Bachelorthesis

Fabian Lemke

An der Fachhochschule Dortmund im Fachbereich Informatik im Studiengang Praktische Informatik erstellte Bachelorthesis zur Erlangung des akademischen Grades Bachelor of Science

Matrikelnummer 7105783
geboren am 14.06.1995
Abgabedatum: 20. Juni 2022

1. Prüfer*in: Prof. Dr. Sabine Sachweh
2. Prüfer*in: Philip Wizenty, M.Sc.

Titel: Konzeption und Umsetzung eines Ansatzes zur Identifikation von Antipattern in Microservice-Architekturen

Title: Design and Implementation of an Approach for the Identification of Anti-Patterns in Microservice Architecture

Zusammenfassung. Vor dem Hintergrund des wachsenden Interesses für Microservices wird bei der Verwendung eines sauberen Stils immer wichtiger. Um im Kontext des Model-Driven-Engineering frühzeitig potentielle Antipattern zu identifizieren wird ein erweiterbarer Ansatz zur Identifikation konzeptioniert und mit LEMMA umgesetzt. Darauf aufbauend wird die Funktionsweise der Umsetzung verifiziert. Dabei werden 14 verschiedene Antipattern betrachtet und für Analysezwecke bewertet.

Abstract. Against the backdrop of growing interest in microservices, it is becoming increasingly important to use a clean style. In order to identify potential antipatterns at an early stage in the context of model-driven engineering, an expandable approach to identification is conceptualized and implemented with LEMMA. Based on this, the functionality of the implementation is verified. 14 different antipatterns are considered and evaluated for analysis purposes.

Schlüsselwörter: Antipatterns · Microservices · LEMMA

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

API Application Programming Interface
ESB Enterprise Service Bus
DDML Domain Data Modeling Language
DSL Domain Specific Language
LEMMA Language Ecosystem for Modeling Microservice Architecture
MDE Model Driven Engineering
OML Operation Modeling Language
SML Service Modeling Language
TML Technology Modeling Language

Listingverzeichnis

1 Einleitung

Zu Beginn werden in den Abschnitten 1.1 und 1.2 Motivation und Ziel der Arbeit umrissen. Anschließend wird das Vorgehen der Arbeit in Abschnitt 1.3 dargestellt.

1.1 Motivation und Problemstellung

Microservices sind in den letzten Jahren zu einem beliebten Architekturstil geworden. So setzen immer mehr Unternehmen auf diesen Architekturstil. Nach einer Umfrage von GitLab[Git21] arbeitete 2021 jedes dritte Unternehmen bereits mit Microservices.[New21b]
Was einen Microservice definiert wird in Abschnitt 2.1 betrachtet.

Im Laufe der Zeit stellten sich für unterschiedliche Architekturstile Pattern heraus. Diese Pattern stellen Lösungen für bekannte Probleme dar. Neben diesen Pattern gibt es jedoch auch sogenannte Antipattern, welche zwar eine gute Lösung zu sein scheinen, jedoch viel mehr eine „schlechte“ Lösung darstellen. Zu definierten Antipattern findet sich zur Prävention auch oft ein sauberer Lösungsansatz. Im Rahmen von statischen Analysen können Antipattern auch automatisiert erkannt werden.[PM15; FRS15b; FRS15c]

Daraus ergibt sich, dass bei der Entstehung einer Softwarelösung nicht nur sinnvolle Pattern betrachtet werden müssen, sondern auch die Nicht-Verwendung von Antipattern.[FRS15d; FRS15c]
Eine Identifizierung von Antipattern kann sich also als durchaus sinnvoll erweisen..

Die Forschungsfrage, die diese Arbeit verfolgt ist:

> Wie kann ein Ansatz zur Identifikation von Antipattern in Microservice-Architekturen aussehen?

1.2 Zielsetzung

Ziel dieser Arbeit ist es, Antipattern aus dem Umfeld der Microservices zu identifizieren. Dabei soll die Identifikation anhand von Modellen dargestellt werden. Für die Erstellung der Modelle sowie die Analyse wird als Grundlage *LEMMA*[1] verwendet. Eine Darstellung von LEMMA ist in Abschnitt 2.3 zu finden.

Es soll ein Konzept erstellt werden, welches die Funktionsweise für eine automatisierte Identifikation darstellt und aufzeigt für welche Antipattern eine automatisierte Identifikation möglich ist. Weiterhin soll das Konzept zur Analyse umgesetzt werden.

[1] github.com/SeelabFhdo/lemma

Dabei soll sich die Lösung in die bestehenden Strukturen von LEMMA integrieren.

Am Ende sollen also anhand von LEMMA-Modellen, Antipattern oder zumindest die Gefahr, dass ein Antipattern vorliegt identifiziert werden und mit der bestehenden Analyseinfrastruktur ausgegeben werden.

1.3 Vorgehensweise

Im Nachfolgenden werden in Kapitel 2 für die Arbeit relevante Grundlagen geschaffen. Es werden Microservices, Model-Driven-Engineering (MDE) sowie LEMMA[2] betrachtet.

Zunächst werden in Kapitel 3 Antipattern für Microservices betrachtet. Zur besseren Verständlichkeit werden die Antipattern anhand einer Fallstudie in Abschnitt 3.1 dargestellt. Neben der eigentlichen Betrachtung in Abschnitt 3.3 gibt es in Abschnitt 3.2 eine Differenzierung zwischen den Begriff *Antipattern* und *Smell*.
Bei der eigentlichen Betrachtung wird dabei das Antipattern beschrieben, sofern möglich mit LEMMA dargestellt und mögliche Lösungswege beschrieben.

Danach folgt in Kapitel 4 die Umsetzung mit LEMMA. Dabei wird eingangs in Abschnitt 4.1 die bestehende Komponente zur Analyse betrachtet und dann in den folgenden Abschnitten auf der Basis des vorher Erarbeiteten die notwendige Anpassung in LEMMA konzeptioniert und umgesetzt.

Eine Verifizierung darüber, ob das Erarbeitete funktioniert, findet in Kapitel 5 statt. Dabei werden die in Abschnitt 3.3 aufgeführten Beispiele zur Fallstudie analysiert und die Ergebnisse mit den zuvor erwarteten Ergebnissen abgeglichen.

In Kapitel 6 werden dann verwandte Arbeiten dargestellt und in Bezug gesetzt. Abschließend folgt ein Fazit in Kapitel 7 sowie ein Ausblick in Kapitel 8.

Der Aufbau der Arbeit ist in Abbildung 1.1 dargestellt.

[2] github.com/SeelabFhdo/lemma

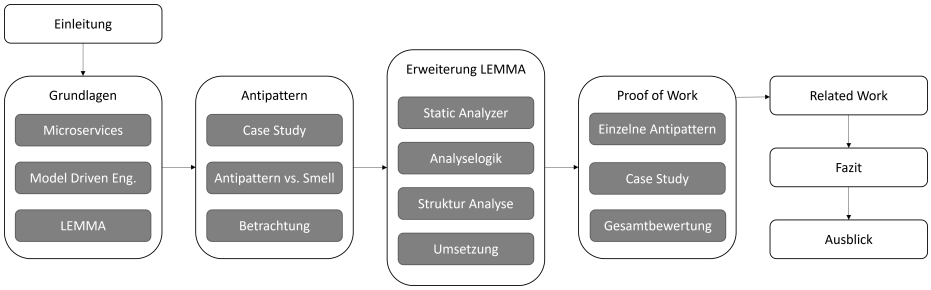

Abb. 1.1. Darstellung der Struktur der Arbeit mit Ablauf und wesentlichen inhaltlichen Kapiteln. (Eigene Darstellung)

2 Grundlagen

In diesem Kapitel werden für die Arbeit relevante Grundlagen dargestellt. In Abschnitt 2.1 wird betrachtet, was einen Microservice definiert. Darauf aufbauend wird in Abschnitt 2.2 das Konzept von *Model Driven Engineering*[GDD09] dargestellt. Zuletzt wird in Abschnitt 2.3 *LEMMA*[RS21] erläutert. Dabei handelt es sich unter anderem um eine Lösung für das zuvor dargestellte Model Driven Engineering.

2.1 Microservices

Das Konzept von Microservices[Wol18; Fam15; New21b] gewinnt immer mehr an Relevanz. So werden, wie in Abbildung 2.1 dargestellt, in 71% der Fälle zumindest teilweise Microservices eingesetzt.

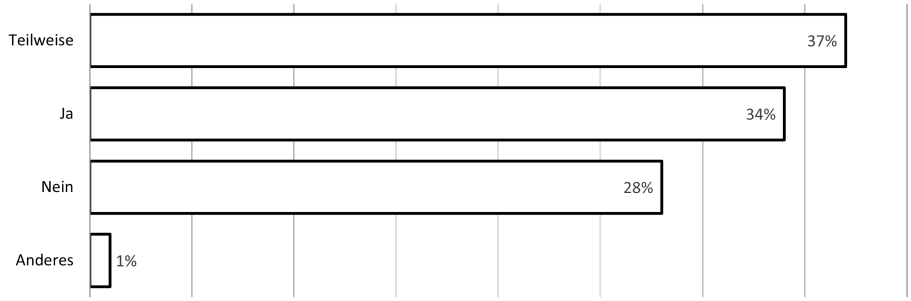

Abb. 2.1. Umfrageergebnis zur Verwendung von Microservices von [Git21](Eigene Darstellung)

Wie der Name *Microservice* bereits nahelegt, ist die Identifikation als Microservice auf Basis der Größe des Services festgemacht.

Ein mögliches Maß für die Größe eines Services sind Codezeilen. Jedoch sind die Zeilen auch von anderen Einflussfaktoren, wie der verwendeten Sprache, abhängig. So kann dieselbe Logik, je nach Technologie, einen unterschiedlichen Umfang haben. Codezeilen eignen sich somit nur sehr begrenzt zur Größenbestimmung eines Services. Weiterhin handelt es sich bei Microservices um einen Architekturansatz und nicht um ein technisches Konzept. Die Servicegrößen werden dabei durch die fachliche Domäne definiert und nicht durch ein technisches Maß.[Wol18; Fam15]

Wichtig ist die Erkenntnis, dass „Micro“ nicht für die Softwaregröße oder den Ressourcenverbrauch steht, sondern für die fachliche Größe, also den Funktionsumfang eines Microservices.[Wol18; Fam15]

Microservices sind nach Familiar[Fam15] autonom und isoliert. Sie stehen komplett für sich und sind nur lose über Schnittstellen und Datenformate miteinander verknüpft. Sie können unabhängig gepflegt werden und den Softwareentwicklungsprozess durchlaufen.

Weiterhin wird beschrieben, dass ein Microservice „Elastic, Resilient, and Responsive“[Fam15] sei. So können Microservices skalieren und Fehler abfangen. Es muss zudem eine kontextbezogene angemessene Performance gegeben sein.

Außerdem beschreibt Familiar[Fam15] die Aspekte der *Nachrichten-Orientierung* und *Programmierbarkeit*. Dabei ist das Prinzip von definierten Datenobjekten zur Kommunikation, also in den Nachrichten, nichts Neues. So findet die Kommunikation oft nach *Representational State Transfer* (ReST) über HTTP statt. Alles was für den aufrufenden Dienst relevant ist, ist die API, alles dahinter ist eine *Black Box*. Die Logik und wie die Daten bearbeitet werden, kann somit nicht eingesehen werden und darf keine Rolle bei der Anbindung spielen.

Des Weiteren muss ein Microservice *konfigurierbar* sein. So besteht ein Microservice in der Realität oft aus mehr als einem Backend und einer API. Es gibt administrative Funktionalitäten, Monitoringfunktionalitäten und weitere Funktionalitäten. Es muss also meist mehr als nur eine API mit verschiedenen Rechten angebunden werden.

Zuletzt wird von [Fam15] Automatisierbarkeit aufgeführt. Der Lebenszyklus eines Microservices soll komplett automatisiert ablaufen. Es sind somit keine manuellen Aktionen notwendig.

Nachfolgend eine Auflistung der von [Fam15] beschriebenen und in den vorherigen Absätzen dargestellten Eigenschaften von Microservices.

- Autonom
- Isoliert
- Skalierbar
- Resilient
- Responsive
- Konfigurierbar
- Automatisierbar

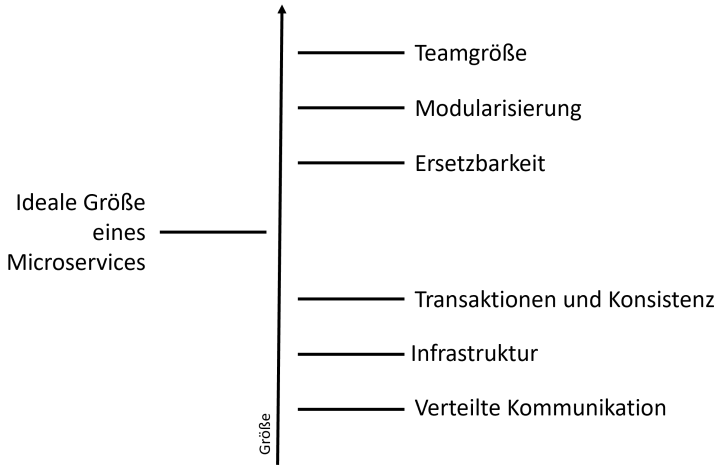

Abb. 2.2. Faktoren zur Größe eines Microservices nach [Wol18](Eigene Darstellung)

Die Größe eines Microservices wird nach Wolff[Wol18] von verschiedenen Einflussfaktoren bestimmt. Dabei werden verschiedene untere und obere Grenzen beschrieben.

Eine obere Grenze ist die Teamgröße. So darf ein Microservice keinesfalls so groß sein, dass mehrere Teams an ihm arbeiten müssen. Weiter eingeschränkt wird dieser durch das Konzept der Modularisierung. So sollte er maximal so groß sein, dass ein Entwickelnder ihn verstehen kann. Ein weiteres Kriterium zur maximalen Größe ist, dass ein Microservice ersetzbar bleiben muss. Für Ersetzbarkeit muss ein komplettes Verständnis einfach möglich sein.

Neben den oberen Grenzen gibt es aber auch untere Begrenzungen für die minimale Servicegröße. So kann bei vielen kleinen Microservices die Bereitstellung notwendiger Infrastruktur ein größeres Problem mit sich bringen als einige größere Services. Auch die notwendige verteilte Kommunikation nimmt mit kleineren Microservices zu und sorgt für Overhead. Transaktionen und Konsistenz begrenzen die Microservicegröße ebenfalls nach unten, da zur Gewährleistung dieser es nicht über mehrere Microservices geschehen darf.[Wol18]
Diese Einflussfaktoren auf die Microservicegröße sind in Abbildung 2.2 dargestellt.

Vorteile: Neben der Frage, was ein Microservice ist, stellt sich jedoch die Frage, welche Vorteile diese bringen. In [IBM21] sind die Ergebnisse von 1200 Befragten zu dem Thema dargestellt. Dabei werden verschiedene Vorteile gesehen. So fallen Argumente wie Sicherheit, schnelle Reaktion auf den Markt, bessere Performance, Skalierbarkeit, schnellere Rollouts und eine generell erhöhte Kundenzufriedenheit. Es sticht jedoch keiner dieser Vorteile klar hervor. Fast alle abgefragten Punkte bewegen sich zwischen 13% und 30% Zustimmung.

So sind zwar viele Vorteile erkennbar, jedoch sticht kein Vorteil besonders hervor.

Auch in der Literatur sind Vorteile zu finden. So beschreibt Familiar[Fam15] folgende Vorteile:

Evolutionär: Durch Microservices ist eine schnelle Erweiterung, auch von bestehenden monolithischen Lösungen möglich. So sind schnellere Releases möglich.

Offen: Durch die Kommunikation über Data Contracts und offene APIs kann jeder Microservice einen eigenen, an das Team angepassten Technologiestack haben. Es gibt keine Begrenzungen.

Hohe Geschwindigkeit: Durch einen überschaubaren Funktionsumfang sind schnellere Releaseiterationen möglich.

Wiederverwendbar und zusammensetzbar: Durch die lose Koppelung können Microservices schnell in anderen Umgebungen und fachlichen Kontexten verwendet werden.

Flexibel: Skalierung und Auslieferung auf verschiedene Systeme wie Produktion oder Test ist flexibel möglich.

Versionierbar und ersetzbar: Da jeder Service für sich releast wird und ein flexibles Deployment möglich ist, lassen sich auch verschiedene Versionen zur Abwärtskompatibilität beibehalten.

Einem Team zugeordnet: Der komplette Produktlebenszyklus liegt bei einem Team. So lassen sich auch agile Praktiken gut umsetzen.

Herausforderungen: Trotz der vielfältigen Vorteile bringt die Arbeit mit Microservices auch Herausforderungen mit sich.

In [IBM21] werden auch abgefragte Herausforderungen dargestellt. Alle Herausforderungen bewegen sich im Bereich von 47% und 52%. Die größte Herausforderung ist die Verfügbarkeit beziehungsweise Kosten von Wissensträgern. Obwohl die Sicherheit von vielen als Vorteil gesehen wird, ist es auch eine der primären Herausforderungen.

Eine strukturierte Darstellung von Herausforderungen bietet Familiar [Fam15]:

(Um)organisation: Bestehende organisatorische Strukturen eigenen Sicht oft nicht für einen Microservice basierten Ansatz. In diesem Rahmen müssen alte Gewohnheiten abgelegt werden und neue Konzepte gelebt werden.

Plattform: Eine Umgebung zur Ausführung von Microservices bereitzustellen kann kostenintensiv und aufwendig sein. Es kann von Vorteil sein eine Cloud-Umgebung zu betrachten.

Identifikation: Identifizieren der fachlichen Teile eines Microservices kann eine Herausforderung sein. Hier bietet es sich an, ein Domänenmodell nach [Eva04] zu erstellen.

Testen: Testmethoden, die ohne eine Microservice-Architektur greifen, können weiterhin angewendet werden. Jedoch haben weitere Aspekte und Testmethoden zusätzlich Relevanz. Diese müssen etabliert und geschult werden.

Auffindbarkeit: In verteilten Umgebungen kann die Auffindbarkeit einer Anwendung Probleme bereiten. So muss die Anwendung nicht immer unter den gleichen Adressen verfügbar sein. Hier bietet sich oft eine „Service-Discovery“ an. Diese stellt Informationen bereit, wo verschiedene Dienste abrufbar sind.[New21a]

2.2 Model Driven Engineering

Die Ursprünge des Model Driven Engineering(MDE) kommen aus ähnlichen Gedanken wie die objektorientierte Programmierung. So schrieb Bézivin bereits 2005:

> „Everything is a model.“ [Béz05]

Ein Modell ist dabei eine Sammlung formaler Elemente. Diese werden dazu benutzt, spezifische Entwicklungen zu analysieren und darzustellen.[GDD09]

Selic stellte dabei im Jahr 2003 fünf Charakteristiken auf, welche ein Modell erfüllen muss. Diese sind in Tabelle 2.1 dargestellt.[Sel03]

Charakteristik	Beschreibung
Abstraktion	Es handelt sich bei einem Modell immer um eine reduzierte Übertragung des repräsentierten Systems.
Verständlichkeit	Ziel muss sein, dass das entstehende Modell nicht nur Details wegabstrahiert sondern auch intuitiv verständlich ist.
Genauigkeit	Die relevanten Features müssen realistisch dargestellt werden.
Vorhersagbar	Anhand des Modells müssen sich nicht offensichtliche Eigenschaften ableiten lassen.
Preiswert	Das Modell muss deutlich günstiger zu erstellen und analysieren sein als das modellierte System.

Tabelle 2.1. Fünf Schlüssel Charakteristiken von Modellen nach [Sel03]

Zur Definition von Modellen können Modellierungssprachen[Com+17] verwendet werden. Dabei stellt die Modellierungssprache selber Modelle zur Modellierung anderer Modelle. Diese Modelle bestehen nach Combemale u. a.[Com+17] aus...

... der Syntax, also wie die Modelle dargestellt werden.

... der Semantik, welche Bedeutung die Modelle haben.
... der Pragmatik, welche beschreibt, wie die Modellierungssprache verwendet wird.

Dabei haben Modellierungssprachen oft eine konkrete Syntax und eine abstrakte Syntax. Die konkrete Syntax ist dabei für einen Modellierenden. Die abstrakte Syntax dagegen dient zur internen Darstellung.
Damit eignet sich die konkrete Syntax beispielsweise, um von einem Entwickelnden genutzt zu werden. Für die Interpretation durch den Rechner eignet sich eine Übertragung in eine abstrakte Syntax.[Com+17]

2.3 LEMMA

Bei LEMMA handelt es sich um ein Projekt des SEELAB[3]. Ausgeschrieben steht LEMMA für ***L**anguage **E**cosystem for **M**odeling **M**icroservice **A**rchitecture*. Es handelt sich dabei um eine Anwendung die beim MDE, wie in Abschnitt 2.2 beschrieben unterstützt. So lässt mit einer Modellierungssprache die Struktur von Microservice-Architekturen beschreiben. Durch die Darstellung in solchen Sprachen eignet sich die Anwendung zur kollaborativen Zusammenarbeit von verschiedenen Beteiligten, wie Domänen-Spezialisierten oder Entwickelnden der Microservices.

Durch die Integration in der Entwicklungsumgebung Eclipse sind Features wie eine Autovervollständigung oder Echtzeitvalidierung möglich. Ein weiteres Feature ist die Erweiterbarkeit von LEMMA.[RS21]

Im Rahmen dieser Arbeit sollen identifizierte Antipattern in LEMMA dargestellt und anhand identifizierter Eigenschaften erkannt werden.

In LEMMA werden dabei verschiedene Modellierungssprachen für unterschiedliche logische Typen verwendet. Diese sind in Tabelle 2.2 aufgelistet und beschrieben. Für alle von LEMMA unterstützten Modellierungssprachen sind im offiziellen LEMMA GitHub-Repository[4] Beispiele zu finden.
Es gibt die folgenden Modellierungssprachen:

- *DDML*: Domain Data Modeling Language
- *TML*: Technology Modeling Language
- *SML*: Service Modeling Language
- *OML*: Operation Modeling Language

Im Folgenden werden für diese Modellierungssprachen die angegebenen Abkürzungen verwendet.
LEMMA unterstützt neben der reinen Modellierung auch Quellcodegenerierung, Analysen und weiteres. Es handelt sich also um ein komplettes Ökosystem um Modellierungssprachen.[RS21; WR22]

[3] seelab.fh-dortmund.de
[4] github.com/SeelabFhdo/lemma/tree/main/examples

Typ	Endung	Beschreibung
DDML	*.data	In diesen Dateien werden die Datenmodelle beschrieben.
TML	*.technology	In Dateien dieses Formats werden Technologien definiert. Technologien sind dabei sehr breit gefasst. So können es Programmiersprachen, Frameworks, Deployment-Technologien oder spezifische Anwendungen, beispielsweise eine Service-Discovery, sein.
SML	*.service	In diesen Dateien findet die Definition von (Micro-)Services statt. So können zum Beispiel Schnittstellen, Abhängigkeiten und verwendete Datenmodelle beschrieben werden.
OML	* operation	In Dateien dieses Typs wird das Deployment beschrieben. Das Deployment ist dabei immer technologiespezifisch. Es können Zusammenhänge, verwendete Technologien und weitere Aspekte dargestellt werden.

Tabelle 2.2. Für die Arbeit relevante Modellierungssprachen in LEMMA sowie deren Verwendung[WR22]

Neben den oben aufgeführten Sprachen gibt es weiterhin sogenannte *Intermediate-Modelle*[Rad22]. Diese Modelle sind im *XML Metadata Interchange* Format [Wei09] abgelegt. Diese Modelle werden aus den in 2.2 dargestellten Typen von Domänensprachen generiert. Es handelt sich dabei um ein „Zwischenformat“, welches die Daten strukturiert und ideal zur Analyse enthält. Es handelt sich dabei um eine abstrakte Modellierungssyntax. Im Verlauf dieser Arbeit wird dieses Format zur Analyse von Modellen verwendet. Mehr dazu ist in Abschnitt 4.1 beschrieben.

3 Antipattern bei Microservices

Dieses Kapitel betrachtet Antipattern von Microservices sowie deren Darstellung in LEMMA (vgl. Abschnitt 2.3). Zum besseren Verständnis werden die Antipattern im Kontext einer in Abschnitt 3.1 beschriebenen Fallstudie gesetzt. Bevor in Abschnitt 3.3 die identifizierten Antipattern betrachtet werden, wird in Abschnitt 3.2 der Unterschied zwischen Antipattern und Smells dargestellt.

3.1 Case Study

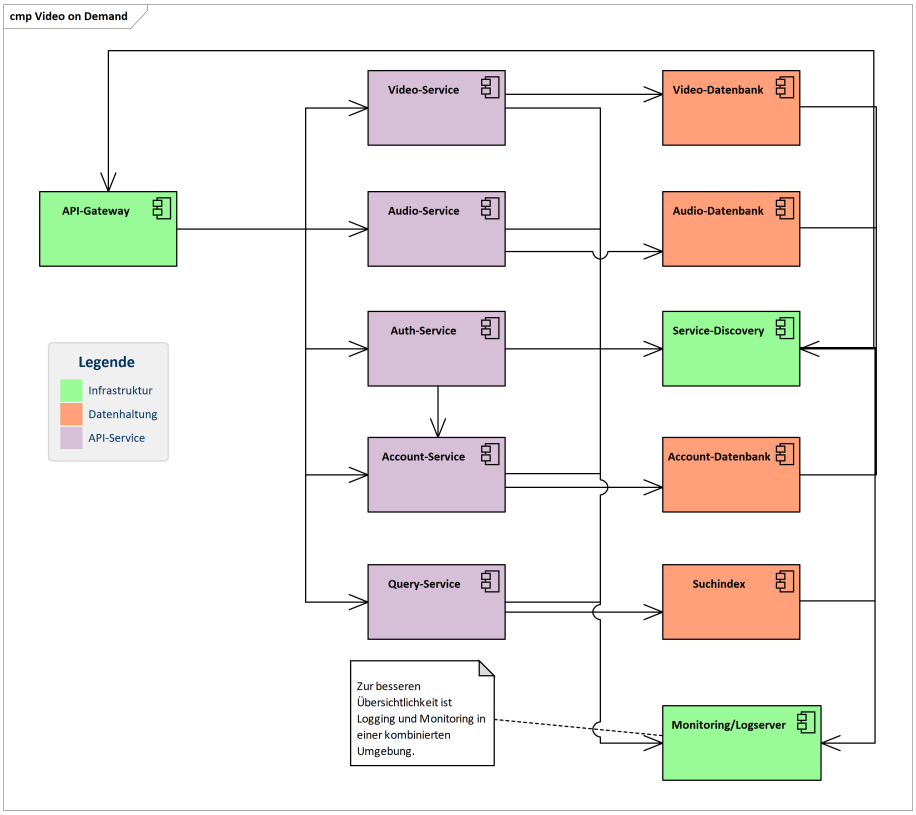

Abb. 3.1. Komponenten einer beispielhaften „Video on Demand“ Lösung angelehnt an [Net21] (Eigene Darstellung)

Die Betrachtung der Antipattern findet im Kontext eines Beispiels statt. Das dargestellte Beispiel ist eine „Video on Demand“ Plattform. Dabei stellt die Beispielanwendung ein vereinfachtes Modell dar. Die Beispielanwendung hat im

Folgenden zur besseren Lesbarkeit den Namen „VODY“.

Die in Abbildung 3.1 dargestellte Microservice-Umgebung kommuniziert über ein zentrales API-Gateway. Die Anfragen an das API-Gateway werden an die verschiedenen API-Services weitergeleitet.
Ein API-Service definiert sich dabei dadurch, dass dieser eine Anwendungs-API bereitstellt.
Die API-Services haben dabei zum großen Teil eine Datenhaltung. So werden Audio- und Video-Daten aus einer Datenbank bezogen. Aber auch Daten für Suchqueries und Accounts haben eine eigene Datenhaltung.

Im Beispiel werden die Funktionen Audio- und Video-Wiedergabe, Suche, Accountverwaltung sowie Authentifizierung dargestellt.
Es ist Nutzenden somit möglich, sich zu authentifizieren und nach gewünschtem Inhalt zu suchen. Nachdem das Gewünschte gefunden wurde, kann es wiedergegeben werden.

Das gewählte Beispiel dient lediglich der Veranschaulichung der in Abschnitt 3.3 beschriebenen Antipattern. Es ist nicht das Ziel, eine ideale oder vollständige Architektur für eine Video-on-Demand-Lösung darzustellen.

3.2 Antipattern oder Smell?

Ein Antipattern ist, im Gegensatz zu einem Design Pattern (vgl. [FRS15d]), eine „schlechte“ Lösung für ein Problem. Diese sollten in der Regel vermieden werden. Antipattern bringen Nachteile mit sich und oft gibt es eine etablierte, sinnvolle Lösung. So wie ein Design Pattern das Softwaredesign verbessert, verschlechtert ein Antipattern das Softwaredesign.[PM15; FRS15c]

Neben Antipattern gibt es noch sogenannte „Smells“. Dies beschreibt eine Eigenschaft, die auf Probleme hindeutet. Es muss in einem solchen Fall nicht immer ein Problem geben, jedoch sollte geprüft werden, ob ein Problem vorliegt. Die Bezeichnung „(Code) Smell“ wurde im Rahmen des Werks [Fow99b] von Martin Fowler und Kent Beck etabliert.[Fow06]

Bei einer Betrachtung der Definitionen fällt auf, dass eine Unterscheidung nicht so einfach ist, wie es möglicherweise erst scheint. Denn wann ist Code wirklich „schlecht“? So gibt es auch bei Verwendung des Begriffs „Antipattern“, wie bei den Smells, oft Gründe für die Verwendung in einem entsprechenden Kontext.
Bei der Sichtung der Literatur zur Identifikation von Antipattern stellte sich heraus, dass dasselbe Pattern mal als Smell und mal als Antipattern bezeichnet wird.

Die Begriffe Smell und Antipattern haben zwar jeweils eigene Eigenschaften, die sich aufgrund der überschneidenden Eigenschaften schnell vermischen.

In dieser Arbeit wird aus diesem Grund **keine** Unterscheidung zwischen Smell und Antipattern vorgenommen.

3.3 Betrachtung der Antipattern

Mögliche Antipattern und Smells wurden aus recherchierter Literatur identifiziert. So haben sich in Vergangenheit bereits Veröffentlichungen mit dem Thema von Antipattern in Microservice-Umgebungen beschäftigt. Dabei wurden folgende Veröffentlichungen herangezogen: [TL18; TLP20; PM15]

Hierbei werden im Folgenden die Antipattern und Smells anhand der Fallstudie aus Abschnitt 3.1 dargestellt, beschrieben und, sofern möglich, eine Modell-Definition mit LEMMA dargestellt.

API-Versionierung: VODY ermöglicht eine Suche nach Elementen. Ab Version 2 wurde jedoch die Suche erweitert. Abbildung 3.2 stellt die entsprechenden Suchobjekte davor und danach dar.

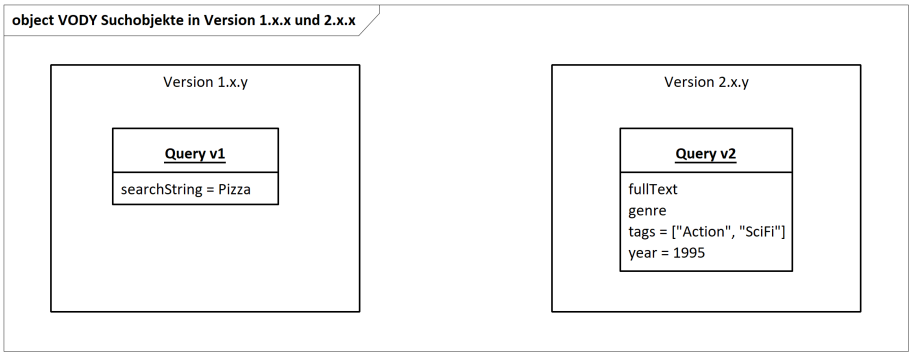

Abb. 3.2. Darstellung der Suchanfrageobjekte in den Major-Versionen 1 und 2 der Beispielanwendung VODY.(Eigene Darstellung)

APIs sind ein zentrales Element zur Kommunikation von Microservices. Da sich diese auch verändern können, muss klar sein, welche Version angesprochen wird. Beim zuvor genannten Beispiel wird eine Suchanfrage ansonsten wahrscheinlich nicht beantwortet werden können. Es muss klar sein, welche Anfrage von der API verarbeitet werden kann. Um Schnittstellenänderungen an der Version zu erkennen, eignet sich das Konzept der semantischen Versionierung von Preston-Werner[Pre21]. So ist klar erkennbar, ob Änderungen vorliegen, die die bisherige Funktionalität nicht mehr sicher stellen. Durch inkompatible API-Versionen können Kommunikationsprobleme auftreten. So kann beispielsweise das Antwort-

objekt ein abweichendes Format haben oder die Schnittstelle nicht verfügbar sein.[TL18; TLP20]

Darstellung in LEMMA: Ob Versionen bei Änderungen immer angepasst werden lässt sich nicht am Modell erkennen. Was jedoch erkennbar ist, ist, ob eine Version bei einem Microservice hinterlegt ist. Dabei sind Versionen dann relevant, wenn Endpunkte definiert sind, mit denen kommuniziert werden kann. Die Definition einer solchen Version eines Microservices ist in Listing 3.1 dargestellt.

```
1  import datatypes from "api-versioning.data" as queryData
2  import technology from "../technology/shared.technology" as
       shared
3
4  @technology(shared)
5  @endpoints(shared::_protocols.rest: "example";)
6  functional microservice vody.QueryService version v2_x_y {
7    interface Interface {
8      public query(
9        sync in request : queryData::query.Request,
10       sync out reply : queryData::query.Result
11     );
12   }
13 }
```

Listing 3.1. Query-Microservice mit definierter Version und einer einfachen Implementierung

Der Microservice ist im Listing am Ende von Zeile fünf versioniert.

Mögliche Behebung des Problems: Für eine Behebung des Problems sollte organisatorisch ein Versionsschema festgelegt werden, sodass ein einheitliches Verständnis besteht, was eine Versionsänderung bedeutet.
So kann beispielsweise die erste Stelle für Änderungen stehen, welche zu einer Inkompatibilität der API führt, die zweite Stelle für neue Funktionalität mit Kompatibilität zu dem Bestehenden und die dritte Stelle für Fehlerbehebung mit Kompatibilität zu dem Bestehenden stehen.[Pre21]

Zyklische Abhängigkeiten: Bei einem wachsenden System kommt es zu neuen Abhängigkeiten unter den verschiedenen Microservices. So kann ein Zirkelbezug entstehen. Das heißt, dass im Endeffekt alle beteiligten Komponenten von einander abhängig sind. Dies sorgt für eine kompliziertere Wartung. Weiterhin ist die Wiederverwendbarkeit sowie Isolierung nicht gegeben. Ein Zirkelbezug kann aus zwei oder mehr Komponenten bestehen.[TLP20; TL18]

Abbildung 3.3 stellt einen solchen Zirkelbezug anhand von VODY grafisch dar. Der Query-Service greift zur Laufzeit auf die Schnittstellen des Auth-Services zu, um zu prüfen, ob die Person authentifiziert ist. Während der Auth-Service für

seine Aufgabe auf die Schnittstellen des Account-Service zugreift. Der Account-Service wiederum greift für eine Interessenbewertung auf die Daten des Query-Services zu. Dadurch besteht eine zyklische Abhängigkeit. Dargestellt ist dies in Abbildung 3.3.

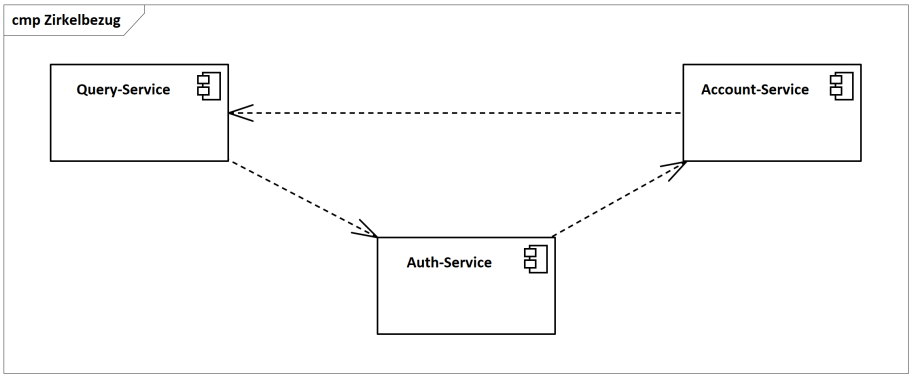

Abb. 3.3. Darstellung eines Zirkelbezugs von Microservices (Eigene Darstellung)

Darstellung in LEMMA: Ein Zirkelbezug lässt sich in LEMMA automatisiert erkennen, da Abhängigkeiten angegeben sind. Der entsprechende LEMMA-Code sieht bei dem beschriebenen Szenario mit der SML auf Serviceebene so wie in Listing 3.2 dargestellt aus.

```
  functional microservice vody.QueryService{
  required microservices {
    AuthService
  }
  // ...
}

functional microservice vody.AccountService {
  required microservices {
    QueryService
  }
  // ...
}

functional microservice vody.AuthService {
  required microservices {
    AccountService
  }
  // ...
```

```
}
```

Listing 3.2. VODY Zirkelbezug unter verschiedenen Microservices mit SML

Diese lässt sich jedoch nicht nur mit der SML, sondern auch mit der OML darstellen. So werden die Abhängigkeiten erst auf Deployment-Ebene definiert. Ein Beispiel dafür ist in Listing 3.3 zu finden.

```
// ...
@technology(generic)
QyeryService is generic::_infrastructure.generic
  depends on nodes AuthService{}

@technology(generic)
AccountService is generic::_infrastructure.generic
  depends on nodes QyeryService{}

@technology(generic)
AuthService is generic::_infrastructure.generic
  depends on nodes AccountService{}
```

Listing 3.3. VODY Zirkelbezug unter verschiedenen Microservices mit OML

Mögliche Behebung des Problems: Ein Zirkelbezug weist darauf hin, dass keine klare Trennung der Domäne stattgefunden hat. Die betroffene Logik sollte hierarchisch umgebaut werden. Das heißt, dass einzelne Funktionalitäten in den übergeordneten Service wandert oder nach unten delegiert wird. Auch eine komplett neue Komponente zur Kapselung der Logik ist möglich und kann sinnvoll sein.
Martin Fowler stellt dabei in dem Buch „Refactoring“ verschiedene Möglichkeiten zur Generalisierung und Lösung des Problems dar. So gibt es Konzepte, um Logiken eine Ebene nach oben zu ziehen, eine Ebene tiefer zu schieben oder aber komplett zu extrahieren.[Fow99a]

Feste Endpunkte: Die Verwendung von statisch definierten Ports oder IP-Adressen sollte vermieden werden. Bei einer Verschiebung auf ein anderes System oder einer anderen Anpassung mit Auswirkungen auf das Netzwerk führt diese Umsetzung zu Problemen.[TL18; TLP20]

Wenn also bei VODY beispielsweise der Auth-Service statisch auf den Account-Service zugreift besteht direkt kein Problem. Wenn dieser jedoch abgelöst wird oder im Rahmen einer Migration oder Systemanpassung zu einer anderen Adresse umzieht, entstehen schnell Fehler, da der Account-Service nicht mehr erreichbar ist. Dies führt im Endeffekt dazu, dass keine Authentifizierung bei dem System möglich ist.

Darstellung in LEMMA: Das Problem lässt sich anhand einer fehlenden Service-Discovery identifizieren. Damit anhand des LEMMA-Modells nachvollzogen werden kann, ob es sich bei einem Service im Deployment um eine Service-Discovery

handelt, muss dies anhand des Modells erkennbar sein. Dafür wird bei der Technologie ein generischer Aspekt definiert. Dieser Aspekt ist, wie im Folgenden in Abschnitt 4.2 beschrieben, für verschiedene Anwendungsfälle allgemeingültig. Die Aspektdefinition für die Service-Discovery *Eureka*[5] ist in Listing 3.4 dargestellt.

```
technology Eureka {
  infrastructure technologies {
    Eureka {
      // ...
    }
  }
  operation aspects {
    aspect ArchitecturePattern for infrastructure{
      string name <mandatory>;
    }
  }
}
```

Listing 3.4. Definition eines generischen Aspects „ArchitecturePattern“ zur Identifikation der Service-Art in der Technologie

Der generische Aspekt wird dann im Deployment als Service-Discovery festgelegt. Relevant ist dafür in Listing 3.5 die in Zeile acht dargestellte Definition des Deploymentknotens als „ServiceDiscovery“

```
import technology from "../technology/eureka.technology" as
    Eureka
import technology from "generic.technology" as generic

@technology(Eureka)
ServiceDiscovery is Eureka::_infrastructure.Eureka
  with operation environment "openjdk:11-jdk-slim"{
  aspects{
    Eureka::_aspects.ArchitecturePattern(name="
    ServiceDiscovery");
  }
// ...
}

@technology(generic)
AccountService is generic::_infrastructure.generic
depends on nodes ServiceDiscovery {}

@technology(generic)
AuthService is generic::_infrastructure.generic
depends on nodes AccountService {}
```

Listing 3.5. Zuweisung des in Listing 3.4 definierten Aspekts in der Deployment-Konfiguration

[5] github.com/Netflix/eureka

Der Service *AccountService* verwendet, die hinterlegte Service-Discovery. Der Service *AuthService* hat jedoch keine Abhängigkeit zur Service-Discovery, sondern lediglich zu dem *AccountService*. Somit erfüllt der *AuthService* nicht das Kriterium eine Service-Discovery zu benutzen.
Der Aspekt *ArchitecturePattern* kann ebenfalls in anderen Kontexten verwendet werden. Beispielsweise auch zur Prüfung, ob andere Komponenten vorhanden sind. Eine Auflistung aller unterstützten Werte des Aspekts ist in Abschnitt 4.2 zu finden.

Mögliche Behebung des Problems: Statt statischer Verbindungsdaten sollte eine Service-Discovery verwendet werden um die Verbindung zu Endpunkten dynamisch zu gestalten.[TL18; TLP20]
Abbildung 3.4 stellt den Ablauf bei der Erreichung des VODY Account-Services über eine Service Discovery dar.

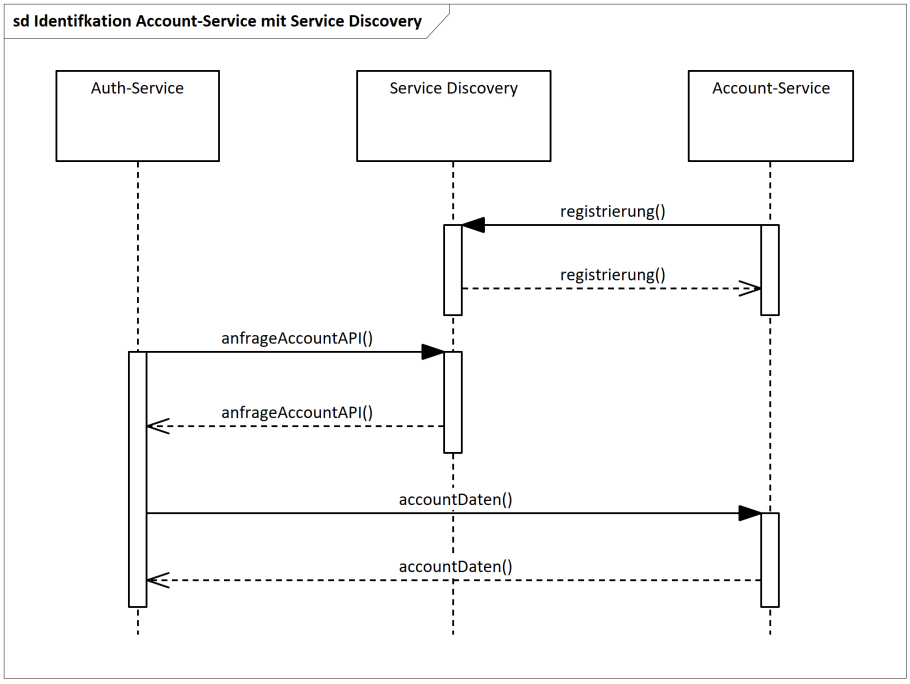

Abb. 3.4. Microservices sind über API-Gateway zugreifbar. Auch die Kommunikation zwischen den beiden Microservices sollte nicht direkt, sondern über das API-Gateway stattfinden. (Eigene Darstellung)

Unangebrachte Service Intimität: Hinter diesem Begriff steckt das Verhalten, dass ein Microservice auf privaten Daten eines anderen Microservices arbei-

tet. Dies ist beispielsweise an API-Anfragen für private Daten oder an der Verwendung der Datenbank eines anderen Microservices erkennbar.[TL18; TLP20]

Dies würde bei VODY beispielsweise der Fall sein, wenn der Auth-Service für benötigte Daten direkt aus der Account-Datenbank abfragt. So würde bei einer Veränderung des Datenbankmodells der Auth-Service nicht mehr funktionieren und der Service wäre nicht mehr alleinstehend.

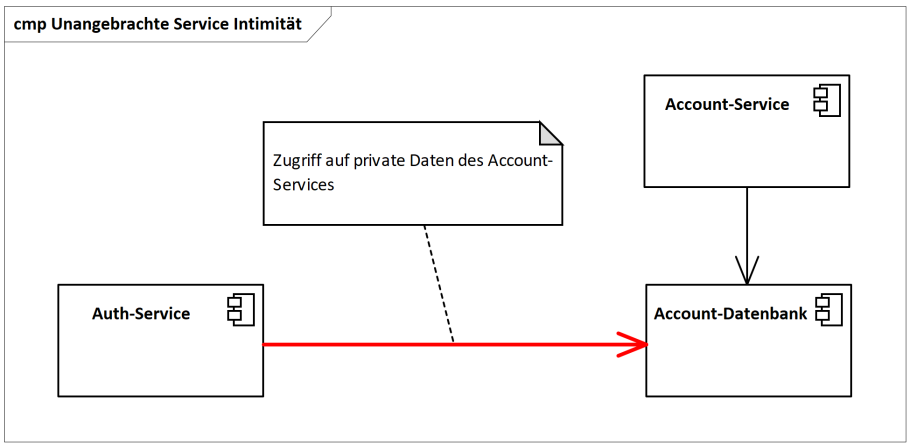

Abb. 3.5. Auth-Service greift auf die privaten Daten von Account-Service zu. (Eigene Darstellung)

Darstellung in LEMMA: Auf der Strukturebene lassen sich API-Aufrufe nicht erkennen, jedoch sorgt die Verwendung einer Datenbank von zwei Microservices dafür, dass diese Fälle identifizierbar sind. Ein Beispiel ist in Abbildung 3.5 dargestellt. Wie dies mit OML in LEMMA aussieht, ist in Listing 3.6 zu sehen.

```
1  import technology from "database.technology" as DB
2  import technology from "../technology/javaWithSpring.
       technology" as Spring
3
4  @technology(DB)
5  AccountDatenbank is DB::_infrastructure.Postgres {
6      aspects{
7        DB::_aspects.ArchitecturePattern(name="Database");
8    }
9  }
10
11 @technology(Spring)
12 AccountService is Spring::_infrastructure.SpringBootAdmin
13   depends on nodes AccountDatenbank{
```

```
14   // ...
15 }
16
17 @technology(Spring)
18 AuthService is Spring::_infrastructure.SpringBootAdmin
19   depends on nodes AccountDatenbank{
20   // ...
21 }
```

Listing 3.6. Zugriff auf die selbe Datenbank von zwei Diensten

Ein Problem an der Darstellung ist, , dass der Zugriff auf eine gemeinsame Datenbank-Instanz durchaus sinnvoll sein kann, sofern die Daten in unterschiedlichen Tabellen oder Schemas liegen und somit getrennt voneinander sind.[TL18]

Mögliche Behebung des Problems: Wie bei den zyklischen Abhängigkeiten sollte hier die Logik, bei welcher dies notwendig ist, umstrukturiert werden. Wie zuvor sei auch an dieser Stelle auf die von Martin Fowler beschriebenen Techniken zur Generalisierung aus [Fow99a] verwiesen.

Microservice Gier: Macht es Sinn für eine einzelne statische Website einen eigenen Service zu erstellen? So könnte es beispielsweise eine Werbewebseite im Rahmen einer VODY-Marketingaktion geben.

Wie in Abschnitt 2.1 dargestellt, ist die Größe eines Microservices nicht einfach zu bestimmen. Dies kann dazu führen, dass kleine Aufgaben wie die Darstellung einer einzelnen, statischen Seite als Microservice umgesetzt werden. Dies kann zu einem Wildwuchs von Microservices führen. Ein System kann so aufgrund der Größe unwartbar werden.[TL18; PM15]

Darstellung in LEMMA: Aufgrund dessen, dass die korrekte Größe eines Microservices von verschiedenen Faktoren abhängt, ist eine Ermittlung anhand des reinen Modells nicht möglich.

Mögliche Behebung des Problems: Funktionalitäten mehrerer Microservices werden in einen zusammengefasst. So kann es beispielsweise einen Microservice zur Anzeige von statischen Seiten geben oder die statische Seite von einem bestehenden Microservice angezeigt werden, welcher zur gleichen Domänenklasse gehört. Außerdem sollte bei der Erstellung eines neuen Microservices abgewogen werden, ob diese benötigt wird.[TL18]

Im angeführten Beispiel wäre es sinnvoll, einen konfigurierbaren Service für statische Webinhalte allgemein bereitzustellen und nicht für jeden statischen Inhalt einen eigenen.

Kein API-Gateway: Die Microservices kommunizieren direkt untereinander. Im schlimmsten Fall greifen sogar die Servicenutzenden direkt auf die Microservices zu. Dadurch wachsen Wartung und Komplexität des Systems. Weiterhin kann es bei einer hohen Anzahl von Microservices aus Erfahrungsberichten zu Kommunikations- und Wartungsproblemen kommen.[TL18; TLP20]
Bezüglich der Kommunikation von Microservices untereinander gibt es unterschiedliche Meinungen, so ist auch die Sicht eines API-Gateways rein für die externe Kommunikation vertreten.[New21a]
Kritisch wird es in jedem Fall, wie zuvor in diesem Absatz bereits dargelegt, bei direkter Kommunikation von außerhalb des Systems. Im Folgenden wird dabei deshalb ausschließlich die Kommunikation von außerhalb des Systems betrachtet.

Darstellung in LEMMA: Wie bereits bei der ServiceDiscovery und Datenbanken findet die Identifikation über einen Aspekt statt. Ein API-Gateway hat jedoch nur Relevanz, wenn Endpunkte bereitgestellt werden, zu welchen eine Verbindung überhaupt möglich ist. Sofern eine Komponente also Endpunkte hat, jedoch keine Abhängigkeit zu einem API-Gateway, sollte genauer geprüften werden. Eine Definition eines korrekten Modells ist in Listing 3.7 zu finden.

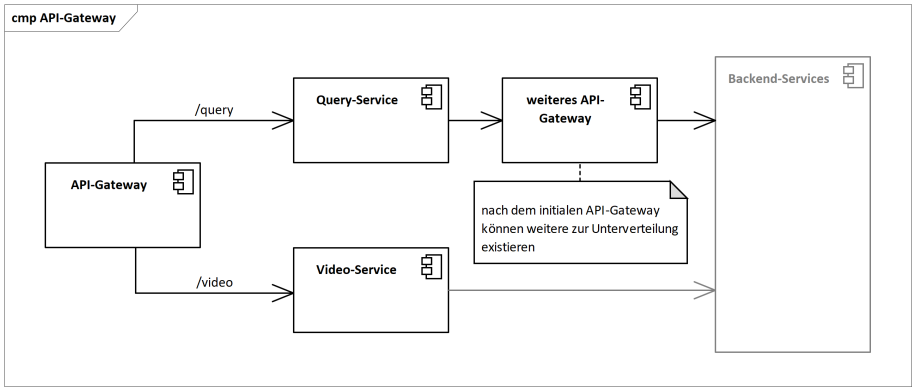

Abb. 3.6. Microservices sind über API-Gateway zugreifbar. So kann es ein oder mehrere API-Gateways geben.[Ama21] (Eigene Darstellung)

```
import technology from "zuul.technology" as Zuul
import technology from "zuul.technology" as Zuul
import technology from "../technology/javaWithSpring.
    technology" as Spring

@technology(Zuul)
Zuul is Zuul::_infrastructure.Zuul
    with operation environment "openjdk:11-jdk-slim"
```

```
8       used by nodes AuthService {
9       aspects{
10          Zuul::_aspects.ArchitecturePattern(name="API-Gateway"
        );
11      }
12      // ...
13  }
14
15  @technology(Spring)
16  AuthService is Spring::_infrastructure.SpringBootAdmin
17      depends on nodes Zuul{
18      // ...
19    endpoints{
20      shared::_protocols.rest: "example";
21    }
22  }
```

Listing 3.7. Darstellung eines Services mit Verwendung eines API-Gateways

Mögliche Behebung des Problems: Die Lösung bei einem fehlenden API-Gateway ist der Einbau eines solchen und die Umstellung jeglicher Kommunikation über diesen.
Dieser Aufbau wird durch das API-Gateway Pattern beschrieben. So gibt es eine Ausprägung mit einem oder mehreren API-Gateways. Grundlegend ähnelt dies Pattern damit dem Facade-Pattern.[Ama21; FRS15a; New21a]
Abbildung 3.6 stellt einen einfachen Aufbau mit API-Gateway dar.

Geteilte Bibliotheken: Microservices können gleiche Bibliotheken und Abhängigkeiten verwenden. So entsteht eine enge Kopplung untereinander. Weiterhin entsteht Kommunikationsaufwand zwischen den Teams bei Anpassung der geteilten Abhängigkeiten.[TL18; TLP20]
Falls also in VODY Account-Service und Query-Service auf gleiche Bibliotheken zugreifen, müssten sich die Verantwortlichen absprechen und bei Änderungen an diesen Bibliotheken müssen immer beide Services getestet werden. Die Kapselung der Services ist also aufgebrochen.

Darstellung in LEMMA: Eine Darstellung geteilter Bibliotheken mit LEMMA ist nicht vorgesehen, da keine Spezifizierung bis zu dieser Tiefe in dem Modell standardisiert vorgesehen ist. Dementsprechend ist **keine** Analyse möglich.

Mögliche Behebung des Problems: Eine Lösung ist die geteilten Bibliotheken in einen eigenen neuen Microservice zu überführen. Dieser wird dann wie die anderen Microservices und deren APIs semantisch versioniert.[TL18; Pre21]

Geteilte Persistenz: Verschiedene Microservices verwenden die gleiche Datenbank. Möglicherweise werden sogar die gleichen Entitäten verwendet. Dies sorgt für eine starke Verknüpfung der Microservices und verringert damit Service- und Team-Unabhängigkeit.[TL18]

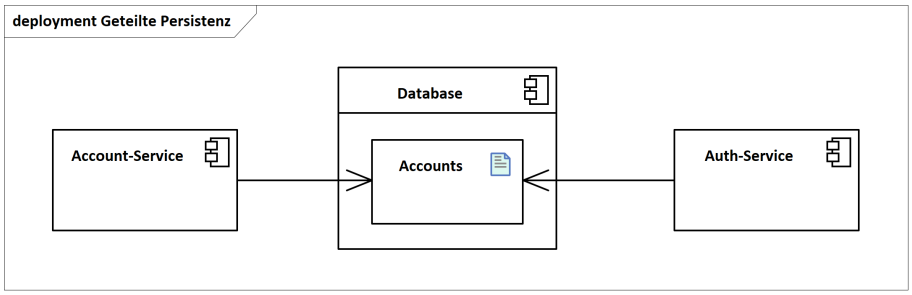

Abb. 3.7. Mehrere Microservices verwenden die selbe Datenhaltung (Eigene Darstellung)

Darstellung in LEMMA: Dieses Antipattern wird identisch zu dem Pattern *unangemessene Service Intimität* an vorheriger Stelle in Abschnitt 3.3 dargestellt. Um redundanten Informationen vorzubeugen, findet keine erneute Darstellung statt.

Mögliche Behebung des Problems: Es bieten sich verschiedene Ansätze an. Zum einen können für jeden Service einzelne Datenbanken verwendet werden. Ein weiterer Ansatz ist, dass die Tabellen in der Datenbank nur durch einzelne Microservices verwendet werden. Die dritte Möglichkeit besteht darin, dass die Daten zwar in einer Datenbank, jedoch in getrennten Entitäten hinterlegt sind.[TL18]

Eine Prüfung auf zugegriffene Tabellen und Entitäten ist in LEMMA nicht praktikabel möglich, da keine Definition auf dieser Abstraktionsebene stattfindet. Lediglich der Zugriff auf die identische Datenbank ist feststellbar.

Zu viele Standards: Es wird eine Vielzahl von verschiedenen Sprachen, Protokollen, Frameworks, und weiteren technologischen Aspekten verwendet. Grundsätzlich erlauben Microservices die Verwendung verschiedener Technologien. Trotzdem kann es zu Problemen führen, da das Wissen zu einzelnen Technologien ungleich verteilt ist. Speziell beim Weggang eines Wissenstragenden ist dies ein Problem.[TL18]

Darstellung in LEMMA: Aufgrund des Modells der Technologien in LEMMA lassen sich die Anzahl der verwendeten Standards prüfen. Es ist jedoch nicht genau messbar, was zu viele Standards sind. Es gibt jedoch keine klar erkennbare Metrik. Im Rahmen einer Prüfung müsste jedoch eine fixe Zahl oder ein fixes Verhältnis als „zu viel“ gelten. Diese ist jedoch zum Beispiel abhängig von Teamgröße und im Team vorhandenem Wissen.

Aus diesen Gründen wird dieses Antipattern **nicht** in LEMMA dargestellt und es erfolgt keine Analyse in dieser Arbeit.

Mögliche Behebung des Problems: Die Verwendung abweichender oder neuer Technologien sollte vorsichtig und bewusst geprüft werden und deren Mehrwert und Wartbarkeit gegenübergestellt werden.[TL18]

Falsche Aufteilung: Im Rahmen der Entwicklung können die Verantwortlichen die Funktionsaufteilung anhand technischer Aspekte durchführen. Die Aufteilung eines Microservices sollte jedoch, wie in Abschnitt 2.1 beschrieben, anhand fachlicher Aspekte erfolgen. Eine Missachtung dessen kann zu einer komplexeren Datenverteilung und eine Trennung der Fachlichkeit führen.[TL18]

Ein Beispiel dafür wäre, wenn der VODY Query-Service auch direkt im Account-Profil Daten ändern kann, um dort die letzten Suchanfragen zu speichern.

Darstellung in LEMMA: Hierbei handelt es sich, wie bei „Zu viele Standards“ um ein Antipattern, welches keine klar prüfbare Logik hat. So ist eine falsche Aufteilung aus einer Nichtbeachtung der Fachlichkeit hervorgegangen. Diese Nicht-Beachtung der Fachlichkeit lässt sich automatisiert jedoch nicht erkennen, da dies nur durch eine Auswertung der fachlichen Bedeutung der Elemente geschehen kann, welche jedoch nicht unterstützt ist.

Aus diesen Gründen wird dieses Antipattern ebenfalls **nicht** in LEMMA dargestellt und es erfolgt auch keine Auswertung.

Mögliche Behebung des Problems: Es sollte eine Analyse der tatsächlichen Geschäftsprozesse und zugehörigen Ressourcen stattfinden. So kann die Planung und Aufteilung von Microservices stattfinden.[TL18]

Megaservice: Ein solcher Service hat sehr viele Aufgaben und wird auch als Monolith bezeichnet. Es sind verschiedene Geschäftsprozesse implementiert. Ein Megaservice besteht oft aus mehreren Modulen und wird von mehreren Entwickelnden oder sogar Teams umgesetzt. Es macht damit die Vorteile einer Microservice Architektur nichtig und hat die Nachteile eines Monolithen.[TLP20]
Ein weiteres Problem eines solchen Services sind häufig hohe Antwortzeiten.[PM15]

Darstellung in LEMMA: Wie bei dem Antipattern „Zu viel Standards“ müsste auch hier eine Metrik festgelegt sein, was „zu viel“ ist. Diese Metrik kann jedoch nur schwer erarbeitet werden, da sie auch stark von der fachlichen Aufgabe eines Dienstes abhängig ist.

Aus diesen Gründen wird dieses Antipattern, wie die vorherigen beiden, **nicht** in LEMMA dargestellt und es erfolgt ebenfalls keine Auswertung.

Mögliche Behebung des Problems: Ein solcher Megaservice sollte in mehrere einzelne Microservices aufgeteilt werden, welche jeweils nur einen fachlichen Aspekt abbilden.[TLP20]

Lokales Logging: Microservices die Log-Meldungen lokal, beispielsweise im Container persistieren, verstecken damit Informationen. So sind im Fehlerfall Log-Meldungen nicht einfach ersichtlich.[TLP20]

Darstellung in LEMMA: Ein zentrales Logging wird, wie Service-Discovery und API-Gateway, über einen Aspekt dargestellt.

```
import technology from "fluentd.technology" as Fluentd

@technology(Fluentd)
LogServer is Fluentd::_infrastructure.fluentd{
    aspects{
        Fluentd::_aspects.ArchitecturePattern(name="LogServer
    ");
    }
}
```

Listing 3.8. Definition des Aspekts „LogServer“

Ohne eine Abhängigkeit an ein zentrales Logging mit dem Aspekt „LogServer“ sollte entsprechend gewarnt werden.

Mögliche Behebung des Problems: Eine mögliche Lösung des Problems ist ein zentraler Logging-Server[TLP20]. Dieser erhält Log-Meldungen von allen Diensten.

Mangel an Monitoring: Die Verwendung eines Monitoring-Systems sorgt für eine automatische Erkennung, sofern Dienste nicht verfügbar sind oder in einem unsauberen Zustand sind. So kann ein Dienst offline sein, ohne, dass die Verantwortlichen es bemerken, wenn es nicht händisch geprüft wird.[TLP20]

Darstellung in LEMMA: Wie zum Beispiel beim zentralen Logging kann geprüft werden, ob ein entsprechendes Objekt mit dem Aspekt „Monitoring“ vorhanden ist.

```
import technology from "monitoring.technology" as Monitoring

@technology(Monitoring)
Monitoring is Monitoring::_infrastructure.GenericMonitoring{
  aspects{
    Monitoring::_aspects.ArchitecturePattern(name="Monitoring
    ");
  }
}
```

Listing 3.9. Definition des Aspekts „Monitoring“

Mögliche Behebung des Problems: Die Lösung für das Problem ist die Einrichtung und Konfiguration eines Monitoring-Systems.[TLP20]

Verwendung Enterprise Service Bus: Ein Enterprise Service Bus (ESB) ist eine Komponente, die Nachrichten verteilt und routet. So wird entstehender Overhead bei der Verarbeitung gehandlet.[CBP16]

Das Problem an der Verwendung eines ESBs ist die größere Komplexität bei Registrierung und Abmeldung von Services an diesem. Oft reicht ein einfacher Message Bus aus.[TL18; TLP20].

Darstellung in LEMMA: Diese findet, wie zuvor andere, mit einem Aspekt statt. Eine mögliche Logik ist in Listing 3.10 zu finden.

```
1  import technology from "Mule.technology" as esb
2
3  @technology(esb)
4  ESB is esb::_infrastructure.Mule{
5    aspects{
6      esb::_aspects.ArchitecturePattern(name="ESB");
7    }
8  }
```

Listing 3.10. Definition eines Deployments mit „ESB“-Aspekt

In dem Fall, dass ein Service mit diesem Aspekt existiert, sollt gewarnt werden. Hierbei handelt es sich um den einzigen Aspekt in dieser Ausführung, welcher nicht verwendet werden sollte.

Mögliche Behebung des Problems: Wie bereits oben in der Beschreibung dargestellt kann ein einfacher Message Bus zur Übermittlung von Nachrichten ausreichen. Dies sollte geprüft und bei Bedarf umgesetzt werden.

4 Erweiterung von LEMMA zur automatischen Analyse

In Abschnitt 2.3 wurde bereits LEMMA eingeführt. Ein Teil dieser Anwendung ist der *Static Analyzer*, der im Rahmen dieser Arbeit um die Analyse einiger Antipattern erweitert wird. In Abschnitt 4.1 wird der Static Analyzer dargestellt. In Abschnitt 4.2 wird dann analysiert, wie eine Analyse aussehen kann. Zuletzt findet die Umsetzung in Abschnitt 4.4 statt.

4.1 LEMMA „Static Analyzer“

Zur Prüfung des geschriebenen LEMMA-Codes wird eine statische Code-Analyse durchgeführt.
Mit einer statischen Codeanalyse kann die Qualität von Quellcode festgestellt und Metriken ermittelt werden. In diesem Rahmen werden technische Schulden identifiziert und ausgewertet. Der Ansatz ist, dass die Kosten, dies in der Zukunft zu bereinigen höher sind als eine zeitnahe Umsetzung.[Rui+20]

Der *Static Analyzer*[Rad22] ist dabei die Komponente von LEMMA, welche entsprechend statische Codeanalysen durchführt. Es werden dabei verschiedene Metriken ausgewertet. Details dazu sind in der angegebenen Literatur zu finden.[AZ11; Eng+18; Hau+17; HCA09]
Der Static Analyzer besteht aus zwei Komponenten.
Dies ist zum einen der Analyzer[6] an sich. Dies ist die Anwendung, welche auch zur Analyse ausgeführt wird. So wird auch das Ausgabeformat in dieser Komponente definiert.
Die tatsächliche Analyse Logik befindet sich jedoch in einer Analyzer Bibliothek[7]. In dieser wird dementsprechend auch die Analyselogik zur Identifizierung von Antipattern implementiert.

Analysen werden dabei auf den in Abschnitt 2.3 beschriebenen *Intermediate*-Modellen durchgeführt. Dabei sind die Logiken für Operation- und Service-Intermediate-Modelle komplett getrennt.

4.2 Analyselogik von Microservice-Antipattern

In Abschnitt 3.3 wurden diverse Antipattern und Smells betrachtet und eine mögliche Erkennung in LEMMA dargestellt.
In Tabelle 4.1 sind die ermittelten Antipattern zusammengefasst.
Auffällig in Abschnitt 3.3, dass die beiden Antipattern *Service Intimität* und *Geteilte Persistenz* zwar verschiedene Bedeutungen haben, jedoch die Erkennung mit LEMMA identisch ist. Dementsprechend wird das Antipattern *Geteilte Persistenz* ausschließlich betrachtet, da es mit der Analysestrategie einen Zugriff auf die gleiche Datenbank zu prüfen spezifischer auf das Schema passt. Es findet

[6] github.com/SeelabFhdo/lemma/tree/main/de.fhdo.lemma.analyzer
[7] github.com/SeelabFhdo/lemma/tree/main/de.fhdo.lemma.analyzer.lib

Antipattern	Kurzbeschreibung
API-Versioning	APIs sollten versioniert sein
Zyklische Abhängigkeiten	Microservice sollte keine Abhängigkeitskette besitzen, welche im Endeffekt auf die Ursprüngliche weist
Verwendung ESB	Enterprise Service Bus sollte nicht verwendet werden
Feste Endpunkte	Andere Microservices und Dienste sollten nicht über feste Endpunkte angesprochen werden
Service Intimität	Kein Zugriff auf private Daten eines anderen Microservices
Kein API-Gateway	Mindestens für die Kommunikation von außerhalb der Anwendung muss ein API-Gateway existieren
Geteilte Persistenz	Zwei Services sollten nicht auf identische Daten zugreifen
Lokales Logging	Logdateien sind nur lokal beim Microservice verfügbar
Monitoring-Mangel	Kein Monitoring der Microservices

Tabelle 4.1. Auflistung von in Abschnitt 3.3 ermittelter zu prüfender Microservice-Antipattern beziehungsweise Smells

aus diesem Grund keine Prüfung für *Service Intimität* statt. Die Alternative dazu ist, dass immer identische Antipattern aufgedeckt werden, was auch keinen Mehrwert bringen würde.

Für eine effizientere Umsetzung werden im Folgenden gleichartige zu prüfende Elemente zusammengefasst und nach Lösungswegen kategorisiert.
Zur besseren Übersichtlichkeit ist dies in Tabelle 4.2 aufgeführt.

Kategorisierung	betroffene Antipattern	Typ
Versionsprüfung	API-Versioning	SML
Rekursive Suche	Zyklische Abhängigkeiten	SML, OML
Zusätzlicher Aspekt	Verwendung ESB	OML
Aspektzugriff	Service Intimität; Geteilte Persistenz	OML
Fehlender Aspekt	Feste Endpunkte; Kein API-Gateway; Lokales Logging; Monitoring-Mangel	OML

Tabelle 4.2. Nach Analyseverfahren kategorisierte Antipattern. Der Typ beschreibt auf welchem Modell aus Tabelle 2.2 die Analyse stattfindet.

Schnell fällt auf, dass alle bis auf zwei Elemente an dem Feature der Aspekte festgemacht sind. Für eine bessere Übersicht wurden die Aspekte alle in derselben Form definiert. Diese Definition ist in Listing 4.1 dargestellt.

```
1 technology Foo{
2   // ...
3   operation aspects {
```

```
4       aspect ArchitecturePattern for infrastructure{
5         string name <mandatory>;
6       }
7     }
8   }
```

Listing 4.1. Generische Form der Aspektsdefinition „ArchitecturePattern“

Diese ist jeweils in der TLM hinterlegt. Das Feld „name“ kann dabei im Rahmen dieser Arbeit folgende unterstützte Werte annehmen:

- **ESB**: Bei der Komponente handelt es sich um einen Enterprise Service Bus
- **ServiceDiscovery**: Diese Komponente ist eine ServiceDiscovery (vgl. [New21a])
- **API-Gateway**: Diese Komponente ist ein API-Gateway (vgl. [New21a; Ama21])
- **Database**: Diese Komponente dient zur Datenhaltung
- **Monitoring**: Dieser Aspekt stellt eine Monitoring-Lösung dar
- **LogServer**: Dieser Aspekt stellt eine zentrale Logging-Lösung dar

Im Folgenden werden die einzelnen Analysekategorien betrachtet und deren Ablauf beschrieben.

Versionsprüfung: Diese Kategorie beschreibt die Prüfung, ob für einen Microervice mit der Bereitstellung einer API eine Version angegeben ist.
Das heißt, sofern für ein Microservices mit Endpoints existiert, sollte eine Version angeben sein, sodass deutlich ist, welche API bereitgestellt wird. Der beschriebene Ablauf ist in Abbildung 4.1 dargestellt.

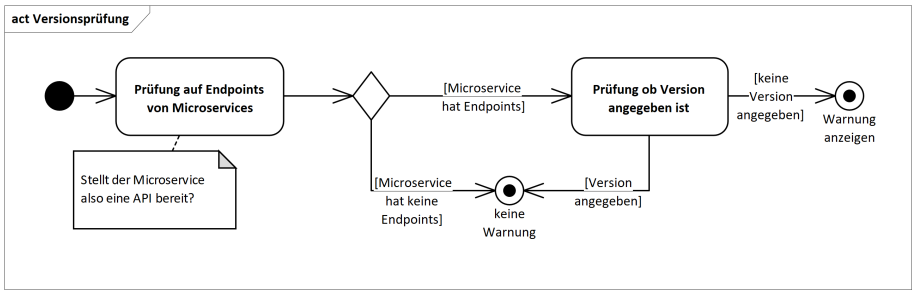

Abb. 4.1. Ablauf um sicherzustellen, dass keine unversionierte API definiert ist, um klar zu stellen welche Schnittstellen eine entfernte API bereitstellt (Eigene Darstellung)

Dabei ist jedoch zu beachten, dass diese Versionsprüfung nur Sinn ergibt, sofern die Versionen entsprechend in dem LEMMA-Model gepflegt und geplant werden. Weiterhin muss bei der Ansteuerung von APIs auch beachtet werden, dass eine API der entsprechenden Version angesprochen wird. Dies wird durch die aktuelle Prüfroutine nicht abgedeckt.

Rekursive Suche: Um zyklische Bezüge aufzudecken, muss die Abhängigkeitshierarchie zyklisch durchsucht werden. Das heißt für jeden Microservice werden die Abhängigkeiten rekursiv geprüft und verifiziert, ob über Abhängigkeiten wieder auf den Microservice verwiesen wird. Der Ablauf ist in Abbildung 4.2 dargestellt.

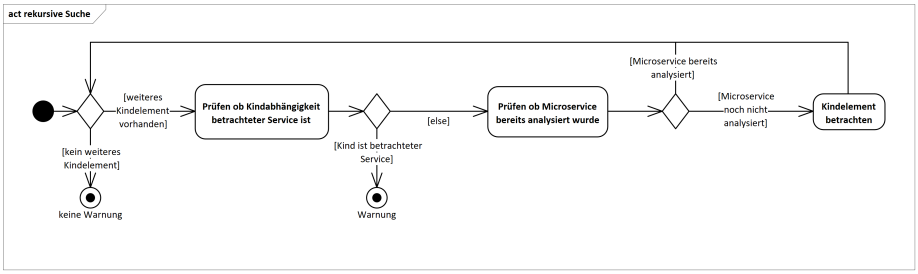

Abb. 4.2. Ablauf um Sicherzustellen, dass kein Zirkelbezug in der LEMMA-Definition besteht. (Eigene Darstellung)

Es besteht die Möglichkeit, dass ein Zirkel innerhalb der Abhängigkeiten besteht, mit welchem das geprüfte Objekt nichts zu tun hat. Damit diese Zirkel nicht zu einer Warnung oder Endlosschleife bei der Analyse führen, wird vor der Analyse auf weitere Kindabhängigkeiten geprüft und ob diese schon betrachtet wurden. Um diese Logik umzusetzen, wird eine Liste betrachteter Elemente in der Implementierung benötigt. Weiterhin wird für die Ausgabe eine Liste der beteiligten Elemente gepflegt, sodass bei Ausgabe der Warnung der Zirkel angezeigt werden kann.

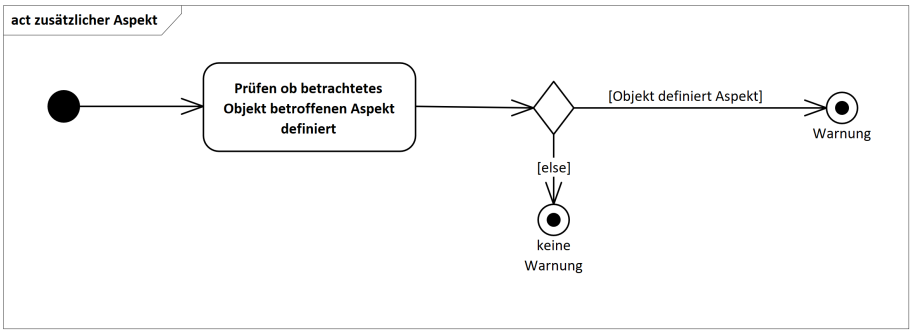

Abb. 4.3. Ablauf zur Prüfung der Verwendung zusätzlicher Aspekte. (Eigene Darstellung)

Zusätzlicher Aspekt: Diese Kategorie von Antipattern beziehungsweise Smells bei welcher eine Komponente in der Anwendung existiert, welche direkt ein Antipattern oder Smell ist. So kann die Verwendung dieser Komponente beispielsweise auf eine unnötige Komplexität hindeuten.
Falls eine Komponente also einen entsprechenden Aspekt definiert, sollte eine Warnung ausgegeben werden.
Das Vorgehen dafür ist in Abbildung 4.3 dargestellt.

Aspektzugriff: Services mit bestimmten Aspekten sollten nur durch einen Microservice benutzt werden, da sonst die klare Kapselung dieser aufgebrochen wird. Dies ist der Fall bei der Datenhaltung. Wenn zwei oder mehr Microservices auf dieselben Datenstrukturen zugreifen, muss bei Anpassung dieser Datenstruktur sichergestellt werden, dass alle Microservices kompatibel bleiben.
Anders als bei der Kategorie „Zusätzlicher Aspekt“ reicht es hier nicht nur das Objekt mit dem Aspekt zu betrachten. Es muss sichergestellt werden, dass nicht mehr als ein Microservice eine Datenbank oder andere betroffene Aspekte verwendet werden. Der Ablauf zur Prüfung ist in Abbildung 4.4 dargestellt. Rele-

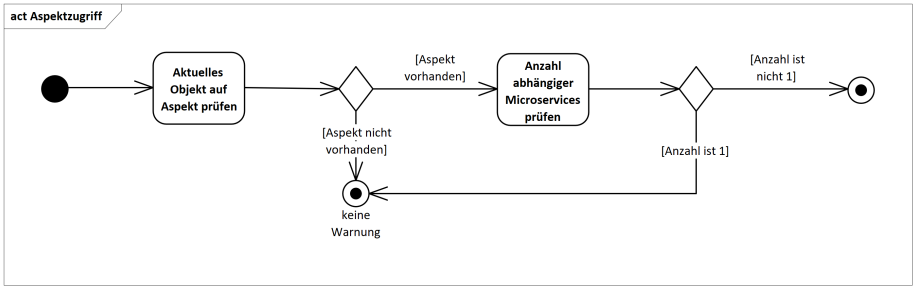

Abb. 4.4. Ablauf zur Prüfung des Zugriffs mehrerer oder keiner Dienste auf die aktuelle Ressource. Nur der Fall mit genau einem zugreifenden Dienst ist ohne Warnung. (Eigene Darstellung)

vant für diese Prüfung ist die Definition der Knoten, die es eine Abhängigkeit zu den Komponenten haben.

Fehlender Aspekt: Bei den Antipattern beziehungsweise Smells dieser Kategorie fehlen Komponenten, welche von Microservices verwendet werden. Dies können Infrastrukturkomponenten, wie eine Service-Discovery oder ein API-Gateway sein. Eine Implementierung dieser Komponenten sollte im Deployment verwendet werden. Der Ablauf für diese Prüfung ist in Abbildung 4.5 dargestellt.

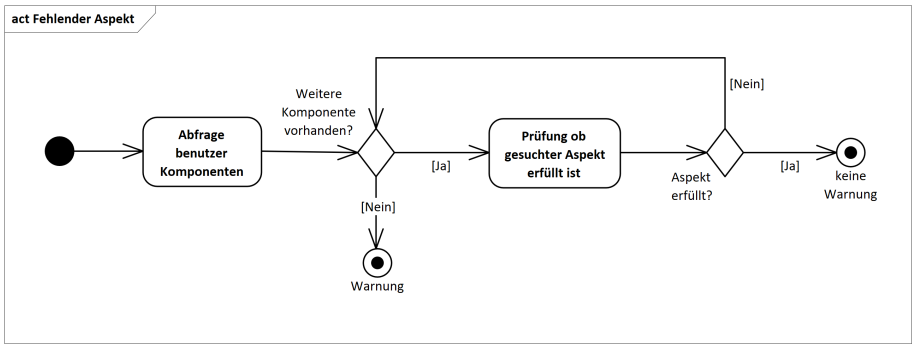

Abb. 4.5. Ablauf zur Prüfung auf die Verwendung benötigter Drittkomponenenten. (Eigene Darstellung)

4.3 Strukturierung Antipatternanalyse

Aufgrund dessen, dass es keine feste Anzahl von Antipattern und Smells gibt, macht die Möglichkeit zur Erweiterbarkeit um weitere Prüfroutinen Sinn.

In Abschnitt 4.2 wurden die betrachteten Pattern bereits kategorisiert und analysiert und eine Logik zur Identifikation dargestellt.

Für diese Analyse-Strategien bietet sich das „Strategie-Pattern“[FRS15b] an. Bei diesem Pattern können standardisiert zur Laufzeit mehrere Logiken definiert und dynamisch ausgeführt werden. Dabei ist der Aufruf immer identisch.[FRS15b] Für jede zuvor identifizierte Kategorie wird eine Strategie definiert. Möglicherweise benötigte Parameter, wie der zu prüfende Wert des Aspekts, werden bei der Definition der Strategien festgelegt. Während der Durchführung des in Abschnitt 4.1 beschriebenen Analyzers werden alle definierten Analyse-Strategien nacheinander auf dem Datenmodell durchgeführt.

Das Ergebnis einer Analyse umfasst dann die gefundenen Antipattern und ein Text zur Beschreibung des aufgetretenen Problems. Der Aufbau dieser Verarbeitungslogik ist in Abbildung 4.6 für die ODM-Dateien dargestellt.

4.4 Umsetzung in LEMMA

Nachdem die in den Abschnitten 4.2 und 4.3 erarbeite Logik zur Antipattern-Analyse in der Analyzer Bibliothek umgesetzt wurde, muss in den Klassen *IntermediateOperationAntipatternAnalysis*[8] und *IntermediateServiceAntipatternAnalysis* die Ausgabe definiert werden. Da diese Logik identisch ist, wird die Ausgabe

[8] Dies ist auch in Abbildung 4.6 abgebildet

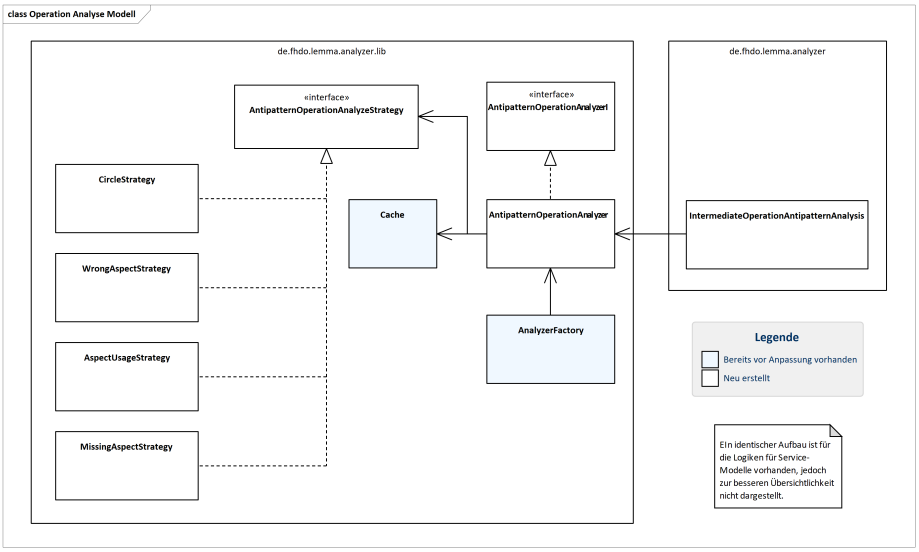

Abb. 4.6. Darstellung des Strategie-Patterns zur Analyse von den identifizierten Antipattern gemäß Tabelle 4.2. (Eigene Darstellung)

über eine zentrale Implementierung definiert. Der dadurch entstehende Aufbau ist in Abbildung 4.7 zu finden.

Eine Ausgabe sieht dann beispielsweise wie in Listing 4.2 abgebildet aus.

```
1  ****************************************************
2  Antipattern analysis result - Type: Operation
3  ****************************************************
4  Disclaimer: Identified Antipattern are not always bad
       practices, but can be a hint!
5  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6  Statistic - Identified antipatterns
7  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
8  Hardcodierte Endpoints:  3
9  Lokales Logging:   1
10 Mangelndes Monitoring:   5
11 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
12 List - All identified antipatterns
13 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
14 Hardcoded Endpoints:   A service discovery should be used.
       Object Zuul has no connection to a Operation-Node with
       ArchitecturePattern-Aspect and name 'ServiceDiscovery'
15 Hardcoded Endpoints:   A service discovery should be used.
       Object AuthService has no connection to a Operation-Node
       with ArchitecturePattern-Aspect and name '
       ServiceDiscovery'
```

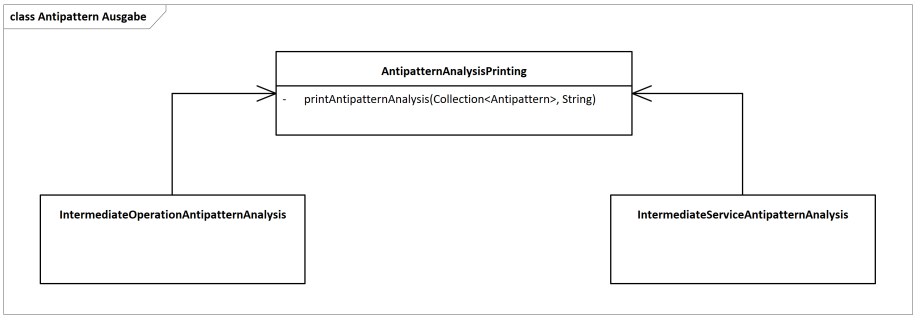

Abb. 4.7. Helper-Klasse zur Ausgabe der identifizierten Antipattern. (Eigene Darstellung)

```
16 // ...
```

Listing 4.2. Beispiehafte Ausgabe mit *AntipatternAnalysisPrinting*

Der gesamte entstandene Quellcode ist auf GitHub in dem Fork „Kikkirej/lemma“ im Branch „antipattern-fle“ abgelegt. Direkt erreichbar ist dieser Stand über github.com/Kikkirej/lemma/tree/antipattern-fle. Im Anhang unter B sind weiterhin alle implementierten Dateien dargestellt.

5 Proof of Work

In Abschnitt 3.3 wurden zur Erläuterung der verschiedenen Antipattern Modelle erstellt. Neben den im Dokument dargestellten Ausschnitten ist der komplette Code für die Modelle auf GitHub[9] veröffentlicht. Die erarbeiten Modelle werden von der, im Rahmen von Kapitel 4 angepassten, statischen Code Analyse geprüft. Das Vorgehen umfasst dabei folgende Schritte:

1. Beschreibung erwartetes Verhalten
2. Ausführung des *Static Analyzer* (vgl. Abschnitt 4.1)
3. Bewertung des Ergebnisses

Da die einzelnen betrachteten Modelle oft speziell für ein Antipattern erstellt wurden und andere Antipattern nicht beachtet wurden, findet keine Betrachtung gerade nicht relevanter Prüfroutinen statt.

Die einzelnen Modelle werden in Abschnitt 5.1 analysiert. In Abschnitt 5.2 wird das Modell zur Fallstudie aus Abschnitt 3.1 analysiert.
Abschließend findet in Abschnitt 5.3 eine Gesamtbewertung des Proof of Works statt.

5.1 Betrachtung Modelle einzelner Antipattern

Da sich die Beispiele aus Abschnitt 3.3 auf spezifische Antipattern beziehen und es sich um minimale Modelle zur Visualisierung und Prüfung des jeweiligen Patterns handelt, werden immer nur Strategien zu dem betrachteten Antipattern aufgeführt. Dafür werden bei den einzelnen Analysen die in 4.6 dargestellten Strategien entsprechend nicht initialisiert.

Weiterhin liegt die Frage nah, warum die Antipattern und nicht die einzelnen Analysekategorien aus Abschnitt 4.2 betrachtet werden. Der Grund liegt in der unterschiedlichen Parametrisierung der Analysestrategien. So muss beispielsweise bei der Strategie *Fehlender Aspekt* in einigen Fällen auf vorhandene Endpunkte geprüft werden und in anderen nicht.

API-Versionierung: Zum Testen der API-Versionierung wird der Code aus Listing 3.1 verwendet. Um auch den anderen Fall zu testen, wird bei einem zweiten Durchlauf geprüft, wie das Verhalten ohne Version ist.

Durchführung der Analyse: In den Listings 5.1 und 5.2 sind die Ausgaben dargestellt.

```
1 ************************************************************
2 Antipattern analysis result - Type: Service
3 ************************************************************
```

[9] github.com/Kikkirej/lemma-models-ba

```
No potential antipatterns identified. Everything seems to be
    clean.
```

Listing 5.1. Ausgabe Static Analyzer API-Versionierung - Version angegeben

```
************************************************************
Antipattern analysis result - Type: Service
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Statistic - Identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
API-Versioning:  1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
List - All identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
API-Versioning:  vody.QueryService has open endpoints, but no
     version. API-Changes wouldn't be noticed.
```

Listing 5.2. Ausgabe Static Analyzer API-Versionierung - Version nicht angegeben

Die Ergebnisse entsprechen somit den im Vorfeld erwarteten Ergebnissen.

Zyklische Abhängigkeiten: Diese lassen sich sowohl in SML und OML darstellen. Aus diesem Grund wird für beide Sprachen ein Modell geprüft. Für SML wird das Beispiel aus Listing 3.2 und für OML das Beispiel aus Listing 3.3 verwendet.

Durchführung der Analyse: Listing 5.3 und 5.4 stellen die Ausgaben der statischen Analyse dar.

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Statistic - Identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Cyclic Dependencies:   1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
List - All identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Cyclic Dependencies:   A cyclic dependency was detected. The
    cyclic dependency consists of [QyeryService, AuthService,
     AccountService]
```

Listing 5.3. Ausgabe Static Analyzer zyklische Abhängigkeit - OML

```
1  ************************************************************
2  Antipattern analysis result - Type: Service
3  ************************************************************
4  Disclaimer: Identified Antipattern are not always bad
       practices, but can be a hint!
5  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6  Statistic - Identified antipatterns
7  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
8  Cyclic Dependencies:   1
9
10 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
11 List - All identified antipatterns
12 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
13 Cyclic Dependencies:   A cyclic dependency was detected. The
       cyclic dependency consists of [vody.QueryService, vody.
       AuthService, vody.AccountService].
```

Listing 5.4. Ausgabe Static Analyzer zyklische Abhängigkeit - SML

In beiden Fällen wurde die zyklische Abhängigkeit identifiziert und die involvierten Services aufgelistet. Es gibt keine doppelten Einträge. Diese wurden erfolgreich heraussortiert.

Feste Endpunkte: Zur Identifikation wird nach der Verwendung einer Service Discovery gesucht. Für den Funktionstest wird das Modell aus Listing 3.5 verwendet. Dabei hat der *AccountService* eine Abhängigkeit zur Service-Discovery, der *AuthService* jedoch nicht.

Durchführung der Analyse: Listing 5.5 stellt die Ausgabe der Analyse dar.

```
1  ************************************************************
2  Antipattern analysis result - Type: Operation
3  ************************************************************
4  Disclaimer: Identified Antipattern are not always bad
       practices, but can be a hint!
5  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6  Statistic - Identified antipatterns
7  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
8  Hardcoded Endpoints:   1
9
10 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
11 List - All identified antipatterns
12 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
13 Hardcoded Endpoints:   A service discovery should be used.
       Object AuthService has no connection to a Operation-Node
       with ArchitecturePattern-Aspect and name '
       ServiceDiscovery'
```

Listing 5.5. Ausgabe Static Analyzer Feste Endpunkte

Es wird, wie erwartet, der Service *AuthService* ohne Verbindung zur Service-Discovery identifiziert.

Kein API-Gateway: Der Beispielcode aus Listing 3.7 hat ein API-Gateway enthalten. Bei statischer Analyse dieses Modells dürfte dementsprechend kein Fehler auftreten. Damit nicht nur der Fall ohne Auftreten des Patterns getestet wird, wird zuerst der Aspekt entfernt und eine weitere Analyse durchgeführt. Danach werden auch die Endpunkte entfernt und es wird eine letzte Analyse durchgeführt.
Das erwartete Ergebnis ist somit:

1. Durchlauf: Kein Antipattern, da API-Gateway vorhanden.
2. Durchlauf: Antipattern wird gefunden, da Endpunkte existieren, jedoch kein API-Gateway.
3. Durchlauf: Kein Antipattern, da ohne Endpunkte kein API-Gateway benötigt wird.

Durchführung der Analyse: Die Ausgabe der Analysen ist in den Listings 5.6 und 5.7 angegeben.

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
No potential antipatterns identified. Everything seems to be
    clean.
```

Listing 5.6. Ausgabe Static Analyzer API-Gateway - Durchläufe 1 und 3

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Statistic - Identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
No API-Gateway:  1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
List - All identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
No API-Gateway:  Although the service has endpoints no API-
    Gateway is connected. Object AuthService has no
    connection to a Operation-Node with ArchitecturePattern-
    Aspect and name 'API-Gateway'
```

Listing 5.7. Ausgabe Static Analyzer API-Gateway - Durchläufe 2

In den Listings ist erkennbar, dass die erwarteten Ergebnisse eingetreten sind. So sind beim ersten und dritten Durchlauf keine Antipattern identifiziert worden und beim zweiten wiederum schon.

Geteilte Persistenz: Da, wie in Abschnitt 4.2 beschrieben, die Erkennung von *Service Intimität* und *Geteilter Persistenz* identisch ist, wird an dieser Stelle das Beispiel zur Service-Intimität aus Listing 3.6 verwendet. Dabei greifen zwei Dienste auf die identische Datenbank zu.

Durchführung der Analyse: Die Ausgabe der Prüfung des Beispielmodells ist in Listing 5.8 zu finden.

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Statistic - Identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shared Persistence:  1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
List - All identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Shared Persistence:  Only one Service should access a data
    storage. In case of data separation, like different
    tables this can be ignored. Too many Nodes are dependend
    on AccountDatenbank with ArchitecturePattern Database.
```

Listing 5.8. Ausgabe Static Analyzer geteilte Persistenz

Das konstruierte Antipattern konnte erfolgreich identifiziert werden. Der Testlauf war somit erfolgreich.

Lokales Logging und Monitoring-Mangel: Diese beiden Antipattern werden zusammengefasst betrachtet, da die Logik und die Parametrisierung dieser nahezu identisch ist. Lediglich der Wert des Felds *name* im Aspekt *ArchitecturePattern* ist unterschiedlich.
Zur Prüfung dieser Logik werden die Listings 3.8 und 3.9 geprüft. Dabei muss bei korrekter Funktionsweise aufgrund der gekapselten Beispiele beim Test des Monitoring-Beispiels das zentrale Logging fehlen und beim Test auf zentrales Logging der Monitoring-Server fehlen.

Durchführung der Analyse: Die Ergebnisse sind in den Listings 5.9 und 5.10 zu finden.

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
Statistic - Identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Missing Monitoring:  1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
List - All identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Missing Monitoring:  Services should be monitored central.
    Object LogServer has no connection to a Operation-Node
    with ArchitecturePattern-Aspect and name 'Monitoring'
```

Listing 5.9. Ausgabe Static Analyzer zentrales Logging

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Statistic - Identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Local Logging:   1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
List - All identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Local Logging:   Log-Data should be stored central. Object
    Monitoring has no connection to a Operation-Node with
    ArchitecturePattern-Aspect and name 'LogServer'
```

Listing 5.10. Ausgabe Static Analyzer Monitoring-Mangel

Beide Ergebnisse entsprechen dem erwarteten Verhalten.

Verwendung ESB: Anders als bei den anderen Analysen, wird hier eine Warnung geworfen, sofern generell ein Knoten mit diesem Aspekt existiert. Die Motivation dahinter ist, dass die Verwendung eines ESBs auf eine möglicherweise nicht notwendige Komplexität hinweist. Sofern andere solcher Knotentypen identifiziert werden sollten, ließen diese sich jedoch mit der identischen Strategie analysieren.

Durchführung der Analyse: Das Ergebnis der statischen Analyse ist in Listing 5.11 zu finden.

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

```
Statistic - Identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Usage Enterprise Service Bus:  1

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
List - All identified antipatterns
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Usage Enterprise Service Bus:  Using an 'Enterprise Service
    Bus' can make a solution too complex. The service with
    the aspect is ESB
```

Listing 5.11. Ausgabe Static Analyzer Enterprise Service Bus

Wie am Listing erkennbar wurde die Verwendung des Enterprise Service Bus erkannt. Der identifizierte Service hat den Namen *ESB*, da er dem Beispiel zum selbigen entstammt.

5.2 Betrachtung Case Study: VODY

Neben den einzelnen Antipatternmodellen wird auch das Modell zu VODY analysiert. Der entsprechende Code zu VODY ist im Anhang A verfügbar.

Dabei findet sowohl eine Analyse der SML-Modelle aus Listing A.1 und der OML-Modelle aus Listing A.2 statt.
Bei einer Analyse sollten keine Antipattern erkennbar werden. Sofern doch Antipattern erkennbar sind, sollte erklärbar sein, warum diese trotzdem aufgetreten sind.

SML-Modelle: Die Ausgabe für die statische Analyse der SML-Modelle ist in Listing 5.12 dargestellt.

```
************************************************************
Antipattern analysis result - Type: Service
************************************************************
No potential antipatterns identified. Everything seems to be
    clean.
```

Listing 5.12. Ausgabe Static Analyzer für *vody.services*

Das gegebene SML-Modell scheint gemäß des Analyseergebnisses keines der betrachteten Antipattern zu enthalten. Dies entspricht dem erwarteten Ergebnis.

OML-Modelle: Die Ausgabe für die statische Analyse der OML-Modelle ist in Listing 5.13 dargestellt.

```
************************************************************
Antipattern analysis result - Type: Operation
************************************************************
Disclaimer: Identified Antipattern are not always bad
    practices, but can be a hint!
```

```
5  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
6  Statistic - Identified antipatterns
7  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
8  Hardcoded Endpoints:   1
9
10 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
11 List - All identified antipatterns
12 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
13 Hardcoded Endpoints:   A service discovery should be used.
       Object MonitoringAndLogging has no connection to a
       Operation-Node with ArchitecturePattern-Aspect and name '
       ServiceDiscovery'
```

Listing 5.13. Ausgabe Static Analyzer für *vody.operation*

Bei der Betrachtung der Ausgabe fällt auf, dass ein Antipattern identifiziert wurde. So hat die Komponente für Monitoring und Logging keine Abhängigkeit zur Service-Discovery. Dies hat den Grund, dass sonst ein Zirkelbezug bestehen würde, da beide aufeinander verweisen. Es ist jedoch in der Tat eine identifizierte Unschönheit in dem Modell, da so die anderen Services nicht über die Service-Discovery die Lösung für Monitoring und Logging identifizieren können.
Das gefundene Antipattern ist somit korrekt identifiziert und es besteht ein Fehler im Design aus Abschnitt 3.1.

5.3 Gesamtbewertung der Ergebnisse

In den vorhergehenden Abschnitten 5.1 und 5.2 wurden verschiedene Modelle durch den angepassten Static Analyzer geprüft. Bei den Einzelprüfungen entsprachen alle Ausgaben dem erwarteten Verhalten. Es gab keine abweichenden Ergebnisse. Die erwartete Funktionsweise ist somit erfüllt. Bei der Analyse der Fallstudie aus Abschnitt 3.1 wurde jedoch ein Antipattern identifiziert. Das identifizierte Antipattern wurde jedoch korrekt identifiziert und hätte im Rahmen der Erstellung der Fallstudie anders konzeptioniert werden müssen.

Diese Testfälle garantieren jedoch nicht, dass jedes der überprüften Antipattern sicher gefunden wird. Verschiedene Gründe können dazu führen, dass ein Antipattern nicht identifiziert wird. Ein Grund kann sein, dass die Analyse dieses Antipatterns einfach nicht vorgesehen ist. Ein anderer Grund kann sein, dass das Antipattern zwar vorliegt, es aber so modelliert ist, dass die vorhandenen Logiken das Antipattern nicht identifizieren können. Eine große Gefahr liegt dabei in den Analysen, die auf einem Aspekt basieren. Der verwendete Aspekt *ArchitecturePattern* ist in vorhandenen Modellen bisher nicht verwendet, da er erst in dieser Arbeit eingeführt wurde. In Abbildung 5.1 ist eine Gruppierung der Analysekategorien gegeben. Aus dieser geht hervor, dass der Großteil der Analyselogiken auf eben diesem Aspekt beruht.

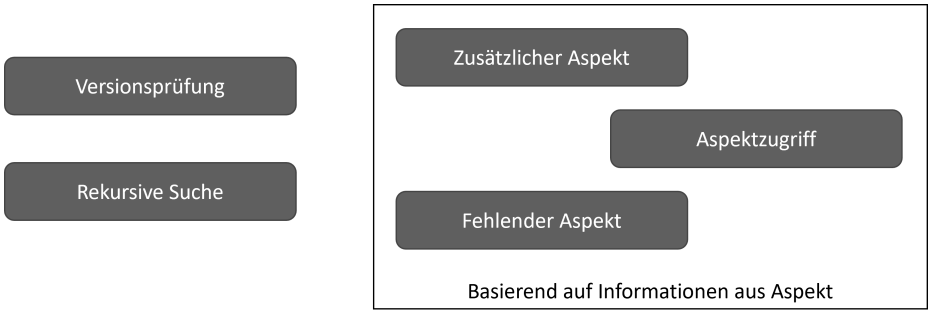

Abb. 5.1. Gruppierung von Analysekategorien aus Tabelle 4.2 (Eigene Darstellung)

6 Related Work

Zu dem Thema Antipattern und deren Analyse gibt es bereits eine Reihe von Betrachtungen, deren Ergebnisse auch in diese Arbeit eingeflossen sind.
So beschäftigten sich Taibi und Lenarduzzi[TL18] bereits 2018 mit dem Thema Antipattern bei Microservices. Viele der in Abschnitt 3.3 betrachteten Microservices wurden dieser Veröffentlichung entnommen. Die Ergebnisse des Artikels beruhen auf der Befragung verschiedener Entwickelnder. Dabei entstanden ist eine Liste verschiedener Antipattern aus diverser Literatur.

Palma und Mohay[PM15] hatten sich bereits im Jahr 2015 mit Antipattern von Services auseinandergesetzt. Dabei lag der Fokus noch nicht auf dem Begriff *Microservice*. Jedoch decken sich viele der identifizierten Antipattern mit denen bei Microservices. So werden beispielsweise zu kleine oder große Services als ein Beispiel aufgeführt.

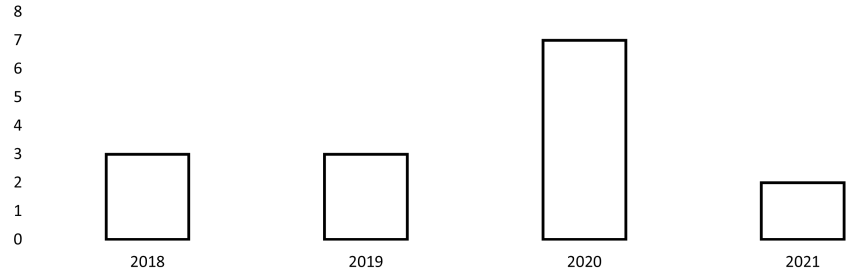

Abb. 6.1. Veröffentlichte Paper zum Themenkomplex Microservice-Antipattern nach [POF22]

Einen guten Überblick über die Entwicklungen zu dem Thema bieten Pinheiro, Oliveria und Figueiredo[POF22]. Diese beschäftigten sich unter anderem mit

dem Themen wann und in welcher Art und Weise sich mit Microservice Smells beschäftigt wurde. Abbildung 6.1 stellt die identifizierten veröffentlichten Paper der letzten vier Jahre dar. Die verwendete Methodik der Paper aus diesem Zeitraum wird in Abbildung 6.2 dargestellt.

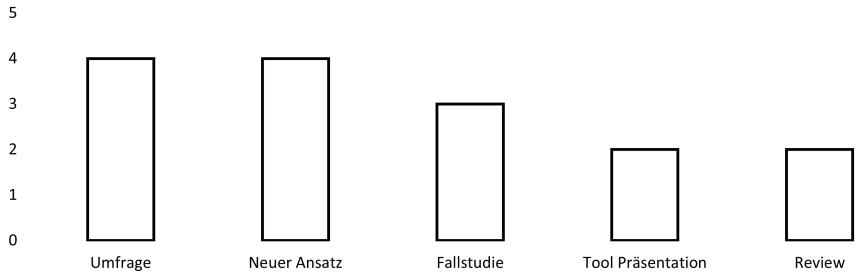

Abb. 6.2. Art der Veröffentlichungen zu Microservice-Antipattern nach [POF22] (Eigene Darstellung)

Weiterhin wird von Pinheiro, Oliveria und Figueiredo[POF22] eine Übersicht aller in den aufgeführten Papern identifizierten Antipattern aufgelistet und kategorisiert. Viele der angegeben Antipattern decken sich mit den in dieser Arbeit verwendeten. Jedoch gibt es auch weitere. Insgesamt wurden $^2/_3$ dieser Antipattern in dieser Arbeit exemplarisch betrachtet.

Neben der Betrachtung von Antipattern von Microservices wurde auch die statische Codeanalyse von domänenspezifischen Sprachen betrachtet. Einen Überblick dazu bieten Ruiz-Rube u. a.[Rui+20]. So wird ein Ansatz dazu beschrieben. Wie bereits in Abschnitt 4.1 dargestellt, findet eine Umsetzung mit LEMMA statt.

7 Zusammenfassung und Fazit

Ziel dieser Arbeit war die Konzeption und Umsetzung eines Ansatzes zur Identifikaton von Antipattern in Microservice-Architekturen. Erkannt werden diese Modelle anhand der DSLs von LEMMA. So findet eine Architekturplanung für eine Microservice mit den LEMMA-DSLs statt. Nach der Erstellung dieser Modelle wird eine statische Code-Analyse durchgeführt. Die statische Code-Analyse wurde im Rahmen dieser Arbeit um die Analyse einiger identifizierter Antipattern erweitert. Dadurch wird bei der Durchführung einer statischen Analyse neben den bereits vorhandenen Analysen weiterhin auf Antipattern geprüft und mögliche Antipattern auf der Konsole ausgegeben.

Für eine bessere Wart- und Erweiterbarkeit wurde sich dazu entschieden, die zu analysierenden Antipattern in parametrisierbare Kategorien einzuteilen. So sind Logiken einfach wiederverwendbar. Weiterhin wurde die Funktion dieser Logiken in verschiedenen Tests sichergestellt.

Im Rahmen der Arbeit wurden 14 Antipattern betrachtet. Nach Pinheiro, Oliveria und Figueiredo[POF22] findet man in der Literatur der letzten Jahre insgesamt 21 Antipattern.
Es wurde also nur eine Teilmenge der Antipattern überhaupt gesichtet. Aufgrund des Umfangs und Ansatzes der Arbeit wurden jedoch auch nicht alle der 14 Antipattern analysiert. Neun Antipattern wurden letztendlich analysiert. Die anderen fünf betrachteten wurden aufgrund unklarer Identifikation nicht weiter betrachtet. Eine sinnvolle Identifikation dieser Antipattern wurde in dieser Arbeit nicht betrachtet.

Im Rahmen der Analyse der Antipattern hängt eine Identifikation in vielen Fällen auch von der Verwendung eines Aspekts statt. Dabei handelt es sich um ein Feature zur Parametrisierung der Modelle. Über einen Aspekt kann im Kontext dieser Arbeit ein Service beispielsweise als *API-Gateway* oder als eine andere Technologie markiert werden.
Ein Nachteil an der Verwendung der Aspekte ist, dass diese nicht allgemeingültig sind. So werden diese Aspekte immer in den verwendeten Technologiemodellen hinterlegt. Dadurch sind die Antipattern die anhand dieser Metriken erkennbar sind, nur bei Projekten anwendbar, welche die Aspekte entsprechend gepflegt haben. Antipattern, welche nicht vom gesetzten Aspekt abhängen funktionieren jedoch ohne Anpassung der Modelle.

Das Ziel der Arbeit, einen Ansatz zur Identifikation zu Antipattern zu finden, war somit erfolgreich. So sind in der aktuellen Umsetzung zum Teil Anpassungen der Modelle notwendig. Auch sind nicht alle in der Literatur identifizierten Antipattern betrachtet. Das Ziel der Arbeit war jedoch, einen Ansatz mit Verifizierung dessen zu erarbeiten. Das Ziel ist somit erfüllt.

8 Ausblick

Wie bereits in Kapitel 7 beschrieben war, gibt es einige Schwachstellen an der Umsetzung, welche in Zukunft behoben werden können. Jedoch sind auch andere Anpassungen und Erweiterungen in Zukunft möglich.

Ein zentrales Problem liegt darin, dass die Modelle und die darin verwendeten Technologien extra Anpassung der Modelle benötigen. Hier würde eine Standardisierung in LEMMA Sinn ergeben. So wäre die hinterlegte Technologieart beziehungsweise das hinterlegte Architekturpattern direkt in der Modellierungsprache definierbar. Dies würde beispielsweise bei der Definition der Technologien sinnvoll sein, da diese häufig bereits festlegen, um welchen Typ es sich handelt. So ist beispielsweise bei der Technologie *PostgreSQL*[10] klar, dass es sich um eine Datenbank handelt, bei *Eureka*[11] ist erkennbar, dass eine Service-Discovery vorliegt und bei *Traefik*[12] ein API-Gateway naheliegend.
Alternativ wäre auch eine Festlegung solcher Technologien in LEMMA denkbar. Das also die Technologien gar nicht aus externen Quellen importiert werden müssen, sondern im Standardpaket oder über Plug-Ins ladbar sind. Dies würde zu einer größeren Standardisierung innerhalb der Modelle führen.

Weiterhin ergibt eine Erweiterung der Analyse anderer Antipattern Sinn. So können Metriken für die fünf betrachteten, jedoch nicht umgesetzten Antipattern ausgearbeitet werden. Mit diesen Metriken wäre eine Analyse der Antipattern möglich. So könnte zum Beispiel ausgewertet werden, ob in einer Microservice-Architektur „zu viele“ Technologien verwendet werden.
Aber auch eine Erweiterung um noch nicht betrachtete Antipattern kann empfohlen werden. Möglicherweise lassen sich diese in den bestehenden Kategorien einpflegen und würden somit für einen geringen zusätzlichen Aufwand sorgen. Falls eine Einordnung in die bestehende Kategorisierung nicht möglich ist, lässt sich die bestehende Struktur jedoch ebenfalls um weitere Analysestrategien erweitern.

Auch sinnvoll kann eine Konfigurationsmöglichkeit sein. So können nur relevante Antipattern betrachtet werden. Dies ist sinnvoll, falls zum Beispiel in den Modellen die Monitoring-Lösung nicht betrachtet wird.
In diesem Kontext ist auch die Möglichkeit denkbar, einzelne Antipattern in den Modellen als nicht relevant zu markieren. In einem Fall, wo ein mögliches Antipattern erkannt wurde, es jedoch bewertet wurde und entschieden wurde, dass es in diesem Fall bewusst so umgesetzt sein soll, erscheint es nicht sinnvoll dieses Antipattern wiederholt bei der Analyse auszugeben.

[10] www.postgresql.org
[11] github.com/Netflix/eureka
[12] traefik.io

Literaturverzeichnis

[Ama21] Amazon Web Services. *API gateway pattern*. 11/01/2021. URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-integrating-microservices/api-gateway-pattern.html (besucht am 02. 04. 2022).

[AZ11] Dionysis Athanasopoulos und Apostolos V. Zarras. „Fine-Grained Metrics of Cohesion Lack for Service Interfaces". In: *2011 IEEE International Conference on Web Services (ICWS 2011). Washington, DC, USA, 4-9 July 2011*. 2011 IEEE International Conference on Web Services (ICWS) (Washington, DC, USA,). Hrsg. von Ian Foster. ICWS und IEEE International Conference on Web Services. Piscataway, NJ: IEEE, 2011, S. 588–595. ISBN: 978-1-4577-0842-8. DOI: 10.1109/ICWS.2011.27.

[Béz05] Jean Bézivin. „On the unification power of models". En;en. In: *Software & Systems Modeling* 4.2 (2005). PII: 79, S. 171–188. ISSN: 1619-1374. DOI: 10.1007/s10270-005-0079-0.

[CBP16] Nikhil Chaudhari, Robin Singh Bhadoria und Siddharth Prasad. „Information Handling and Processing Using Enterprise Service Bus in Service-Oriented Architecture System". In: *2016 8th International Conference on Computational Intelligence and Communication Networks (CICN 2016). Tehri, India, 23-25 December 2016*. 2016 8th International Conference on Computational Intelligence and Communication Networks (CICN) (Tehri, India,). Piscataway, NJ: IEEE, 2016, S. 418–421. ISBN: 978-1-5090-1144-5. DOI: 10.1109/CICN.2016.88.

[Com+17] Benoit Combemale u. a. „What's a Modeling Language?" In: *Engineering Modeling Languages. Turning Domain Knowledge into Tools*. 2017, S. 20–44. ISBN: 978-1-4665-8373-3.

[Eng+18] Thomas Engel u. a. „Evaluation of Microservice Architectures: A Metric and Tool-Based Approach". In: *Information systems in the big data era. CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings / edited by Jan Mendling, Haralambos Mouratidis*. Hrsg. von Jan Mendling und Haralambos. editor Mouratidis. Bd. 317. Lecture Notes in Business Information Processing, 1865-1348 317. Cham: Springer, 2018, S. 74–89. ISBN: 978-3-319-92900-2. DOI: 10.1007/978-3-319-92901-9_8.

[Eva04] Eric Evans. *Domain-driven design. Tackling complexity in the heart of software / Eric Evans*. Boston und London: Addison-Wesley, 2004. ISBN: 0321125215.

[Fam15] Bob Familiar. „What Is a Microservice?" In: *Microservices, IoT, and Azure. Leveraging DevOps and microservice architecture to deliver SaaS solutions / Bob Familiar*. Hrsg. von Bob Familiar. Berkeley, CA: Apress, 2015, S. 9–19. ISBN: 978-1-4842-1276-9. DOI: 10.1007/978-1-4842-1275-2_2.

[Fow06] Martin Fowler. *CodeSmell*. 9/02/2006. URL: https://www.martinfowler.com/bliki/CodeSmell.html (besucht am 02. 04. 2022).

[Fow99a] Martin Fowler. „Chapter 11: Dealing with Generalization“. In: *Refactoring. Improving the design of existing code / Martin Fowler ; with contributions by Kent Beck ... [et al.]* Hrsg. von Martin Fowler. The Addison-Wesley object technology series. Reading, Mass. und Harlow: Addison-Wesley, 1999, S. 319–358. ISBN: 978-0-201-48567-7.

[Fow99b] Martin Fowler, Hrsg. *Refactoring. Improving the design of existing code / Martin Fowler ; with contributions by Kent Beck ... [et al.]* The Addison-Wesley object technology series. Reading, Mass. und Harlow: Addison-Wesley, 1999. ISBN: 978-0-201-48567-7.

[FRS15a] Eric Freeman, Elisabeth Robson und Kathy Sierra. „Anpassungsfähigkeit beweisen. das Adapter- und Facade-Muster“. In: *Entwurfsmuster von Kopf bis Fuss*. Hrsg. von Eric Freeman, Elisabeth Robson und Kathy Sierra. 2. Aufl., aktuell zu Java 8. Heidelberg: O'Reilly, 2015. ISBN: 978-3-95561-986-2.

[FRS15b] Eric Freeman, Elisabeth Robson und Kathy Sierra. „Einführung“. In: *Entwurfsmuster von Kopf bis Fuss*. Hrsg. von Eric Freeman, Elisabeth Robson und Kathy Sierra. 2. Aufl., aktuell zu Java 8. Heidelberg: O'Reilly, 2015. ISBN: 978-3-95561-986-2.

[FRS15c] Eric Freeman, Elisabeth Robson und Kathy Sierra. „Entwurfsmuster in der realen Welt. besser leben mit Mustern“. In: *Entwurfsmuster von Kopf bis Fuss*. Hrsg. von Eric Freeman, Elisabeth Robson und Kathy Sierra. 2. Aufl., aktuell zu Java 8. Heidelberg: O'Reilly, 2015, S. 583–616. ISBN: 978-3-95561-986-2.

[FRS15d] Eric Freeman, Elisabeth Robson und Kathy Sierra, Hrsg. *Entwurfsmuster von Kopf bis Fuss*. 2. Aufl., aktuell zu Java 8. Heidelberg: O'Reilly, 2015. 653 S. ISBN: 978-3-95561-986-2.

[GDD09] Dragan Gaševic, Dragan Djuric und Vladan Devedžic. „Model Driven Engineering“. en. In: *Model Driven Engineering and Ontology Development*. Springer, Berlin, Heidelberg, 2009, S. 125–155. DOI: 10.1007/978-3-642-00282-3_4.

[Git21] GitLab. „A maturing DevSecOps landscape. 2021 Global Survey results“. In: (2021).

[Hau+17] Florian Haupt u. a. „A Framework for the Structural Analysis of REST APIs“. In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017 IEEE International Conference on Software Architecture (ICSA) (Gothenburg, Sweden,). [Place of publication not identified]: IEEE, 2017, S. 55–58. ISBN: 978-1-5090-5729-0. DOI: 10.1109/ICSA.2017.40.

[HCA09] Mamoun Hirzalla, Jane Cleland-Huang und Ali Arsanjani. „A Metrics Suite for Evaluating Flexibility and Complexity in Service Oriented Architectures“. en. In: International Conference on Service-Oriented Computing. Springer, Berlin, Heidelberg, 2009, S. 41–52. DOI: 10.1007/978-3-642-01247-1_5.

[IBM21] IBM. *Microservices in the enterprise. Real benefits, worth the challenges*. 2021. URL: https://www.ibm.com/account/reg/us-en/signup?formid=urx-49970 (besucht am 26. 03. 2022).

[Net21] Netflix Technology Blog. „The Netflix Cosmos Platform“. In: *Netflix TechBlog* (1. Apr. 2021).

[New21a] Sam Newman. „Implementing Microservice Communication“. In: *Building microservices. Designing fine-grained systems / Sam Newman*. Hrsg. von Sam Newman. Second edition. Beijing: O'Reilly, 2021, S. 121–174. ISBN: 978-1-492-03402-5.

[New21b] Sam Newman. „What are Microservices?“ In: *Building microservices. Designing fine-grained systems / Sam Newman*. Hrsg. von Sam Newman. Second edition. Beijing: O'Reilly, 2021, S. 3–34. ISBN: 978-1-492-03402-5.

[PM15] Francis Palma und Naouel Mohay. „A study on the taxonomy of service antipatterns“. In: *2015 IEEE 2nd Workshop on Patterns Promotion and Anti-Patterns Prevention (PPAP 2015). Montréal, Québec, Canada, 2 March 2015*. 2015 IEEE 2nd Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP) (Montreal, QC, Canada,). Hrsg. von IEEE Workshop on Patterns Promotion Prevention und Anti-Patterns. Hrsg. von Surafel Lemma Abebe. IEEE Workshop on Patterns Promotion and Anti-Patterns Prevention u. a. Piscataway, NJ: IEEE, 2015, S. 5–8. ISBN: 978-1-4673-6920-6. DOI: 10.1109/PPAP.2015.7076848.

[POF22] Denis Pinheiro, Johnatan Oliveria und Eduardo Figueiredo. „Microservice Smells and Automated Detection Tools: A Systematic Literature Review“. In: *Microservices 2022*. International Conference on Microservices (Paris). 2022.

[Pre21] Tom Preston-Werner. *Semantic Versioning 2.0.0*. 14/08/2021. URL: https://semver.org/ (besucht am 02. 04. 2022).

[Rad22] Florian Rademacher. *LEMMA Static Analyzer*. 2022. URL: https://github.com/SeelabFhdo/lemma/tree/main/de.fhdo.lemma.analyzer (besucht am 01. 05. 2022).

[RS21] Florian Rademacher und Jonas Sorgalla. *LEMMA*. 10/18/2021. URL: https://github.com/SeelabFhdo/lemma/blob/main/README.md (besucht am 02. 04. 2022).

[Rui+20] Iván Ruiz-Rube u. a. „Applying static code analysis for domain-specific languages“. En;en. In: *Software & Systems Modeling* 19.1 (2020). PII: 729, S. 95–110. ISSN: 1619-1374. DOI: 10.1007/s10270-019-00729-w.

[Sel03] B. Selic. „The pragmatics of model-driven development“. In: *IEEE Software* 20.5 (2003), S. 19–25. ISSN: 0740-7459. DOI: 10.1109/MS.2003.1231146.

[TL18] Davide Taibi und Valentina Lenarduzzi. „On the Definition of Microservice Bad Smells“. In: *IEEE Software* 35.3 (2018), S. 56–62. ISSN: 0740-7459. DOI: 10.1109/MS.2018.2141031.

[TLP20] Davide Taibi, Valentina Lenarduzzi und Claus Pahl. „Microservices Anti-patterns: A Taxonomy“. In: *Microservices*. Hrsg. von Antonio Bucchiarone u. a. Cham: Springer International Publishing, 2020, S. 111–128. ISBN: 978-3-030-31645-7. DOI: 10.1007/978-3-030-31646-4_5.

[Wei09] Michael Weiss. „XML Metadata Interchange“. en. In: *Encyclopedia of Database Systems*. Springer, Boston, MA, 2009, S. 3597. DOI: 10.1007/978-0-387-39940-9_902.

[Wol18] Eberhard Wolff. „Microservices: Was, warum und warum vielleicht nicht?“ In: *Microservices. Grundlagen flexibler Softwarearchitekturen*. Hrsg. von Eberhard Wolff. 2., aktualisierte Auflage. Heidelberg: dpunkt.verlag, 2018, S. 29–96. ISBN: 9783960884132.

[WR22] Philip Wizenty und Florian Rademacher. „Towards Viewpoint-Based Microservice Architecture Reconstruction“. In: *Microservices 2022*. International Conference on Microservices (Paris). 2022.

A VODY-Modell

```
import technology from "../technology/shared.technology" as
    shared

@technology(shared)
@endpoints(shared::_protocols.rest: "/video";)
functional microservice vody.VideoService version v_1{
  interface foo{
    foo();
  }
}

@technology(shared)
@endpoints(shared::_protocols.rest: "/audio";)
functional microservice vody.AudioService version v_1{
  interface foo{
    foo();
  }
}

@technology(shared)
@endpoints(shared::_protocols.rest: "/auth";)
functional microservice vody.AuthService version v_1{
  required microservices{
    AccountService
  }
  interface foo{
    foo();
  }
}

@technology(shared)
@endpoints(shared::_protocols.rest: "/account";)
functional microservice vody.AccountService version v_1{
  interface foo{
    foo();
  }
}

@technology(shared)
@endpoints(shared::_protocols.rest: "/query";)
functional microservice vody.QueryService version v_1{
  interface foo{
    foo();
  }
}
```

Listing A.1. vody.services

```
import technology from "generic.technology" as generic
import technology from "../technology/shared.technology" as shared
import microservices from "vody.services" as vs
import technology from "../technology/container_base.technology" as container_base


@technology(container_base)
container VideoContainer
  deployment technology container_base::_deployment.Kubernetes
    with operation environment "golang"
    deploys vs::v_1.vody.VideoService
    depends on nodes API_Gateway, MonitoringAndLogging, ServiceDiscovery, VideoDatabase
{}

@technology(container_base)
container AudioContainer
  deployment technology container_base::_deployment.Kubernetes
    with operation environment "golang"
    deploys vs::v_1.vody.AudioService
    depends on nodes API_Gateway, MonitoringAndLogging, ServiceDiscovery, AudioDatabase
{}

@technology(container_base)
container AuthContainer
  deployment technology container_base::_deployment.Kubernetes
    with operation environment "golang"
    deploys vs::v_1.vody.AuthService
    depends on nodes API_Gateway, MonitoringAndLogging, ServiceDiscovery, AccountContainer
{}

@technology(container_base)
container AccountContainer
  deployment technology container_base::_deployment.Kubernetes
    with operation environment "golang"
    deploys vs::v_1.vody.AccountService
    depends on nodes API_Gateway, MonitoringAndLogging, ServiceDiscovery, AccountDatabase
{}

@technology(container_base)
container QueryContainer
```

```
41   deployment technology container_base::_deployment.
       Kubernetes
42     with operation environment "golang"
43     deploys vs::v_1.vody.QueryService
44     depends on nodes API_Gateway, MonitoringAndLogging,
       ServiceDiscovery, SearchDatabase
45 {}
46
47 @technology(generic)
48 API_Gateway is generic::_infrastructure.generic
49   depends on nodes MonitoringAndLogging, ServiceDiscovery{
50   aspects{
51     generic::_aspects.ArchitecturePattern(name="API-Gateway")
       ;
52   }
53 }
54
55 @technology(generic)
56 ServiceDiscovery is generic::_infrastructure.generic
57 depends on nodes MonitoringAndLogging{
58   aspects{
59     generic::_aspects.ArchitecturePattern(name="
       ServiceDiscovery");
60   }
61 }
62
63 @technology(generic)
64 MonitoringAndLogging is generic::_infrastructure.generic{
65   aspects{
66     generic::_aspects.ArchitecturePattern(name="Monitoring");
67     generic::_aspects.ArchitecturePattern(name="LogServer");
68   }
69 }
70
71 @technology(generic)
72 VideoDatabase is generic::_infrastructure.generic
73   depends on nodes MonitoringAndLogging, ServiceDiscovery,
       API_Gateway{
74   aspects{
75     generic::_aspects.ArchitecturePattern(name="Database");
76   }
77 }
78
79 @technology(generic)
80 AudioDatabase is generic::_infrastructure.generic
81   depends on nodes MonitoringAndLogging, API_Gateway,
       ServiceDiscovery{
82   aspects{
83     generic::_aspects.ArchitecturePattern(name="Database");
84   }
```

```
}

@technology(generic)
AccountDatabase is generic::_infrastructure.generic
  depends on nodes MonitoringAndLogging, API_Gateway,
    ServiceDiscovery{
  aspects{
    generic::_aspects.ArchitecturePattern(name="Database");
  }
}

@technology(generic)
SearchDatabase is generic::_infrastructure.generic
  depends on nodes MonitoringAndLogging, API_Gateway,
    ServiceDiscovery{
  aspects{
    generic::_aspects.ArchitecturePattern(name="Database");
  }
}
```

Listing A.2. vody.operation

B Anhang: Quellcode

In diesem Anhang wird der wesentliche entstandene Quellcode aufgelistet. Dieser ist auch auf GitHub unter github.com/Kikkirej/lemma/tree/antipattern-fle zu finden.

B.1 de.fhdo.lemma.analyzer.lib

de.fhdo.lemma.analyzer.lib.analyzers.AntipatternServiceAnalyzerI

```
package de.fhdo.lemma.analyzer.lib.analyzers

import de.fhdo.lemma.analyzer.lib.AnalyzerI
import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    Antipattern

interface AntipatternServiceAnalyzerI : AnalyzerI {
    fun checkExistingAntipattern(): Collection<Antipattern>
}
```

de.fhdo.lemma.analyzer.lib.analyzers.AntipatternOperationAnalyzerI

```
package de.fhdo.lemma.analyzer.lib.analyzers

import de.fhdo.lemma.analyzer.lib.AnalyzerI
import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    Antipattern

interface AntipatternOperationAnalyzerI : AnalyzerI{
```

```
    fun checkExistingAntipattern(): Collection<Antipattern>
}
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.Antipattern

```
package de.fhdo.lemma.analyzer.lib.impl.antipattern

enum class AntipatternType(val displayName: String) {
    API_VERSIONING("API-Versioning"),
    CYCLIC_DEPENDENCY("Cyclic Dependencies"),
    USAGE_ESB("Usage Enterprise Service Bus"),
    HARDCODED_ENDPOINTS("Hardcoded Endpoints"),
    NO_API_GATEWAY("No API-Gateway"),
    SHARED_PERSISTENCE("Shared Persistence"),
    LOCAL_LOGGING("Local Logging"),
    NO_MONITORING("Missing Monitoring"),
}

data class Antipattern(val type: AntipatternType, val message
    : String)
```

**de.fhdo.lemma.analyzer.lib.impl.antipattern.
AntipatternOperationAnalyzer**

```
package de.fhdo.lemma.analyzer.lib.impl.antipattern

import de.fhdo.lemma.analyzer.lib.analyzers.
    AntipatternOperationAnalyzerI
import de.fhdo.lemma.analyzer.lib.impl.
    AbstractSingleModelTypeAnalyzer
import de.fhdo.lemma.analyzer.lib.impl.Cache
import de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies
    .operation.*
import de.fhdo.lemma.operation.intermediate.
    IntermediateOperationModel
import de.fhdo.lemma.operation.intermediate.
    IntermediateOperationNode

internal class AntipatternOperationAnalyzer :
    AbstractSingleModelTypeAnalyzer<
    IntermediateOperationModel>(IntermediateOperationModel::
    class.java), AntipatternOperationAnalyzerI {
    private var strategies = setOf(
        CircleStrategy(),
        WrongAspectStrategy("ESB", AntipatternType.USAGE_ESB,
     "Using an 'Enterprise Service Bus' can make a solution
    too complex"),
        MissingAspectStrategy("ServiceDiscovery",
    AntipatternType.HARDCODED_ENDPOINTS, "A service discovery
     should be used"),
        AspectUsageStrategy("Database", AntipatternType.
    SHARED_PERSISTENCE, "Only one Service should access a
```

```
    data storage. In case of data separation, like different
    tables this can be ignored"),
16      MissingAspectStrategy("LogServer", AntipatternType.
    LOCAL_LOGGING, "Log-Data should be stored central"),
17      MissingAspectStrategy("Monitoring", AntipatternType.
    NO_MONITORING, "Services should be monitored central"),
18      MissingAspectStrategy("API-Gateway", AntipatternType.
    NO_API_GATEWAY, "Although the service has endpoints no
    API-Gateway is connected", true)
19  )
20
21  override fun checkExistingAntipattern(): Collection<
    Antipattern>{
22      val allInfrastuctureNodes = Cache.
    allInfrastuctureNodes()
23      val allContainer = Cache.allContainer()
24      val operationNodes = mutableListOf<
    IntermediateOperationNode>()
25      operationNodes.addAll(allContainer)
26      operationNodes.addAll(allInfrastuctureNodes)
27      val result = mutableListOf<Antipattern>()
28      strategies.forEach { result.addAll(it.
    analyzeOperationNodes(operationNodes)) }
29      return result
30  }
31 }
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.AntipatternServiceAnalyzer

```
1  package de.fhdo.lemma.analyzer.lib.impl.antipattern
2
3  import de.fhdo.lemma.analyzer.lib.analyzers.
       AntipatternServiceAnalyzerI
4  import de.fhdo.lemma.analyzer.lib.impl.
       AbstractSingleModelTypeAnalyzer
5  import de.fhdo.lemma.analyzer.lib.impl.Cache
6  import de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies
       .service.APIVersionStrategy
7  import de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies
       .service.AntipatternServiceAnalyzerStrategy
8  import de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies
       .service.CircleStrategy
9  import de.fhdo.lemma.service.intermediate.
       IntermediateServiceModel
10
11 internal class AntipatternServiceAnalyzer :
       AbstractSingleModelTypeAnalyzer<IntermediateServiceModel
       >(
12      IntermediateServiceModel::class.java),
       AntipatternServiceAnalyzerI {
```

```
13
14      private var strategies = listOf<
        AntipatternServiceAnalyzerStrategy>(CircleStrategy(),
        APIVersionStrategy())
15
16      override fun checkExistingAntipattern(): Collection<
        Antipattern> {
17          val microservices = Cache.allMicroservices()
18          val result = mutableListOf<Antipattern>()
19          strategies.forEach { result.addAll(it.
        analyzeOperationNodes(microservices)) }
20          return result
21      }
22  }
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.AspectExtractor

```
1   package de.fhdo.lemma.analyzer.lib.impl.antipattern
2
3   import de.fhdo.lemma.data.intermediate.
        IntermediateImportedAspect
4   import org.eclipse.emf.common.util.EList
5
6   /**
7    * Encapsulated logic for information extraction in aspects
8    * Created in context of antipattern analysis
9    */
10  object AspectExtractor {
11      fun isArchitecturePatternDefined(aspects: EList<
        IntermediateImportedAspect>, architectureName: String):
        Boolean {
12          val names = aspects.filter { it.name.equals("
        ArchitecturePattern") }.flatMap { it.propertyValues }
13              .filter { it.property.name.equals("name") }
14          for (name in names) {
15              if (architectureName.equals(name.value)) {
16                  return true
17              }
18          }
19          return false
20      }
21  }
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.operation.AntipatternOperationAnalyzerStrategy

```
1   package de.fhdo.lemma.analyzer.lib.impl.antipattern.
        strategies.operation
2
3   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
        Antipattern
```

```
import de.fhdo.lemma.operation.intermediate.
    IntermediateOperationNode

interface AntipatternOperationAnalyzerStrategy {
    fun analyzeOperationNodes(allOperationNodes: Iterable<
    IntermediateOperationNode>): Collection<Antipattern>
}
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.operation.AspectUsageStrategy

```
package de.fhdo.lemma.analyzer.lib.impl.antipattern.
    strategies.operation

import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    Antipattern
import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    AntipatternType
import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    AspectExtractor
import de.fhdo.lemma.operation.intermediate.
    IntermediateOperationNode

class AspectUsageStrategy(val architecturePatternName: String
    , val antipatternType: AntipatternType, val description:
    String) : AntipatternOperationAnalyzerStrategy {
    override fun analyzeOperationNodes(allOperationNodes:
    Iterable<IntermediateOperationNode>): Collection<
    Antipattern> {
        val antipatterns = mutableListOf<Antipattern>()
        for (node in allOperationNodes) {
            if(AspectExtractor.isArchitecturePatternDefined(
    node.aspects,architecturePatternName)){
                val tooManyDependent = getConnectedElements(
    node, allOperationNodes)
                if(tooManyDependent){
                    antipatterns.add(Antipattern(
    antipatternType, "$description. Too many Nodes are
    dependend on ${node.name} with ArchitecturePattern
    $architecturePatternName."))
                }
            }
        }
        return antipatterns
    }

    private fun getConnectedElements(node:
    IntermediateOperationNode, allOperationNodes: Iterable<
    IntermediateOperationNode>):Boolean {
        var count=node.usedByNodes.size;
        for (operationNode in allOperationNodes) {
```

```
            val dependonnodes = operationNode.dependsOnNodes
            for (dependonnode in dependonnodes) {
                if(dependonnode.name.equals(node.name)){
                    count++
                }
            }
        }
        val dependendOnMoreThan1 = count > 1
        return dependendOnMoreThan1

    }
}
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.operation.CircleStrategy

```
package de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.operation

import de.fhdo.lemma.analyzer.lib.impl.antipattern.Antipattern
import de.fhdo.lemma.analyzer.lib.impl.antipattern.AntipatternType
import de.fhdo.lemma.operation.intermediate.IntermediateOperationNode

class CircleStrategy : AntipatternOperationAnalyzerStrategy {
    override fun analyzeOperationNodes(allOperationNodes: Iterable<IntermediateOperationNode>): Collection<Antipattern> {
        val operationNodeMap = getOperationNodeMap(allOperationNodes)
        val setOfOperationNodeLists = mutableSetOf<List<String>>() //Hier sind die Loops drin
        for (key in operationNodeMap.keys) {
            recursiveCall(key, key, listOf(key), setOfOperationNodeLists, mutableSetOf(),operationNodeMap)
        }
        val result = mutableSetOf<Antipattern>()
        setOfOperationNodeLists.forEach{result.add(Antipattern(AntipatternType.CYCLIC_DEPENDENCY, "A cyclic dependency was detected. The cyclic dependency consists of $it"))}
        return result
    }

    private fun recursiveCall(
        parentKey: String,
        checkedKey: String,
        viewedElements: List<String>,
        setOfLoopLists: MutableSet<List<String>>,
```

```
24          alreadyViewedElements: MutableSet<String>,
25          operationNodeMap: MutableMap<String,
    IntermediateOperationNode>
26      ) {
27          val operationNode = operationNodeMap[checkedKey]
28          for (requiredNode in operationNode!!.dependsOnNodes)
    {//TODO hier kombinieren depends on und requiredBy
29              if(alreadyViewedElements.contains(requiredNode.
    name)){
30                  continue
31              }
32              if(requiredNode.name.equals(parentKey)){
33                  viewedElementsAddToSetIfNoEquivalentInSet(
    setOfLoopLists, viewedElements)
34                  continue
35              }
36              alreadyViewedElements.add(requiredNode.name)
37              val list = viewedElements + listOf(requiredNode.
    name)
38              recursiveCall(parentKey, requiredNode.name, list,
     setOfLoopLists, alreadyViewedElements, operationNodeMap)
39          }
40      }
41
42      private fun viewedElementsAddToSetIfNoEquivalentInSet(
43          setOfLoopLists: MutableSet<List<String>>,
44          viewedElements: List<String>
45      ) {
46          for (existingList in setOfLoopLists) {
47              if(existingList.size != viewedElements.size){
48                  continue
49              }
50              var cont = false
51              for (viewedElement in viewedElements) {
52                  if(!existingList.contains(viewedElement)){
53                      cont=true
54                  }
55              }
56              if(cont){
57                  continue
58              }
59              return
60          }
61          setOfLoopLists.add(viewedElements)
62      }
63
64      private fun getOperationNodeMap(operationNodes: Iterable<
    IntermediateOperationNode>): MutableMap<String,
    IntermediateOperationNode> {
```

```
65          val map = mutableMapOf<String,
    IntermediateOperationNode>()
66          operationNodes.forEach { map[it.name] = it }
67          return map
68      }
69  }
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.operation.MissingAspectStrategy

```
1   package de.fhdo.lemma.analyzer.lib.impl.antipattern.
    strategies.operation
2
3   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    Antipattern
4   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    AntipatternType
5   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    AspectExtractor
6   import de.fhdo.lemma.operation.intermediate.
    IntermediateOperationNode
7   import de.fhdo.lemma.operation.intermediate.
    IntermediateOperationNodeReference
8
9   class MissingAspectStrategy(val architecturePatternName:
    String, val antipatternType: AntipatternType, val
    description: String, val endpointsRelevant: Boolean =
    false) : AntipatternOperationAnalyzerStrategy {
10      override fun analyzeOperationNodes(allOperationNodes:
    Iterable<IntermediateOperationNode>): Collection<
    Antipattern> {
11          val result = mutableSetOf<Antipattern>()
12          for (node in allOperationNodes) {
13              var skip = false
14              if(endpointsRelevant && node.endpoints.size == 0)
    {
15                  continue
16              }
17              if(AspectExtractor.isArchitecturePatternDefined(
    node.aspects, architecturePatternName)){
18                  continue
19              }
20
21              for(dependendOnNodeReference in node.
    dependsOnNodes) {
22                  val analyzedNode = getNodeWithName(
    dependendOnNodeReference, allOperationNodes)
23                  if(AspectExtractor.
    isArchitecturePatternDefined(analyzedNode.aspects,
    architecturePatternName)){
24                      skip = true
```

```
                }
            }
            if (skip)continue
            for(usedByNodeReference in node.usedByNodes) {
                val analyzedNode = getNodeWithName(
    usedByNodeReference, allOperationNodes)
                if(AspectExtractor.
    isArchitecturePatternDefined(analyzedNode.aspects,
    architecturePatternName)){
                    skip=true
                }
            }
            if (skip)continue
            result.add(Antipattern(antipatternType, "
    $description. Object ${node.name} has no connection to a
    Operation-Node with ArchitecturePattern-Aspect and name '
    $architecturePatternName'"))
        }
        return result
    }

    private fun getNodeWithName(
        nodeReference: IntermediateOperationNodeReference,
        allOperationNodes: Iterable<IntermediateOperationNode
    >
    ): IntermediateOperationNode{
        for (node in allOperationNodes) {
            if(node.name.equals(nodeReference.name)){
                return node
            }
        }
        throw RuntimeException("This should never happen")
    }
}
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.operation.WrongAspectStrategy

```
package de.fhdo.lemma.analyzer.lib.impl.antipattern.
    strategies.operation

import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    Antipattern
import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    AntipatternType
import de.fhdo.lemma.analyzer.lib.impl.antipattern.
    AspectExtractor
import de.fhdo.lemma.operation.intermediate.
    IntermediateOperationNode

/**
```

```
 * This aspect describes a usage
 */
class WrongAspectStrategy(val architecturePatternName: String, val antipatternType: AntipatternType, val description: String) : AntipatternOperationAnalyzerStrategy {
    override fun analyzeOperationNodes(allOperationNodes: Iterable<IntermediateOperationNode>): Collection<Antipattern> {
        val result = mutableListOf<Antipattern>()
        for (operationNode in allOperationNodes) {
            if(AspectExtractor.isArchitecturePatternDefined(operationNode.aspects,architecturePatternName)){
                result.add(Antipattern(antipatternType, "${description}. The service with the aspect is ${operationNode.name}"))
            }
        }
        return result
    }
}
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.service.AntipatternServiceAnalyzerStrategy

```
package de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.service

import de.fhdo.lemma.analyzer.lib.impl.antipattern.Antipattern
import de.fhdo.lemma.service.intermediate.IntermediateMicroservice

interface AntipatternServiceAnalyzerStrategy {
    fun analyzeOperationNodes(microservices: Iterable<IntermediateMicroservice>): Collection<Antipattern>
}
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.service.APIVersionStrategy

```
package de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.service

import de.fhdo.lemma.analyzer.lib.impl.antipattern.Antipattern
import de.fhdo.lemma.analyzer.lib.impl.antipattern.AntipatternType
import de.fhdo.lemma.service.intermediate.IntermediateMicroservice

class APIVersionStrategy : AntipatternServiceAnalyzerStrategy {
```

```
8       override fun analyzeOperationNodes(microservices:
        Iterable<IntermediateMicroservice>): Collection<
        Antipattern> {
9           val antipatterns = mutableSetOf<Antipattern>()
10          for (microservice in microservices) {
11              if (microservice.endpoints.size>0 && microservice
        .version==null){
12                  antipatterns.add(Antipattern(AntipatternType.
        API_VERSIONING, "${microservice.name} has open endpoints,
         but no version. API-Changes wouldn't be noticed."))
13              }
14          }
15          return antipatterns
16      }
17  }
```

de.fhdo.lemma.analyzer.lib.impl.antipattern.strategies.service.CircleStrategy

```
1   package de.fhdo.lemma.analyzer.lib.impl.antipattern.
        strategies.service
2
3   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
        Antipattern
4   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
        AntipatternType
5   import de.fhdo.lemma.service.intermediate.
        IntermediateMicroservice
6
7   class CircleStrategy : AntipatternServiceAnalyzerStrategy {
8       override fun analyzeOperationNodes(microservices:
        Iterable<IntermediateMicroservice>): Collection<
        Antipattern> {
9           val microserviceMap = getMicroserviceMap(
        microservices)
10          val setOfServiceLists: MutableSet<List<String>> =
        mutableSetOf()
11          for (key in microserviceMap.keys) {
12              recursiveCall(key, key, listOf(key),
        setOfServiceLists, mutableSetOf<String>(),
        microserviceMap)
13
14          }
15          val result = mutableSetOf<Antipattern>()
16          setOfServiceLists.forEach{result.add(Antipattern(
        AntipatternType.CYCLIC_DEPENDENCY, "A cyclic dependency
        was detected. The cyclic dependency consists of $it."))}
17          return result
18      }
19
20      private fun recursiveCall(
```

```
        parentKey: String,
        checkedKey: String,
        viewedElements: List<String>,
        setOfLoopLists: MutableSet<List<String>>,
        alreadyViewedElements: MutableSet<String>,
        microserviceMap: MutableMap<String,
    IntermediateMicroservice>
    ) {
        val microservice = microserviceMap[checkedKey]
        for (requiredMicroservice in microservice!!.
    requiredMicroservices) {
            if(alreadyViewedElements.contains(
    requiredMicroservice.qualifiedName)){
                continue
            }
            if(requiredMicroservice.qualifiedName.equals(
    parentKey)){
                viewedElementsAddToSetIfNoEquivalentInSet(
    setOfLoopLists,viewedElements)
                continue
            }
            alreadyViewedElements.add(requiredMicroservice.
    qualifiedName)
            val list = viewedElements + listOf(
    requiredMicroservice.qualifiedName)
            recursiveCall(parentKey, requiredMicroservice.
    qualifiedName, list ,setOfLoopLists,
    alreadyViewedElements,microserviceMap)
        }

    }

    private fun viewedElementsAddToSetIfNoEquivalentInSet(
        setOfLoopLists: MutableSet<List<String>>,
        viewedElements: List<String>
    ) {
        for (existingList in setOfLoopLists) {
            if(existingList.size != viewedElements.size){
                continue
            }
            var cont = false
            for (viewedElement in viewedElements) {
                if(!existingList.contains(viewedElement)){
                    cont=true
                }
            }
            if(cont){
                continue
            }
            return
```

```
62          }
63          setOfLoopLists.add(viewedElements)
64      }
65
66      private fun getMicroserviceMap(microservices: Iterable<
        IntermediateMicroservice>): MutableMap<String,
        IntermediateMicroservice> {
67          val map = mutableMapOf<String,
        IntermediateMicroservice>()
68          microservices.forEach { map[it.qualifiedName] = it }
69          return map
70      }
71  }
```

B.2 de.fhdo.lemma.analyzer

de.fhdo.lemma.analyzer.analyses. IntermediateOperationAntipatternAnalysis

```
1   package de.fhdo.lemma.analyzer.analyses
2
3   import de.fhdo.lemma.analyzer.*
4   import de.fhdo.lemma.analyzer.lib.Analyzers
5   import de.fhdo.lemma.analyzer.lib.analyzers.
        AntipatternOperationAnalyzerI
6   import de.fhdo.lemma.analyzer.lib.createAnalyzer
7   import de.fhdo.lemma.analyzer.modules.AbstractAnalysisModule
8   import de.fhdo.lemma.analyzer.modules.ModuleInfo
9   import de.fhdo.lemma.model_processing.annotations.Before
10  import de.fhdo.lemma.model_processing.annotations.
        IntermediateModelValidator
11  import de.fhdo.lemma.operation.intermediate.
        IntermediateOperationModel
12  import de.fhdo.lemma.operation.intermediate.
        IntermediatePackage
13  import org.eclipse.emf.ecore.resource.Resource
14
15
16  @IntermediateModelValidator
17  internal class IntermediateOperationAntipatternAnalysis :
        AbstractAnalysisModule<IntermediateOperationModel>() {
18      companion object{
19          const val MODULE_CLI_OPTION_NAME = "operation-
        antipattern-analysis"
20      }
21
22      private lateinit var analyzer:
        AntipatternOperationAnalyzerI
23
```

```
24      override fun moduleInfo() = ModuleInfo("Operation
    Antipattern Analysis", MODULE_CLI_OPTION_NAME)
25
26      @Before
27      private fun createAnalyzer(resource: Resource) {
28          analyzer = Analyzers.OPERATION_ANTIPATTERN.
    createAnalyzer() as AntipatternOperationAnalyzerI
29      }
30
31      override fun analysis(args: Map<String, String>) {
32          analyzer.setAnalysisModels(loadedModels)
33          val result = analyzer.checkExistingAntipattern()
34          AntipatternAnalysisPrinting.printAntipatternAnalysis(
    result, "Operation")
35      }
36
37      override fun getLanguageNamespace() = IntermediatePackage
    .eNS_URI
38  }
```

**de.fhdo.lemma.analyzer.analyses.
IntermediateServiceAntipatternAnalysis**

```
1   package de.fhdo.lemma.analyzer.analyses
2
3   import de.fhdo.lemma.analyzer.AntipatternAnalysisPrinting
4   import de.fhdo.lemma.analyzer.lib.Analyzers
5   import de.fhdo.lemma.analyzer.lib.analyzers.
        AntipatternServiceAnalyzerI
6   import de.fhdo.lemma.analyzer.lib.createAnalyzer
7   import de.fhdo.lemma.analyzer.modules.AbstractAnalysisModule
8   import de.fhdo.lemma.analyzer.modules.ModuleInfo
9   import de.fhdo.lemma.model_processing.annotations.Before
10  import de.fhdo.lemma.model_processing.annotations.
        IntermediateModelValidator
11  import de.fhdo.lemma.service.intermediate.IntermediatePackage
12  import de.fhdo.lemma.service.intermediate.
        IntermediateServiceModel
13  import org.eclipse.emf.ecore.resource.Resource
14
15  @IntermediateModelValidator
16  internal class IntermediateServiceAntipatternAnalysis :
        AbstractAnalysisModule<IntermediateServiceModel>(){
17      companion object{
18          const val MODULE_CLI_OPTION_NAME = "operation-
    antipattern-analysis"
19      }
20
21      private lateinit var analyzer:
    AntipatternServiceAnalyzerI
22
```

```
23      override fun moduleInfo() = ModuleInfo("Service
    Antipattern Analysis", MODULE_CLI_OPTION_NAME)
24
25      @Before
26      private fun createAnalyzer(resource: Resource) {
27          analyzer = Analyzers.SERVICE_ANTIPATTERN.
    createAnalyzer() as AntipatternServiceAnalyzerI
28      }
29
30      override fun analysis(args: Map<String, String>) {
31          analyzer.setAnalysisModels(loadedModels)
32          val antipattern = analyzer.checkExistingAntipattern()
33          AntipatternAnalysisPrinting.printAntipatternAnalysis(
    antipattern, "Service")
34      }
35
36      override fun getLanguageNamespace() = IntermediatePackage
    .eNS_URI
37  }
```

de.fhdo.lemma.analyzer.AntipatternAnalysisPrinting

```
1   package de.fhdo.lemma.analyzer
2
3   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
        Antipattern
4   import de.fhdo.lemma.analyzer.lib.impl.antipattern.
        AntipatternType
5
6   internal object AntipatternAnalysisPrinting {
7
8       fun printAntipatternAnalysis(antipattern: Collection<
        Antipattern>, analysisType: String){
9           section("Antipattern analysis result - Type:
        $analysisType")
10          if(antipattern.isEmpty()){
11              println("No potential antipatterns identified.
        Everything seems to be clean.")
12              blankLine()
13              return
14          }
15          println("Disclaimer: Identified Antipattern are not
        always bad practices, but can be a hint!")
16          subsection("Statistic - Identified antipatterns")
17          val statistic = getStatistic(antipattern)
18          statistic.keys.forEach {println("${it.displayName}:\t
         ${statistic.get(it)}")}
19          blankLine()
20          subsection("List - All identified antipatterns")
21          antipattern.forEach{ println("${it.type.displayName
    }:\t ${it.message}") }
```

```
22          blankLine()
23      }
24
25      private fun getStatistic(antipatterns: Collection<
        Antipattern>): Map<AntipatternType, Int> {
26          val result = mutableMapOf<AntipatternType, Int>()
27          for (antipattern in antipatterns) {
28              val count = result.get(antipattern.type)
29              if (count == null){
30                  result[antipattern.type] = 1
31              }else{
32                  result[antipattern.type] = count + 1
33              }
34          }
35          return result
36      }
37  }
```

Eidesstattliche Erklärung zur Bachelorthesis

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

$Unterschrift$: $Ort, Datum$: