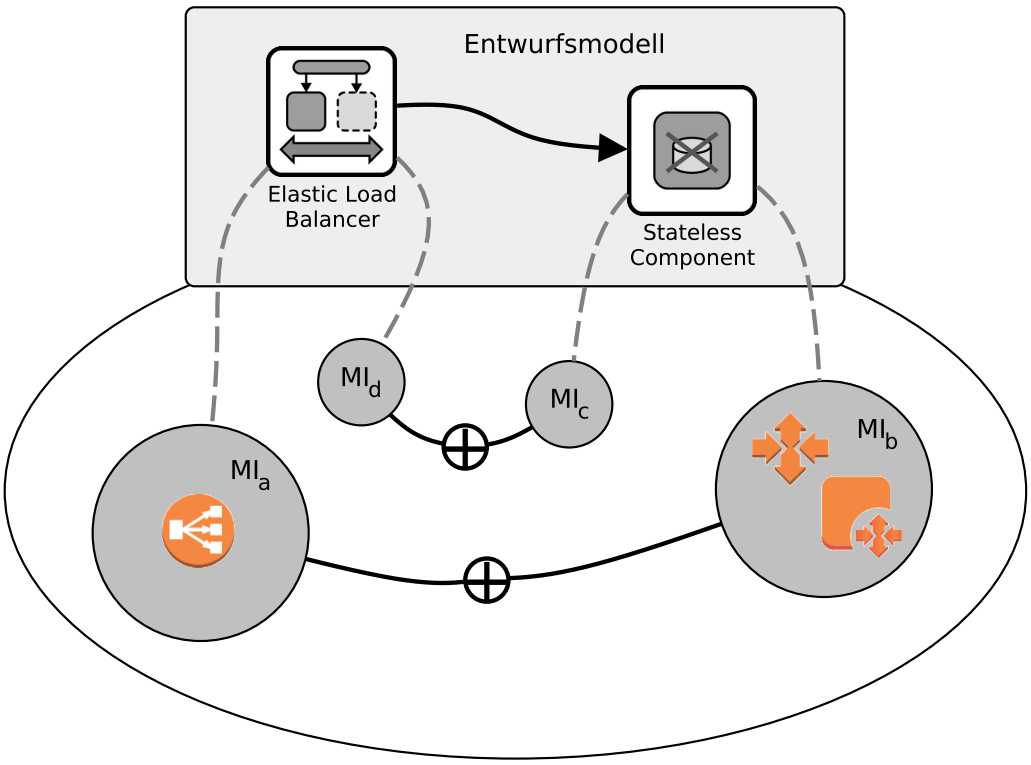

Abbildung 3.2.: Der Architekturentwurf des Cloud Computing Patterns Referenzszenario. Visualisierung der Lösungssprache und den Musterimplementierungen $\texttt{MI}_\texttt{a}$ und $\texttt{MI}_\texttt{b}$ mit Amazon Web Services-Produkten.

Amazon Relational Database Service [Amaa]. Eine relationale Datenbank kann so, wie auch weitere Cloud-Ressourcen, mittels eines AWS CloudFormation Templates provisioniert werden. Bei AWS CloudFormation Templates handelt es sich um eine deklarative Provisionierungssprache für die Beschreibung von AWS Cloud-Ressourcen und deren Eigenschaften in Form von JavaScript Object Notation (JSON) oder YAML Ain't Markup Language (YAML)-Dateien [Amac; BL18].

Weitere anbieterneutrale und -spezifische deklarative Provisionierungsmöglichkeiten sind denkbar. Beispiele für anbieterneutrale Alternativen sind (Open)TOSCA [BEK+16] oder Terraform [Has]. Eine anbieterspezifische Alternative ist beispielsweise der Cloud Deployment Manager [Goob] für die GCP [Gooc]. Eine weitere Möglichkeit ist die Nutzung verschiedener Programmierschnittstellen (APIs), die durch ein generiertes Script oder eine Workflow Engine aufgerufen werden. Hierdurch sind zudem Verwaltungsaufgaben möglich, was bereits mit Management Planlets gezeigt werden konnte [BBKL13]. Management Planlets sind generische Bausteine, welche Verwaltungsfunktionen eigenständig und wiederverwendbar kapseln.

Die Abbildung 3.2 zeigt den Architekturentwurf sowie die Lösungssprache für dieses Referenzszenario. { MI_a, MI_b } und { MI_c, MI_d } stellen alternative Implementierungsmöglichkeiten dar. Um den Umfang dieser Arbeit zu begrenzen, wurde dieses Referenzszenario auf eine Aggregation für AWS CloudFormation Templates beschränkt. Die Anwendungsarchitektur aus Abbildung 3.1 soll hierfür in einem AWS CloudFormation Template abgebildet werden. Ein CloudFormation Template besteht aus mehreren Abschnitten, in denen die Konfiguration für ein komplettes Deployment von Softwaresystemen möglich ist. Relevant für die Musterimplementierungen in diesem Referenzszenario ist der Abschnitt *Resources*. Im Folgenden werden die Musterimplementierungen vorgestellt und anschließend eine Aggregation durchgeführt. Um die Quellcodes kompakt zu halten, wird nur ein minimales Beispiel mit dem relevanten Code präsentiert. Aus diesem Grund sind Netzwerkkonfigurationen und weitere Details bewusst nicht enthalten. Dem *Elastic Load Balancer* wird als MIA der Code aus Listing 3.1 zugeordnet, dies entspricht der MI_a aus Abbildung 3.2.

```
1 "MyELB": {
2   "Type": "AWS::ElasticLoadBalancing::LoadBalancer"
3 }
```

Listing 3.1: Die Definition eines Elastic Load Balancers für ein CloudFormation Template als Musterimplementierung für einen Elastic Load Balancer.

Als Implementierung der *Stateless Component* wird das MIA aus Listing 3.2 genutzt. Diese besteht aus einer *Autoscaling Group*, für die automatisierte Skalierung von Virtuelle Maschinen (VMs), und einer *Launch Configuration*, für die Auswahl und Konfiguration der VM. Dies entspricht der MI_b aus Abbildung 3.2.

```
1 "MyASG": {
2   "Type": "AWS::AutoScaling::AutoScalingGroup",
3   "Properties": {
4     "LoadBalancerNames": [ {
5       "Ref": "<TODO>"
6     } ],
7     "LaunchConfigurationName": {
8       "Ref": "MyLC"
9     }
```

```
10    }
11  },
12  "MyLC": {
13    "Type": "AWS::AutoScaling::LaunchConfiguration"
14  }
```

Listing 3.2: Die Definition einer Autoscaling Group für die Skalierung von VMs und einer Launch Configuration für die Spezifizierung der VM als Musterimplementierung für eine Stateless Component.

Die Aggregation der zwei MIAs erfolgt durch den Aggregationsoperator $\oplus$ zwischen $\texttt{MI}_a$ und $\texttt{MI}_b$ aus Abbildung 3.2. Der Aggregationsoperator muss die folgenden drei Schritte durchführen um ein funktionsfähiges Aggregat zu erhalten. (i) Die zwei MIAs werden, mit einem Komma als Trenner, aneinander gefügt. (ii) Der Wert für die Referenz zum *Elastic Load Balancer* in Listing 3.2 Zeile 5 muss korrekt gesetzt werden. (iii) Abschließend muss das bisherige Aggregat als Wert für die `Resources` in Listing 3.3 Zeile 3 eingesetzt werden.

```
1  {
2    "AWSTemplateFormatVersion": "2010-09-09",
3    "Resources": {
4      "MyELB": {
5        "Type": "AWS::ElasticLoadBalancing::LoadBalancer"
6      },
7      "MyASG": {
8        "Type": "AWS::AutoScaling::AutoScalingGroup",
9        "Properties": {
10         "LoadBalancerNames": [ {
11           "Ref": "MyELB"
12         } ],
13         "LaunchConfigurationName": {
14           "Ref": "MyLC"
15         }
16       }
17     },
18     "MyLC": {
19       "Type": "AWS::AutoScaling::LaunchConfiguration"
20     }
21   }
22 }
```

Listing 3.3: Das Referenzszenario der Cloud Computing Patterns implementiert als Amazon Web Services CloudFormation Template.

3.1.2. Anforderungen

Aus dem aufgezeigten Referenzszenario ergeben sich mehrere Anforderungen an eine Software, die diesen Prozess der Aggregation automatisiert. Die grundlegende und erste Anforderung umfasst die Erstellung und Veränderung eines Architekturentwurfs. Dieses stellt die Softwarearchitektur

mittels Mustern dar. Die Anforderung *A1 - Der Nutzer kann einen Architekturentwurf erstellen und verändern* umfasst somit einen Editor, mit dem es möglich ist Muster zu platzieren und Beziehungen zwischen diesen auszudrücken. Die Darstellung kann dabei äquivalent zu Abbildung 3.1 sein.

Damit eine Aggregation stattfinden kann, werden die zu aggregierenden Musterimplementierungen benötigt. Ein MIA muss implementiert werden. Um die Musterimplementierungen mit den Mustern zu verbinden, sollen diese mittels eines MID mit einer Referenz auf das Muster versehen werden. So können Musterimplementierungen bereitgestellt werden. Diese Anforderung wird in *A2 - Erstellung von Musterimplementierung und Zuordnung zu Muster* ausgedrückt und als Musterimplementierung (Kreis) und Referenz zum Muster (gestrichelte Linie) in Abbildung 3.2 dargestellt. Damit der Nutzer die passenden Musterimplementierungen zum eigenen Architekturentwurf wählen kann, wird in der Anforderung *A3 - Der Nutzer hat Einfluss auf die Selektion der Musterimplementierung* festgehalten, dass eine Selektion derselben durch den Nutzer möglich sein muss.

Für die Aggregation selbst muss ein Aggregationsoperator existieren. Dieser muss mindestens in der Lage sein (i) zwei Textfragmente zusammenzufügen und gegebenenfalls weiteren Text hinzuzufügen, (ii) Referenzen zu ergänzen oder Anpassungen des Textes vorzunehmen und (iii) dies gegebenenfalls in ein weiteres Textfragment einzubetten. Diese drei Funktionalitäten werden in der Anforderung *A4 - Aggregationsoperator für die Aggregation der Musterimplementierung* zusammengefasst.

3.2. Referenzszenario: Enterprise Integration Patterns

Das zweite Referenzszenario umfasst die Mustersprache der EIP. Die EIP von Hohpe und Woolf umfassen 65 Muster mit einem Fokus auf der Integration von Softwaresystemen mittels Messaging. Die EIP enthalten einen Abschnitt „Examples“, der von Hohpe und Woolf sehr ausführlich behandelt wurde und häufig eine oder mehrere Beispielimplementierungen enthält. [HW04]

Die EIP lassen sich nach ihrem Anwendungskontext in fünf Gruppen einteilen. In der ersten Gruppe finden sich Integrationsstile, welche die Muster *File Transfer*, *Shared Database*, *Remote Procedure Invocation* und *Messaging* beinhalten. Alle anderen Muster beschreiben Teilaspekte des Integrationsstils *Messaging*. Eine zweite Gruppe umfasst die für das Messaging grundlegenden Muster, wie *Message Broker*, *Message Bus*, *Message Channel*, *Message Endpoint* und *Message*. Die dritte Gruppe enthält Muster, die beschreiben wie fachliche Aspekte als Inhalt von Nachrichten repräsentiert werden können. Dazu zählen *Command Message*, *Document Message*, *Event Message* und *Format Indicator*. Zur vierten Gruppe zählen alle Muster, welche typischerweise in einer Clientanwendung implementiert werden. Diese Muster umfassen beispielsweise *Channel Adapter*, *Envelop Wrapper*, *Idempotent Receiver*, *Invalid Message Channel*, *Message Sequence*, *Messaging Bridge*, *Messaging Gateway*, *Messaging Mapper* und *Selective Consumer*. Die fünfte Gruppe enthält die übrigen 43 Muster der EIP, welche in einem Message Broker implementiert werden können. [GG19]

In diesem Referenzszenario soll ein Onlineshop mit dem Warenwirtschaftssystem integriert werden. Der Onlineshop sendet alle Bestellungen als Nachricht. Hierbei sind auch Bestellungen enthalten, die noch nicht abgeschlossen sind, vom Kunden allerdings (zwischen)gespeichert wurden. Diese zwischengespeicherten, aber noch nicht abgeschlossenen, Bestellungen müssen ignoriert werden. Hierfür wird eine Komponente benötigt, die diese Nachrichten verwirft. Die Firma betreibt zwei Versandlager mit unterschiedlichen Arten von Ware. Das eine Lager kann hierbei beispielsweise

ein Außenlager sein, welches schwere Speditionsgüter mit Logistikunternehmen versendet, die im zweiten Lager aufgrund ihrer Größe keinen Platz finden. Jedes Versandlager hat dabei sein eigenes Warenwirtschaftssystem. Deshalb müssen die Nachrichten entsprechend aufgeteilt werden und die Bestellungen an das passende Versandlager übermittelt werden.

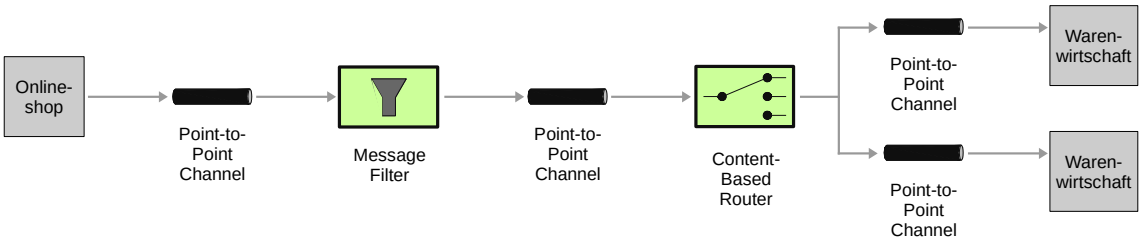

Abbildung 3.3.: Der Architekturentwurf des Enterprise Integration Patterns Referenzszenario zur Integration eines Onlineshops mit zwei Warenwirtschaftssystemen. Die Richtung der Kante stellt den Nachrichtenfluss dar.

Die Architektur der Systemintegration in diesem Referenzszenario mittels EIP ist in Abbildung 3.3 dargestellt. Der Onlineshop sendet die Nachrichten mit den Bestellungen. Der *Point-to-Point Channel* garantiert, dass alle Bestellungen beim *Message Filter* ankommen. Dieser sortiert die nicht abgeschlossenen Bestellungen aus und sendet nur die abgeschlossenen Bestellungen an den *Point-to-Point Channel* weiter. Der *Content-Based Router* empfängt die Nachrichten und sortiert die Bestellungen anhand der bestellten Ware und gibt die Nachricht mit der Bestellung an den *Point-to-Point Channel* der Warenwirtschaft an Standort A oder B weiter. Beide Warenwirtschaftssysteme empfangen dann die Nachrichten mittels ihres *Point-to-Point Channels*. Die Richtungen der Kanten stellen in Abbildung 3.3 den Nachrichtenfluss dar und nicht unbedingt die Beziehung zwischen den Mustern. Technisch sendet der *Point-to-Point Channel* die Nachricht nicht weiter, sondern der *Message Filter* empfängt die Nachricht von dort.

3.2.1. Implementierung

Die grundlegenden Muster für Messaging, wie beispielsweise *Message*, werden von jeder MOM implementiert. Für die Implementierung des vorgestellten Referenzszenarios ist darüber hinausgehend eine umfangreiche Unterstützung der brokerseitigen Muster von Vorteil. Mittels Apache Camel [Apaa] unterstützt Apache ActiveMQ [Apab] die vorgestellten brokerseitigen Muster [Apa20]. So lassen sich von den brokerseitigen Mustern 41 vollständig und zwei teilweise mit Apache ActiveMQ implementieren. Damit bietet Apache ActiveMQ weit bessere Integrationsfähigkeit als andere open-source MOMs, wie Apache Kafka oder RabbitMQ. Die implementierbaren EIP können für Apache ActiveMQ entweder mittels einer Domänenspezifische Sprache (DSL) in Java oder einer Extensible Markup Language (XML)-Konfiguration umgesetzt werden [Apa20]. Aufgrund der guten Integrationsfähigkeit von EIP wird für die Implementierung Apache ActiveMQ als MOM gewählt. [GG19]

Im Folgenden wird eine mögliche Implementierung und Aggregation für Apache ActiveMQ im Detail vorgestellt. Für das Beispiel wird die Java DSL gewählt, da sie etwas kompakter ist als die XML-Konfiguration. Funktional sind beide Möglichkeiten gleichwertig. Zuerst werden die Musterimplementierungen für die drei Muster vorgestellt und zugeordnet. Anschließend werden sie dem Architekturentwurf entsprechend aggregiert.

```
1 from("activemq:<TODO>")
2   .to("activemq:<TODO>");
```

Listing 3.4: Eine Musterimplementierung eines Point-to-Point Channels für Apache ActiveMQ.

Listing 3.4 zeigt eine Musterimplementierung des *Point-to-Point Channels*. Die zwei Platzhalter `<TODO>` müssen entsprechend bei der Aggregation durch die korrekten Referenzen ersetzt werden.

```
1 from("activemq:<TODO>")
2   .filter(xpath("/sensor/value > -50")) // TODO: adapt filter rule
3     .to("activemq:<TODO>");
```

Listing 3.5: Eine Musterimplementierung eines Message Filters für Apache ActiveMQ.

Listing 3.5 zeigt die gewählte Musterimplementierung des *Message Filters* mittels Java DSL. Die Platzhalter `<TODO>` müssen ebenfalls entsprechend bei der Aggregation durch die korrekten Referenzen ersetzt werden.

```
1 from("activemq:<TODO>")
2   .choice()
3     .when(xpath("/sensor/type = 'temp-celsius'")) // TODO: adapt filter rule
4       .to("activemq:<TODO>")
5     .otherwise()
6       .to("activemq:deadletterchannel");
```

Listing 3.6: Eine Musterimplementierung eines Content-Based Router für Apache ActiveMQ.

Die Java Musterimplementierung des *Message Filters* ist in Listing 3.6 dargestellt. Dass dieses Referenzszenario einen Graphen anstatt einer Sequenz darstellt, wird bei der Aggregation dieser Musterimplementierung deutlich. Ein *Content-Based Router* kann die eingehende Nachricht an einen von vielen *Point-to-Point Channels* senden. Die Anzahl ist nicht begrenzt und es wird somit erforderlich, dass Zeile 3 und 4 entsprechend oft wiederholt werden. Nachdem dies geschehen ist, müssen, wie bei den zwei bisherigen Musterimplementierungen, die Platzhalter `<TODO>` durch die korrekten Referenzen ersetzt werden.

Die sechs einzelnen Java-Fragmente müssen nun zusammengefügt werden. Davon viermal die Musterimplementierung des *Point-to-Point Channel* und jeweils einmal die anderen beiden Musterimplementierungen. Außerdem müssen die Referenzen so gesetzt werden, dass der Nachrichtenfluss entsprechend des Entwurfs konfiguriert wird. Das zusammengesetzte Fragment muss nun in einer Klasse innerhalb der `configure()`-Methode eingefügt werden. Die aggregierte Java-Datei ist in Listing 3.7 dargestellt.

```
1  public class MyRouteBuilder extends RouteBuilder {
2    public void configure() {
3      from("activemq:onlineshop-out").to("activemq:message-filter-in");
4      from("activemq:message-filter-in")
5        .filter(xpath("/sensor/value > -50")) // TODO: adapt filter rule
6          .to("activemq:message-filter-out");
7      from("activemq:message-filter-out").to("activemq:content-based-router-in");
8      from("activemq:content-based-router-in")
9        .choice()
10         .when(xpath("/sensor/type = 'temp-celsius'")) // TODO: adapt filter rule
11           .to("activemq:content-based-router-out-a")
12         .when(xpath("/sensor/type = 'temp-celsius'")) // TODO: adapt filter rule
13           .to("activemq:content-based-router-out-b")
14         .otherwise()
15           .to("activemq:deadletterchannel");
16     from("activemq:content-based-router-out-a").to("activemq:wawi-in-a");
17     from("activemq:content-based-router-out-b").to("activemq:wawi-in-b");
18   }
19 }
```

Listing 3.7: Das Referenzszenario der Enterprise Integration Patterns für die Konfiguration von Apache ActiveMQ implementiert mittels anwendungsspezifischem Java.

Die Filterlogik des *Message Filters* und die Bedingungslogik des *Content-Based Routers* müssen schließlich noch entsprechend der Geschäftslogik angepasst werden. Nachdem die Zeilen 5, 10 und 12 angepasst sind, kann der `TODO`-Kommentar entfernt werden. Nun kann Apache ActiveMQ mit dieser Java-Datei so konfiguriert werden, dass das dargestellte Referenzszenario abgebildet ist.

3.2.2. Anforderungen

Äquivalent zum ersten Referenzszenario zeigt auch dieses Referenzszenario ähnliche Anforderungen auf. Grundlegend bleiben die Anforderungen *A1 - Der Nutzer kann einen Architekturentwurf erstellen und verändern*, *A2 - Erstellung von Musterimplementierung und Zuordnung zu Muster* sowie *A3 - Der Nutzer hat Einfluss auf die Selektion der Musterimplementierung*. Im ersten Referenzszenario war die Auswahl der Implementierung pro Muster ausreichend. Im zweiten Referenzszenario kommt hingegen dasselbe Muster mehrfach vor. Dies wird in der Anforderung *A5 - Mehrere Instanzen eines Musters* festgehalten. Diese Musterinstanzen haben eigenständige Beziehungen zu anderen Musterinstanzen. Zwei unterschiedliche Instanzen eines Musters können prinzipiell auch unterschiedlich implementiert werden. Deshalb soll die Musterimplementierung der Musterinstanz zugewiesen werden können. In diesem Referenzszenario kommt außerdem hinzu, dass der Architekturentwurf nicht nur eine Sequenz ist, sondern einen Graphen darstellt. Das bedeutet für die Aggregation der Musterimplementierung des *Content-Based Router* mit den Musterimplementierungen der *Point-to-Point Channels*, dass die Anzahl der Bedingungen an die Anzahl der verbunden Musterinstanzen angepasst werden muss. Bei der Definition der Musterimplementierungen muss deshalb die Anforderung *A6 - Der Architekturentwurf kann ein Graph sein* berücksichtigt werden.

3.3. Referenzszenario: Mehrere Mustersprachen

Das dritte Referenzszenario beschreibt ein Integrationsszenario. Das Softwaresystem in diesem Integrationsszenario soll nicht auf eigener Hardware betrieben werden, sondern hauptsächlich in einer Public Cloud. Ein möglicher Anwendungsfall ist die Verarbeitung von Wetterdaten. Eine Softwareanwendung empfängt Wetterdaten von verschiedenen Sensoren. Hierzu gehören auch Sensoren für Schneehöhen und Eisdicken auf Fahrbahnen, welche nur Daten senden sobald Schnee oder Eis vorhanden ist. Die Daten kommen in unterschiedlichen Formaten bei der Anwendung an. Bei Längen können diese Daten in den Einheiten Millimeter, Zentimeter oder Zoll angegeben sein. Entsprechend bei Temperaturen in den Einheiten Celsius oder Fahrenheit. Die Daten müssen deshalb normalisiert und auf ein einheitliches Format gebracht werden. Immer wieder gibt es auch fehlerhafte Messwerte, die verworfen werden sollen. Anschließend sollen die Daten öffentlichen und privatwirtschaftlichen Partnern zur Verfügung gestellt werden.

Die Anwendung für den Empfang der Sensordaten soll aufgrund der schwankenden Datenintensität auf einer *Elastic Platform* gehostet werden. Diese Anwendung ist mittels ihres *Message Endpoint* an eine MOM angebunden und sendet die Nachrichten an einen *Point-to-Point Channel*. Ein *Normalizer* empfängt diese Nachrichten und normalisiert die Einheiten. Anschließend wird die Nachricht mit den normalisierten Messwerten weitergesendet. Ungültige Nachrichten werden nun vom *Message Filter* verworfen. Die Nachrichten mit validen Messwerten werden auf einem *Publish-Subscribe Channel* publiziert. Dieser kann von internen und externen Anwendungen mittels *Message Endpoint* abonniert werden. Die Abbildung 3.4 visualisiert den Architekturentwurf, welcher auch die Kombination mehrerer Mustersprachen verdeutlicht.

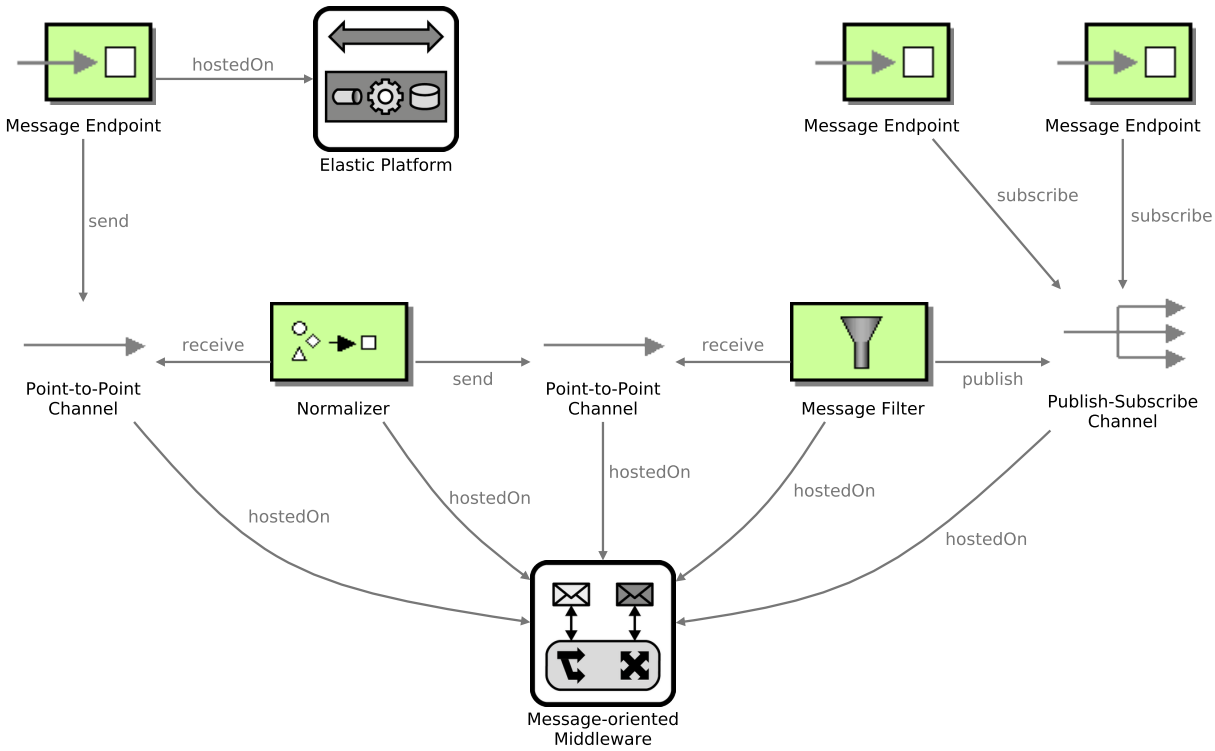

Abbildung 3.4.: Ein Architekturentwurf mit Mustern aus unterschiedlichen Mustersprachen.

3.3.1. Implementierung

CCP und EIP unterscheiden sich in den Abstraktionsebenen und der Domäne. Dennoch ist, wie im Referenzszenario dargestellt, eine Kombination der zwei Mustersprachen möglich und sinnvoll. Die Integration unterschiedlicher Mustersprachen erhöht die Komplexität bei der Aggregation der Musterimplementierungen. Im Gegensatz zu den bisherigen Referenzszenarien ist hier eine Mehrzahl unterschiedlicher Artefakte erforderlich. Während in weniger komplexen Szenarien eine Konfigurationsdatei für Apache ActiveMQ oder ein AWS CloudFormation Template eine mögliche Implementierung darstellten, besteht die Lösung in diesem Fall aus mehreren Dateien in unterschiedlichen Formaten. Die *Message Endpoints* können als Java-Klassen generiert werden. Die *Point-to-Point Channels*, der *Publish-Subscribe Channel*, der *Normalizer* und der *Message Filter* können durch eine XML-Konfigurationsdatei für Apache ActiveMQ implementiert werden. Dazu kann ein AWS CloudFormation Template generiert werden, welches eine *Message-oriented Middleware* mit der entsprechenden Konfiguration und einen *Message Endpoint* auf einer *Elastic Platform* hostet. Ein besonderes Merkmal dieses Referenzszenarios sind die Stellen, an denen verschiedene Mustersprachen aufeinander treffen. Wenn technisch möglich und fachlich sinnvoll, sollen die Musterimplementierungen über die Grenzen von Mustersprachen hinweg aggregiert werden können. Beispielsweise ist es in diesem Fall möglich, dass eine XML-Konfiguration für Apache ActiveMQ in das CloudFormation Template eingebettet wird. Ebenfalls denkbar ist die Definition einer ElasticBeanstalk Application im CloudFormation Template, sodass der Message Endpoint entsprechend bereitgestellt werden kann. Da die Implementierung dieses Referenzszenarios entsprechend umfangreich ist, wurde diese als Anhang A.1 dieser Arbeit angefügt.

3.3.2. Anforderungen

Die bereits vorgestellten Anforderungen aus den ersten beiden Referenzszenarien sind auch bei diesem Referenzszenario relevant. Zusätzlich kommen durch dieses Referenzszenario neue Anforderungen hinzu. Diese werden im Folgenden vorgestellt.

Aufgrund der Vielzahl an Kombinationsmöglichkeiten wird entweder ein Aggregator mit enormem Funktionsumfang benötigt oder eine Möglichkeit für mehrere einzelne Aggregatoren. Wenn einzelne Aggregatoren zum Einsatz kommen sollen, müssen diese passend zu den aggregierenden Musterimplementierungen gewählt werden. Hierfür wird ein automatisierter Selektionsmechanismus nötig. Mehrere einzelne Aggregatoren bieten den Vorteil, dass sie einfacher zu warten und zu erweitern sind. Deshalb wird auf mehrere einzelne Aggregatoren gesetzt, die passend der Musterimplementierungen gewählt werden. Diese Anforderungen sind in *A7 - Mehrere Aggregationsoperatoren* und *A8 - Erweiterbarkeit der Aggregationsoperatoren* festgehalten. Somit ist zudem die Erweiterbarkeit der Lösung sichergestellt.

Während bei den vorherigen Beispielen jeweils ein Aggregat in Form einer Ausgabedatei ausreichend war, benötigen wir in diesem Fall mehrere Ausgabedateien. Der Message Endpoint ist eine separate Datei, genauso wie die Konfiguration für Apache ActiveMQ oder AWS CloudFormation. Daher wird in der Anforderung *A9 - Mehrere Ausgabedateien* festgelegt, dass beliebig viele Ausgabedateien möglich sind.

3.4. Übersicht der Anforderungen

Im Folgenden sind alle Anforderungen aus den drei Referenzszenarien zusammengefasst. Die Anforderungen sind nummeriert und kurz erläutert. Hierdurch können sie in den nächsten Kapiteln eindeutig referenziert werden. Diese Anforderungen dienen als Grundlage für das Konzept und den Prototyp.

A1 - Der Nutzer kann einen Architekturentwurf erstellen und verändern

Ein Architekturentwurf aus Mustern muss erstellbar sein. Ebenso soll der Architekturentwurf ergänzt und verändert werden können. Dies umfasst das Hinzufügen und Entfernen von Mustern und den Beziehungen zwischen den Mustern.

A2 - Erstellung von Musterimplementierung und Zuordnung zu Muster

Einem Muster müssen sich eine oder mehrere Musterimplementierungen zuordnen lassen. Diese Musterimplementierungen bestehen in den vorgestellten Referenzszenarien aus Textfragmenten, wie beispielsweise Quellcode in Java. Prinzipiell sollen hier aber auch andere Formate, wie beispielsweise Binärdateien möglich sein.

A3 - Der Nutzer hat Einfluss auf die Selektion der Musterimplementierung

Der Nutzer benötigt Einfluss auf die gewählten Musterimplementierungen, damit der eigene Anwendungsfall bestmöglich erfüllt wird. Hierzu zählt beispielsweise die Wahl der Technologie der zu aggregierenden Musterimplementierungen.

A4 - Aggregationsoperator für die Aggregation der Musterimplementierung

Es soll mindestens ein Aggregationsoperator implementiert werden, der Musterimplementierungen aggregieren kann. Wie in den Referenzszenarien beschrieben, sollen beispielsweise Textfragmente zusammengeführt und verändert werden können.

A5 - Mehrere Instanzen eines Musters

Mehrere Instanzen eines Musters müssen im Architekturentwurf vorkommen dürfen. Beziehungen zwischen Mustern beziehen sich dabei nicht global auf das Muster, sondern auf die jeweilige Instanz des Musters.

A6 - Der Architekturentwurf kann ein Graph sein

Eine Sequenz an Mustern genügt nicht in allen Fällen um einen Architekturentwurf darzustellen. Ein Architekturentwurf kann ein Graph aus Mustern sein. Das bedeutet eine Musterinstanz kann mehrere ein- und ausgehende Kanten haben.

A7 - Mehrere Aggregationsoperatoren

Ein Aggregator kombiniert die Musterimplementierungen. Für eine bessere Erweiter- und Wartbarkeit, sollen mehrere Aggregatoren möglich sein. Der jeweils passende Aggregator wird mittels eines Selektionsmechanismus gewählt, abhängig von den zu aggregierenden Musterimplementierungen.

A8 - Erweiterbarkeit der Aggregationsoperatoren

Die Menge der Aggregatoren soll einfach erweiterbar sein. Um neue Musterimplementierungen für neue Technologien oder Mustersprachen zu unterstützen.

A9 - Mehrere Ausgabedateien

Bei der Aggregation eines Architekturentwurfs sollen mehrere Ausgabedateien möglich sein. Dies ermöglicht die Aggregation unterschiedlicher Technologien innerhalb eines Architekturentwurfs.

4. Konzept

Muster werden für neue Anwendungsfälle immer wieder neu implementiert. Hierfür werden Muster in der Softwarearchitektur identifiziert und Implementierungen für diese Muster erstellt oder, sofern vorhanden, bestehende wiederverwendet und gegebenenfalls angepasst [FBBL19]. Wenn hierfür kein Wissensmanagement besteht und existierende Implementierungen nicht bekannt oder verfügbar sind, entsteht ein hoher manueller Aufwand für die Neuimplementierung einer Lösung [FBBL19]. Im Folgenden wird ein Ansatz vorgestellt um diesen manuellen Aufwand zu reduzieren. Das Konzept hierfür umfasst drei essentielle Aspekte. (i) Die Architektur der Software kann in einem Entwurfsmodell mittels Muster und deren Beziehungen abgebildet werden. (ii) Musterimplementierungen sollen mit den Mustern verknüpft werden können. (iii) Der Prozess für die Aggregation soll weitgehend automatisiert werden. Diese drei Aspekte sind in Abschnitt 3.4 wiederum in neun Anforderungen gegliedert und detailliert beschrieben. Um das Ziel der automatisierten Aggregation von Musterimplementierungen zu erreichen, werden im folgenden Kapitel die hierzu genutzten Konzepte eingeführt und definiert. Zunächst werden die zwei Konzepte Entwurfsmodell und Musterinstanzen definiert. Anschließend wird die Zuordnung von Musterimplementierungen zu den Musterinstanzen betrachtet. Abschließend wird die Automatisierung der Aggregation der Musterimplementierungen beschrieben.

4.1. Entwurfsmodell

In den Referenzszenarien wurden Systementwürfe gezeigt um die Softwarearchitektur mit Mustern zu modellieren. Hierfür wird eine Modellierungssprache benötigt um diese Systementwürfe ausdrücken zu können. Die im Wesentlichen zu lösenden Herausforderungen sind in den Anforderungen in Abschnitt 3.4 beschrieben. Insbesondere die Anforderungen *A1 - Der Nutzer kann einen Architekturentwurf erstellen und verändern*, *A5 - Mehrere Instanzen eines Musters* und *A6 - Der Architekturentwurf kann ein Graph sein* sind für die Modellierung eines Systementwurfs relevant.

Zur Erfüllung der Anforderungen an die Modellierung von Systementwürfe mit Mustern werden Entwurfsmodelle eingeführt. Ein Entwurfsmodell ist ein Modell um mittels Beziehungen zwischen Mustern die Architektur einer Software zu beschreiben. Da ein Muster, beispielsweise ein *Point-to-Point Channel*, mehr als einmal in einem Softwaresystem auftreten kann, werden in einem Entwurfsmodell Instanzen von Muster verwendet. Instanzen von Mustern werden im Folgenden auch Musterinstanzen genannt. Dabei hat jede Musterinstanz einen eindeutigen Identifikator innerhalb des Entwurfsmodells. So wird das mehrfache Vorkommen und die eindeutige Identifizierbarkeit von Musterinstanzen innerhalb eines Entwurfsmodells ermöglicht. Hiermit wird die Anforderung *A5 - Mehrere Instanzen eines Musters* adressiert.

Eine Kante im Entwurfsmodell beschreibt die Beziehung zwischen zwei Musterinstanzen. Das kann beispielsweise „sende“ oder „empfange“ im Kontext von Messaging bei den EIP oder „hostedOn“ im Kontext der CCP sein. An den EIP zeigt sich auch direkt, dass die Semantik von expliziten Beziehungen wichtig ist, denn es ist ein wesentlicher Unterschied, ob eine Musterinstanz Nachrichten sendet oder empfängt. Die Art der Beziehung wird in diesem Konzept Beziehungstyp genannt und hat eine eindeutige Semantik. Die Beziehungen sind gerichtet und eine Musterinstanz kann beliebig viele ein- oder ausgehende Beziehungen besitzen. Um eine Eindeutigkeit sicherzustellen sind Mehrfachkanten, also mehrere Kanten zwischen zwei Musterinstanzen, nicht erlaubt. Ein Entwurfsmodell wird als gerichteter Graph ohne Mehrfachkanten definiert, bestehend aus Musterinstanzen (den Knoten) und ihren Beziehungen (den Kanten). Abbildung 4.1 zeigt das entsprechende Metamodell des Entwurfsmodells. So kann die Anforderung *A6 - Der Architekturentwurf kann ein Graph sein* erfüllt werden. Die Anforderung *A1 - Der Nutzer kann einen Architekturentwurf erstellen und verändern* wird insoweit erfüllt, dass ein Architekturentwurf mittels Entwurfsmodell modelliert werden kann.

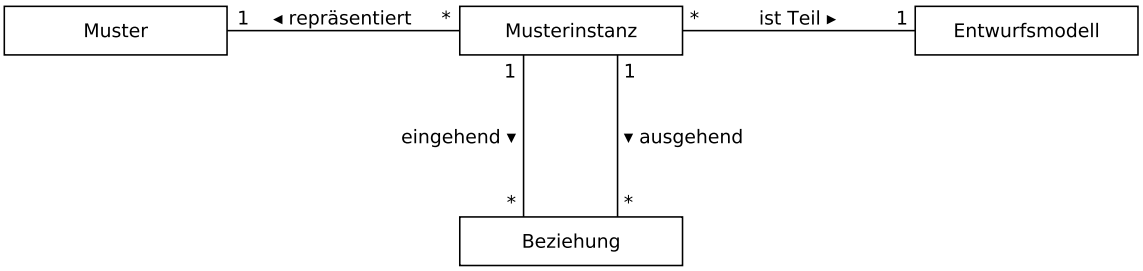

Abbildung 4.1.: Das Metamodell eines Entwurfsmodells.

Die Implementierung eines Editors zur Unterstützung der Erstellung und Bearbeitung von Entwurfsmodellen wird in Kapitel 5 ergänzt. Dazu zählt unter anderem eine gleichbleibende Positionierung von Musterinstanzen um eine gute Nutzbarkeit zu gewährleisten. Wird die Position der Musterinstanz vom Benutzer geändert, soll diese automatisch gespeichert werden.

4.2. Musterimplementierungen

Um Muster für neue Anwendungsfälle nicht wiederholt implementieren zu müssen, sollen Musterimplementierungen als wiederverwendbare Bausteine ermöglicht werden. Hierfür ist es für Musterimplementierungen nötig die Anforderung *A2 - Erstellung von Musterimplementierung und Zuordnung zu Muster* umzusetzen. Außerdem muss die Anforderung *A6 - Der Architekturentwurf kann ein Graph sein* berücksichtigt werden. Wenn fachlich korrekt, soll eine Musterimplementierung mehrere ein- oder ausgehende Kanten unterstützen. So können Entwurfsmodelle, die nicht einer Sequenz entsprechen, korrekt implementiert werden.

Eine Musterimplementierung besteht aus zwei Teilen wie in Abbildung 4.2 visualisiert. Der eine Teil ist das Musterimplementierungsartefakt (MIA) und der andere Teil der Musterimplementierungsdeskriptor (MID). Der MID enthält Metainformationen der Musterimplementierung. Eine der enthaltenen Metainformationen ist die Zuordnung zum implementierten Muster. In der Software Pattern Atlas lassen sich alle Muster über einen fachliche Identifikator in Form eines Uniform Resource

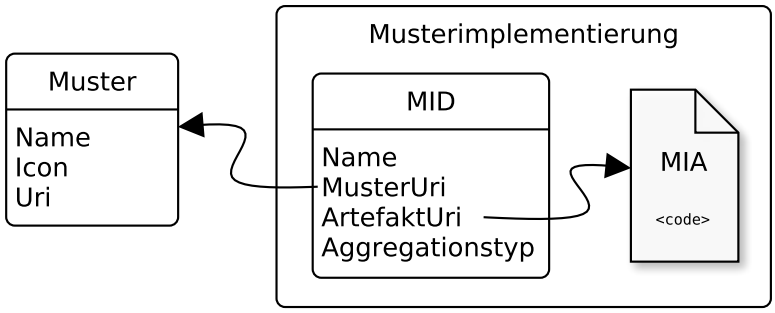

Abbildung 4.2.: Eine Musterimplementierung besteht aus einem Musterimplementierungsdeskriptor und einem Musterimplementierungsartefakt. Das implementierte Muster wird mittels ihres URI referenziert.

Identifier (URI) identifizieren. Beispielsweise entspricht der URI bei einer *Recipient List* dem Wert `https://patternpedia.org/patternLanguages/eip/recipientList`. Dieser URI wird als Fremdschlüssel im MID genutzt um die Musterimplementierungen dem Muster zuzuordnen. Auf diese Weise kann eine neue Musterimplementierung erstellt werden und, ohne das Muster zu verändern, diesem zugeordnet werden. Durch diese Zuordnung können dem Nutzer mögliche Musterimplementierungen für ein Muster zur Auswahl vorgeschlagen werden. Eine weitere Metainformation stellt die Referenz auf das MIA dar. Das MIA stellt die eigentliche Implementierung in Form eines Quellcodes, einer Konfigurationsdatei, eines ausführbaren Programms oder auch einen physischen Gegenstands dar. Bei physischen Gegenständen kann die Referenz eine Ortsangabe sein, wie beispielsweise die Nummer eines Lagerplatzes. Für die automatisierte Aggregation und digital verfügbaren Implementierungen bietet sich hierfür ein URI in Form eines Uniform Resource Locator (URL) an. So können diese Referenzen von den Aggregationsoperatoren aufgelöst und die Implementierungen abgerufen werden. Zu den weiteren enthaltenen Metainformationen gehört ein aussagekräftiger Name, damit der Nutzer verschiedene Implementierungen unterscheiden kann. Somit wird die Anforderung *A2 - Erstellung von Musterimplementierung und Zuordnung zu Muster* erfüllt.

Zur Erfüllung der Anforderung *A6 - Der Architekturentwurf kann ein Graph sein* müssen MIA entsprechend anpassbar sein. Beispielsweise erlaubt eine *Recipient List* beliebig viele Empfänger. Hierfür kann eine Liste der *Point-to-Point Channels* dienen, welche von den Empfängern gelesen werden. Ein MIA für Apache ActiveMQ mit der Java DSL für einen Empfänger kann wie folgt gestaltet sein:

```
from(...).recipientList(constant("activemq:<TODO>"));
```

Bei zwei Empfängern wird dies entsprechend angepasst:

```
from(...).recipientList(constant("activemq:<TODO>,activemq:<TODO>"));
```

Weitere Empfänger können kommasepariert angehängt werden. Um diese Anforderung zu erfüllen, sollen MIA eine einfache Möglichkeit für Bedingungen und Schleifen bieten. Auf diese Weise können MIA so flexibel entwickelt werden, dass sie sich im Kontext von Graphen verwenden lassen. Ein Aggregationsoperator kann das MIA bei der Aggregation dynamisch an die Anzahl der Kanten anpassen.

4.3. Selektion einer Musterimplementierung

Ein Entwurfsmodell, wie in Abschnitt 4.1 definiert, kann mehrere Instanzen eines Musters enthalten. Abbildung 4.3 zeigt ein solches Entwurfsmodell, mit zwei Musterinstanzen von M_2. Für Muster können, wie in Abschnitt 4.2 aufgezeigt, Musterimplementierungen entwickelt und zugeordnet werden. Dementsprechend existieren für ein Muster beliebig viele Musterimplementierungen. Der Nutzer soll zwischen den Alternativen eine Auswahl treffen können, damit für den konkreten Anwendungsfall die Musterimplementierungen passend sind. Hierbei muss berücksichtigt werden, dass diese Selektion für unterschiedliche Musterinstanzen möglicherweise unterschiedlich ausfällt. Beispielsweise ist die Auswahl von MI_c für eine Musterinstanz von M_2 und MI_d für eine andere Instanz von M_2 denkbar.

Um für Musterinstanzen mögliche Musterimplementierungen vorzuschlagen, muss zuvor das Muster ermittelt werden. Anschließend werden Musterimplementierungen gesucht, welche dieses Muster implementieren. Die gefundenen Musterimplementierungen können dem Nutzer entsprechend zur Selektion vorgeschlagen werden. Da mehrere Musterinstanzen eines Musters im Entwurfsmodell vorkommen, benötigt jede Musterinstanz einen eindeutigen Identifikator. So kann die Selektion der Musterimplementierung für jede einzelne Musterinstanz festgehalten werden.

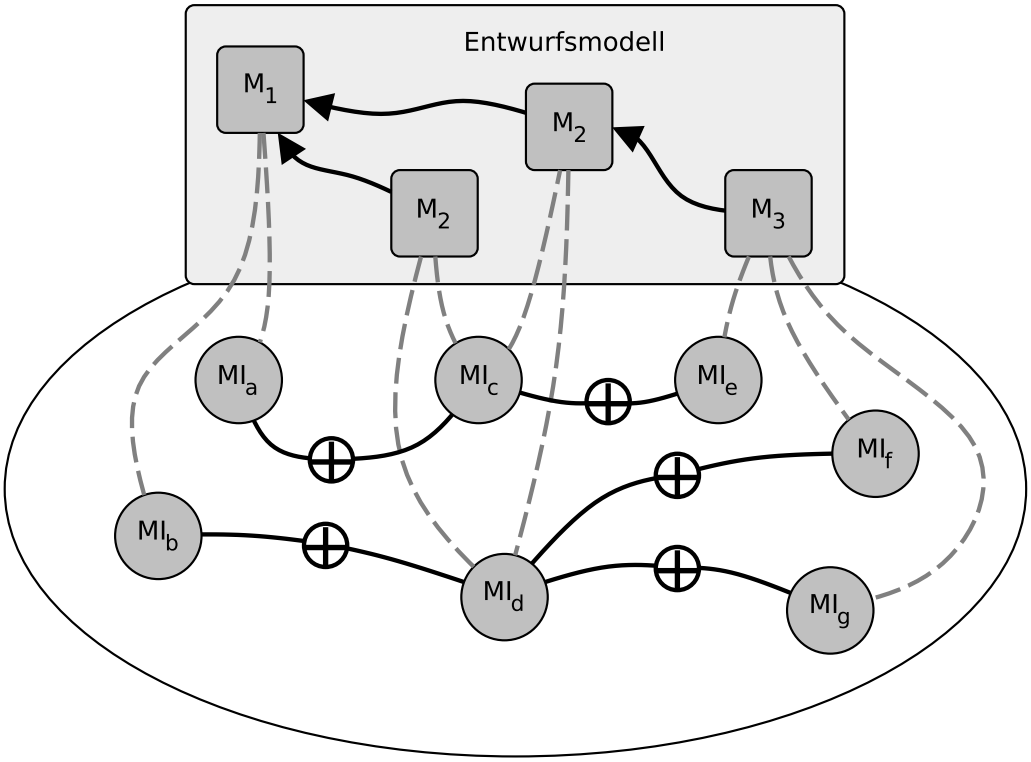

Abbildung 4.3.: Ein Entwurfsmodell aus Mustern mit den zugeordneten Musterimplementierung. Die Musterimplementierungen sind durch mögliche Aggregationsoperatoren verbunden.

Die Selektion ist damit eine Abbildung von der Musterinstanz auf die Musterimplementierung. Sei M die Menge der Musterinstanzen und MI die Menge der Musterimplementierungen, dann ist die Selektion eine Abbildung:

$$s : M \rightarrow MI$$

Auf diese Weise kann die Selektion der gewünschten Musterimplementierung durch den Nutzer beeinflusst werden und dieser hat die Kontrolle über die, für das Aggregat verwendeten, Musterimplementierungen. Beispielsweise kann der Nutzer beim in Abbildung 4.3 dargestellten Entwurfsmodell die folgende Auswahl treffen: Für M_1 wird MI_a, für beide Instanzen von M_2 wird MI_c und für M_3 wird MI_e selektiert. So kann anschließend der Graph $MI_c \rightarrow MI_a \leftarrow MI_c \leftarrow MI_e$ durch die Aggregationsoperatoren aggregiert werden. Dies erfüllt die Anforderungen *A3 - Der Nutzer hat Einfluss auf die Selektion der Musterimplementierung* und *A5 - Mehrere Instanzen eines Musters*.

4.4. Übersicht und Klassendiagramm

Dieser Abschnitt gibt eine kurze Übersicht des bisher vorgestellten Konzepts. Während die Mustersprache und die Muster bereits im Pattern Atlas implementiert sind, sollen die anderen Teile in einem darauf aufbauenden Prototyp implementiert werden. Hierfür wurde das Entwurfsmodell bestehend

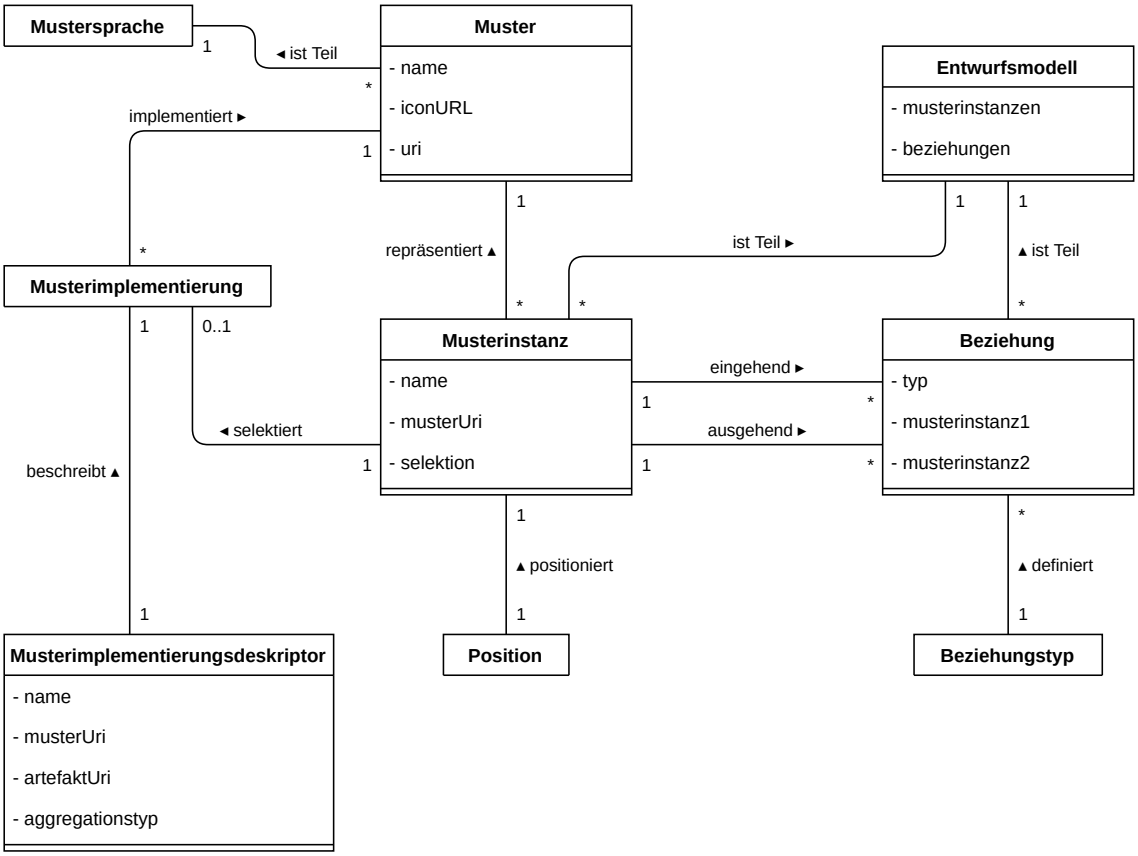

Abbildung 4.4.: Das Klassendiagramm des Konzepts mit den wichtigsten Attributen.

aus Musterinstanzen und Beziehungen in Abschnitt 4.1 konzipiert. Musterimplementierung und die Bestandteile MID und MIA wurden in Abschnitt 4.2 ergänzt. Abbildung 4.4 zeigt die relevanten Klassen des Konzepts und ihr Verhältnis zueinander. Auf dieser Grundlage wird im Folgenden die automatisierte Aggregation betrachtet.

4.5. Aggregationsgrundlagen

Die Musterimplementierungen und deren Selektion wurden im Vorigen konzipiert. Um die einzelnen selektierten Musterimplementierungen zu integrieren, wird ein Konzept für die Aggregation der Einzelteile benötigt. Das Konzept muss dabei die Anforderungen der Referenzszenarien erfüllen. Im Folgenden werden dafür Aggregationstypen und Aggregationsoperatoren vorgestellt. Darauf aufbauend wird das Konzept für die Aggregation eines Entwurfsmodells entworfen.

4.5.1. Aggregationstyp

Für die Kombination von Musterimplementierungen kommen Aggregationsoperatoren zum Einsatz. Falkenthal et al. [FBB+14a] stellten hierfür das Konzept von Aggregationsoperatoren vor, im Folgenden auch Aggregatoren genannt. Prinzipiell kann ein „Superaggregator“ existieren, welcher jegliche Musterimplementierungen aggregieren kann. Oder im Gegensatz dazu ein Aggregator, welcher nur zwei bestimmte Muster aggregieren kann. Um die Wartbarkeit und Erweiterbarkeit zu verbessern und die Komplexität zu reduzieren, sollen mehrere Aggregationsoperatoren möglich sein. Die Grenzen können so gezogen werden, dass zusammenhängende Musterimplementierungen mit ähnlichen Charakteristiken gruppiert werden können. Falkenthal et al. [FBBL19] definieren hierfür eine Lösungsalgebra und Trägermenge. Eine Trägermenge ist eine Teilmenge der Lösungssprache und umfasst Musterimplementierungen mit ähnlichen essentiellen Charakteristiken. Diese Charakteristik kann beispielsweise dieselbe DSL sein. Eine Lösungsalgebra ist ein 2-Tupel, bestehend aus einem Tupel von Trägermengen und einem Tupel von Aggregationsoperatoren.

Trägermengen und dazu passende Aggregatoren können die Entwicklung von Software für die automatisierte Aggregation vereinfachen [FBBL19]. Dies ergibt sich daraus, dass verschiedene Technologien unterschieden beziehungsweise ähnliche Charakteristiken der Musterimplementierungen gruppiert werden. Hierfür werden Trägermengen aus Musterimplementierungen anhand eines Aggregationstyps definiert. Aggregationstypen sind Schlüsselwerte, die gleichartige Musterimplementierungen gruppieren und somit einer Trägermenge zuordnen. Diese können beispielsweise Technologien wie AWS CloudFormation Template oder eine Apache ActiveMQ Konfiguration umfassen. Abbildung 4.5 stellt die Trägermengen $\mathtt{TM}_1$, $\mathtt{TM}_2$, $\mathtt{TM}_3$ und $\mathtt{TM}_4$ dar, die unterschiedlichen Technologien für die Implementierung unterschiedlicher Mustersprachen dienen. Zur Definition der Zugehörigkeit einer Musterimplementierung zu einer Trägermenge, wird das Attribut Aggregationstyp im MID verwendet, wie in Abbildung 4.2 dargestellt.

4.5.2. Aggregationsoperatoren

Alle Musterimplementierungen, die Muster einer Mustersprache implementieren, sind in einer Lösungssprache enthalten. Die Trägermengen einer Lösungsalgebra sind als Teilmengen der Lösungssprache definiert [FBBL19]. Die Musterimplementierungen dieser Teilmengen sind anhand ihres Aggregationstyps definiert und zugeordnet. Da die Aggregationstypen die Charakteristik der Musterimplementierungen berücksichtigen, sind diese technisch ähnlich. Deshalb kann ein Aggregationsoperator zwei oder mehr Elemente einer Trägermenge in ein Element der Trägermenge abbilden [FBBL19]. Ein Beispiel hierfür ist das Zusammenführen von zwei Fragmenten von AWS CloudFormation Templates in ein neues AWS CloudFormation Template.

In Abschnitt 3.3 wurde aufgezeigt, dass es sinnvoll sein kann, mehrere Mustersprachen zu kombinieren. In dem Referenzszenario ist eine Aggregation von AWS CloudFormation Template und Apache ActiveMQ XML-Konfiguration möglich und fachlich sinnvoll. Deshalb sollen Aggregationsoperatoren prinzipiell Musterimplementierungen aus unterschiedlichen Lösungssprachen oder Trägermengen aggregieren können. Beispielsweise wird dies in Abbildung 4.5 mit dem Aggregationsoperator zwischen TM_1 und TM_3 veranschaulicht. Die Muster M_1 und M_2 stammen aus der Mustersprache MS_1. Die Musterimplementierungen von M_1 und M_2 sind jeweils in der Trägermenge TM_1 und TM_2 enthalten. MI_e und MI_f bilden separate Trägermengen als Teil einer anderen Mustersprache MS_2. Deshalb wird

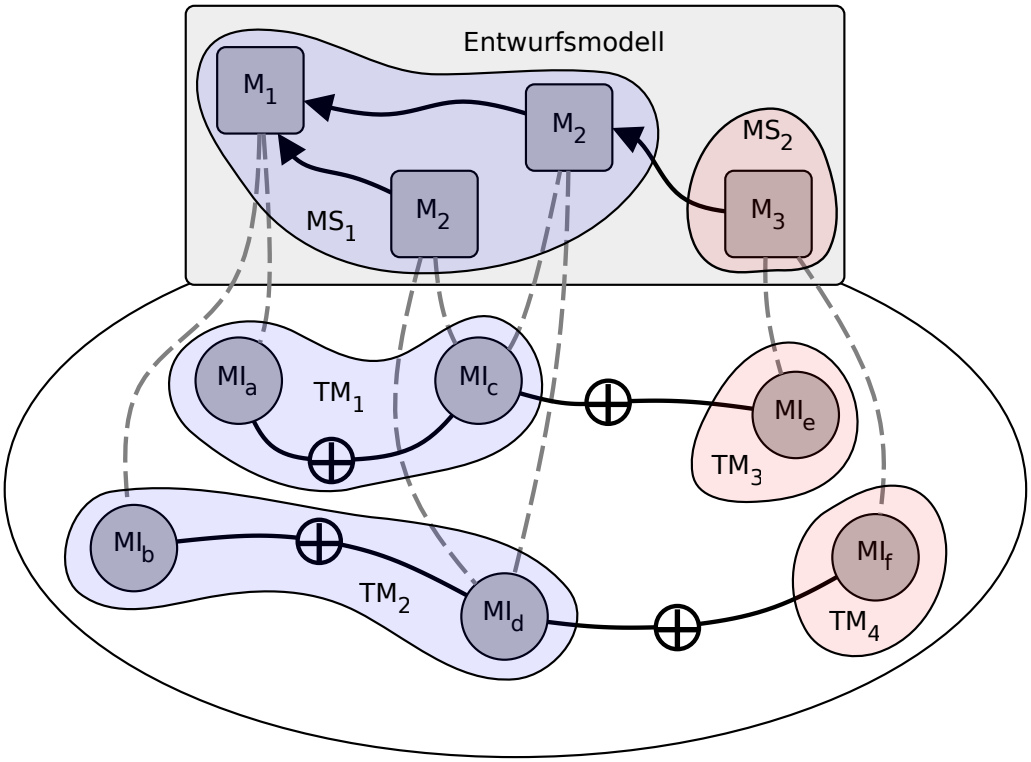

Abbildung 4.5.: Ein Entwurfsmodell aus Mustern mit den zugeordneten Musterimplementierungen. Aggregationstypen gruppieren Musterimplementierungen anhand ähnlichen Charakteristiken, beispielsweise anhand derselben domänenspezifische Sprache.

zusätzlich vorgesehen, dass Aggregationsoperatoren zwischen Trägermengen existieren können. Damit besteht die Lösungsalgebra im Beispiel von Abbildung 4.5 aus den Trägermengen $\mathtt{TM}_1$ bis $\mathtt{TM}_4$ und den vier Aggregationsoperatoren $\oplus$.

Die Aggregatoren müssen um die Informationen bereichert werden, welche Musterimplementierungen aggregiert werden können. Hierfür werden die Aggregationstypen genutzt. So wird definiert, welche Trägermenge(n) ein Aggregator aggregieren kann. Auf diese Weise können bei der Aggregation die passenden Aggregationsoperatoren anhand des Aggregationstyps ausgewählt werden. Beispielsweise kann ein Aggregationsoperator für AWS CloudFormation Templates existieren. Möglich sind auch zwei oder mehr verschiedene Aggregationstypen. Auf diese Art können unterschiedliche Trägermengen beziehungsweise Technologien aggregiert werden, wie in Abbildung 4.5 dargestellt.

Die Größe einer Trägermenge und der Funktionsumfang eines Aggregationsoperators können sich gegenseitig beeinflussen. Je größer die Varianz der Charakteristiken innerhalb einer Trägermenge, desto umfangreicher muss der Aggregator sein, um die Unterschiedlichkeiten der einzelnen Musterimplementierungen zu beherrschen.

Semantik der Beziehungen im Entwurfsmodell

Ein Entwurfsmodell aus Mustern der EIP wird mit expliziten Beziehungen modelliert. Dies ist nötig um ein eindeutiges Entwurfsmodell zu erhalten. Beispielsweise ist es, wie im zweiten Referenzszenario erwähnt, ein relevanter Unterschied, ob ein *Message Filter* eine Nachricht empfängt oder sendet. Insofern ist der Beziehungstyp und die Richtung der Beziehung für die Semantik des Entwurfsmodells relevant und muss bei der Aggregation berücksichtigt werden. Die Semantik kann entweder im Aggregationsoperator behandelt werden oder der Graph aus Musterimplementierungen kann aus technischer Sicht vereinheitlicht werden. Beispielsweise können die Richtung und der Typ der Beziehungen so gespiegelt werden, dass der Nachrichtenfluss abgebildet wird. Wird die Semantik auf diese Weise normalisiert, bevor die Musterimplementierungen aggregiert werden, muss die Beziehungssemantik nicht in jedem Aggregationsoperator behandelt werden.

Konzept für die Aggregationsoperatoren

Die mathematischen Eigenschaften von Aggregationsoperatoren variieren je nach Domäne. Beispielsweise sind TOSCA Topology Models kommutativ, aber nicht assoziativ. Management Planlets hingegen sind weder kommutativ noch assoziativ. [FBBL19]

Betrachtet man die Musterimplementierungen der EIP ohne eine definierte Reihenfolge, so ist die Richtung des Nachrichtenflusses unklar. Deshalb wird diese Information bei der Aggregation von zwei Musterimplementierungen benötigt. Sei TM eine Trägermenge, A ein Artefakt und der Aggregator mit $\oplus : TM \times TM \rightarrow A, (mi_1, mi_2) \mapsto a$ definiert, dann muss eine Konvention für den Nachrichtenfluss festgelegt sein. Diese kann in Richtung von mi_1 zu mi_2 oder umgekehrt sein. Damit ist $\oplus$ nicht kommutativ. Die *Message Channels* benötigen bei der Aggregation eindeutige Namen, sonst kann kein Nachrichtenfluss sichergestellt werden, der dem Entwurfsmodell entspricht. Die Reihenfolge der einzelnen Musterimplementierungen innerhalb einer Apache ActiveMQ XML-Konfiguration oder dem Äquivalent in der Java DSL ist unbedeutend. Allerdings werden beispielsweise bei einem *Content-Based Router* alle „nachrichtensendende“ Beziehungen

benötigt, damit das Konfigurationsfragment korrekt generiert werden kann. Hierfür müssen diese bekannt sein um entsprechend eingefügt werden zu können. Diese Beziehungen können mit einer separaten Suche auf dem Graphen oder durch die Beachtung der Reihenfolge bei der Aggregation ermittelt werden.

Aufgrund der dargelegten Charakteristiken der Musterimplementierungen, wird dem Konzept im Rahmen dieser Arbeit zugrunde gelegt, dass Aggregationsoperatoren nicht kommutativ oder assoziativ sind. So lassen sich mehr Anwendungsfälle abdecken und die Implementierung bleibt generischer. Die untersuchten CCP und EIP lassen sich so als AWS CloudFormation Template beziehungsweise Apache ActiveMQ Konfigurationen aggregieren.

Allgemein lässt sich der Aggregationsoperator als Funktion darstellen. Sei A eine Menge von Artefakten, beispielsweise XML-Fragmente. Seien TM_i und TM_j eine Trägermenge und Teilmenge von A. Der Aggregationsoperator $\oplus$ ist eine Abbildung:

$$\oplus : TM_i \times TM_j \rightarrow A$$

Dabei ist $\oplus$ im Allgemeinen nicht assoziativ und nicht kommutativ.

Es folgen drei spezifische Beispiele um diese Definition zu erläutern. Werden zwei Musterimplementierungen vom Typ Apache ActiveMQ XML-Konfiguration aggregiert, gilt $i = j$ und A kann auf ihre Teilmenge TM_i beschränkt werden. In diesem Fall sei TM_1 die Menge aller XML-Fragmente für Apache ActiveMQ. Dann gilt:

$$\oplus_1 : TM_1 \times TM_1 \rightarrow TM_1$$

Daran wird sichtbar, dass dieser Aggregationsoperator wiederholt angewandt werden kann, um eine dritte Musterimplementierung aus TM_1 mit dem Ergebnis zu aggregieren. Nun soll das Ergebnis in ein AWS CloudFormation Template eingebettet werden. Sei TM_2 die Menge der AWS CloudFormation Template-Fragmente, so gilt für das Aggregieren der zwei unterschiedlichen Technologien:

$$\oplus_2 : TM_1 \times TM_2 \rightarrow TM_2$$

Ebenfalls kann ein Aggregator zwei unterschiedliche Quellcodefragmente verknüpfen und diese kompilieren. Auf diese Weise kann ein Artefakt entstehen, welches nicht weiter aggregierbar ist. Sei TM_3 die Menge der Quellcodefragmente, dann gilt:

$$\oplus_3 : TM_3 \times TM_3 \rightarrow A \setminus TM_3$$

Die Anforderung *A4 - Aggregationsoperator für die Aggregation der Musterimplementierung* wird durch die festgelegten Eigenschaften der Aggregatoren erfüllt. Aufgrund der Anforderungen *A7 - Mehrere Aggregationsoperatoren* und *A8 - Erweiterbarkeit der Aggregationsoperatoren* werden die Aggregatoren so zugeschnitten, dass sie ähnliche Musterimplementierungen aggregieren können. Im Rahmen dieser Arbeit wird jeweils ein Aggregator für die Aggregation von AWS CloudFormation Templates und ein Aggregator für die Java-DSL beziehungsweise XML-Konfiguration von Apache ActiveMQ erstellt.

4.6. Konzept für den Aggregationsalgorithmus

In diesem Abschnitt wird der Prozess für die Aggregation der Musterimplementierungen eines Entwurfsmodells beschrieben und abschließend um alternative Konzepte ergänzt. Nachdem ein Nutzer ein Entwurfsmodell erstellt und die Selektion der Musterimplementierungen abgeschlossen hat, kann die Aggregation beginnen. Zur Vorbereitung werden die selektierten Musterimplementierungen auf den Graph aus Musterinstanzen übertragen. Der daraus resultierende Graph ist in Abbildung 4.6i dargestellt. Als Vorbereitung wird die Semantik der Beziehungen vereinheitlicht. Beispielsweise bietet sich für die Beziehungen zwischen Musterinstanzen der EIP an, eine einheitliche Semantik durch die Richtung der Kante herzustellen.

Die Semantik der Beziehung aus dem Entwurfsmodell soll nicht in den Aggregationsoperatoren gehandhabt werden. Die Aggregationsoperatoren arbeiten auf dem vereinheitlichten Graph aus Musterimplementierungen. Dieser Graph aus Musterimplementierungen entspricht der Semantik des Entwurfsmodells, kann sich aber teilweise in den Richtungen der Kanten unterscheiden. Bei der Aggregation werden zunächst Musterimplementierungen, ohne eingehende Kanten, mit ihren Nachbarn aggregiert. Die Aggregationsschritte werden wiederholt, wobei der Graph bei jedem Schritt um eine Musterimplementierung reduziert wird. Daten, welche möglicherweise in folgenden Aggregationsschritten benötigt werden, können in einem Datencontainer gespeichert werden. Dieser Datencontainer wird im Folgenden Kontext genannt und existiert bis die Aggregation abgeschlossen ist. Der Kontext wird für Zwischenergebnisse in Form von Textfragmenten, zu setzende Referenzen oder ähnliches verwendet. Eine Referenz ist ein Identifikator einer Komponente, beispielsweise der eindeutige Namen eines *Message Channels*.

Im Folgenden wird der beschriebene Aggregationsprozess exemplarisch an Abbildung 4.6 durchgeführt. Im ersten Schritt wird $\texttt{MI}_\texttt{c}$ und $\texttt{MI}_\texttt{e}$, verbunden durch die Kante $\texttt{E}_3$, aggregiert. Zunächst werden die Aggregationstypen der Musterimplementierungen ausgewertet. Unter der Annahme von Abbildung 4.5 sind dies zwei unterschiedliche Aggregationstypen, im Folgenden $\texttt{AT}_1$ bei $\texttt{MI}_\texttt{c}$ und $\texttt{AT}_2$

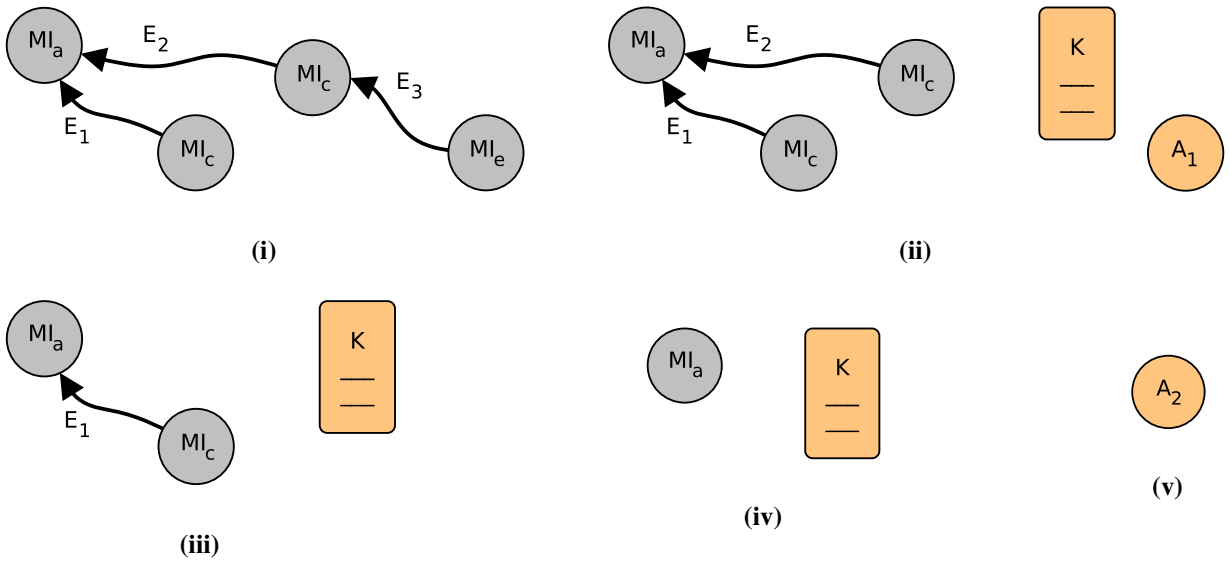

Abbildung 4.6.: Schematische Darstellung des Aggregationsprozesses.

bei MI_e. Nun wird nach einem passenden Aggregator gesucht, welcher AT_1 mit AT_2 aggregieren kann. Ist der Aggregator gefunden, werden die zwei Musterimplementierungen MI_c und MI_e übergeben. Der Aggregationsoperator speichert für weitere Aggregationsschritte relevante Daten im Kontext. Zusätzlich gibt er ein Aggregat A_1 zurück, wie in Abbildung 4.6ii dargestellt. A_1 ist das Ergebnis des ersten Aggregationsschritts in Form einer Datei. Die Datei wird im weiteren Verlauf des Aggregationsprozesses von den Aggregatoren nicht mehr berücksichtigt. Anschließend wird dieser Schritt für die Knoten der Kante E_2 wiederholt. Dieses Mal handelt es sich um zwei Musterimplementierungen mit demselben Aggregationstyp AT_1. Ein passender Aggregationsoperator kann nun die Musterimplementierung MI_c für MI_a anpassen und gegebenenfalls mit Daten aus dem Kontext zusammenfügen oder anreichern. Das Zwischenergebnis wird ebenfalls im Kontext abgelegt, da noch kein endgültiges Aggregat feststeht, beispielsweise wenn weitere XML-Fragmente hinzugefügt werden sollen. Das Vorgehen wird für die Musterimplementierungen MI_c aus Abbildung 4.6iii wiederholt. Ein Aggregator für den Aggregationstyp AT_1 passt die Musterimplementierung MI_c an MI_a an und führt MI_c mit dem bestehenden Zwischenergebnis zusammen. Abschließend wird der Aggregationsschritt für MI_c aus Abbildung 4.6iv durchgeführt. Nun wird ein Aggregator benötigt, der Musterimplementierungen mit Aggregationstyp AT_1 „abschließen“ kann. Der Aggregator passt MI_a an und fügt es mit dem bisherigen Zwischenergebnis zusammen, wodurch das Aggregat A_2 entsteht, wie in Abbildung 4.6v gezeigt. Die Artefakte A_1 und A_2 sind somit das Ergebnis des Aggregationsalgorithmus. Hierin sind die vier Musterimplementierungen vereint. Diese zwei Artefakte werden nun an den Nutzer zurückgegeben. Der Vorteil dieser Lösung ist die Nutzung der Semantik des Entwurfsmodells um Aggregatoren nutzen zu können, welche nicht direkt die Datenstruktur eines Graphen handhaben müssen.

Ein alternatives Konzept für den Aggregationsprozess, um die Aggregation stärker von der Semantik des Entwurfsmodells zu entkoppeln, ist die Kontraktionsreihenfolge im Graphen der Musterimplementierung anhand anderer Metriken zu bestimmen. Dazu kann eine Operatorrangfolge, also eine Priorisierung für die Aggregationsoperatoren, definiert werden. Die bekannteste Operatorrangfolge ist vermutlich die Punkt- vor Strichrechnung bei den Grundrechenarten. Überträgt man dieses Prinzip auf die Aggregation, gibt die Operatorrangfolge die Reihenfolge vor, in der zwei Musterimplementierungen aggregiert werden. So kann für jede Kante ein Gewicht ermittelt und anhand dessen die Sortierung für die Aggregation vorgenommen werden. Neu entwickelte Aggregatoren müssen mit korrekter Priorisierung eingeordnet werden. Während bei einer Sequenz ein Tupel an Musterimplementierungen an den Aggregator übergeben werden kann, sind bei einem Graphen zusätzlich die Kanten relevant. Nachteilig an diesem Vorgehen ist, dass der Kontext für die Aggregation vorab generiert werden muss oder der Aggregator einen (Teil-)Graphen als Eingabe erhält, anstatt eines Tupels aus Musterimplementierungen. Um beispielsweise das XML-Fragment für einen *Content-Based Routers* zu generieren, müssen alle ausgehenden Nachrichtenflüsse bekannt sein. Aus diesen Gründen wurde diese zweite Alternative verworfen.

Eine dritte Alternative ist ein Datenmodell, welches das XML-Schema abbildet. Dieses kann dann mit jeder Musterimplementierung ergänzt werden und abschließend in eine XML-Datei serialisiert werden. Äquivalent kann dies für JSON-Dateien umgesetzt werden. Diese Alternative wurde verworfen, da umfangreiche Datenmodelle für die entsprechenden Domänen erstellt werden muss, was einen entsprechend hohen Aufwand an Entwicklung und Wartung nach sich zieht.

Die erste Variante bietet die Möglichkeit den Graphen, durch Nutzung der vorhandenen Semantik des Entwurfsmodells, zu aggregieren und erfüllt damit auch für die Aggregation die Anforderung *A6 - Der Architekturentwurf kann ein Graph sein*. Die Komplexität des Graphen muss nicht in den

Aggregatoren gehandhabt werden. Dies bezieht die Anforderungen *A7 - Mehrere Aggregationsoperatoren* und *A8 - Erweiterbarkeit der Aggregationsoperatoren* mit ein. Außerdem können wie gezeigt *A9 - Mehrere Ausgabedateien* generiert werden.

5. Prototyp

Dieses Kapitel stellt den für diese Arbeit entwickelten Prototyp vor, insbesondere den Editor für Entwurfsmodelle und den Aggregationsalgorithmus. In diesem Zusammenhang wird auch auf Implementierungsdetails und spezielle Herausforderungen eingegangen. Abschließend wird der Prototyp hinsichtlich der Anforderungen evaluiert, welche in Kapitel 3 aus den Referenzszenarien erhoben wurden.

5.1. Softwarearchitektur des Prototyps

Der Prototyp lässt sich in vier Komponenten unterteilen, wie in Abbildung 5.1 dargestellt. Die open-source Software Pattern Atlas besteht aus zwei Komponenten. Die Pattern Atlas UI bildet die Benutzerschnittstelle in Form einer Webanwendung und wird mit dem Framework Angular [Gooa] entwickelt. In dieser Komponente ist beispielsweise der grafische Editor für Entwufsmodelle enthalten. Für die Webanwendung stellt die Komponente Pattern Atlas API eine Representational State Transfer (REST)-Schnittstelle bereit. Die Pattern Atlas API ist eine Java Anwendung, die mit dem Framework Spring Boot [VMw] entwickelt wird. Diese Komponente dient der Verwaltung der Daten. Dazu zählen die Mustersprachen, Muster, Entwurfsmodelle, Musterinstanzen, Beziehungen, etc. Für die Persistenz der Daten kommt eine relationale Datenbank zum Einsatz, die mittels Structured Query Language (SQL) angesprochen wird. Dies ist standardmäßig die PostgreSQL-Datenbank [The]. Für die Verwaltung des MIA wird ein Git-Repository genutzt. Das Repository [Grab] für die MIAs,

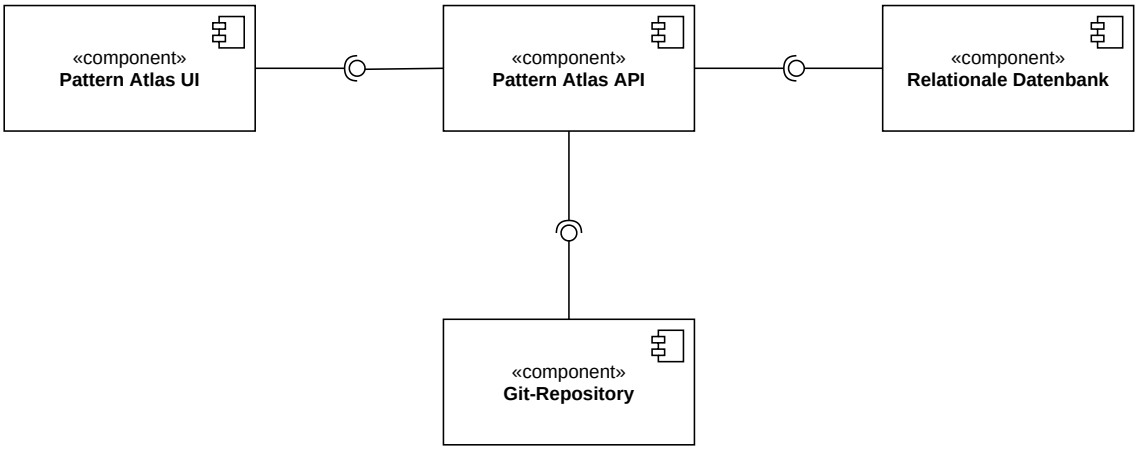

Abbildung 5.1.: Das Komponentenmodell des Prototypen, bestehend aus der Benutzerschnittstelle Pattern Atlas UI, der Pattern Atlas API für die Datenverwaltung, einer relationalen Datenbank für die Persistenz der Daten und einem Git-Repository für die Verwaltung von Musterimplementierungsartefakten.

die im Rahmen des Prototypen entwickelt wurden, wird auf GitHub [Gita] gehostet. Prinzipiell sind auch andere Speicherorte möglich, sofern das MIA mittels einer URL abgerufen werden kann. Git-Repositories bieten einen gewohnten Arbeitsfluss bei Softwareentwicklern und ermöglichen die einfache Weiterentwicklung des MIA, ohne den MID anzupassen. Bei der Aggregation der Musterimplementierungen kann die Pattern Atlas API die benötigten MIA mittels Hypertext Transfer Protocol (HTTP) aus dem Git-Repository laden.

5.2. Entwurfsmodell

Bisher unterstützt Pattern Atlas die Darstellung von Mustern und ihren Beziehungen. Für die Erstellung eines Entwurfsmodells wurde ein grafischer Editor, aufbauend auf Pattern Atlas, entwickelt. So können die bestehenden Muster für die Erstellung eines Entwurfsmodells genutzt werden.

Hierfür wurde die Software erweitert, so dass die in Abschnitt 4.1 vorgestellten Musterinstanzen genutzt werden können. Die bestehende Visualisierungskomponente für Mustergraphen wurde um die benötigen Anforderungen weiterentwickelt. So wird beim Hinzufügen eines Musters zum Entwurfsmodell eine Musterinstanz aus dem Muster generiert und gespeichert. Diese Musterinstanzen können im Systementwurf frei positioniert werden. Um das Entwurfsmodell im aktuellen Erscheinungsbild zu speichern, wurde eine Persistenz für die Positionierung der Musterinstanzen hinzugefügt. Wird eine Musterinstanz oder eine Beziehung nicht mehr benötigt, so kann diese wieder gelöscht werden. Insgesamt wurden für einen Editor typische Funktionen ergänzt, wie beispielsweise das Hinzufügen, das Löschen und die Positionsänderung von Musterinstanzen sowie das Entfernen von Kanten.

Mit diesem Editor können Nutzer einen Architekturentwurf eines Softwaresystems mittels Mustern entwerfen und bearbeiten. Dabei können Muster mehrfach vorkommen und über eine beliebige Anzahl an Beziehungen zu anderen Musterinstanzen verfügen. Somit lassen sich auch Architekturentwürfe mit der Charakteristik eines Graphen darstellen. Abbildung 5.2 zeigt das Entwurfsmodell für das dritte Referenzszenario aus Abschnitt 3.3, wie es mit dem Editor in Pattern Atlas modelliert werden kann.

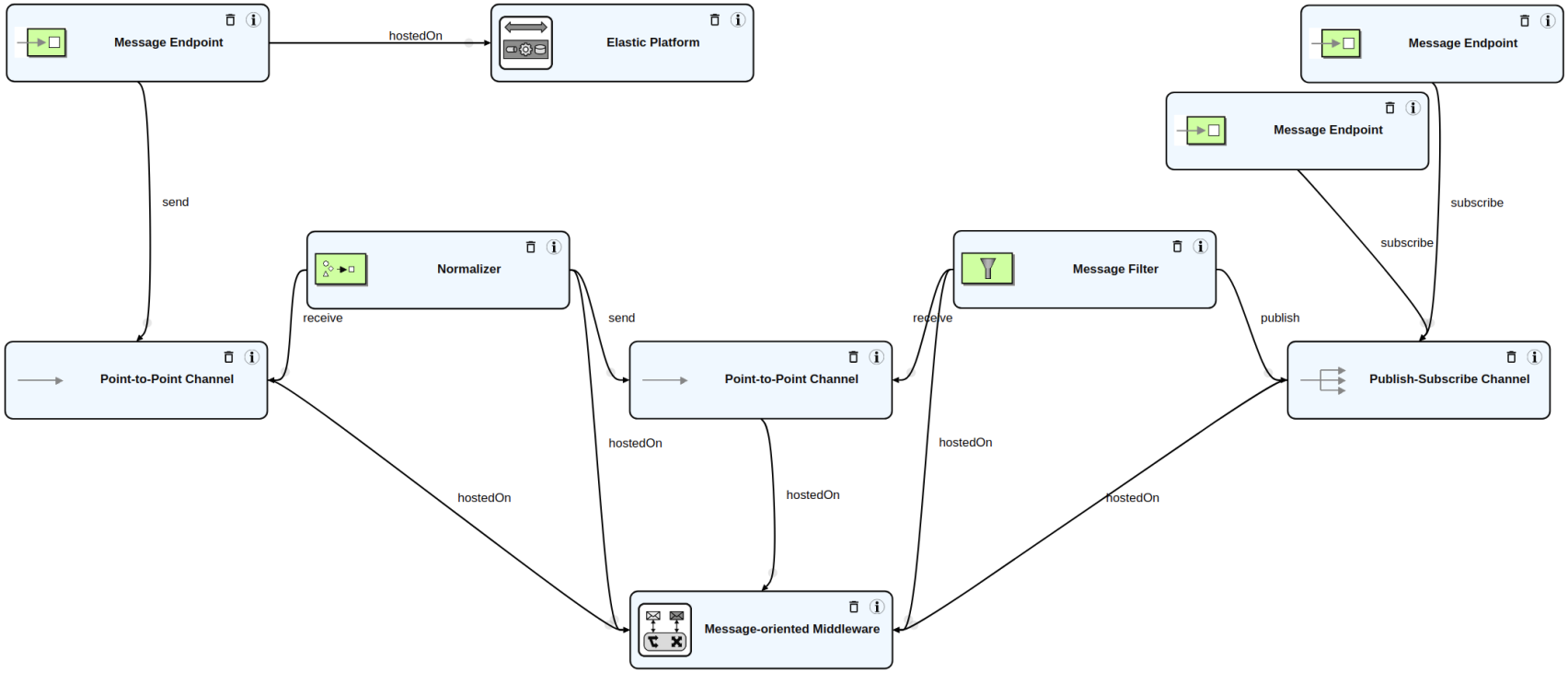

Abbildung 5.2.: Darstellung des Entwurfsmodells für das dritte Referenzszenario im Pattern Atlas. Die Modellierung erfolgt mit Musterinstanzen und deren expliziten Beziehungen.

5.3. Musterimplementierungen

Die Musterimplementierungen bestehen aus zwei Teilen. Der MID enthält alle benötigten Metadaten für eine Musterimplementierung und das MIA stellt das eigentliche Implementierungsartefakt dar. Der MID besteht aus einem Namen, einer URI um das implementierte Muster zu referenzieren, den Aggregationstyp und der URI des MIA. Diese Daten werden in der relationalen Datenbank gespeichert.

Das MIA wird im MID anhand einer URI referenziert. Für die Musterimplementierungen in diesem Prototyp wird hierfür ein URL genutzt. Damit kann das MIA flexibel bereitgestellt werden, beispielsweise in einem Git-Repository, im lokalen Dateisystem oder auf einem Webserver. Für open-source MIAs bieten sich öffentliche Repositories an, solche werden unter anderem von GitHub oder GitLab [Gitb] angeboten. Proprietäre MIA können entsprechend wie nichtöffentlicher Quellcode behandelt werden. Die Nutzung eines Versionsverwaltungssystems für MIA bietet Vorteile. Die Veränderung eines MIA kann nachvollzogen werden und die weitverbreitete und weitreichend etablierte Zusammenarbeit zwischen Softwareentwicklern mit Merge Requests und Reviews wird ermöglicht.

Grundsätzlich bedingen sich das Format des MIA und der Aggregator gegenseitig, denn dieser muss das Format verstehen um es verarbeiten und mit anderen MIAs aggregieren zu können. In den Aggregatoren in diesem Prototyp werden Textformate und ein Templatesystem unterstützt. Als Templatesystem wird StringTemplate [Par] genutzt, da es die Generierung von Quellcode verschiedener Sprachen ermöglicht. So können beispielsweise sowohl XML-Dateien, als auch Java-Dateien generiert werden. Listing 5.1 zeigt ein solches Template mit einer Schleife um einen *Content-Based Router* für Apache ActiveMQ zu konfigurieren, wie es beispielsweise im Listing 3.6 vorgestellt wurde.

```
1  from("activemq:$input$")
2    .choice()
3  $output:{op |
4      .when(xpath("/sensor/type = 'temp-celsius'")) // TODO: adapt filter rule
5        .to("activemq:$op$")
6  }$
7      .otherwise()
8        .to("activemq:deadletterchannel");
```

Listing 5.1: Ein Musterimplementierungsartefakt eines Content-Based Router für die Konfiguration von Apache ActiveMQ.

5.4. Selektion einer Musterimplementierung

Die Selektion einer Musterimplementierung ist dreistufig aufgebaut. Anhand des im Entwurfsmodell verwendeten Musters wird nach passenden Musterimplementierungen für eine Musterinstanz gesucht. So werden nur passende, dieses Muster implementierende, Musterimplementierungen vorgeschlagen. Optional kann der Nutzer einen globalen Filter nutzen um die Vorschläge an Musterimplementierungen für das gesamte Entwurfsmodell einzuschränken. Damit lässt sich die Selektion der Musterimplementierungen für alle Musterinstanzen gleichzeitig beeinflussen. Dieser Filter

ermöglicht auch komplexe Filterungen mittels JavaScript-Unterstützung. Die dritte Stufe ist eine Selektion der Musterimplementierung individuell für einzelne Musterinstanzen. So kann für zwei Musterinstanzen des gleichen Musters zwei unterschiedliche Musterimplementierungen selektiert werden. Diese Selektion überschreibt, sofern vorhanden, die globalen Filterergebnisse, sodass beides kombiniert genutzt werden kann und die größtmögliche Flexibilität bei der Selektion geboten wird.

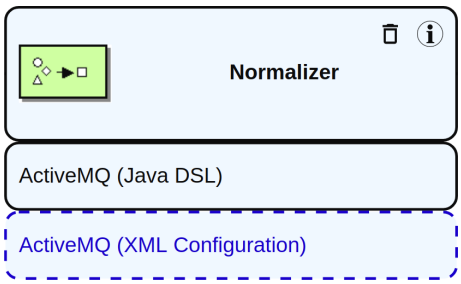

Abbildung 5.3.: Eine Musterinstanz in Pattern Atlas mit zwei vorgeschlagenen Musterimplementierungen, wovon die Zweite selektiert wurde.

Abbildung 5.3 zeigt die Selektion einer Musterimplementierung anhand eines *Normalizers*. Hier werden dem Nutzer zwei mögliche Musterimplementierungen für Apache ActiveMQ präsentiert, eine Java-DSL und eine XML-Konfiguration. In diesem Fall wurde die zweite Musterimplementierung vom Nutzer selektiert. Für die Aggregation kann dann die Selektion in Form einer Abbildung der Musterinstanzen auf die Musterimplementierungen genutzt und an die Pattern Atlas API übertragen werden.

5.5. Aggregationsoperatoren

Die Aggregationsoperatoren werden in der Programmiersprache Java erstellt. Für die Durchführung einer Aggregation wird der Aggregator anhand der zu aggregierenden Musterimplementierungen beziehungsweise deren Aggregationstypen bestimmt. Die Aggregatoren werden deshalb mit einer Annotation `@AggregatorMetadata` versehen. Beispielsweise kann ein Aggregator mit der Annotation `@AggregatorMetadata(sourceTypes = {"ActiveMQ-Java"}, targetTypes = {"ActiveMQ-Java"})` zwei Musterimplementierungen vom Aggregationstyp „ActiveMQ-Java“ aggregieren. Außerdem muss der Aggregator das Interface `Aggregator` implementieren. Dieses Interface definiert die Schnittstelle in Form einer Methode `aggregate`.

Während die Aggregation von zwei Musterimplementierungen durchgeführt wird, können Daten in einem Kontext abgelegt werden. Dieser Kontext ist während der Aggregation eines Entwurfsmodells gültig und für die Aggregationsoperatoren abrufbar. Für nachfolgende Aggregationsschritte sind diese Daten somit verfügbar, da sie aus dem Kontext gelesen werden können. Wie im Konzept vorgestellt ist, wird der Kontext für Zwischenergebnisse in Form von Textfragmenten, zu setzende Referenzen oder ähnliches verwendet. Eine solche Referenz kann der Namen eines *Message Channels* sein, an den ein *Message Filter* Daten senden soll. In Abschnitt 5.6.3 wird die Aggregation von zwei Musterimplementierungen mit dem Aggregationstyp „ActiveMQ-Java“ im Detail beschrieben.

Steht ein Ergebnis fest, kann ein Aggregator eine Datei zurückgeben. Hierfür wird ein Dateiname, das Dateiformat als Multipurpose Internet Mail Extensions (MIME)-Typ und der Inhalt der Datei angegeben. Die so erstellten Dateien werden dem Nutzer gesammelt zum Download angeboten.

5.6. Aggregationsalgorithmus

Im folgenden Abschnitt werden der Aggregationsalgorithmus sowie Implementierungsdetails vorgestellt. Das zugrundeliegende Konzept wurde in Abschnitt 4.6 beschrieben. Es werden drei Algorithmen vorgestellt, die sich auf unterschiedliche Abstraktionsebenen der Aggregation beziehen. Die ersten zwei Algorithmen werden bei jeder Aggregation im Prototypen genutzt. Algorithmus 5.1 nutzt als Eingabe das Entwurfsmodell und die gewählte Selektion der Musterimplementierungen. Algorithmus 5.2 wird daraufhin für zwei Knoten des Graphen aufgerufen und sucht den passenden Aggregator. Der dritte Algorithmus ist domänenspezifisch und abhängig von den Aggregationstypen des Quell- und Nachbarknotens. Als Beispiel wurde die Java-DSL von Apache ActiveMQ gewählt. Algorithmus 5.3 zeigt, wie die Aggregation in diesem Fall durchgeführt wird.

Für den Prototyp wurden mehrere Aggregatoren entwickelt. Darunter zwei Aggregatoren für Apache ActiveMQ Konfigurationen, davon einer für die Aggregation der Java-DSL und einer für die Aggregation der XML-Konfiguration. Außerdem ein Aggregator für AWS CloudFormation Templates, welcher auch Apache ActiveMQ XML-Konfigurationen einbetten kann. Aufgrund des Umfangs des Quellcodes wird dieser nicht im Einzelnen vorgestellt. Die Implementierungen aller Aggregatoren sind online als open-source Software im Rahmen des Pattern Atlas dauerhaft verfügbar [Graa].

Um den Prototyp zu evaluieren, wurden mehrere MIAs für Java DSL, XML-Konfiguration und AWS CloudFormation Template erstellt. Das entsprechende Git-Repository [Grab] ist ebenfalls Teil des Pattern Atlas Projekts.

5.6.1. Aggregationsalgorithmus für das Entwurfsmodell

Wird die Aggregation eines Entwufsmodells vom Nutzer gestartet, werden die Musterinstanzen und deren Beziehungen des Entwurfsmodells sowie die Selektion der Musterimplementierungen an den Algorithmus 5.1 übergeben. Zuerst werden die Musterimplementierungen, nach Vorgabe durch die Selektion, mit den Musterinstanzen verknüpft. Die Semantik der Kanten im Entwurfsmodell drückt die Beziehung der Muster zueinander aus. Deshalb wird für die Aggregation diese Semantik technisch vereinheitlicht. Beispielsweise kann das „senden“ und „empfangen“ von Nachrichten auf diese Weise bei der Aggregation einheitlich behandelt werden, wodurch die Aggregatoren nicht die Semantik des Entwurfsmodells behandeln müssen. Ausführlicher beschrieben ist dies in Abschnitt 4.5.2. Anschließend wird der Kontext, der (Zwischen-)Speicher für die Aggregatoren, und eine Menge für die erzeugten Artefakte initialisiert.

So lange im Graphen noch Knoten vorhanden sind, werden die folgenden Schritte wiederholt. Es wird ein Quellknoten gesucht, der gemeinsam mit seinem Nachbarknoten und der Kante zwischen den beiden Knoten zurückgegeben wird. Quell- und Nachbarknoten werden zusammen mit dem Kontext nun an Algorithmus 5.2 zur Aggregation weitergereicht. Wenn ein Artefakt vollständig ist, wird eine Datei zurückgegeben, die gemeinsam mit den anderen erzeugten Artefakten gesammelt

Algorithmus 5.1 Aggregation der Musterimplementierungen eines Entwurfsmodells

Eingabe M: Die Menge der Musterinstanzen
K: Die Menge der Kanten
S: Die Abbildung der Selektion
Ausgabe $artefakte$: Generierte Dateien

```
1:  procedure AGGREGIERE(M, K, S)
2:      M ← VERBINDEMUSTERIMPLEMENTIERUNGEN(M, S)
3:      K ← VEREINHEITLICHEKANTEN(K)
4:      kontext ← {}                          // (Zwischen-)Speicher für Aggregatoren
5:      artefakte ← ∅                         // Menge der generierten Dateien
6:      while M ≠ ∅ do
7:          quelle, nachbar, kante ← QUELLUNDNACHBARKNOTEN(M, K)
8:          a ← AGGREGIERE(quelle, nachbar, kontext)
9:          artefakte ← artefakte ∪ {a}       // ggf. erzeugte Datei hinzufügen
10:         K ← K \ {kante}                   // Kante entfernen
11:         if ¬HATKANTE(quelle) then         // Knoten entfernen, wenn keine Kante
12:             M ← M \ {quelle}
13:         end if
14:     end while
          return artefakte
15: end procedure
```

wird. Anschließend wird die Kante zwischen den aggregierten Knoten entfernt. Falls der Quellknoten nun keine Kanten mehr hat, wird auch dieser entfernt. Sind noch weitere Knoten vorhanden, werden diese Schritte wiederholt.

Abschließend werden alle erzeugten Artefakte an den Nutzer zurückgegeben. Diese stellen die aggregierten Implementierungen des Entwurfsmodells dar.

5.6.2. Aggregationsalgorithmus für zwei Musterimplementierungen

Um den passenden Aggregator für die Aggregation von zwei Musterimplementierungen zu finden, wird dieser anhand der Aggregationstypen der Musterimplementierungen gesucht. Die Auswahl und Nutzung des passenden Aggregators erfolgt in Algorithmus 5.2. Dazu werden die Aggregationstypen von Quell- und Nachbarkonten gelesen und mit den Annotationen der vorhandenen Aggregatoren abgeglichen. Ein Aggregator implementiert die Methode `aggregiere`, welche die Musterimplementierungen aggregieren kann. Wird ein passender Aggregator gefundenen, wird dessen Aggregationsmethode mit den zwei Knoten und dem Kontext aufgerufen. Wird kein passender Aggregator gefunden, wird eine Ausnahme geworfen, da die korrekte Aggregation der Implementierungen nicht gewährleistet werden kann.

Algorithmus 5.2 Aggregation von zwei Musterimplementierungen

Eingabe $quelle$: Eine Musterimplementierung (Quellknoten)
$nachbar$: Eine Musterimplementierung (Nachbarknoten)
$kontext$: Der Kontext für die Aggregatoren
Ausgabe $artefakt$: Generierte Datei

```
1: procedure AGGREGIERE(quelle, nachbar, kontext)
2:     aggregator ← sucheAggregator(quelle.aggregationstyp,
                                    nachbar.aggregationstyp)
3:     if aggregator ≠ null then
4:         artefakt ← aggregator.AGGREGIERE(quelle, nachbar, kontext)
5:     else
6:         WERFEAUSNAHME(Kein passender Aggregator vorhanden)
7:     end if
       return artefakt
8: end procedure
```

5.6.3. Aggregationsalgorithmus für Apache ActiveMQ Java-Konfiguration

Nachdem ein passender Aggregator gefunden wurde, werden als Parameter der Quellknoten, der Nachbarknoten sowie der Kontext an die laufende Aggregation übergeben. In diesem Abschnitt wird hierfür der Aggregator für die Apache ActiveMQ Java-DSL-Konfiguration beispielhaft beschrieben. Dieser und alle weiteren Aggregatoren sind im Git-Repository [Graa] von Pattern Atlas verfügbar. Algorithmus 5.3 zeigt diesen Algorithmus, um eine gültige Java-Konfiguration für Apache ActiveMQ zu generieren, welcher im Folgenden beschrieben wird.

Anhand der `artefaktUrl` aus dem MID, welche auf das MIA verweist, kann diese Datei eingelesen werden. Da die Datei anhand einer URL referenziert wird, kann diese beispielsweise in einem Git-Repository im lokalen Dateisystem oder auf einem Server gespeichert sein. Bei den MIAs für die Java-DSL handelt es sich um Templates. Diese enthalten Variablen, wodurch unter anderem die Referenzen für die *Message Channels* gesetzt werden können. Im nächsten Schritt werden die Variablen individualisiert, damit diese für jede Musterinstanz individuell ausgefüllt werden können. Anschließend werden später benötigte Referenzen auf die besagten *Message Channels* im Kontext gespeichert. Das MIA wird daraufhin mit dem bisherigen Zwischenergebnis der Konfiguration verkettet, sofern dieses bereits vorhanden ist. Ist der Aggregationstyp des Nachbarknotens ebenfalls `ActiveMQ-Java`, ist der Aggregationsschritt hier zu Ende. Andernfalls kann die Java-Konfiguration für Apache ActiveMQ generiert werden. Dazu wird das bisherige Ergebnis aus den MIAs mit den im Kontext gespeicherten Variablenwerten ausgefüllt und so die Konfiguration erstellt. Abschließend muss diese Konfiguration in eine Java-Klasse eingesetzt werden, um funktionsfähigen Quellcode zu erhalten. Hierfür wird das Template einer Java-Klasse geladen und anschließend die Konfiguration eingesetzt. Nun ist die Aggregation mit dem Ergebnis einer Java-Klasse beendet.

Algorithmus 5.3 Aggregator für die Aggregation einer Apache ActiveMQ Java-Konfiguration

Eingabe $quelle$: Eine Musterimplementierung (Quellknoten)
$nachbar$: Eine Musterimplementierung (Nachbarknoten)
$kontext$: Der Kontext für die Aggregatoren
Ausgabe $artefakt$: Generierte Datei

1: **procedure** AGGREGIERE($quelle, nachbar, kontext$)
2: $id \leftarrow quelle.musterinstanzId$
3: $mid \leftarrow quelle.musterimplementierung$
4: $mia \leftarrow$ LESEDATEI($mid.artefaktUrl$)
5: $mia \leftarrow$ INDIVIDUALISIEREVARIABLEN(mia, id)
6: $kontext \leftarrow$ ERWEITEREUMCHANNELIDS($quelle, nachbar, kontext$)
7: $kontext.camel \leftarrow$ VERKETTE($mia, kontext.camel$)
8: **if** $nachbar.aggregationstyp$ = ActiveMQ-Java **then**
return null
9: **end if**
10: $konfiguration \leftarrow$ RENDERTEMPLATE($kontext.camel, kontext$)
11: $wrapper \leftarrow$ LESEDATEI($camelJavaWrapperUrl$)
12: $javaKlasse \leftarrow$ RENDERTEMPLATE($wrapper, \{camel : konfiguration\}$)
return ARTEFAKT(RouteBuilder.java, text/x-java, $javaKlasse$)
13: **end procedure**

5.7. Evaluation

In diesem Abschnitt wird die Verwendung des Prototyps im Hinblick auf die in Kapitel 3 vorgestellten Referenzszenarien bewertet. Anschließend wird dieser Prototyp auf Einhaltung der Anforderungen überprüft, die zuvor aus den Referenzszenarien gefolgert wurden.

5.7.1. Evaluation der Referenzszenarien

Zur Vermeidung von Redundanz wird in diesem Abschnitt der Einsatz des Prototyps für das Referenzszenario mit mehreren Mustersprachen aus Abschnitt 3.3 betrachtet. Denn dieses Referenzszenario enthält sowohl Muster der CCP als auch der EIP. Das heißt, bei der Aggregation müssen Musterimplementierungen der CCP und der EIP untereinander und miteinander aggregiert werden. Damit sind auch die ersten zwei Referenzszenarien von den technischen Aspekten der Aggregation abgedeckt.

Um das dritte Referenzszenario mit dem Prototyp umzusetzen, wird zuerst ein neues Entwurfsmodell angelegt. Anschließend kann das Entwurfsmodell entsprechend dem beschriebenen Referenzszenario modelliert werden. Hierfür werden die Muster ausgewählt und per Drag and Drop der Arbeitsfläche des Editors hinzugefügt. Anschließend können die Beziehungen zwischen den, zuvor hinzugefügten, Musterinstanzen modelliert werden. Die Visualisierung dieses Entwurfsmodells im Editor ist in Abbildung 5.2 dargestellt. Nach Abschluss der Modellierung, können die Musterimplementierungen angezeigt werden. Wie in Abschnitt 5.4 beschrieben, werden passende vorhandene Musterimplementierungen zu den jeweiligen Musterinstanzen angezeigt. Mit einem Filter für alle

Musterinstanzen oder individuell für jede Musterinstanz kann der Nutzer die gewünschten Musterimplementierungen selektieren. Die Selektion einer Musterimplementierung für eine individuelle Musterinstanz ist in Abbildung 5.3 abgebildet. Nachdem die Selektion abgeschlossen ist, kann der Nutzer die automatisierte Aggregation starten.

Im Hintergrund läuft nun die automatisierte Aggregation der selektierten Musterimplementierungen. Der Algorithmus und wie dieser implementiert ist, wird in Abschnitt 5.6 beschrieben. Im Fall vom dritten Referenzszenario werden fünf Dateien generiert. Es wird eine XML-Konfiguration des Camel Contexts für die Konfiguration von Apache ActiveMQ erstellt. Eine weitere generierte Datei ist das AWS CloudFormation Template mit eingebetteter XML-Konfiguration für Apache ActiveMQ. Außerdem werden drei Java-Dateien erzeugt, jeweils eine für jeden der drei *Message Endpoints*. Die Java-Dateien sind entsprechend mit individuellen Namen für die genutzten *Message Channels* angepasst. Die vom Prototyp generierten Dateien sind in Anhang A.1 an diese Arbeit angehängt. Sie stellen das Ergebnis der Aggregation des Entwurfsmodells aus Abbildung 5.2 dar.

Mit dem Prototyp konnte gezeigt werden, dass es möglich ist mit Mustern in einem Entwurfsmodell eine Softwarearchitektur zu modellieren. Hierfür wird das Entwurfsmodell genutzt, welches aus Musterinstanzen und deren Beziehungen zueinander besteht. Wie beschrieben, kann darauf aufbauend eine Selektion der Musterimplementierungen durchgeführt werden. Auf diese Weise werden mit dem Prototyp die Grundlagen für die automatisierte Aggregation von Musterimplementierungen gelegt. Darauf aufbauend kann mit dem Prototyp eine automatisierte Aggregation durchgeführt werden. Dabei konnte anhand des genannten Referenzszenarios demonstriert werden, dass die Aggregation mehrerer Mustersprachen möglich ist. Somit deckt der entwickelte Prototyp den kompletten Vorgang von der Modellierung bis zur Aggregation ab.

Die Herausforderung stellt dabei die Konzeption und Implementierung des Aggregationsalgorithmus dar, damit Musterimplementierungen in jeglicher Konstellation korrekt aggregiert werden. Hierbei ist besonders die Struktur des Entwurfsmodells in Form eines Graphen zu handhaben. Bei Graphen ergibt sich die Komplexität daraus, dass kein Sequenzen vorgegeben ist, sondern sich die Aggregationsreihenfolge aus dem Entwurfsmodell und den Musterimplementierungen ergibt. Ebenfalls muss eine Musterimplementierung an verschiedene Konstellationen in einem Graphen angepasst werden können. Beispielsweise ist es für manche Musterimplementierungen erforderlich, dass diese mit einer beliebigen Anzahl von anderen Musterimplementierungen aggregiert werden können. Dies wurde unter anderem anhand des *Content-Based Router* diskutiert oder im demonstrierten Referenzszenario anhand der MOM gezeigt.

5.7.2. Evaluation der Anforderungen

An die Evaluation des Prototyps in Bezug auf die Referenzszenarien, wird hier die Einhaltung der Anforderungen aus Abschnitt 3.4 diskutiert. Jede Anforderung wird hierzu anhand des Prototyps gesondert geprüft.

A1 - Der Nutzer kann einen Architekturentwurf erstellen und verändern

Als Teil des Prototyps wurde für Pattern Atlas ein Editor für Entwurfsmodelle entwickelt, der die Erstellung und Veränderung derselben ermöglicht. Hierfür wurde das Entwurfsmodell definiert. Mit dem Prototyp konnte die Erstellung eines Entwurfsmodells zur Abbildung eines Architekturentwurfs aus Mustern aufgezeigt werden. Diese Anforderung trifft somit zu und konnte mit dem Prototyp realisiert werden.

A2 - Erstellung von Musterimplementierung und Zuordnung zu Muster

Musterimplementierungen bestehen aus dem MID und dem MIA. Im MID wird der URI des Musters angegeben, welches implementiert wird. Auf diese Weise können einem Muster beliebig viele Musterimplementierungen zugeordnet werden.

Das MIA wird im Prototyp mittels einer URL referenziert. Damit können Dateien von verschiedenen Orten gelesen werden. Für Textdateien wurde dies im Prototyp demonstriert. Weitere Formate sind hier ebenfalls möglich, sofern dafür Aggregatoren entwickelt werden, welche mit den Formaten umgehen können. Die Ausgabe von Artefakten ist flexibel gestaltet und unterstützt verschiedene Formate. Somit ist diese Anforderung im Prototyp erfüllt.

A3 - Der Nutzer hat Einfluss auf die Selektion der Musterimplementierung

Der Nutzer definiert die Selektion, welche eine Abbildung von Musterinstanz auf Musterimplementierung darstellt. Im Prototyp kann die Auswahl anhand des globalen Filters oder individuell für jede Musterinstanz erfolgen. Somit hat der Nutzer weitreichenden Einfluss auf die Selektion der Musterimplementierungen, wodurch diese Anforderung eingehalten werden kann.

A4 - Aggregationsoperator für die Aggregation der Musterimplementierung

Im Rahmen dieser Arbeit wurden mehrere Aggregationsoperatoren zur Aggregation von Musterimplementierungen implementiert. In Abschnitt 5.6.3 wurde einer dieser Aggregatoren für die Java-DSL von Apache ActiveMQ vorgestellt. Für den Prototyp wurden neben diesem noch ein Aggregator für die Apache ActiveMQ XML-Konfiguration sowie ein Aggregator für AWS CloudFormation Templates entwickelt. Die Funktionsfähigkeit wurde anhand des dritten Referenzszenarien demonstriert und somit die Einhaltung dieser Anforderung bestätigt.

A5 - Mehrere Instanzen eines Musters

Das Entwurfsmodell ist so definiert, dass mehrere Instanzen eines Musters möglich sind. Jede Musterinstanz ist eindeutig identifizierbar. Damit werden individuelle Beziehungen zu anderen Musterinstanzen unterstützt. Somit wurde diese Anforderung per Definition eingehalten. Mit dem Prototyp konnte dies demonstriert und bestätigt werden.

A6 - Der Architekturentwurf kann ein Graph sein

Bei der Konzeption sowie der Entwicklung des Prototyps wurde ebenfalls berücksichtigt, dass es sich bei einem Architekturentwurf um einen Graphen handeln kann. Somit können Architekturentwürfe, welche einem Graphen entsprechen, modelliert und die Musterimplementierungen aggregiert werden. Hierdurch wird gewährleistet, dass Musterimplementierungen eine beliebige Anzahl von Nachbarknoten haben können. Anhand der Aggregation des dritten Referenzszenarios konnte dies demonstriert werden. Hierdurch ist auch diese Anforderung im Prototypen erfüllt.

A7 - Mehrere Aggregationsoperatoren

Zur Aggregation von Musterimplementierungen wurden im Prototypen mehrere Aggregationsoperatoren implementiert. Je nach zu aggregierenden Musterimplementierungen wird der passende Aggregator anhand der Aggregationstypen ausgewählt. Auf diese Weise können mehrere Aggregationsoperatoren umgesetzt werden. Dies ermöglicht die Aggregatoren so zuzuschneiden, dass sie Musterimplementierungen mit ähnlichen Charakteristiken aggregieren können. Im Prototyp sind bereits mehrere Aggregatoren realisiert, damit wird diese Anforderung erfüllt.

A8 - Erweiterbarkeit der Aggregationsoperatoren

Um auch dieser Anforderung gerecht zu werden, sind Musterimplementierungen im Prototypen anhand des Aggregationstyps in Trägermengen gruppiert. Wird ein neuer Aggregator für neue Trägermengen benötigt, kann dieser erstellt und mit den entsprechenden Aggregationstypen annotiert werden. Implementiert der Aggregator außerdem das Java-Interface `Aggregator`, so wird der Aggregator automatisch bei der Suche nach Aggregatoren berücksichtigt. Falls der neue Aggregator passend ist, wird die Aggregationsmethode mit den entsprechenden Knoten aufgerufen. Auf diese Weise können neue Aggregatoren ohne aufwändige Integration ergänzt werden. Dadurch wird diese Anforderung erfüllt.

A9 - Mehrere Ausgabedateien

Bei jeder Aggregationsoperation kann der Aggregator aus den zuvor aggregierten MIA eine Gesamtkomposition erstellen und diese in Form einer Datei ausgeben. Diese Dateien werden über den kompletten Aggregationsprozess gesammelt und dem Nutzer nach Abschluss der Aggregation zur Verfügung gestellt. Auf diese Weise werden, neben der Anforderungserfüllung mehrere Ausgabedateien zu ermöglichen, auch unterschiedliche Technologien innerhalb eines Architekturentwurfs unterstützt. Somit wird auch diese Anforderung erfüllt.

5.7.3. Zusammenfassung der Evaluation

Den Anforderungen, welche aus den Referenzszenarien erarbeitet und in Abschnitt 3.4 präsentiert werden, konnte entsprochen werden. Das Konzept und der darauf aufbauende Prototyp kann somit als passgenau zu den gestellten Anforderungen bezeichnet werden. Der Prototyp ist geeignet um Architekturentwürfe mit Mustern zu modellieren. Diese Entwurfsmodelle können Graphen

aus Mustern verschiedener Mustersprachen sein, welchen Musterimplementierungen zugeordnet werden können. Diese Musterimplementierung bilden die Grundlage für die automatisierte Aggregation zu Artefakten, welche die Softwarearchitektur abbilden. Damit ist der Prototyp für die gestellten Anforderungen geeignet und zeigt die Umsetzbarkeit der automatisierten Aggregation von Musterimplementierungen auf.

6. Fazit und Ausblick

Abgeschlossen wird diese Arbeit durch ein Fazit und ein Ausblick auf weitere Forschungsmöglichkeiten im Bereich der automatisierten Aggregation von Musterimplementierungen.

6.1. Fazit

Um wiederkehrende Problemstellungen und bewährte und abstrakte Lösungen zu beschreiben haben sich in der Softwaretechnik Muster bewährt. Diese Muster werden für neue Anwendungsfälle jedoch häufig neu implementiert. In dieser Arbeit wurde daher untersucht, wie dieser Prozess automatisiert werden kann. Hierfür wurden die Grundlagen anhand verwandter Arbeiten erforscht. Darauf aufbauend wurden die Anforderungen anhand von drei Referenzszenarien betrachtet und erhoben. Aus den Grundlagen und Anforderungen wurde ein Konzept erarbeitet und ein Prototyp entwickelt. Mit diesem Prototyp können Softwarearchitekturen mit Mustern und ihren Beziehungen modelliert werden. Passend zum konkreten Anwendungsfall können zudem die Musterimplementierungen anhand der Musterinstanzen des Entwurfsmodells selektiert werden. Dazu wurden Musterimplementierungen der CCP und der EIP sowie Aggregatoren für die Aggregation dieser Musterimplementierungen konzipiert und implementiert. Insbesondere wurden Aggregatoren für AWS Cloud Formation Templates und Apache ActiveMQ Java-DSL sowie XML-Konfigurationen implementiert. Hierfür wurde ein Algorithmus aufgezeigt und im Pattern Atlas integriert, um diese Musterimplementierungen automatisiert zu aggregieren. Dabei hat sich gezeigt, dass die Semantik des Entwurfsmodells in die Aggregation der Musterimplementierungen einfließen muss. So wirkt sich die Semantik aus dem Entwurfsmodell beispielsweise in einer konkreten Apache ActiveMQ Konfiguration in der Richtung des Nachrichtenflusses aus. Außerdem konnte gezeigt werden, dass die Aggregation von Mustern unterschiedlicher Mustersprachen fachlich richtig und technisch möglich sein kann. Beispielhaft wurde dies an einem umfangreichen Referenzszenario demonstriert. Somit konnte das Zusammenwirken der Modellierungssprache und der Aggregationsoperatoren für die automatisierte Aggregation von Musterimplementierungen anhand eines auf Pattern Atlas basierenden Prototyps gezeigt werden.

6.2. Ausblick

Im Folgenden werden Anknüpfungspunkte an diese Arbeit vorgestellt. Damit wird ein Ausblick auf weitere mögliche Forschung gegeben. Sie reichen von Aspekten des Prototyps bis hin zur Untersuchung weiterer Mustersprachen aus anderen Anwendungsbereichen.

6.2.1. Verbesserung der Nutzerfreundlichkeit des Prototyps

Der Prototyp kann zur Erhöhung der Nutzerfreundlichkeit weiterentwickelt werden. Hierzu kann bei der Selektion einer Musterimplementierung direkt geprüft werden, ob für zwei benachbarte Musterimplementierungen ein Aggregator gefunden werden kann. Falls nicht, kann dies entsprechend an der Kante zwischen den Musterinstanzen visualisiert werden. Außerdem können alle erzeugten Artefakte in ein ZIP-Archiv gepackt werden, nachdem die Aggregation abgeschlossen ist. Dies bietet sowohl den Vorteil, dass dem Nutzer nur eine Datei angeboten wird, als auch die Möglichkeit hierarchische Ordnerstrukturen abzubilden. Das erscheint insbesondere bei umfangreichen Modellen mit vielen generierten Artefakten sinnvoll.

6.2.2. Konkretisierung von Mustern

Muster können unterschiedlich abstrakt sein, dies zeigt sich sowohl innerhalb von Mustersprachen, als auch im Vergleich unterschiedlicher Mustersprachen. Je spezifischer und konkreter ein Muster ist, um so einfacher kann es in eine Implementierung übersetzt werden. Falkenthal et al. [FBB+16] haben aufgezeigt, dass auf diese Art eine Konkretisierung von Mustern vorgenommen werden kann, bis schließlich eine Musterimplementierung gewählt wird. Hierzu kann das Entwurfsmodell weiter entwickelt werden. So ist zur Konkretisierung eines Musters die Substitution dieses Musters durch einen Graphen aus konkreteren Mustern denkbar. Beispielsweise kann ein *Normalizer* durch einen *Router* und mehrere *Translators* substituiert werden [HW04]. Dementsprechend können dann Musterimplementierungen für *Router* und *Translators* anstatt der Musterimplementierung für einen *Normalizer* genutzt werden. Dies bietet gegebenenfalls erhöhte Flexibilität, falls für eine bestimmte Technologie keine Musterimplementierung verfügbar oder umsetzbar ist. Beispielsweise zeigen verschiedene open-source MOMs unterschiedliche umfängliche Integrationsfähigkeiten der EIP auf [GG19].

6.2.3. Automatisierte Selektion der Musterimplementierungen

Für die Auswahl der passenden Musterimplementierungen sind neben dem konkreten Anwendungsfall oft weitere Rahmenbedingungen entscheidend. Dies können Einschränkung auf bereits genutzte Cloud-Anbieter oder aufgrund rechtlicher Vorgaben sein. Zur Unterstützung des Nutzers bei der Auswahl der passenden Musterimplementierungen kann eine automatisierte Selektion genutzt werden. Entweder um die Selektion vollständig zu übernehmen oder um einen Vorschlag für die Selektion der Musterimplementierungen zu unterbreiten. Hierfür ist es möglich, den Ansatz von Falazi [Fal17] zu erweitern und zu integrieren. Falazi untersuchte die automatisierte Selektion von Musterimplementierungen für eine Mustersequenz. Dazu entwickelte Falazi einen Algorithmus bestehend aus zwei Phasen. In diesen werden zuerst alle möglichen Pfade durch eine Menge von Musterimplementierungen generiert und anschließend anhand nutzerspezifizierter Anforderungen gefiltert. Diese nutzerspezifischen Anforderungen können beispielsweise auch Unternehmensrichtlinien oder bereits genutzten Cloud-Anbieter abbilden. Damit kann eine automatisierte Vorauswahl der Musterimplementierungen erfolgen, welche vom Nutzer direkt übernommen werden kann und den eigenen Anwendungsfall erfüllt.

6.2.4. Untersuchungen und Evaluation mit weiteren Anwendungsgebieten

Anwendungsfälle und Anforderungen für weitere Mustersprachen aus unterschiedlichen Domänen können ebenfalls betrachtet werden. Hierzu gehören auch Mustersprachen ohne direkten informationstechnischen Bezug. Beispielsweise beschreiben die Costume Patterns nach Schumm et al. [SBLE12] verschiedene Lösungsmuster für Kostüme, wie beispielsweise einen *Wild West Sheriff*. Da ein Kostüm nicht direkt als Aggregat von einem Computer erstellt werden kann, bieten sich manuelle Instruktionen an. Hierdurch wird unter Anderem definiert, wie Kostüme kombinierbar sind und was des Weiteren beachtet werden muss. Der vorgestellte Prototyp ist prinzipiell auch in der Lage, Handlungsanweisungen für die Aggregation von Kostümen zu erstellen, indem die Kombination von verschiedenen Instruktionen ermöglicht wird. Da die Handlungsanweisung von Menschen gelesen und verstanden werden, sind hier keine besonderen informationstechnische Anforderungen hinsichtlich eines Dateiformats gegeben. Beispielsweise ist im einfachsten Fall die Generierung einer Textdatei mit Handlungsanweisungen in Form einer Aufgabenliste möglich. Dabei können auch weitere Informationen hinzugefügt werden, wie einzelne Teile abgewandelt werden oder was beachtet werden muss, damit die Kostüme derselben Zeitära oder Region entspringen. Hier kann untersucht werden, inwiefern solche Handlungsanweisungen sinnvoll und hilfreich sind um die Umsetzung eines Musters zu vereinfachen und Fehler zu reduzieren.

Muster werden in unterschiedlichen Anwendungsgebieten genutzt. Erste Muster wurden im Rahmen von Stadt- und Gebäudearchitektur entwickelt. Heutzutage sind Muster auch im Kontext von Softwarearchitektur und -entwicklung weitverbreitet. Diese Arbeit konnte hierdurch einen Beitrag zur automatisierten Aggregation von Musterimplementierungen leisten. Neben vielen spezifischen Anknüpfungspunkten in der Softwaretechnik, zeigen sich dabei auch Anknüpfungspunkte dieses Themas für andere Anwendungsgebiete. So kann das gewonnene Wissen aus der Domäne der Softwaretechnik auch in andere Anwendungsgebiete einfließen.

Literaturverzeichnis

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel. *A Pattern Language - Towns, Buildings, Construction*. Oxford University Press, 1977. ISBN: 978-0-19-501919-3 (zitiert auf S. 15, 19).

[Amaa] Amazon Web Services, Inc. *Amazon Relational Database Service (RDS)*. URL: https://aws.amazon.com/rds (besucht am 17. 11. 2020) (zitiert auf S. 29).

[Amab] Amazon Web Services, Inc. *Amazon Web Services AWS - Server Hosting und Cloud Services*. URL: https://aws.amazon.com (besucht am 19. 11. 2020) (zitiert auf S. 20).

[Amac] Amazon Web Services, Inc. *AWS CloudFormation concepts*. URL: https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/cfn-whatis-concepts.html (besucht am 16. 11. 2020) (zitiert auf S. 29).

[Apaa] Apache Software Foundation. *Apache Camel*. URL: https://camel.apache.org (besucht am 03. 11. 2020) (zitiert auf S. 32).

[Apab] Apache Software Foundation. *Flexible & Powerful Open Source Multi-Protocol Messaging - Apache ActiveMQ*. URL: https://activemq.apache.org (besucht am 24. 11. 2020) (zitiert auf S. 32).

[Apa20] Apache Software Foundation. *Enterprise Integration Patterns*. 30. Sep. 2020. URL: https://camel.apache.org/components/latest/eips/enterprise-integration-patterns.html (besucht am 25. 11. 2020) (zitiert auf S. 32).

[BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. „Pattern-based Runtime Management of Composite Cloud Applications“. Englisch. In: *CLOSER 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science, Aachen, Germany, 8-10 May, 2013*. Hrsg. von F. Desprez, D. Ferguson, E. Hadar, F. Leymann, M. Jarke, M. Helfert. SciTePress, Mai 2013, S. 475–482. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2013-20 (zitiert auf S. 29).

[BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. „The OpenTOSCA Ecosystem - Concepts & Tools“. Englisch. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges*. SCITEPRESS - Science und Technology Publications, Dez. 2016, S. 112–130. ISBN: 978-989-758-207-3. DOI: 10.5220/0007903201120130. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2016-26 (zitiert auf S. 29).

[BL18] U. Breitenbücher, F. Leymann. „Cloud Computing“. Lecture slides. 2018 (zitiert auf S. 29).

[Fal17] G. Falazi. „A concept for describing concrete solutions to support their automated selection from patterns“. English. Master Thesis. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, und Information Technology, Germany, Okt. 2017, S. 103. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=MSTR-2017-25 (zitiert auf S. 24, 66).

[FBB+14a] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. „Efficient Pattern Application: Validating the Concept of Solution Implementations in Different Domains“. English. In: Bd. 7. 3&4. Xpert Publishing Services (XPS), Dez. 2014, S. 710–726. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2014-13 (zitiert auf S. 15, 44).

[FBB+14b] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann. „From Pattern Languages to Solution Implementations“. English. In: *Proceedings of the 6th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), Mai 2014, S. 12–21. ISBN: 978-1-61208-343-8. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2014-37 (zitiert auf S. 15, 19, 22, 24).

[FBB+16] M. Falkenthal, J. Barzen, U. Breitenbücher, C. Fehling, F. Leymann, A. Hadjakos, F. Hentschel, H. Schulze. „Leveraging Pattern Applications via Pattern Refinement“. English. In: *Proceedings of the International Conference on Pursuit of Pattern Languages for Societal Change (PURPLSOC)*. Hrsg. von P. Baumgartner, T. Gruber-Muecke, R. Sickinger. Krems: epubli GmbH, Okt. 2016, S. 38–61. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2016-47 (zitiert auf S. 23, 24, 66).

[FBBL19] M. Falkenthal, U. Breitenbücher, J. Barzen, F. Leymann. „On the algebraic properties of concrete solution aggregation“. English. In: *SICS Software-Intensive Cyber-Physical Systems* 34.2-3 (Feb. 2019), S. 117–128. DOI: 10.1007/s00450-019-00400-1. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=ART-2019-04 (zitiert auf S. 16, 25, 39, 44–46).

[FBFL15] C. Fehling, J. Barzen, M. Falkenthal, F. Leymann. „PatternPedia - Collaborative Pattern Identification and Authoring“. German. In: *Proceedings of the International Conference on Pursuitof Pattern Languages for Societal Change (PURPLSOC)*. epubli GmbH, Juni 2015, S. 252–284. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2015-51 (zitiert auf S. 26).

[FL17] M. Falkenthal, F. Leymann. „Easing Pattern Application by Means of Solution Languages“. English. In: *Proceedings of the 9th International Conference on Pervasive Patterns and Applications*. Xpert Publishing Services (XPS), Feb. 2017, S. 58–64. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2017-10 (zitiert auf S. 23, 25).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns - Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014. ISBN: 978-3-7091-1567-1. DOI: 10.1007/978-3-7091-1568-8. URL: https://doi.org/10.1007/978-3-7091-1568-8 (zitiert auf S. 15, 20, 21, 27).

[Fow06] M. Fowler. *Writing Software Patterns*. 1. Aug. 2006. URL: https://www.martinfowler.com/articles/writingPatterns.html (besucht am 17. 11. 2020) (zitiert auf S. 19).

[GG19] T. Gutierrez, M. Graf. „Eine Analyse von open-source Message-oriented Middlewares hinsichtlich der Integrationsfähigkeit von Enterprise Integration Patterns“. Masterfachstudie. Okt. 2019. URL: https://share.nerdsupport.de/Eine%20Analyse%20von%20open-source%20Message-oriented%20Middlewares%20hinsichtlich%20der%20Integrationsf%C3%A4higkeit%20von%20Enterprise%20Integration%20Patterns.pdf (besucht am 27. 11. 2020) (zitiert auf S. 31, 32, 66).

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Hrsg. *Design Patterns: Elements of Reusable Object-Oriented Software*. Englisch. Addison-Wesley professional computing series. Boston, Mass.; Munich [u.a.]: Addison-Wesley, 1995, S. 395. ISBN: 0-201-63361-2 (zitiert auf S. 15, 19).

[Gita] GitHub Inc. *GitHub: Where the world builds software*. URL: https://github.com (besucht am 10. 12. 2020) (zitiert auf S. 52).

[Gitb] GitLab Inc. *DevOps Platform Delivered as a Single Application - GitLab*. URL: https://about.gitlab.com (besucht am 10. 12. 2020) (zitiert auf S. 54).

[Gooa] Google. *Angular*. URL: https://angular.io (besucht am 27. 11. 2020) (zitiert auf S. 51).

[Goob] Google Ireland Limited. *Cloud Deployment Manager - Google Cloud*. URL: https://cloud.google.com/deployment-manager (besucht am 19. 11. 2020) (zitiert auf S. 29).

[Gooc] Google Ireland Limited. *Cloud-Computing-Dienste - Google Cloud*. URL: https://cloud.google.com (besucht am 19. 11. 2020) (zitiert auf S. 20, 29).

[Good] Google Ireland Limited. *Google Maps*. URL: https://maps.google.com (besucht am 17. 11. 2020) (zitiert auf S. 20).

[Graa] M. Graf. *pattern-atlas-api/src/main/java/com/patternpedia/api/util/aggregator at master · PatternAtlas/pattern-atlas-api*. URL: https://github.com/PatternAtlas/pattern-atlas-api/tree/master/src/main/java/com/patternpedia/api/util/aggregator (besucht am 12. 12. 2020) (zitiert auf S. 56, 58).

[Grab] M. Graf. *PatternAtlas/pattern-atlas-pattern-implementations: Pattern Implementations for Pattern Atlas*. URL: https://github.com/PatternAtlas/pattern-atlas-pattern-implementations (besucht am 12. 12. 2020) (zitiert auf S. 51, 56).

[Has] HashiCorp, Inc. *Terraform*. URL: https://www.terraform.io (besucht am 25. 11. 2020) (zitiert auf S. 29).

[HBF+18] L. Harzenetter, U. Breitenbücher, M. Falkenthal, J. Guth, C. Krieger, F. Leymann. „Pattern-based Deployment Models and Their Automatic Execution“. Englisch. In: *11th IEEE/ACM International Conference on Utility and Cloud Computing UCC 2018, 17–20 December 2018, Zurich, Switzerland*. IEEE Computer Society, Dez. 2018, S. 41–52. DOI: 10.1109/UCC.2018.00013. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2018-49 (zitiert auf S. 24).

[HW04] G. Hohpe, B. Woolf. *Enterprise Integration Patterns*. Addison Wesley, 1. Jan. 2004. 480 S. ISBN: 0321200683. URL: https://www.ebook.de/de/product/2779126/gregor_hohpe_bobby_woolf_enterprise_integration_patterns.html (zitiert auf S. 15, 19, 21, 22, 31, 66).

[Kon] Konsortium PlanQK. *PlanQK - Plattform und Ökosystemfür Quantenunterstützte Künstliche Intelligenz*. URL: https://planqk.de (besucht am 18. 11. 2020) (zitiert auf S. 26).

[Kri18] C. Krieger. „Semantic querying of distributed pattern and solution repositories“. English. Master Thesis. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, und Information Technology, Germany, Juni 2018, S. 75. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=MSTR-2018-49 (zitiert auf S. 22, 23).

[Ley] F. Leymann. *PatternPedia - Wiki-based Pattern Repository*. URL: https://www.iaas.uni-stuttgart.de/forschung/projekte/patternpedia/ (besucht am 03. 11. 2020) (zitiert auf S. 26).

[Mar] MariaDB Foundation. *MariaDB Server: The open source relational database*. URL: https://mariadb.org (besucht am 16. 11. 2020) (zitiert auf S. 28).

[MG11] P. Mell, T. Grance. *The NIST definition of cloud computing*. Techn. Ber. 2011. DOI: 10.6028/nist.sp.800-145. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (besucht am 04. 11. 2020) (zitiert auf S. 20).

[Mic] Microsoft Corporation. *Cloud-Computing-Dienste - Microsoft Azure*. URL: https://azure.microsoft.com (besucht am 19. 11. 2020) (zitiert auf S. 20).

[Ora14] Oracle. *Java Platform, Enterprise Edition: The Java EE Tutorial. Overview of the JMS API*. 2014. URL: https://docs.oracle.com/javaee/7/tutorial/jms-concepts001.htm (besucht am 26. 11. 2020) (zitiert auf S. 21).

[Par] T. Parr. *StringTemplate*. URL: https://www.stringtemplate.org/ (besucht am 03. 12. 2020) (zitiert auf S. 54).

[Pata] Pattern Atlas Contributors. *Pattern Atlas*. URL: https://github.com/PatternAtlas (besucht am 20. 11. 2020) (zitiert auf S. 16).

[Patb] Pattern Atlas Contributors. *Pattern Atlas API Repository*. URL: https://github.com/PatternAtlas/pattern-atlas-api (besucht am 13. 11. 2020) (zitiert auf S. 26).

[Patc] Pattern Atlas Contributors. *Pattern Atlas UI Repository*. URL: https://github.com/PatternAtlas/pattern-atlas-ui (besucht am 13. 11. 2020) (zitiert auf S. 26).

[QuA] QuAntiL Docs Contributors. *Welcome to The QuAntiL Documentation*. URL: https://quantil.readthedocs.io/en/latest/ (besucht am 18. 11. 2020) (zitiert auf S. 26).

[SBLE12] D. Schumm, J. Barzen, F. Leymann, L. Ellrich. „A Pattern Language for Costumes in Films“. English. In: *Proceedings of the 17th European Conference on Pattern Languages of Programs (EuroPLoP 2012)*. Hrsg. von C. Kohls, A. Fiesser. New York NY USA: ACM, Juli 2012, S. 1–25. ISBN: 9781450329439. DOI: 10.1145/2602928.2603083. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2012-19 (zitiert auf S. 67).

[The] The PostgreSQL Global Development Group. *PostgreSQL: The World's Most Advanced Open Source Relational Database*. URL: https://www.postgresql.org (besucht am 16. 11. 2020) (zitiert auf S. 28, 51).

[VMw] VMware, Inc. *Spring Boot*. URL: https://spring.io/projects/spring-boot (besucht am 05. 12. 2020) (zitiert auf S. 51).

[WBB+20] M. Weigold, J. Barzen, U. Breitenbücher, M. Falkenthal, F. Leymann, K. Wild. „Pattern Views: Concept and Tooling for Interconnected Pattern Languages“. In: *CoRR* abs/2003.09127 (2020). arXiv: 2003.09127. URL: https://arxiv.org/abs/2003.09127 (zitiert auf S. 26).

A. Quellcode und -abschnitte

A.1. Referenzszenario: Mehreren Mustersprachen

Dieser Abschnitt enthält die aggregierten Musterimplementierungen des dritten Referenzszenarios bestehend aus Mustern verschiedener Mustersprachen. Die gezeigten Artefakte wurden mit dem Prototyp aus dem Entwurfsmodell generiert, welches in Abbildung 5.2 dargestellt ist.

A.1.1. Apache ActiveMQ XML-Konfigurationsdatei

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
    Copyright 2020 University of Stuttgart

    Licensed under the Apache License, Version 2.0 (the "License");
    you may not use this file except in compliance with the License.
    You may obtain a copy of the License at

        http://www.apache.org/licenses/LICENSE-2.0

    Unless required by applicable law or agreed to in writing, software
    distributed under the License is distributed on an "AS IS" BASIS,
    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
    See the License for the specific language governing permissions and
    limitations under the License.
-->
<!--
    Apache Camel documentation of Enterprise Integration Patterns:
    https://camel.apache.org/components/latest/eips/enterprise-integration-patterns.html

    Include this file in your Apache ActiveMQ configuration to enable Apache Camel
    e.g. <import resource="camel.xml"/> in your activemq.xml right before </beans>
-->

<beans
   xmlns="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="
     http://camel.apache.org/schema/spring http://camel.apache.org/schema/spring/camel-spring.xsd
     http://www.springframework.org/schema/beans
↪  http://www.springframework.org/schema/beans/spring-beans.xsd">

    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">

        <!-- The following Spring XML configuration is proudly presented by Pattern Atlas -->
```

```
36          <!-- point-to-point-channel-a5675128-a8b6-486b-b0e0-106c8a4796d1 -->
37          <route>
38              <description>Point to Point Channel</description>
39              <from uri="activemq:point-to-point-channel-a5675128-a8b6-486b-b0e0-106c8a4796d1" />
40              <to uri="activemq:normalizer-45c1239f-f6c9-4960-b313-a75f47fc964d" />
41          </route>
42          <!-- normalizer-45c1239f-f6c9-4960-b313-a75f47fc964d -->
43          <bean id="normalizer124119893" class="TODO: insert bean class pa.ck.age.Class" />
44
45          <route>
46              <description>Normalizer</description>
47              <from uri="activemq:normalizer-45c1239f-f6c9-4960-b313-a75f47fc964d" />
48              <choice>
49                  <when>
50                      <!-- TODO: add filter rule, e.g.
51                      <xpath>/sensor/type = 'temp-celsius'</xpath>
52                      -->
53                      <bean ref="normalizer124119893" method="TODO: insert bean method name or remove
    ↪  element, if no processing is required" />
54                      <to uri="activemq:point-to-point-channel-8867a184-68f5-4c04-a2ce-801c3d898fef" />
55                  </when>
56
57                  <otherwise>
58                      <!-- TODO: may adapt dead letter channel -->
59                      <to uri="activemq:deadletterchannel"/>
60                  </otherwise>
61              </choice>
62          </route>
63          <!-- point-to-point-channel-8867a184-68f5-4c04-a2ce-801c3d898fef -->
64          <route>
65              <description>Point to Point Channel</description>
66              <from uri="activemq:point-to-point-channel-8867a184-68f5-4c04-a2ce-801c3d898fef" />
67              <to uri="activemq:message-filter-08c86128-c3e6-49ac-ab95-4599af87ef3d" />
68          </route>
69          <!-- message-filter-08c86128-c3e6-49ac-ab95-4599af87ef3d -->
70          <route>
71              <description>Message Filter</description>
72              <from uri="activemq:message-filter-08c86128-c3e6-49ac-ab95-4599af87ef3d" />
73              <filter>
74                  <!-- TODO: add filter rule, e.g.
75                  <xpath>/sensor/value > -50</xpath>
76                  -->
77                  <to uri="activemq:publish-subscribe-channel-ed67648b-9fa5-4147-8f2a-6c082d72b60a" />
78              </filter>
79          </route>
80          <!-- publish-subscribe-channel-ed67648b-9fa5-4147-8f2a-6c082d72b60a -->
81          <route>
82              <description>Publish-Subscribe-Channel</description>
83              <from uri="activemq:publish-subscribe-channel-ed67648b-9fa5-4147-8f2a-6c082d72b60a" />
84              <multicast>
85                  <to uri="activemq:message-endpoint-057355c9-c130-4aff-bf45-42ef09c811e7" />
86                  <to uri="activemq:message-endpoint-567764eb-641a-4d0c-abaa-80f67591ac3f" />
87
88              </multicast>
89          </route>
90
91      </camelContext>
```

```
92
93 </beans>
```

Listing A.1: Eine aggregierte Apache ActiveMQ XML-Konfigurationsdatei

A.1.2. AWS CloudFormation Template

```
1  {
2      "AWSTemplateFormatVersion": "2010-09-09",
3      "Metadata": {},
4      "Resources": {
5          "AmazonMQBroker2046028219": {
6              "Type": "AWS::AmazonMQ::Broker",
7              "Properties": {
8                  "AutoMinorVersionUpgrade": false,
9                  "BrokerName": "ActiveMQ-Broker-2046028219",
10                 "Configuration": {
11                     "Id": {
12                         "Ref": "AmazonMQConfiguration2046028219"
13                     },
14                     "Revision": {
15                         "Fn::GetAtt": [
16                             "AmazonMQConfiguration2046028219",
17                             "Revision"
18                         ]
19                     }
20                 },
21                 "DeploymentMode": "SINGLE_INSTANCE",
22                 "EngineType": "ACTIVEMQ",
23                 "EngineVersion": "5.15.0",
24                 "HostInstanceType": "mq.t2.micro",
25                 "PubliclyAccessible": false,
26                 "Users": [{
27                     "Username": "test",
28                     "Passwort": "TODO: Change me!"
29                 }]
30             }
31         },
32         "AmazonMQConfiguration2046028219": {
33             "Type": "AWS::AmazonMQ::Configuration",
34             "Properties": {
35                 "Data": {
36                     "Fn::Base64": "<?xml version=\"1.0\" encoding=\"UTF-8\"?> <beans
↪  xmlns=\"http://www.springframework.org/schema/beans\"
↪  xmlns:xsi=\"http://www.w3.org/2001/XMLSchema-instance\"
↪  xsi:schemaLocation=\"http://www.springframework.org/schema/beans
↪  http://www.springframework.org/schema/beans/spring-beans.xsd
↪  http://activemq.apache.org/schema/core
↪  http://activemq.apache.org/schema/core/activemq-core.xsd\"> <bean
↪  class=\"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer\"> <property
↪  name=\"locations\"> <value>file:${activemq.conf}/credentials.properties</value> </property>
↪  </bean> <bean id=\"logQuery\" class=\"io.fabric8.insight.log.log4j.Log4jLogQuery\"
↪  lazy-init=\"false\" scope=\"singleton\" init-method=\"start\" destroy-method=\"stop\"> </bean>
↪  <broker xmlns=\"http://activemq.apache.org/schema/core\" brokerName=\"localhost\"
↪  dataDirectory=\"${activemq.data}\"> <destinationPolicy> <policyMap> <policyEntries> <policyEntry
↪  topic=\">\" > <pendingMessageLimitStrategy> <constantPendingMessageLimitStrategy
```

```
↪ limit=\"1000\"/> </pendingMessageLimitStrategy> </policyEntry> </policyEntries> </policyMap>
↪ </destinationPolicy> <managementContext> <managementContext createConnector=\"false\"/>
↪ </managementContext> <persistenceAdapter> <kahaDB directory=\"${activemq.data}/kahadb\"/>
↪ </persistenceAdapter> <systemUsage> <systemUsage> <memoryUsage> <memoryUsage
↪ percentOfJvmHeap=\"70\" /> </memoryUsage> <storeUsage> <storeUsage limit=\"100 gb\"/>
↪ </storeUsage> <tempUsage> <tempUsage limit=\"50 gb\"/> </tempUsage> </systemUsage>
↪ </systemUsage> <transportConnectors> <transportConnector name=\"openwire\"
↪ uri=\"tcp://0.0.0.0:61616?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600\"/>
↪ <transportConnector name=\"amqp\"
↪ uri=\"amqp://0.0.0.0:5672?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600\"/>
↪ <transportConnector name=\"stomp\"
↪ uri=\"stomp://0.0.0.0:61613?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600\"/>
↪ <transportConnector name=\"mqtt\"
↪ uri=\"mqtt://0.0.0.0:1883?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600\"/>
↪ <transportConnector name=\"ws\"
↪ uri=\"ws://0.0.0.0:61614?maximumConnections=1000&amp;wireFormat.maxFrameSize=104857600\"/>
↪ </transportConnectors> <shutdownHooks> <bean
↪ xmlns=\"http://www.springframework.org/schema/beans\"
↪ class=\"org.apache.activemq.hooks.SpringContextHook\" /> </shutdownHooks> </broker>
↪ <camelContext><!-- point-to-point-channel-a5675128-a8b6-486b-b0e0-106c8a4796d1 --> <route>
↪ <description>Point to Point Channel</description> <from
↪ uri=\"activemq:point-to-point-channel-a5675128-a8b6-486b-b0e0-106c8a4796d1\" /> <to
↪ uri=\"activemq:normalizer-45c1239f-f6c9-4960-b313-a75f47fc964d\" /> </route> <!--
↪ normalizer-45c1239f-f6c9-4960-b313-a75f47fc964d --> <bean id=\"normalizer745699718\"
↪ class=\"TODO: insert bean class pa.ck.age.Class\" /> <route>
↪ <description>Normalizer</description> <from
↪ uri=\"activemq:normalizer-45c1239f-f6c9-4960-b313-a75f47fc964d\" /> <choice> <when> <!-- TODO:
↪ add filter rule, e.g. <xpath>/sensor/type = 'temp-celsius'</xpath> --> <bean
↪ ref=\"normalizer745699718\" method=\"TODO: insert bean method name or remove element, if no
↪ processing is required\" /> <to
↪ uri=\"activemq:point-to-point-channel-8867a184-68f5-4c04-a2ce-801c3d898fef\" /> </when>
↪ <otherwise> <!-- TODO: may adapt dead letter channel --> <to
↪ uri=\"activemq:deadletterchannel\"/> </otherwise> </choice> </route> <!--
↪ point-to-point-channel-8867a184-68f5-4c04-a2ce-801c3d898fef --> <route> <description>Point to
↪ Point Channel</description> <from
↪ uri=\"activemq:point-to-point-channel-8867a184-68f5-4c04-a2ce-801c3d898fef\" /> <to
↪ uri=\"activemq:message-filter-08c86128-c3e6-49ac-ab95-4599af87ef3d\" /> </route> <!--
↪ message-filter-08c86128-c3e6-49ac-ab95-4599af87ef3d --> <route> <description>Message
↪ Filter</description> <from uri=\"activemq:message-filter-08c86128-c3e6-49ac-ab95-4599af87ef3d\"
↪ /> <filter> <!-- TODO: add filter rule, e.g. <xpath>/sensor/value > -50</xpath> --> <to
↪ uri=\"activemq:publish-subscribe-channel-ed67648b-9fa5-4147-8f2a-6c082d72b60a\" /> </filter>
↪ </route> <!-- publish-subscribe-channel-ed67648b-9fa5-4147-8f2a-6c082d72b60a --> <route>
↪ <description>Publish-Subscribe-Channel</description> <from
↪ uri=\"activemq:publish-subscribe-channel-ed67648b-9fa5-4147-8f2a-6c082d72b60a\" /> <multicast>
↪ <to uri=\"activemq:message-endpoint-057355c9-c130-4aff-bf45-42ef09c811e7\" /> <to
↪ uri=\"activemq:message-endpoint-567764eb-641a-4d0c-abaa-80f67591ac3f\" /> </multicast> </route>
↪ </camelContext> </beans>"
            },
            "EngineType": "ACTIVEMQ",
            "EngineVersion": "5.15.0",
            "BrokerName": "ActiveMQ-Config-2046028219"
        }
    }
    ,
    "ElasticBeanstalkApp1790772165": {
        "Type": "AWS::ElasticBeanstalk::Application",
        "Properties": {
```

```
            }
        },
        "ElasticBeanstalkAppVersion1790772165": {
            "Type": "AWS::ElasticBeanstalk::ApplicationVersion",
            "Properties": {
                "ApplicationName": {
                    "Ref": "ElasticBeanstalkApp1790772165"
                },
                "SourceBundle": {
                    "S3Bucket": "TODO: insert Amazon S3 bucket",
                    "S3Key": "TODO: insert Amazon S3 key"
                }
            }
        },
        "ElasticBeanstalkConfigTemplate1790772165": {
            "Type": "AWS::ElasticBeanstalk::ConfigurationTemplate",
            "Properties": {
                "ApplicationName": {
                    "Ref": "ElasticBeanstalkApp1790772165"
                },
                "EnvironmentId": {
                    "Ref": "ElasticBeanstalkEnvironment1790772165"
                }
            }
        },
        "ElasticBeanstalkEnvironment1790772165": {
            "Type": "AWS::ElasticBeanstalk::Environment",
            "Properties": {
                "ApplicationName": {
                    "Ref": "ElasticBeanstalkApp1790772165"
                }
            }
        }

    }
}
```

Listing A.2: Ein aggregiertes Amazon Web Services CloudFormation Template

A.1.3. Sendender Message Endpoint

```
import javax.jms.*;
import javax.naming.*;

public class QueueProducer {

    private QueueSession session;
    private QueueSender sender;

    public static void main(String[] args) {
        QueueProducer producer = new QueueProducer();
        producer.send("Hello world!");
    }

    public QueueProducer() {
        initialize();
```

```
16      }
17
18      private void initialize() {
19          try {
20              Context jndi = new InitialContext();
21              QueueConnectionFactory connectionFactory = (QueueConnectionFactory)
    ↪  jndi.lookup("ConnectionFactory");
22              QueueConnection connection = connectionFactory.createQueueConnection();
23              QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
24              Queue queue = (Queue)
    ↪  jndi.lookup("point-to-point-channel-a5675128-a8b6-486b-b0e0-106c8a4796d1");
25              QueueSender sender = session.createSender(queue);
26
27              connection.start();
28          } catch (Exception e) {
29              e.printStackTrace();
30          }
31      }
32
33      private void send(String message) {
34          try {
35              TextMessage textMessage = session.createTextMessage();
36              textMessage.setText(message);
37              sender.send(textMessage);
38          } catch (JMSException e) {
39              e.printStackTrace();
40          }
41      }
42  }
```

Listing A.3: Ein nachrichtensendender Message Endpoint

A.1.4. Empfangender Message Endpoint

```
1   import javax.jms.*;
2   import javax.naming.*;
3
4   public class QueueConsumer implements MessageListener {
5
6       public static void main(String[] args) {
7           new QueueConsumer();
8       }
9
10      public QueueConsumer() {
11          initialize();
12      }
13
14      private void initialize() {
15          try {
16              Context jndi = new InitialContext();
17              QueueConnectionFactory connectionFactory = (QueueConnectionFactory)
    ↪  jndi.lookup("ConnectionFactory");
18              QueueConnection connection = connectionFactory.createQueueConnection();
19              QueueSession session = connection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
20              Queue queue = (Queue)
    ↪  jndi.lookup("message-endpoint-057355c9-c130-4aff-bf45-42ef09c811e7");
```

```
21              QueueReceiver receiver = session.createReceiver(queue);
22
23              receiver.setMessageListener(this);
24              connection.start();
25          } catch (Exception e) {
26              e.printStackTrace();
27          }
28      }
29
30      public void onMessage(Message message) {
31          // TODO: implement message processing
32      }
33  }
```

Listing A.4: Ein nachrichtenempfangender Message Endpoint

Die Java-Datei des zweiten empfangenden Message Endpoints ist bis auf die Zeile 20 identisch. In der zweiten Java-Datei lautet der Methodenaufruf:

```
jndi.lookup("message-endpoint-567764eb-641a-4d0c-abaa-80f67591ac3f");
```

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift