

# Implementierung von Abfragen zur Smell-Erkennung in Wissensgraphen der Unternehmensarchitektur

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering und Internet Computing**

eingereicht von

**Armond Alexanian, BSc**

Matrikelnummer 00728633

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Professor Dr. Dominik Bork

Mitwirkung: Associate Professor Dr. rer. nat. Simon Hacks, M.Sc. B.Sc.

Wien, 7. März 2025

Armond Alexanian

Dominik Bork



# Implementation of Smell Detection Queries on Enterprise Architecture Knowledge Graphs

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering and Internet Computing**

by

**Armond Alexanian, BSc**

Registration Number 00728633

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Professor Dr. Dominik Bork

Assistance: Associate Professor Dr. rer. nat. Simon Hacks, M.Sc. B.Sc.

Vienna, 7<sup>th</sup> March, 2025

\_\_\_\_\_  
Armond Alexanian

\_\_\_\_\_  
Dominik Bork



# Erklärung zur Verfassung der Arbeit

Armond Alexanian, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 7. März 2025

---

Armond Alexanian



# Acknowledgements

I would like to express my deepest gratitude to my supervisor, *Associate Professor Dr. Dominik Bork*, and my co-supervisor, *Associate Professor Dr. rer. nat. Simon Hacks, M.Sc., B.Sc.*, for their unwavering guidance, insightful advice, and constant support throughout this journey. Their invaluable support and provision of all necessary resources made the completion of this work possible. It was a true pleasure collaborating with them, and I deeply appreciate the freedom and time they allowed me to learn and progress at my own pace.

Finally, I would like to thank my family for their love and endless support, especially my sister. Without her encouragement, I would not have been able to embark on this journey.



# Kurzfassung

Die Unternehmensarchitektur (Enterprise Architecture, EA) umfasst verschiedene Bereiche, darunter die Geschäftsarchitektur, die Architektur von Informationssystemen sowie die Infrastrukturarchitektur. Sie bietet eine ganzheitliche Sicht auf ein Unternehmen. Die Umsetzung von Enterprise Architecture (EA) gestaltet sich oft komplex, insbesondere bei wachsenden Unternehmen, was Herausforderungen im Design und in der Architektur mit sich bringt. In diesem Zusammenhang wurde kürzlich die Metapher der “Enterprise Architecture Debt” eingeführt, um die potenziellen negativen Auswirkungen einer unausgereiften oder nicht optimal umgesetzten EA zu beschreiben. Dieser Begriff leitet sich von der technischen Schuld (Technical Debt, TD) ab, die sowohl technische als auch geschäftliche Aspekte eines Unternehmens umfasst. Da die Definition der EAD keine konkreten Methoden zur Identifikation und Messung solcher Schulden liefert, wurde das Konzept der EA Smells eingeführt.

EA Smells dienen als Indikatoren, um potenzielle Designfehler und Unzulänglichkeiten innerhalb der Unternehmensarchitektur zu erkennen. Ein Katalog mit 63 EA Smells wurde veröffentlicht, der jedoch noch in einem frühen Entwicklungsstadium ist. Für die meisten dieser Smells fehlen derzeit präzise Definitionen und konkrete Ansätze zu ihrer Erkennung.

EA-Modelle sind entscheidend für den Erfolg von Unternehmen und weit verbreitet in Forschung und Industrie. Dennoch bedarf die Analyse von EA-Modellen, sowie die umfassendere Betrachtung der Modellabläufe, mehr Aufmerksamkeit. ArchiMate ist eine De-facto-Standardmodellierungssprache für EA, bietet jedoch keine integrierten Werkzeuge zur Qualitätsanalyse. Die Untersuchung von EA-Smells in ArchiMate-Modellen unterstützt Unternehmensarchitekten bei der Identifikation von Designfehlern und Mängeln im Modell. Die manuelle Analyse großer Modelle, kann jedoch aufgrund ihrer Komplexität eine Herausforderung darstellen, die zu Fehlberechnungen und übersehenen Details führen kann. Im Gegensatz dazu, kann eine automatisierte Analyse Architekten dabei helfen, solche Fehler zu vermeiden. Ein automatisiertes Analysewerkzeug zur Erkennung von EA-Smells kann somit die Identifikation von Modellfehlern erleichtern und das Bewusstsein für die negativen Folgen dieser Mängel schärfen.

Ein neuartiger Ansatz zur Interpretation von EA-Modellen als Knowledge Graph (KG) wurde vorgestellt, der die Modellanalyse durch EA-Abfragen und Graph-Algorithmen erleichtert.

In unserer Arbeit haben wir die EA-Smells analysiert und KG-Abfragen definiert, um diese Smells in EA-Modellen zu identifizieren. Es gelang uns, EA-Abfragen für sechsunddreißig EA-Smells zu entwickeln. Für jeden dieser Smells haben wir den zugrunde liegenden Smell kurz beschrieben und erläutert, wie wir ihn in die EA-Domäne, insbesondere in die Modellierungssprache ArchiMate, übertragen und erkannt haben. Darüber hinaus haben wir den EA-Smell-Katalog um die definierten Abfragen und ergänzende Erweiterungen erweitert, um die Semantik der EA-Smells dort zu klären, wo dies erforderlich war. Schließlich implementierten wir eine Erkennungsplattform zur automatischen Identifizierung von EA-Smells in KGs, die ArchiMate-Modelle repräsentieren. Wir testeten unsere Erkennungsplattform und evaluierten unseren Ansatz anhand einer großen Menge an EA-Modellen. Das Ergebnis war vielversprechend: Unsere Plattform war in der Lage, zahlreiche EA-Smells in realen Modellen zu erkennen. Die Korrektheit unseres Ansatzes haben wir mithilfe von Präzisions- und Recall-Metriken bewertet und bei beiden Metriken einen hohen Wert erzielt, was auf die Präzision unseres Ansatzes hinweist.

# Abstract

Enterprise Architecture (EA) encompasses various domains, such as business architecture, information system architecture, and infrastructure architecture, providing a holistic view of an enterprise. Implementing EA can be complex, especially as enterprises grow, leading to challenges in design and architecture. Recently, the metaphor of Enterprise Architecture Debt (EAD) has been introduced to address the negative consequences of EA. It is derived from Technical Debt (TD), which covers the enterprise's technical and business aspects. Since this definition does not provide a way to identify and measure possible debts, the concept of EA Smell has been introduced. EA Smell is a means to identify the debt in EA. A catalog consisting of sixty-three EA Smells has been published. However, since the catalog is in its early stages of development, most Smells still do not have precise definitions and concrete approaches for detection.

EA models are crucial for enterprise success. They are widely used in research and industry. Yet, more attention needs to be paid to analyzing EA models and addressing the model flaws in a more formal way. ArchiMate is a de facto standard modeling language for EA, but does not offer quality analysis tools. Examining EA Smells in ArchiMate models will assist enterprise architects in identifying design flaws and deficiencies in the model. However, manually analyzing large models can be challenging due to their complexity, which may lead to miscalculations and overlooked details. In contrast, automated analysis can assist architects in avoiding such errors. Therefore, an automated analysis tool for detecting EA Smells can ease the identification of model flaws and raise awareness about those shortcomings and negative consequences.

A novel approach to interpreting EA models as Knowledge Graphs has been introduced, which facilitates model analysis via KG queries and graph algorithms.

Based on the Knowledge Graph (KG) approach, our thesis analyzed the EA Smells and defined KG queries to identify those Smells in EA models. We were able to define KG queries for thirty-six EA Smells. For each of those Smells, we briefly described the underlying Smell and how we transferred and detected it in the EA domain, particularly the ArchiMate modeling language. Additionally, we enhanced the EA Smell catalog by including the defined queries and supplementary extensions to clarify the semantics of EA Smells where necessary. Finally, we implemented a detection platform to automatically identify EA Smells in KG representing ArchiMate models. We tested our detection platform and evaluated our approach on a large set of EA models. The results were

promising. Our platform was able to detect many EA Smells from real-world models. Finally, we evaluated an EA model given by an EA expert containing different EA Smells. We used precision and recall metrics to evaluate the correctness of our approach and achieved a high ratio for both metrics, which indicates that our approach is very accurate.

# Contents

<b>Kurzfassung</b>	<b>ix</b>
<b>Abstract</b>	<b>xi</b>
<b>Contents</b>	<b>xiii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation and Problem Statement . . . . .	1
1.2 Aim of the Work . . . . .	2
1.3 Methodological Approach . . . . .	3
1.4 Thesis Outline . . . . .	5
<b>2 Background</b>	<b>7</b>
2.1 Technical Debt . . . . .	7
2.2 Enterprise Architecture Debt . . . . .	8
2.3 EA Smells . . . . .	9
2.4 ArchiMate . . . . .	10
2.5 Graph-based analysis of EA Models . . . . .	15
2.6 CM2KG Platform . . . . .	16
2.7 Summary . . . . .	16
<b>3 EA Smells Analysis</b>	<b>19</b>
3.1 Detectable EA Smells via Knowledge Graphs . . . . .	20
3.2 Undetectable EA Smells . . . . .	79
3.3 Out of Scope . . . . .	81
3.4 Summary . . . . .	82
<b>4 Extension of the Enterprise Architecture Smells catalog</b>	<b>83</b>
4.1 Unifying the schema of EA Smells . . . . .	84
4.2 Extending the schema and revising the EA Smells catalog . . . . .	85
4.3 Summary . . . . .	93
<b>5 Integrating the EA Smells catalog into a Knowledge Graph-based detection platform</b>	<b>95</b>
	xiii

5.1	General Overview . . . . .	95
5.2	Implementation of the Detection platform . . . . .	96
5.3	Summary . . . . .	101
<b>6</b>	<b>Evaluation</b>	<b>103</b>
6.1	Methodologies and Metrics . . . . .	103
6.2	Software Testing . . . . .	104
6.3	Empirical Experiment . . . . .	104
6.4	Precision and Recall . . . . .	108
6.5	Summary . . . . .	114
<b>7</b>	<b>Related Work</b>	<b>115</b>
<b>8</b>	<b>Conclusion</b>	<b>117</b>
8.1	Summary . . . . .	117
8.2	Future Work . . . . .	117
<b>A</b>	<b>ArchiMate Elements and Relationships</b>	<b>119</b>
	<b>List of Figures</b>	<b>123</b>
	<b>List of Tables</b>	<b>125</b>
	<b>List of Algorithms</b>	<b>127</b>
	<b>Acronyms</b>	<b>129</b>
	<b>Bibliography</b>	<b>131</b>



# Introduction

This chapter defines the current state of Enterprise Architecture Debt (EAD) and the Enterprise Architecture (EA) Smells catalog. We explain how EA Smells can bridge the analysis gap and help improve the quality of EA and EA models, especially those in the ArchiMate modeling language. Our approach leverages a Knowledge Graph-based method to detect EA Smells via Knowledge Graph queries in ArchiMate models. We then outline the research questions we aim to address and the methodology we use to present and evaluate our approach.

## 1.1 Motivation and Problem Statement

Enterprise Architecture (EA) represents a coherent set of various architectures, including business architecture, information system architecture, and infrastructure architecture [1]. Together, these domains provide a comprehensive view of an enterprise. The implementation of an enterprise can often be complex and not always straightforward [2]. As enterprises grow, their business operations and IT infrastructures become increasingly intricate, complicating both design and architecture [3]. EA models are essential to the success of an enterprise. While modeling of EA is prevalent in both research and industry, more focus is needed on the analysis of these EA models [4, 5]. Moreover, until recently, there was a lack of clear and formal definitions for assessing quality flaws and the negative consequences of EA [2].

Technical Debt (TD) is a concept that addresses the shortcomings in the software development industry that may negatively impact maintainability, development, quality, and long-term success of producing software [6]. The original concept of TD focused on the code level, and Code Smells analyze the source code in order to indicate potential TD. Although TD has evolved and is now subcategorized into software architecture debt, database design debt and documentation debt, it encompasses only the technical aspects of IT and not the business aspects. To address this limitation, Hacks et al. [2] defined

the concept of Enterprise Architecture Debt (EAD), which extends the TD to cover both the technical and organizational aspects of the enterprise. EA Smells [7] are a set of means for measuring and indicating EAD by analyzing the organization from a more holistic viewpoint. It was initially derived from Code Smells that can be adopted in the EA domain. Later, it was extended with some Software Architecture Smells and Business Process Modeling (BPM) anti-patterns, resulting in a catalog of sixty-three EA Smells [8]. Knowledge about EA smells could help EA architects to prevent design mistakes and detect model flaws in the early stages of design.

Generally, each Smell in the catalog consists of a name, description, cause, detection, and other attributes containing all necessary information about the specific Smell. However, the specific definition or approach for identifying the EA Smells in EA models is missing so far. Furthermore, some descriptions and concepts defined in the catalog are too abstract. Therefore, a revision is required to provide a precise approach and a concrete mechanism for detecting EA Smells and to extend the information on EA Smells.

ArchiMate [9] is a *de facto* standard modeling language for EA modeling but does not provide analyzing and visualizing functions for model quality [3]. Measuring the quality of ArchiMate models via EA Smells could give architects indications of imperfections in the model. However, manually analyzing large models can be difficult because the complexity of the models can lead to miscalculations, and some details may be overlooked [10]. On the other hand, the automated analysis can prevent architects from making such errors. Therefore, integrating EA Smells into an automated detection tool will not only increase the quality of the EA and EA models, but also raise awareness about common bad practices.

Knowledge Graphs have recently gained traction in the field of modeling, particularly in EA modeling. A new approach for analyzing EA models within Knowledge Graphs has been introduced [11]. Various algorithms and metrics can be employed to evaluate EA models after converting them into Knowledge Graph (KG) format. KG queries are efficient, even on larger graphs. Therefore, defining KG queries for analyzing EA models has shown promising results [12].

### 1.2 Aim of the Work

The objective of this thesis consists of two parts. In the first part, we will extend the EA Smell catalog [8] by defining KG-based queries and revising existing EA Smells definitions. In the second part, we will create a comprehensive graph-based EA Smell detection platform for automated analysis by transforming ArchiMate models into KG and integrating the EA Smell catalog into that platform.

This work can serve as a guideline or an information base for detecting EA Smells in EA models, especially in the ArchiMate modeling language. All EA Smells from the catalog will be analyzed in the first step. The catalog contains sixty-three Smells. We will study and interpret each EA Smell to determine whether it can be identified via KG queries in

the domain of the ArchiMate modeling language. Based on this analysis, we propose KG queries for detecting EA Smells. Some queries could have one or more parameters as query options. These options are strongly dependent on the enterprise and could differ from organization to organization, but some default values should be defined, which can be seen as a threshold. We will extend the EA Smell catalog by incorporating the results of our analysis. As mentioned in the previous section, most definitions and examples provided in the catalog are either too abstract or taken directly from Code Smells without further adaptation in the EA domain. Therefore, we will revise the existing descriptions to make them more precise.

Once reasonable queries for the EA Smells have been provided, and the EA Smells catalog has been enhanced, we will develop a KG-based detection platform and integrate the EA Smell catalog into it. Our platform will analyze the ArchiMate models as KG and detect possible EA Smells from the given model. In order to detect EA Smells via KG queries, we have to convert the ArchiMate models into KG. To that end, we will use the Conceptual Model to Knowledge Graph (CM2KG) platform [13] for our model transformation. The CM2KG is not our contribution to this thesis, we will just use it as an external service. Our detection platform will deliver a user interface to show analysis results in textual and visual form. Furthermore, the enterprise architects should be able to customize Smell parameters to strengthen or weaken the conditions for Smell detection, considering the size of the enterprise and organization use cases. Enterprise architects with a basic understanding of ArchiMate should be able to use this platform, although familiarity with KG queries is not required. Our detection platform should increase the quality of EA, help enterprise architects identify weaknesses in the models, and gain insight into possible negative consequences in the long term.

In our thesis, we aim to answer the following research questions:

1. What is an appropriate means to detect EA Smells via Enterprise Architecture KG queries?
2. How to achieve an interoperability between the EA Smell catalog and the CM2KG platform?

## 1.3 Methodological Approach

In this thesis, we will utilize the problem-solving paradigms of Design Science Research (DSR) [14]. DSR focuses on addressing real-world problems by developing both intellectual and computational artifacts [14]. It is essential to evaluate the “utility, quality, and efficacy” [14] of the proposed solution. We will adhere to the six process steps outlined by Peffers et al. [15] to create our artifact. Additionally, we will conduct empirical evaluations to ensure the artifact’s quality and usefulness. Overall, the methodologies employed in this thesis are as follows.

- *Literature Analysis*

We will conduct a literature review collecting the relevant knowledge and theoretical background for TD, EAD, Code Smells, EA Smells and the ArchiMate modeling language.

- *Domain Analysis and Context Mapping* We will conduct a comprehensive analysis and assessment of each EA Smell within the framework of the ArchiMate modeling language and provide KG queries for detection. To accomplish this, we will scrutinize the underlying Code Smell or anti-pattern to determine its relevance to EA. If applicable, we will explore potential modifications to adapt the Smell to the new EA domain, specially ArchiMate modeling language.

- *Software Engineering*

**Model Transformation** - In order to detect EA Smells via KG queries, we have to transform the ArchiMate models into a graph-suited format. We will perform an out-place model transformation via the CM2KG platform to convert the ArchiMate Exchange File Format [16] to GraphML format.

**Software Prototype** - After the analysis phase, we will develop a KG-based detection tool to assess the quality of ArchiMate models by detecting EA Smells via KG queries.

- *Evaluation*

In order to evaluate the quality and correctness of our artifact, both in technological and conceptual aspects, and get a holistic picture of our results, we will perform both quantitative and qualitative evaluations. We will use the following methods for our evaluations.

### Qualitative

**Software Testing** - We will perform unit and integration tests to evaluate the behavior of our platform and its interaction with the extended EA Smell catalog and the CM2KG platform.

### Quantitative

**Empirical Experiment** - We will conduct a test using a large set of real-world models to investigate query execution time and the detectability of EA Smells in practice.

**Precision and Recall** - Our detection platform will verify an EA model with a set of EA Smells provided by an EA expert to assess the correctness and relevance of our provided queries. In other words, it will determine the extent to which the queries are semantically correct.

## 1.4 Thesis Outline

The remainder of this thesis is structured as follows:

### Chapter 2 - Background

This chapter provides the background knowledge that forms the basis for our thesis. We explain the concept of TD, EAD, and EA Smells catalog. We briefly introduce the ArchiMate modeling language, KG, the CM2KG platform, and the Neo4j database.

### Chapter 3 - EA Smells analysis and defining knowledge Graph queries

We examine the EA Smells listed in the catalog, explain the process of translating concepts from the foundational level to the EA domain, and outline the KG queries along with their parameters and default values. Additionally, we touch upon the EA Smells that are undetectable in the graph and those that are beyond the scope of this research.

### Chapter 4 - Extension of EA Smell catalog

We extend and unify the meta-model of EA Smells and revise the existing definition based on our findings from the previous chapter.

### Chapter 5 - Integrating the EA Smells catalog into a Knowledge Graph-based detection platform

We introduce our detection platform and describe its interaction with the EA Smell catalog and CM2KG platform.

### Chapter 6 - Evaluation

We assess the quality and accuracy of our work, focusing on both technological and conceptual aspects.

### Chapter 7 - Related Works

This chapter describes previous research and efforts in EAD and EA Smells, identifies the gap filled by our work and outlines our contributions.

### Chapter 8 - Conclusion

The last chapter summarizes our thesis contributions and discusses required improvements and future works.



# CHAPTER 2

## Background

This chapter provides the necessary background information to ensure a common understanding of our thesis. We begin by introducing the concept of Technical Debt (TD) in Section 2.1 and how it relates to the new metaphor of Enterprise Architecture Debt (EAD) (Section 2.2). Section 2.3 delves into the idea of EA Smells and their significance in measuring the level of EAD. In Section 2.4 we introduce the widely accepted ArchiMate modeling language, a *de facto* standard modeling tool for EA models. Section 2.5 discusses the common practice of analyzing EA models as graphs in research. Finally, we present the CM2KG platform (Section 2.6), a model transformation tool for converting ArchiMate models into KG.

### 2.1 Technical Debt

Cunningham [17] introduced the metaphor of Technical Debt (TD) in 1992. TD is a crucial issue in the software industry due to its adverse impacts on the maintenance, quality, progress, and long-term success of software projects [6]. It refers to a circumstance where a developer or a software team might opt for a solution that accelerates short-term development but results in higher future costs in the form of additional work [18].

The metaphor of TD comes from financial debt. Just like financial debt incurs interest payments, TD results in increased future costs for the development and maintenance of software due to earlier suboptimal design and implementation choices. If the interest is not paid in the form of re-architecting, refactoring, or improving the technological gaps to fix these shortcomings, then the interest will become higher. The higher the interest, the more complex the software development lifecycle becomes. Sometimes, the TD is necessary or justified if the interest is paid off on time [19].

Martin [20] stated that TD should only be considered if a decision or design choice is made deliberately. That is, the messy code or unintentional design shortcomings that

arise due to a lack of knowledge should not be classified as TD. However, Fowler [21] states that the distinction should not be between debt and non-debt but rather between reckless and prudent debts. He presented the “Technical Debt Quadrant” (Figure 2.1) and categorized debts as reckless, prudent, deliberate, and inadvertent. TD initially focused on the code level but has been extended to documentation, testing, database design, and many other technical areas.

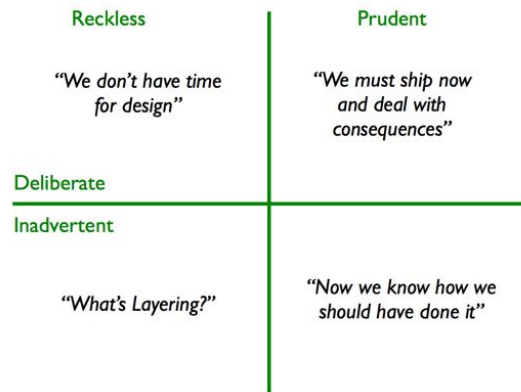


Figure 2.1: Technical Debt Quadrant [21]

## 2.2 Enterprise Architecture Debt

The definition of enterprise diverges significantly [22]. Some understand “business” or “organization,” while others understand “enterprise systems” [23]. The Open Group [24] has defined the enterprise as “any collection of organizations that has a common set of goals and/or a single bottom line.” The same applies to the meaning of architecture. The term varies even more widely. A more common definition of *architecture* is a set of artifacts that model the blueprint of an enterprise [23].

A standard definition of Enterprise Architecture (EA) is a cohesive set of ideas, methods, and models used to design, develop, and implement a company’s organizational structure, business processes, information systems, and infrastructure [25]. EA provides a holistic view of an organization covering both IT and business aspects. Implementing an EA might be challenging due to limited resources and uncertainties during the design phase. TD is considered to raise awareness of possible technical shortcomings and design flaws in the technical domain. However, the TD is limited to the technical and does not include business aspects.

Hacks et al. [2] extended the concept of TD and proposed the metaphor of Enterprise Architecture Debt (EAD) covering the organization’s IT and business aspects. The formal definition of EAD is as follows:

*“Enterprise Architecture Debt is a **metric** that depicts the deviation of the **currently present state** of an enterprise from a **hypothetical ideal state**.”* [2]

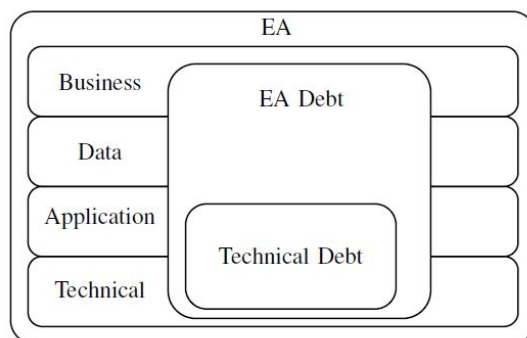


Figure 2.2: Placing of TD within the concept of EAD [2]

Figure 2.2 demonstrates the different layers of an EA. The TD covers the technology and application layers. Beyond that, the EAD encompasses the TD and covers the organization’s data and business layers.

It often arises when short-term solutions or changes are made without considering the long-term impact on the overall architecture. This debt can be manifested as outdated technology, incompatible systems, or inefficient processes, hindering the organization’s ability to adapt, innovate, or meet evolving business needs.

## 2.3 EA Smells

Code Smells are a key component for measuring the level of TD in software systems. They serve as a tool for identifying suboptimal code that may require refactoring. Salentin and Hacks [7] introduced the concept of EA Smells, which are designed to measure the level of EAD. EA Smells serve a similar purpose to Code Smells and anti-patterns in the EA domain. Like Code Smells, EA Smells help assess EAD. The primary goal of EA Smells is to enhance the quality of EA by addressing shortcomings and design flaws in EA while also raising awareness of bad habits.

Salentin and Hacks [8] developed a catalog containing forty-five EA Smells derived primarily from code Smells [7]. Later, Lehman et al. [26] expanded the catalog with eighteen additional EA Smells that stem from process anti-patterns. Finally, Benny Tieu [27] enriched the catalog by transferring three more software architecture Smells into the EA domain.

At the time of writing this thesis, the catalog contains sixty-three EA Smells. It is available as a web application <sup>1</sup>. Figure 2.3 demonstrates an EA Smell from the catalog. Each Smell has a name, aliases, description, consequences, detection, etc. They provide information describing an EA Smell. Detailed information about the current metamodels of the EA Smells in the catalog is given in Section 4.1 (Table 4.1).

<sup>1</sup><https://swc-public.pages.rwth-aachen.de/smells/ea-smells/>

The EA Smell catalog is still in its early stages and includes the Smells relevant to the EA domain. However, many of the EA Smells in the catalog require more specific definitions and a precise approach to identification. Currently, most definitions are too abstract or taken directly from related Code Smells or anti-patterns without being properly adapted to suit the EA context.

For example, the *Combinatorial Explosion* EA Smell, illustrated in Figure 2.3 currently lacks a precise definition of how the Smell could occur and how it should be detected. It is obvious that the field *definition* is overly general, while the *detection* section is empty. Additionally, the provided *example* belongs to the underlying Code Smell and is irrelevant to the EA domain. Many other EA Smells face similar issues and require adjustments and revisions to better align with the specifics of the EA domain.

This thesis aims to address existing gaps by revising current definitions and providing specific detection mechanisms, thereby enhancing the understanding of EA Smells.

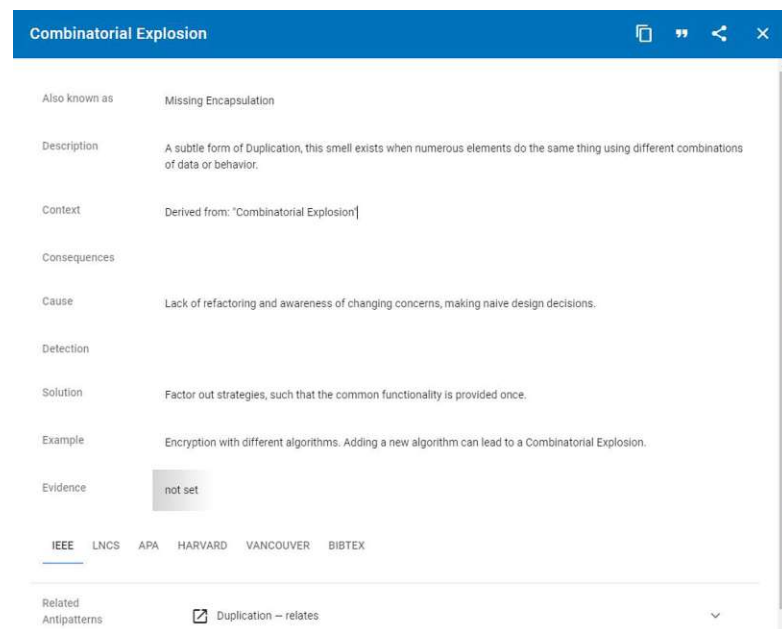


Figure 2.3: Current representation of the *Combinatorial Explosion* EA Smell in the catalog

## 2.4 ArchiMate

*ArchiMate* is a widely adopted, independent modeling language designed for enterprise architecture developed by The Open Group [9]. ArchiMate provides a comprehensive and systematic way to describe, analyze, and visualize the structure and behavior of complex systems within an organization. It is a powerful tool for enterprise architects, business analysts, and other stakeholders to seamlessly communicate and align business and IT

aspects. ArchiMate provides a common language for business processes, organizational structures, IT, and technical infrastructure. ArchiMate's core concepts are elements, relationships, and viewpoints. The elements represent the building blocks of the enterprise, while relationships depict the associations between these elements. Viewpoints offer specific perspectives on the architecture to cater to different stakeholders.

ArchiMate core framework consists of three core layers and three aspects. The layers are business, application, and technology. The *Business Layer* represents services offered to customers, realized by business processes. Business actors perform business processes. In the *Application Layer*, application services realize and support business. The *Technology Layer* provides the necessary infrastructure services to support applications like processing, storage, and communication. These services are realized through devices, communication hardware, and system software. In addition to layers, ArchiMate has three aspects: *active structure*, *behavior*, and *passive structure*. The active structure describes the structural elements in the model. These are the elements that perform a behavior. The behavior aspect represents functions, processes, events, and services. The passive structure signifies the entities upon which behaviors are executed. Since version three, ArchiMate has added one more new aspect and two more layers. The motivation aspect describes the motivation elements, such as requirements, stakeholder concerns, and goals. The strategy layer has elements for describing the strategic choices and directions. Architectures can be implemented and migrated using the implementation and migration elements. Figure 2.4 demonstrates the entire framework of ArchiMate with layers and aspects.

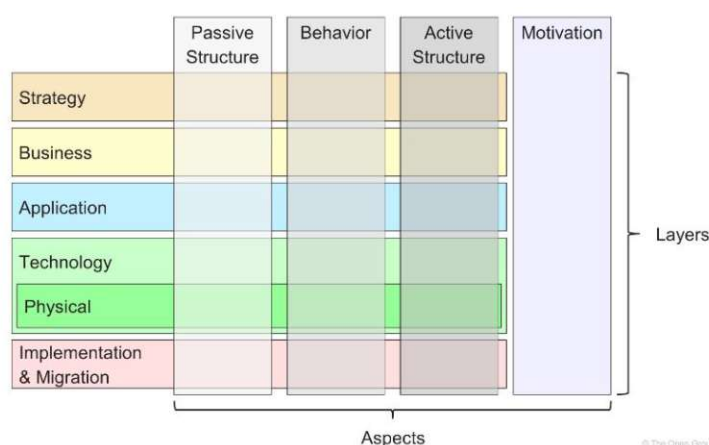


Figure 2.4: The ArchiMate Full Framework taken from [9]

Relationships represent the associations between elements. ArchiMate has four types of relationships.

- **Structural Relationships** build static coherence of the architecture. *Assignment*, *Realization*, *Aggregation* and *Composition* relationships are four types of structural relationship.

- a) **Assignment relationship** links active structures to behaviors. It defines responsibility, the performance of behavior and execution. Figure 2.5 illustrates an example of an *Assignment* relationship in ArchiMate. “Payment Interface” is assigned to “Payment Service,” meaning “Payment Interface” can perform or execute the “Payment Service.”

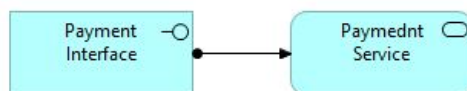


Figure 2.5: ArchiMate Assignment Relationship

- b) **Composition relationship** illustrates that one element is a component of another element in the model. Similar to the composition relationship in the UML class diagram, it signifies an existence dependency between a whole and its constituent parts. Figure 2.6 illustrates an example of *Composition* relationships between elements. The elements “Accounting,” “Payment” and “Billing” are part of the “Financial Processing” element.

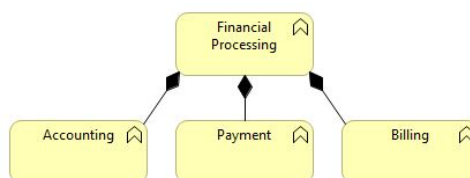


Figure 2.6: ArchiMate Composition Relationship

- c) **Aggregation relationship** also derived from the aggregation relationship in the UML class diagram. Aggregation groups one or more concepts. Unlike composition, the existence of aggregated elements does not depend on the aggregating elements. Figure 2.7 illustrates an example of the *Aggregation* relationship.

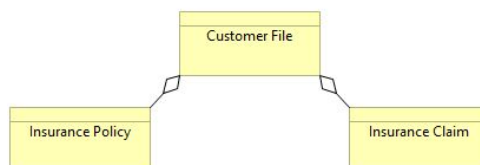


Figure 2.7: ArchiMate Aggregation Relationship

- d) **Realization Relationship** describes the realization of an abstract or logical element by a more concrete element. Figure 2.8 depicts a *Realization* relationship between two elements. The “Transaction Processing” *Business Function* realizes or creates a more abstract element, “Billing Service.”

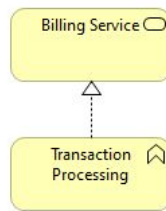


Figure 2.8: ArchiMate Realization Relationship

- **Dependency Relationships** outline how elements depend on each other. *Serving*, *Access*, *Influence* and *Association* are four types of dependency relationships.

- a) **Access relationship** indicates that a process or function interacts with a passive element. Generally, the *Access* relationship represents a data dependency. Figure 2.9 demonstrates two *Access* relationships. The first “Create Invoice” *Business Process* creates an invoice, the second “Send Invoice” *Business Process* reads that invoice.

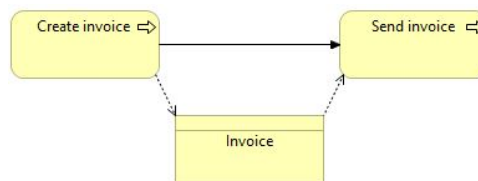


Figure 2.9: ArchiMate Access and Triggering Relationships

- b) **Influence relationship** describes how an element may have positive or negative impact on another element. Figure 2.10 illustrates a requirement “Assign Personal Assessment,” which has a positive impact on “Reduce Workload of Employees” and, at the same time, a negative consequence on “Decrease Costs.”

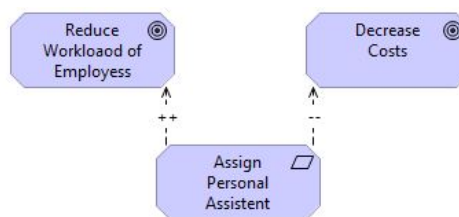


Figure 2.10: ArchiMate Influence relationship

- c) **Serving relationship** illustrates that one element provides functionality to another element. Figure 2.11 illustrates a *Serving* relationship between the *Application Service* and *Business Process* elements. The “Payment Service” serves its functionality to the “Pay Invoices.”

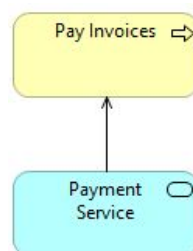


Figure 2.11: ArchiMate Serving Relationship

- d) **Association Relationship** is a more general or unspecified relationship. It may also define a relationship not represented by other relationships in the ArchiMate modeling language. Figure 2.12 illustrates an *Association* relationship between the *Representation* and *Business Object* elements.

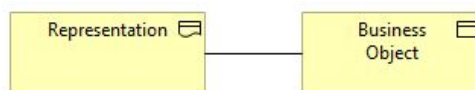


Figure 2.12: ArchiMate Association Relationship

- **Dynamic relationships** depict temporal order between elements. The process chains are described by this type of relationships. *Temporal* and *Flow* are two types of dynamic relationships.

- a) **Triggering relationship** describes the temporal or casual order between elements. Figure 2.13 illustrates an example of a *Triggering* relationship. “Create Invoice” triggers “Send Invoice.”

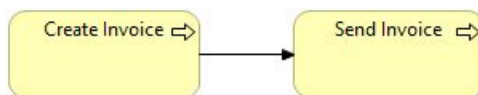


Figure 2.13: ArchiMate Triggering Relationship

- b) **Flow relationship** represents the transfer of data, information, and goods through elements. Figure 2.14 shows an example of a *Flow* relationship. “Claim Assessment” followed by “Claim Settlement.”

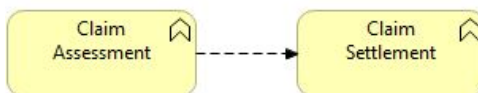


Figure 2.14: ArchiMate Flow Relationship

- **Specialization Relationship** specifies that an element is a specific type of another element. As demonstrated in Figure 2.15, “SMS Notification” and “Mail Notification” are specific types of “Notification.”

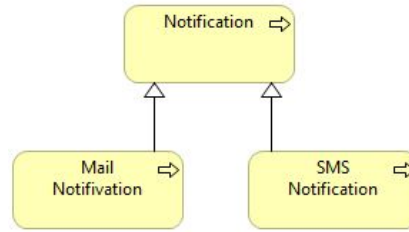


Figure 2.15: ArchiMate Specialization Relationship

## 2.5 Graph-based analysis of EA Models

Interpreting models in EA as graphs is a widely used approach in research [4]. For instance, Aier [28] proposes the EA Builder tool that supports the identification of clusters in graphs. These Clusters are designed to identify potential candidates for services in a service-oriented architecture. Johnson et al. [29] interpret the modeling of EAs as a probabilistic state estimation problem. They propose using a Dynamic Bayesian Network to model the network, allowing for observing network traffic and predicting the most probable configuration of IT infrastructure. Later, Bebenze and Hacks [30] redefined this approach and enhanced it by modeling the network using Hidden Markov Models (HMMs). Hacks and Lichter [31] also provided a Probabilistic Prediction for modeling model uncertainty and contradictory data in EA. Several efforts have been made to leverage graphs for optimizing EA [32, 33, 34]. Recently, Smajevic and Bork [11, 35] introduced an innovative approach for analyzing EA models within KG. They provided a generic platform for transforming EA models into KG. Furthermore, they defined a small set of queries for detecting EA Smells in KG representing the ArchiMate model [3].

A KG is a powerful way of organizing and representing information to capture the complex interrelationships between different entities. At its core, it is a graph structure which consists of nodes, relationships, and their properties. Data is stored and represented as nodes and relationships. Nodes are vertices, and relationships are edges connecting vertices. Both nodes and relationships have properties as key-value pairs that store attributes and information. KG enables a flexible and efficient way of storing, querying, and analyzing data, making it invaluable across various domains and industries. Such information is stored in a graph database and is represented as a graph. KG's more general and formal definition is "a large network of entities, and instances for those entities, describing real-world objects and their interrelations, with specific reference to a domain or an organization" [36, p. 27]. KG is used in a wide range of industries and applications. Well-known corporations like Google, Yahoo, Microsoft, and Facebook have developed their KG to provide powerful semantic searches and an intelligent way to process and distribute data [37]. Other common uses of KG are Artificial Intelligence (AI), Internet of Things (IoT), Machine Learning (ML), and Healthcare [38, 39, 40]. In our work, the same approach is used to analyze the EA models. By interpreting EA models as KGs, we define KG queries to detect EA Smells in the ArchiMate models.

We use the *Neo4j* graph database to store EA models as KGs and analyze EA models by defining the KG queries to detect EA Smells in the ArchiMate models. *Neo4j* is a popular graph database management system, which stores and queries data in a graph format.

## 2.6 CM2KG Platform

The CM2KG is a generic platform for model transformation [13, 11, 35]. It stands for *From Conceptual Models To Knowledge Graphs*. This platform is designed to convert conceptual models based on EMF, ADOxx metamodels, and ECORE-based modeling platforms into graph structures, with both input and output files in XML format. For ArchiMate, the platform processes an Archi file in Open Exchange File Format and converts it to GraphML format. It operates at the *meta*<sup>2</sup> level, facilitating transformation from the meta-meta model, with the potential for further customization at the modeling language level.

Additionally, the CM2KG platform provides an interface for graph analysis tools such as Neo4j, Gephi, yEd, and Stardog. The converted GraphML file can be imported into these tools for analysis. Figure 2.16 depicts a general overview of the platform. In our thesis, we use the CM2KG platform to transform ArchiMate models into graphs.

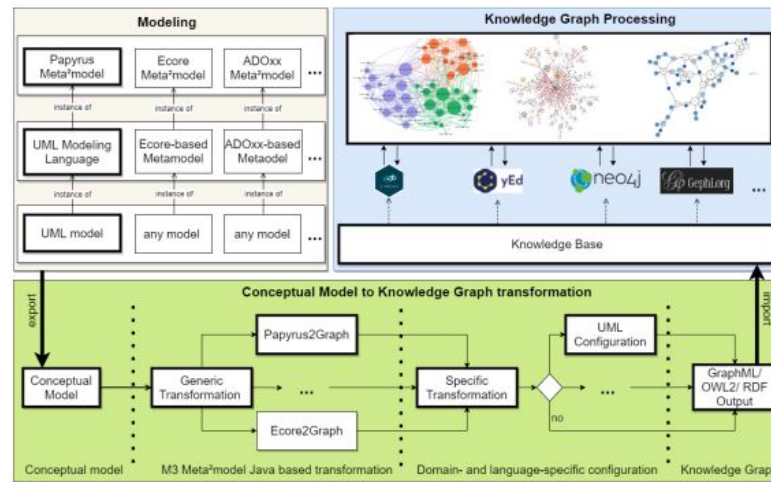


Figure 2.16: CM2KG, a generic platform for transforming models into graphs [11]

## 2.7 Summary

In this section, we provide essential background information to enhance understanding of the thesis. We discuss the limitations of TD in the domain of EA while also introducing the new metaphor of EAD and the concept of EA Smells as a metric for evaluating EAD. We emphasize that interpreting EA models as graphs streamlines analysis through graph

queries and various algorithms. Additionally, ArchiMate and its relationship types are explored. The next chapter examines EA Smells in the context of KG-based ArchiMate models and presents corresponding KG queries.



# EA Smells Analysis

This chapter analyzes and describes how to detect EA Smells in the EA domain, particularly in the ArchiMate modeling language via KG queries. The concept of EA Smell is new to the research field, as are the descriptions in the EA Smell catalog. It requires improvements and clarifications because the existing definition of EA Smells is too abstract and lacks a precise definition and approach for detecting those Smells in the EA domain. This chapter analyzes EA Smells from the catalog by interpreting and studying every Smell and whether they are detectable in the context of the ArchiMate modeling language via KG queries.

EA research has shown that EA models can be interpreted as graphs [4, 41, 28, 29, 42, 3]. Moreover, a KG-based approach for analyzing EA models and detecting a tiny set of EA Smells has yielded a promising result [12]. This approach surpassed the previous Java-based solution [43]. The Smell detection time using the KG query was three to five times faster than the Java-based approach. In this chapter, we use the same KG-based approach to detect EA Smells in graphs derived from ArchiMate models. However, we maintain all KG queries in the Smell catalog and do not store them in the detection platform. We separate the detection logic from the detection platform to ensure long-term maintainability. Furthermore, users do not need to know anything about KG queries. Section 5 describes our detection platform in more detail.

For transforming the ArchiMate models into KG, we use a platform called CM2KG [13, 11, 35]. This is not our contribution to this thesis. We use the platform only as an external service for model transformation. We will discuss it in more detail in Section 5.

Section 3.1 describes the EA Smells in the ArchiMate models for which the KG queries, query options, thresholds, and graphical examples are analyzed and provided.

Section 3.2 describes the EA Smells that are not detectable via KG queries together with those that are out of the scope of this thesis. EA Smells that require more than one EA

model for detection are beyond the scope of this thesis and are also briefly discussed in Section 3.3.

## 3.1 Detectable EA Smells via Knowledge Graphs

For each Smell in the catalog, we briefly describe the underlying Code Smell or anti-pattern and explain how we adapt and transfer this Smell in the EA domain, particularly in the ArchiMate modeling language. Based on our provided logic for detection, we define the KG queries. A KG query may have one or more query parameters. The criteria for detecting EA Smells may vary from organization to organization. Therefore, the queries should be adjusted using query parameters. These parameters always have a default value as a threshold. Enterprise architects may change the threshold to fit the query based on their enterprise concerns. However, it is also possible for a KG query to have no parameters. Alongside each KG query and its corresponding parameters, we provide a graphical example to illustrate the EA Smell clearly and reduce misinterpretations.

To enhance the understanding of the provided KG queries, we will briefly introduce the Neo4j query language. Cypher is a declarative graph query language designed to express patterns and traverse the graph to retrieve data. It offers a visual approach for identifying relationships and patterns within the data. In Cypher, nodes and relationships are defined using parentheses and brackets. Relations in Neo4j are always directed; if a direction is not specified, the Cypher query considers both directions. Listing 1 presents an example of a native Cypher query. The variables *a* and *b* represent nodes, while *r* represents a relationship. The label *Person* within the parentheses identifies a node type, and the property *name*: 'John' inside curly brackets specifies a node attribute. Similarly, *knows* denotes a relationship type, and the syntax *\*1..2* describes the hops degree, indicating the minimum and maximum lengths of relationships between nodes. The query outlined in Listing 1 aims to find the friends and friends of friends of a person named John.

```
MATCH p = (a:Person{name: 'John'})-[r:knows*1..2]-(b:Person)
RETURN p
```

Listing 1: Simple example of a Cypher query

In this thesis, most of the KG queries we provide include query parameters, mainly when dealing with graph traversal or the length of relationships. We have two main reasons for setting the degree of relationships as a query parameter. First of all, the query options for detecting EA Smells can vary between organizations. For instance, some enterprise architects may consider a process chain with only ten edges, while others may be interested in chains with at least twenty edges. Secondly, if no upper limit is specified for path lengths, the query may take considerable time to execute. Therefore, it is essential to set an upper limit to length to prevent performance issues when querying large, densely connected data.

During the writing of this thesis, the latest version of Neo4j is five (5.x). According to the Neo4j documentation, there are some limitations regarding query parameters in the native Cypher query language. Specifically, it is impossible to set the hops degree as a query parameter [44]. As a workaround, we utilize a library called `apoc`<sup>1</sup> (Awesome Procedures on Cypher). This library offers numerous functions and procedures that allow us to parameterize the query in places where it is not possible within the native Cypher query. In this thesis, we not only use native Cypher queries but also leverage some `apoc` procedures to define our query parameters. We use two procedures to specify path lengths as query parameters: `apoc.nodes.cycles` and `apoc.path.expand`.

We use the `apoc.nodes.cycles` to detect cyclic paths in KG's representing the ArchiMate models. Listing 2 illustrates the syntax of this procedure and provides a simple example of how to call it. The procedure accepts two parameters: a list of nodes, and configuration properties in the form of key-value pairs. The configuration properties are not mandatory but filter the length and relation types of the cyclic path. To optimize query performance, we parameterize the length of the paths to prevent adverse impacts on query performance. However, we can modify the value based on the model size. The query provided in Listing 2 looks for all cycles starting from the start nodes with up to three intermediate nodes. Table 3.1 details the configuration parameters.

```

    apoc.nodes.cycles(nodes :: LIST? OF NODE?, config = {} :: MAP?) :: (path
    ↪ :: PATH?)

    MATCH (m1:Start) WITH collect(m1) as nodes CALL apoc.nodes.cycles(nodes,
    ↪ {maxDepth: 3}) YIELD path RETURN path
    
```

Listing 2: Signature and an example of `apoc.nodes.cycles` procedure

Table 3.1: Possible parameters for `apoc.nodes.cycles` procedure

Parameter	Type
Node	<id> Node list
Relationship	[<]EDGE_TYPE1[>][<]EDGE_TYPE2[>]...
MaxDepth	Number

The second `apoc` procedure used for querying EA Smells is `apoc.path.expand`. This procedure allows us to search for a path based on multiple query parameters, including the starting and ending nodes, minimum and maximum path lengths, labels, and the relationships along with their directions. It takes five parameters in total. The first parameter is the starting node, the second and third are relations and nodes. The last two parameters are minimum and maximum values for path length. Table 3.2 illustrates the syntax of the query parameters in more detail. The symbols “>” and “<” associated with the relationships indicate the direction of the edges. Regarding label parameters,

<sup>1</sup><https://neo4j.com/docs/apoc/current>

the symbol “>” or “/” before a label name signifies that the node with this label is an end or termination node. By including a termination filter (“/”), the path will return up to a node with the specified label. In contrast to the termination filter, the end node filter (“>”) returns all the paths that lead to a node with a specified label, but the traversal will continue beyond the specified end node. Additionally, filtering for labels can be done using a whitelist or blacklist, represented by plus (+) and minus (-) signs, respectively. Finally, the *MinLevel* and *MaxLevel* parameters are used to specify the minimum and maximum number of hops during the traversal.

Table 3.2: Possible parameters for `apoc.path.expand` procedure

Parameter	Possible Value(s)
Node	<id> Node list
Relationship	[<]EDGE_TYPE1[>]  [<]EDGE_TYPE2[>]  ...
Label	[+/->]LABEL1 LABEL2[*]
MinLevel	Number
MaxLabel	Number

Listing 3 demonstrates a scenario with different parameters to provide a detailed explanation of the query. A person node named “John” serves as the starting point. The path navigates through nodes beginning with the name “John,” featuring an outgoing relation labeled *FOLLOWS* and a bidirectional relation labeled *KNOWS*. The label *Engineering* is marked by a plus sign (“+”), indicating a whitelist filter, which means that all nodes in the path should have this label. Conversely, the presence of the label *Science* is prohibited, denoted by the minus sign, acting as a blacklist. If there is no whitelist or blacklist specified, all labels will be considered in the path. A label with a backslash sign (“/”) is considered as a final node. The label *Management* is a terminate node, causing the traversal to finish reaching a node with the *Management* label and return the resulting path.

```

MATCH (p:Person {name: "John"})
CALL apoc.path.expand(p, "FOLLOWS>|KNOWS",
↪ "+Engineering|-Science|/Management", 1, 3)
YIELD path
RETURN path

```

Listing 3: A Cypher query using `apoc.path.expand` procedure

We have briefly introduced the Cypher query and `apoc` procedures. Now we will start with the EA Smell analysis, providing definitions and KG-queries.

### 3.1.1 Cyclic Dependency

The *Cyclic Dependency* EA Smell derives from the *Circular Dependency* Code Smell [45]. This Smell occurs when two or more abstractions depend on each other, resulting in a

cyclic path between them. For instance, in software engineering, a cyclic dependency exists between classes A and B, if A has a reference from B and B contains a reference from A. Therefore, any changes in class A may affect B and vice versa. This Smell leads to tight coupling between the classes. We can demonstrate such dependencies using a dependency graph. The *Cyclic Dependency* Smell is also prevalent in microservice architecture design, where there is a cyclic communication between microservices [46]. In the context of the EA, this Smell occurs in the model once a cyclic path between elements exists.

Table 3.3 depicts a graphical example of the cyclic path between *Business Services* 1, 2, and 3. Table 3.3 also features the KG query detecting cyclic paths. We specify a relation degree in the query for detecting cyclic paths. The query checks for cyclic paths until they reach a certain length. We must set an upper limit on length to prevent query performance issues with large, densely connected data. However, in a native Cypher query, there is no possibility of setting the relation depth as a variable. Therefore, we use a method `apoc.nodes.cycles` from the `apoc` library to set the maximum depth for detecting cyclic paths as a parameter. We set ten as the default maximum value for edges between nodes. The relation length is set as a query parameter, and enterprise architects can modify the default value based on their model size. However, setting a large value as hop degree may impact the query performance.

Table 3.3: Graphical example, Cypher query and query options for detecting *Missing Abstraction* EA Smells

Properties	Descriptions
Cypher Query	<pre> MATCH (m) WITH collect(m) as nodes CALL   ↪ apoc.nodes.cycles(nodes, {maxDepth: \$maxDepth})   ↪ YIELD path RETURN path as p </pre>
Query options	<pre> "queryParams": [   {     "name": "maxDepth",     "type": "number",     "default": 10,     "description": "The max length of a cyclic path."   } ] </pre>
Continued on next page	

Table 3.3 – continued from previous page

Property	Description
Graphical ex-ample	<pre> graph LR     BS4[Business Service 4] -.-&gt; BS1[Business Service 1]     BS1 --&gt; BS2[Business Service 2]     BS2 --&gt; BS5[Business Service 5]     BS3[Business Service 3] --&gt; BS1     BS3 --&gt; BS2     BS4 -.-&gt; BO[Business Object]     BS3 --&gt; BO </pre>

### 3.1.2 Incomplete Pairs

The *Incomplete Pairs* EA Smell is derived from the *Incomplete Abstraction* Code Smell. Coherence and completeness are essential aspects of an abstraction. *Incomplete Abstraction* arises when an abstraction fails to provide all complementary methods [45, pp. 34-35]. For example, in the case of a data structure, the abstraction must support adding and removing elements. If either of these interrelated methods is missing, the abstraction is considered incomplete.

We have renamed *Incomplete Abstraction* to *Incomplete Pairs* to better align with the EA domain. We check for completeness between elements within a process. Table 3.5 illustrates an example of *Incomplete Pairs* EA Smell in the ArchiMate modeling language. An element is named “start claim,” but a complementary element “stop claim” is missing. This indicates that the model suffers from the *Incomplete Pairs* EA Smell.

Table 3.5 demonstrates the KG query and a complementary list of key-value pairs as query parameters for detecting the *Incomplete Pairs* EA Smell. If an element name is included in the keys, then the query checks for its counterpart. If the counterpart element does not exist, then the model may suffer from the *Incomplete Abstraction* EA Smell. Girish Suryanarayana et al. [45, p. 37] provided a set of complimentary pairs listed in Table 3.4 for detecting *Incomplete Abstraction* Code Smell. By default, we provide the same symmetric pairs as query parameters. However, enterprise architects may extend or modify the parameters.

Table 3.4: Symmetric pairs taken from [45, p. 37]

Min/max	Open/close	Create/destroy	Get/set
Read/write	Print/scan	First/last	Begin/end
Start/stop	Lock/unlock	Show/hide	Up/down
Source/target	Insert/delete	First/last	Push/pull
Enable/disable	Acquire/release	Left/right	On/off

Table 3.5: Graphical example, Cypher query and query options for detecting *Incomplete Pairs* EA Smells

Incomplete Pairs	
Property	Description
Cypher query	<pre>WITH apoc.map.values(\$pairs, keys(\$pairs), true) as jsonValues MATCH (n) WHERE any (x in keys(\$pairs) WHERE toLower(n.name) starts WITH x) WITH n WITH [x in keys(\$pairs) WHERE toLower(n.name) starts WITH x   x] as ↪ copplementaryPair, n unwind copplementaryPair as selectedKey WITH collect(distinct selectedKey) as complementaryKey, n WITH n, apoc.map.values(\$pairs, complementaryKey, true) as ↪ jsonValues, complementaryKey MATCH (n), (m) WHERE not exists { MATCH (j) WHERE m=j and toLower(n.name) contains complementaryKey and ↪ toLower(j.name) contains apoc.map.values(\$pairs, ↪ complementaryKey, true) } return n</pre>
Query options	<pre>"queryParams": [ {   "name": "pairs",   "type": "map",   "default": { "min": "max" , "open": "close", "start": "stop", ↪ "create": "destroy", "get": "set",   "read": "write", "print": "scan", "first" : "last", ↪ "begin": "end", "lock": "unlock", "show": "hide", ↪ "up": "down", "source": "target", "insert": "delete", ↪ "push": "pull", "enable": "disable", ↪ "acquire": "release", "left": "right", "on": "off"},   "description": "A list of complementary key-value pairs." } ]</pre>
Graphical example	

### 3.1.3 Multifaceted Abstraction

A good abstraction should only be responsible for one specific functionality, and when it encapsulates more than one responsibility, it signifies the *Multifaceted Abstraction* Smell [45, pp. 41-43]. The *Multifaceted Abstraction* Smell violates the single responsibility principle and has a low cohesion, which violates the modularization principle. An example of this Smell at the code level is the `Java.Util.Calendar` package, as it handles both calendar and time functionalities [45, p. 41].

We can transfer this Smell to the EA domain by examining an abstraction performing or realizing more than one behavior. According to the ArchiMate specification [9], the *Assignment Relationship* links active structures to behaviors and states for responsibility, execution, and performing of a behavior. The *Realization Relationship* signifies that an element implements and supplies the abstract element. Table 3.6 demonstrates two possible scenarios leading to the *Multifaceted Abstraction* in the ArchiMate modeling language. The first example depicts a *Business Actor* element responsible for performing two behaviors, “Business Process 1” and “Business Process 2.” The second example illustrates an *Application Component* realizing two different services. Since the *Business Actor* and *Application Component* have more than one responsibility, they are candidates for the *Multifaceted Abstraction* EA Smell.

The KG query for identifying the *Multifaceted Abstraction* EA Smell is also represented in Table 3.6. We consider the number of *Realization* and *Assignment* relations in our KG query for counting the responsibilities of an active element. If an element has two or more relationships of these types, then the model suffers from *Multifaceted Abstraction* EA Smell.

Table 3.6: Graphical example and Cypher query for detecting *Multifaceted Abstraction* EA Smells

<i>Multifaceted Abstraction</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b>   (m) - [r:AssignmentRelationship RealizationRelationship] -&gt; (n) <b>WITH</b> m, COUNT(r) <b>AS</b> rCount <b>MATCH</b> p = (m) - [r:AssignmentRelationship RealizationRelationship]   (n) <b>WHERE</b> rCount &gt; 1 <b>RETURN</b> p           </pre>
Query options	-

Continued on next page

Table 3.6 – continued from previous page

Property	Description
Graphical ex-ample	<pre> graph LR     BA[Business Actor] --&gt; BP1[Business Process 1]     BA --&gt; BP2[Business Process 2]     AC[Application Component] -.-&gt; AS1[Application Service 1]     AC -.-&gt; AS2[Application Service 2] </pre>

### 3.1.4 Data Service

The *Data Service*, also known as *Data Class* Smell, is a SOA anti-pattern. This type of service solely performs data retrieval and has no other capabilities [47]. The data service does not communicate with other services and has no dependency on other services. However, other services may use this service to retrieve data.

We transfer this anti-pattern into the EA domain by searching services that exclusively provide access to a data source and do not initiate interaction with other components. In the ArchiMate modeling language, the *Business Object* and *Data Object* elements serve as data source representations. Under certain circumstances, the element *Artifact* can also represent a database if it realizes a data source element or has a name that specifies a database (e.g., “DBMS”).

Table 3.7 illustrates an example of the *Data Service* EA Smell in ArchiMate. The “Access Customer Data” has access to a data source named “Customer Data Profile.” A *Serving Relationship* from A to B means that A provides its functionality to B, or A is used by B. Consequently, the “Access Customer Data” serves the data retrieval to the two services - “Customer Administration” and “Verify Claim,” but it does not initiate interactions with any services. Therefore, it qualifies for *Data Service* EA Smell.

Table 3.7 illustrates the KG query for identifying the *Data Service* EA Smell. The query verifies whether a service can access a data source element and refrain from engaging with other services. It does not possess specific types of outgoing relations to other elements. The query takes into account *Business Object*, *Data Object*, and *Artifact* as data-source elements.

### 3.1.5 Feature Envy

The *Feature Envy* Smell occurs when an abstraction becomes more focused on the methods and fields of other abstractions rather than its functions and variables [48, p. 293]. This issue typically arises when data and functionality that should be kept together are separated across multiple components [49, p. 66]. For instance, a service may contain business logic, while the data required to perform that logic is stored in

Table 3.7: Graphical example and Cypher query for detecting *Data Service* EA Smells

<i>Data Service</i>	
Property	Description
<i>Cypher query</i>	<pre> <b>MATCH</b> p = (a)-[r:AccessRelationship]-(d) <b>WHERE</b> (a:ApplicationService or a:BusinessService or a:TechnologyService) AND (d:DataObject or d:BusinessObject or ↔ d:Artifact) <b>WITH</b> p, a <b>WHERE NOT EXISTS</b>{ (a)-[rl:AssignmentRelationship CompositionRelationship  AssociationRelationship AggregationRelationship  RealizationRelationship]-(s) <b>WHERE NOT</b> s:ApplicationService or s:BusinessService or s:TechnologyService } <b>RETURN</b> p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	

another service. As a result, the service with the business logic often has to invoke the other service frequently to retrieve the necessary data. The *Feature Envy* Smell is often found in conjunction with another Smell known as *Data Service*.

We transfer this Smell in the ArchiMate modeling language by searching for a service that communicates with a data service to complete its task. Table 3.8 illustrates an example of the *Feature Envy* combined with *Data Service* EA Smell in an ArchiMate model. The “Risk Calculating Service” and “Retrieve Customer Data” both rely on the “Access Customer Data” to obtain the required data to fulfill their tasks. Therefore, they are candidates for the *Feature Envy* EA Smell. The KG query detecting *Feature Envy* EA Smell is depicted in Table 3.8. The query looks for the *Serving* and *Association* relations between the feature and service containing the data.

Table 3.8: Graphical example and Cypher query for detecting *Feature Envy* EA Smells

<i>Feature Envy</i>	
Property	Description
<i>Cypher query</i>	<pre> <b>MATCH</b> p=(f)&lt;-[:ServingRelationship  AssociationRelationship]-(a)-[r:AccessRelationship]-(d) <b>WITH</b> p, a <b>WHERE NOT</b> EXISTS{ (a)-[r1:AssignmentRelationship CompositionRelationship  AssociationRelationship AggregationRelationship  RealizationRelationship]-(s) <b>WHERE NOT</b> s:ApplicationService or s:BusinessService or s:TechnologyService } <b>RETURN</b> p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	

### 3.1.6 Shotgun Surgery

The *Shotgun Surgery* Smell appears when a modification in one abstraction requires changes in a bunch of other abstractions [50]. For example, in `APACHE TOMCAT`, there is a method named `isAsync` within the class `AsyncStateMachine`. This method has been used within 31 classes over 48 methods [51]. Any change in the `isAsync` method force changes in all 48 methods.

We can similarly transfer this Smell in the EA domain. Generally, when multiple abstractions depend on a single abstraction, any changes to the single abstraction may force some modifications in the dependent abstractions. In the ArchiMate modeling language, the relation types describing dependency are *Serving*, *Association*, *Access*, and *Influence*.

To detect this Smell via KG query, we check whether a one-to-many relationship exists using the specified relation types. The definition of ‘many’ may differ depending on the organization or enterprise. To accommodate this variability, we establish a default

threshold number, which can be adjusted by the enterprise architects conducting the detection. The default threshold is set to three. When the number of relationships exceeds the threshold, the query identifies the Smell candidates. Table 3.9 presents two graphical examples of the *Shotgun Surgery* EA Smell. In this context, an *Application Service* shared by multiple services and a specific *Requirement* influences four goals. Any changes to the *Application Service* or the *Requirement* could necessitate modifications to the dependent elements, potentially leading to further changes. Table 3.9 illustrates the KG query and the parameter used for detecting the *Shotgun Surgery* EA Smell.

Table 3.9: Graphical example, Cypher query and query options for detecting *Shotgun Surgery* EA Smells

<i>Shotgun Surgery</i>	
Property	Description
<i>Cypher query</i>	<pre> <b>MATCH</b> (a)-[r:ServingRelationship AssociationRelationship  InfluenceRelationship AccessingRelationship]-() <b>WITH</b> a, COUNT(r) <b>AS</b> relCount <b>MATCH</b> p=(a)-[r]-(b) <b>WHERE</b> relCount&gt; \$abstractionsCnt <b>RETURN</b> p </pre>
<i>Query options</i>	<pre> {   "name": "abstractionsCnt",   "type": "number",   "default": 3,   "description": "The number of abstractions dependent on a ↔ single abstraction." } </pre>
<i>Graphical example</i>	

### 3.1.7 Scattered Parasitic Functionality

The *Scattered Parasitic Functionality* Smell occurs when two or more abstractions are responsible for the same concern while some of them also address orthogonal concerns [52]. Spreading a concern among abstractions violates the principle of separation of concerns.

In the realm of EA, we interpret a concern as a function or a goal. We identify this Smell in the ArchiMate model in two ways. First, in the literal sense, we are looking for *Application Components* responsible for the same function. Besides, at least one of them also performs some independent functions. Second, by looking at the analysis elements called *Assessments* that emerge from stakeholder concerns. Each *Assessment* should deliver a goal. If two *Assessments* influence the same goal and one of them also addresses other goals at the same time, then this can also be a sign of the *Scattered Parasitic Functionality* Smell in ArchiMate.

Table 3.10 shows two examples of the *Scattered Parasitic Functionality* EA Smell in the application and motivation layers of ArchiMate, as well as defines KG query detecting this Smell. We consider four types of relationships between elements. These are *Influence*, *Assignment*, *Realization* and *Association*. Assessments influence goals, components are assigned to functions and association is used to describe relationships between elements that are more general or still unspecified.

Table 3.10: Graphical example and the Cypher query for detecting *Scattered Parasitic Functionality* EA Smells

<i>Scattered Parasitic Functionality</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (a)-[r:AssignmentRelationship AssociationRelationship  InfluenceRelationship RealizationRelationship]-(b) <b>WHERE</b> (a:ApplicationFunction OR a:ApplicationService OR ↔ a:Goal) AND (b:ApplicationComponent OR b:Assessment) <b>WITH</b> a, COUNT (b) <b>AS</b> compCount <b>MATCH</b> (a)-[r:AssignmentRelationship AssociationRelationship  InfluenceRelationship RealizationRelationship]-(b) <b>WHERE</b> (b:ApplicationComponent OR b:Assessment) AND compCount&gt;1 <b>WITH</b> a, b <b>MATCH</b> p = (a)--(b)-[r:AssignmentRelationship AssociationRelationship Inf] ↔ l uenceRelationship RealizationRelationship]-(orthogonal) <b>WHERE</b> a&lt;&gt;orthogonal and (orthogonal:ApplicationFunction or ↔ orthogonal:Goal) <b>RETURN</b> p </pre>
Query options	-

Continued on next page

Table 3.10 – continued from previous page

Property	Description
Graphical ex-ample	

### 3.1.8 Deficient Encapsulation

If an abstraction makes its members accessible to other abstractions in a more permissive way, which is not necessarily required, it suffers from *Deficient Encapsulation* Smell [45, pp. 63]. At the code level, these abstractions are classes with public fields. Therefore, they are directly accessible by other classes, although such access is not necessarily required. The `java.awt.Point` class is an example of suffering from the *Deficient Encapsulation* Smell [45, p. 64]. It has two fields for drawing points, named `x` and `y`. These fields are declared public despite having mutators and accessors methods.

In the EA domain, particularly in the ArchiMate modeling language there are no access modifiers. As a result, we need to slightly adjust the definition to apply this concept to ArchiMate. Instead of modifiers, we try to identify sources of information that contain sensitive data while the rules for accessing those data are more lenient than necessary.

In ArchiMate, the *Business Objects*, *Data Objects*, and *Artifacts* elements serve as data source representations. However, there is no distinction between sensitive and non-sensitive data sources. To address this, we can explore identifying a property that designates a data source as confidential. Additionally, we can establish a threshold for permissible access to sensitive data. If the number of accesses exceeds this threshold, we may interpret it as an indication of a *Deficient Encapsulation* EA Smell.

Table 3.11 depicts an example of the *Deficient Encapsulation* EA Smell in ArchiMate. Three elements accessing one *Business Object* having the property “confident.” We have three access relationships between the data source and other elements. Since the default permissible number of accesses for such data source is two, the model suffers from the *Deficient Encapsulation* EA Smell.

The KG query detecting this Smell, along with its parameters, is outlined in Table 3.11. The query requires two parameters. The first parameter specifies the property names and their corresponding values, allowing users to search for property keys, their values, or both. The second parameter sets the maximum permissible communication with the data source. The default properties for identifying sensitive data in the query are “confident” and “classified.” The query checks for these values as property key and value. The default value for permissible access is two. However, users or enterprise architects can redefine these query parameters as needed.

Table 3.11: Graphical example, Cypher query and query options for detecting *Deficient Encapsulation* EA Smells

<i>Deficient Encapsulation</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (n) <b>WHERE</b> n.ClassName ='DataObject' or ↪ n.ClassName='BusinessObject' or n.ClassName='Artifact' <b>WITH</b> n <b>WHERE ANY</b> (x <b>in</b> keys(n) <b>WHERE</b> n[x] <b>IN</b> \$sensitiveProperties OR ↪ toLower(x) <b>IN</b> \$sensitiveProperties) <b>WITH</b> n <b>MATCH</b> (n)-[r:AccessRelationship]-(m) <b>WITH</b> n, count(r) <b>as</b> relCnt <b>MATCH</b> p=(n)-[r:AccessRelationship]-(m) <b>WHERE</b> relCnt&gt;\$permissibleCount <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [   {     "name": "sensitiveProperties",     "type": "list",     "default": ["confident", "classified", "sensitive"],     "description": "The list of property values specifying ↪ an element as confidential."   },   {     "name": "permissibleCount",     "type": "number",     "default": 2,     "description": "The permissible number of elements for ↪ accessing a confidential data source."   } ] </pre>
Continued on next page	

Table 3.11 – continued from previous page

Property	Description
Graphical ex-ample	<pre> graph TD     BO[Business Object]     BA1[Business Actor]     BA2[Business Actor]     BA3[Business Actor]     BO -.-&gt; BA1     BO -.-&gt; BA2     BO -.-&gt; BA3 </pre> <p>Default properties specifying a data source as confidential: confident : true sensible : true</p>

### 3.1.9 Wrong Cuts

The *Wrong Cuts* Smell is originally a microservice architecture Smell. Martin Fowler [53] defines a microservice as an independent unit with a single business capability that can exist and deploy independently. A monolith application should break down to microservices based on business capabilities and not the technical aspects [54, 46]. The *Wrong Cuts* occurs when a business capability is distributed among many microservices, then it violates the separation of concern design principle and increases data-splitting complexity [46]. It also violated the autonomous nature of the microservice, affecting its business capability since it can not fulfill its responsibility and remains dependent on the existence of other microservices.

We can transfer the concept of *Wrong Cuts* into the EA domain by looking for behaviors or business capabilities that are implemented by more than one component. According to the specification of the ArchiMate, *Application Components* are independently deployable, reusable, self-contained units [9]. They perform some functionality and expose their services. Therefore, we can consider them as microservices.

The *Wrong Cuts* EA Smells can occur in ArchiMate once two or more *Application Components* implement the same *Application Function* or *Application Service*. Table 3.12 illustrates an example of *Wrong Cuts* EA Smell in the ArchiMate modeling language, where two *Application Components* are involved in realizing the same *Application Function*. This means that the same functionality is shared in both application components. Hence, both components violate the single responsibility and separation of concern design principles. The KG query used to identify the *Wrong Cuts* EA Smell is depicted in Table 3.12. The query checks for multiple realizations of the same behavior across different *Application Components*.

Table 3.12: Graphical example and Cypher query for detecting *Wrong Cuts* EA Smells

<i>Wrong Cuts</i>	
Property	Description
<i>Cypher query</i>	<pre> <b>MATCH</b> (c:ApplicationComponent)-[r:RealizationRelationship]-(f:  ↪ ApplicationFunction ApplicationProcess ApplicationService) <b>WITH</b> f, COUNT(c) <b>AS</b> compCount <b>MATCH</b> p = ↪ (c:ApplicationComponent)-[r:RealizationRelationship]-(f) <b>WHERE</b> compCount &gt; 1 <b>RETURN</b> p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	


### 3.1.10 Dead Component

The EA Smell *Dead Component* is inspired by the concept of the Code Smell *Dead Code*. *Dead Code* refers to sections of code that are no longer needed or relevant but remain in the project. This typically happens during refactoring, when certain code segments become obsolete or redundant but are not removed for safety reasons [45, p. 51] [55, p. 49]. These outdated sections linger in the codebase, consuming resources and potentially causing confusion, despite not contributing to the functionality of the application.

In the EA domain, the name of the Smell has been adjusted to *Dead Component* [7]. It might refer to a component or module in an EA system that is no longer in use but is retained for various reasons, even though it serves no purpose in the current system.

We can detect the *Dead Component* EA Smell via KG query when a component has neither incoming nor outgoing edges. Table 3.13 depicts a graphical example of the *Dead Component* EA Smell in the ArchiMate model and the KG query for detection.

Table 3.13: Graphical example and Cypher query for detecting *Dead Component* EA Smells

<i>Dead Component</i>	
Property	Description
<i>Cypher query</i>	<pre> MATCH p = (n) WHERE NOT (n) -- () RETURN p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	

### 3.1.11 Vendor Lock-In

The *Vendor Lock-In* is an anti-pattern when a software product becomes strongly dependent on a technology provided by a vendor [55, pp. 91-92]. Being overly reliant on a vendor's technology impacts the software quality, and the cost of switching to another technology becomes so high that the software is forced to stick with the current vendor. Generally, an Isolation layer is a solution to avoid such a strong coupling between the product and vendor's implementation [56].

In the EA domain, internal and external services expose their functions to users and other services. A vendor product can be an external service, and an interface can act as an isolation layer between the vendor's technology and the rest of the organization.

In ArchiMate, there is no clear distinction between external and internal services. Hence, it is reasonable to presume that a property or element's designation should indicate the service as an external offering from a vendor. The element name should reflect the vendor's product or the element should possess a property signifying its external nature. After identifying such an element within the model, our next step is to verify whether it has an interface functioning as an isolation layer. If such layer is missing, the model suffers from the *Vendor Lock-In* EA Smell.

Table 3.14 illustrates an example of a service offered by a vendor. The external service "Document Management Back-Up" has a property named "vendor" with a value "AWS." Since there is no interface between the vendor project and elements in the model, there is a risk that the whole organization is tightly coupled to the vendor's project, which could be a sign of the *Vendor Lock-In* EA Smell.

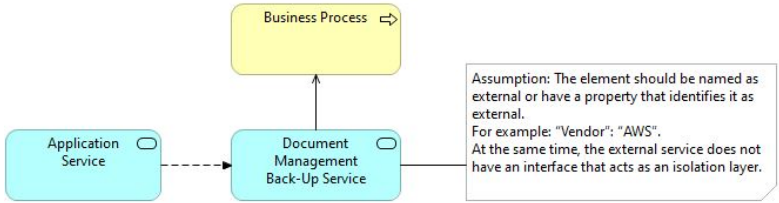
Table 3.14 depicts the KG query and the query parameters for the Smell detection.

The query checks whether or not a property or an element name exists indicating the element is a vendor product. If such an element exists and does not have an isolation interface, then the element is a candidate for *Vendor Lock-In* EA Smell. The query has two parameters. The first parameter is a property name that indicates the element as external. The default values are “external” and “vendor.” The second parameter is the vendor’s name. The default values are “IBM,” “Microsoft,” and “Amazon.” The query searches for these parameters in property names and their values. Nevertheless, a list of words to specify vendor products can be defined by enterprise architects in order to extend or replace the default parameters.

Table 3.14: Graphical example, Cypher query and query options for detecting *Vendor Lock-In* EA Smells

<i>Vendor Lock-In</i>	
Property	Description
Cypher Query	<pre> MATCH p = (n) WHERE n.name in \$external AND NOT EXISTS{   (n)--(i) where toLower(i.classname) contains "interface" } RETURN p UNION MATCH p = (n) where ANY (x in keys(n) WHERE n[x] IN \$vendors OR toLower(x) ↔ IN \$external) AND NOT EXISTS{   (n)--(i) where toLower(i.classname) contains "interface" } RETURN p </pre>
Continued on next page	

Table 3.14 – continued from previous page

Property	Description
Query options	<pre> "queryParams": [ {   "name": "external",   "type": "list",   "default": ["external", "vendor"],   "description": "The name or the property of the element ↪ that marks it as external." }, {   "name": "vendors",   "type": "list",   "default": ["IBM", "Microsoft", "Amazon"],   "description": "The name or the property of the element ↪ that marks it as external." } ] </pre>
Graphical ex-ample	

### 3.1.12 No Legacy

The *No Legacy* anti-pattern refers to a situation where an organization decides to rebuild a system from scratch instead of modernizing the existing legacy systems and integrating them into the new one [57]. Developing and implementing new systems can be a time-consuming and expensive process that may require excessive investments. Furthermore, it may not be necessary if the legacy systems still function well and can be smoothly integrated in the new system. Often, this anti-pattern is combined with the *Big Bang* anti-pattern where the entire system is built all at once [58].

While the ArchiMate modeling language lacks a dedicated syntax or element for denoting legacy components, these can still be inferred based on their names or properties. Additionally, we can pinpoint components that have a limited duration as they are intended to be replaced by new ones.

In ArchiMate, a *plateau* is an element type representing a stable state that exists for a specific period. Table 3.15 illustrates the legacy system “Main Frame CRM,” which is set to replace a new component, “Microservice CRM.” The terms “Baseline” and “Vision”

represent *Plateau* elements. The baseline reflects the current state of the legacy system “Main Frame CRM,” providing a reference point for comparison with the new system’s envisioned state, “Vision.” These states remain relevant only temporarily until the legacy system is fully replaced. The replacement process is managed by the “Replace Main Frame” element, categorized as a *Work Package*. A *Work Package* contains a set of actions aimed at completing a specific task within a defined timeframe, in this case, creating the “Microservice CRM.”

The KG query begins by identifying elements in the model that possess a specific name or property associated with legacy components. If such an element exists, they are flagged as legacy components. The query then evaluates whether these legacy components have relationships with relevant *Plateau* and *Work Package* elements. A default query parameter, “legacy,” is predefined to detect legacy components. However, enterprise architects can modify this parameter to specify a custom list of names for identifying legacy elements in their enterprise models. Table 3.15 illustrates the KG query and its parameters for detecting the *No Legacy* EA Smell.

Table 3.15: Graphical example, Cypher query and query options for detecting *No Legacy* EA Smells

<i>No Legacy</i>	
Property	Description
Cypher Query	<pre> MATCH p=(m)--(n) WHERE any (name in \$legacy where toLower(m.name) CONTAINS ↪ toLower(name)) OR ANY (x in keys(m) WHERE m[x] IN \$legacy OR toLower(x) IN ↪ \$legacy) WITH m MATCH p = (m)--(n:WorkPackage Plateau) return p </pre>
Query options	<pre> "queryParams": [ {   "name": "legacy",   "type": "list",   "default": ["legacy"],   "description": "The element names or property values ↪ specifying legacy components." }] </pre>

Continued on next page

Table 3.15 – continued from previous page

Property	Description
Graphical ex-ample	<p>Element's name or attributes specifying legacy components. Example: legacy: true type: legacy</p>

### 3.1.13 Warm Bodies

When working on a software project, it is common for developers with different skill sets and productivity levels to collaborate in teams. However, if the team size becomes too large, coordination between members becomes difficult, resulting in less efficient decision-making and reduced productivity. Over time, it also leads to a loss of shared insight. This behavior is defined as an anti-pattern called *Warm Bodies* [55, p. 101].

In ArchiMate, we can represent the team concept by *Actor* or *Work package* elements. An *Actor* can represent several people, and a *Work package* can accommodate a group of employees. Work packages in ArchiMate are similar to agile iterations and represent designed actions for achieving a specific task within a given timeframe.

To determine the team size in these elements, we assume the existence of a property specifying the number of team members. If such a property exists and the team size exceeds a defined threshold, we consider the element as a potential instance of the *Warm Bodies* EA Smell.

Table 3.16 demonstrates a graphical example of the *Warm Bodies* EA Smell in the ArchiMate modeling language. In this example, the element “Hardware Update” includes a property named “Members,” with a value indicating a team size of seven. According to Edwards et al. [59], the optimal team size is four. If we set the threshold for the ideal team size at four, the “Hardware Update” element qualifies as a candidate for the *Warm Bodies* EA Smell, because its team size exceeds this threshold.

To detect the presence of the *Warm Bodies* EA Smell, our provided KG query evaluates

elements labeled as *Actor* or *Work Package* with properties indicating a team size greater than the defined threshold, which is set to four by default. If such elements are found, the model suffers from the *Warm Bodies* EA Smell.

We define a list of parameters to include potential property names representing team size, such as “Members,” “Personnel,” and “Staff,” along with a numerical threshold. By default, the threshold is set to four, but enterprise architects can adjust these parameters to align with their organization’s specific needs. Table 3.16 illustrates the KG query and its customizable options for detecting this EA Smell.

Table 3.16: Graphical example, Cypher query and query options for detecting *Warm Bodies* EA Smells

Warm Bodies	
Property	Description
Cypher query	<pre>MATCH (n:Actor WorkPackage) where ANY (x in keys(n) WHERE x IN \$staffProperty and n[x] &gt; \$staffSize) RETURN n</pre>
Query options	<pre>"queryParams": [ {   "name": "staffProperty",   "type": "list",   "default": ["Members", "staff", "Personnel"]   "description": "Property name(s) specifying the size of a team." }, {   "name": "staffSize",   "type": "number",   "default": 4,   "description": "The number of people in an Actor or Work Package element." }]</pre>
Graphical example	

### 3.1.14 Combinatorial Explosion

The *Combinatorial Explosion*, also referred to as the *Missing Encapsulation* Smell, arises when there is a lack of abstraction or hierarchy to encapsulate implementation variations [45, pp. 78-79]. This Smell is typically manifested in two ways. First, when a client is directly dependent on multiple service variations, any modification in existing or creating new services affects the client. Second, whenever there is an effort to create a new service type in the hierarchy, an unnecessary amount of classes are produced, leading to an “explosion of classes.” Such a design shortcoming results in a great amount of duplicated codes with tiny variations in data and behavior.

In the EA domain, the *Combinatorial Explosion* Smell can also occur as a subtle form of duplication. Table 3.17 demonstrates a graphical example of *Combinatorial Explosion* EA Smell in the ArchiMate modeling language. Two application services implement identical behavior but rely on different data sources. According to the ArchiMate specification, if two abstractions realize the same behavior, each can fully implement the behavior. However, both services provide identical functionality in this scenario while accessing separate data sources, leading to duplication. This design flaw results in redundant components with slight variations in data and behavior. The existence of *Combinatorial Explosion* in the EA domain may have different reasons. For example, two services provide the same functionality but get data from two different sources because the data cannot be shared. In such cases, the duplication of services is unavoidable for the enterprise. Table 3.17 illustrates the KG query for detection of *Combinatorial Explosion* EA Smell.

Table 3.17: Graphical example, Cypher query and query options for detecting *Combinatorial Explosion* EA Smells

<i>Combinatorial Explosion</i>	
Property	Description
<i>Cypher query</i>	<pre> MATCH (a)-[r:RealizationRelationship]-&gt;(b) WITH b, COUNT (a) AS implCount MATCH p = (d)-[r1:AccessRelationship]-(a)-[r:RealizationRelationship]-&gt;(b) WHERE implCount&gt;1 RETURN p </pre>
<i>Query options</i>	-

Continued on next page

Table 3.17 – continued from previous page

Property	Description
Graphical example	

### 3.1.15 Stovepipe System

It is common for an enterprise to have multiple systems designed independently at every level of the organization. The *Stovepipe System* anti-pattern emerges when there is little or no coordination and planning among these systems [55]. Thus, the design and development of the different subsystems take place independently, leading to the creation of isolated subsystems with limited interoperability and inhibited reusability. Since common mechanisms for implementing subsystems are missing, reusing the existing abstractions in different subsystems becomes inaccessible or invisible. This shortcoming causes the creation of duplicate abstractions within subsystems, which is a sign of *Duplication Abstraction* Smell [45, p. 54]. At first glance, they may not seem identical to existing ones because they may have different names and interfaces. For this reason, the occurrence of Smell *Alternative Classes with Different Interfaces* becomes more conspicuous. In order to transform the mentioned Smell into the EA domain, we slightly renamed it by replacing the term class with a component, which becomes *Alternative Components with Different Interfaces*.

In ArchiMate, we can detect this Smell by looking for services that provide similar functionality but have different names or interfaces. In order to detect a duplicate abstraction, one can check for identical implementations and terms in the model. We can not evaluate identical implementations within a graph using KG query because it requires much more low-level details. Therefore, a process manager is required to check for identical implementation between components. However, we can evaluate identical names by checking if two components have the same or similar names. Table 3.18 illustrates an example of *Stovepipe System* or *Alternative Components with Different Interfaces* EA Smell in the ArchiMate modeling language. The components “Legal Expense CRM System” and “General CRM System” appear to serve similar purposes but use different interfaces, indicating duplication.

### 3. EA SMELLS ANALYSIS

The KG query and the query parameters for detecting this Smell are shown in Table 3.18. We use the Jaro-Winkler distance algorithm [60] to inspect the similarity of strings. The algorithm produces a similarity score ranging from zero to one, where zero indicates an exact match, and one signifies no similarity between the strings. We set a threshold of “0.4,” meaning that any similarity score between 0 and 0.4 is considered a candidate for duplication.

Table 3.18: Graphical example, Cypher query and query options for detecting *Stovepipe System* EA Smells

<i>Stovepipe System</i>	
Property	Description
<i>Cypher query</i>	<pre> <b>MATCH</b> p1=(a)--(b:ApplicationInterface BusinessInterface Techno ↪ logyInterface), ↪ p2=(c)--(d:ApplicationInterface BusinessInterface Technolo ↪ gyInterface) <b>WHERE</b> a&lt;&gt;c and b&lt;&gt;d and apoc.text.jaroWinklerDistance ↪ (a.name, c.name) &lt; \$score <b>UNWIND</b> [p1,p2] <b>as</b> p <b>RETURN</b> p </pre>
<i>Query options</i>	<pre> <b>"queryParams":</b> [ {   <b>"name":</b> "score",   <b>"type":</b> "number",   <b>"default":</b> 0.4,   <b>"min":</b> 0.0,   <b>"max":</b> 1.0,   <b>"step":</b> 0.1,   <b>"description":</b> "List of synonyms to check whether ↪ components are similiar." } ] </pre>
<i>Graphical example</i>	<pre> graph BT     A[Legal Expense CRM System] --&gt; B[Get Lawyer]     C[General CRM System] --&gt; D[Get Customer Data] </pre>

### 3.1.16 Nanoservices

A nano service is a fine-grained service with a simple operation that implements only a part of an abstraction and exchanges a large amount of information with a group of services to complete an abstraction task [61]. A higher number of services can be advantageous when it comes to scalability and changeability, but maintainability and communication between these services cause considerable overhead. Due to numerous interactions between nano services, often a cyclic path emerges between these kinds of services [54]. This anti-pattern often occurs in microservice architecture when a monolith is divided into too many small services [54].

We translate this anti-pattern in the EA domain by considering a business capability as an abstraction consisting of many small services that are tightly coupled and fulfill the same business capability. Each service implements only a part of that abstraction, and they communicate with each other to fulfill that business capability as a whole.

Table 3.19 depicts an example of three nano-services interacting with each other to fulfill the more abstract application service. A cyclic path between these services is also apparent. Table 3.19 illustrates the KG query identifying this Smell. The query checks if an abstraction that consists of many small services exists. In other words, the query looks for aggregations or composition relations between a single abstraction and services implementing that abstraction. Besides, the query checks whether a cyclic path exists between those services implementing the abstraction. We set the default threshold to three for the number of services, which are part of the abstraction. If the number of services exceeds the threshold and there is a cyclic path between them, then the model suffers from *Nanoservices* EA Smell. We also use the same threshold to specify the maximum length of the cyclic path between those nano-services. The threshold specifying the number of nanoservices, which is also used for detecting cyclic paths, is defined as a query parameter.

Table 3.19: Graphical example, Cypher query and query options for detecting *Nanoservices* EA Smells

<i>Nanoservices</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (a)-[r:CompositionRelationship]-(n) <b>WITH</b> a, COUNT(n) <b>AS</b> cnt <b>WHERE</b> cnt &gt;= \$nanoServiceCount <b>MATCH</b> p=(a)-[r:CompositionRelationship]-(n) <b>RETURN</b> p <b>UNION</b> <b>MATCH</b> (a)-[r:CompositionRelationship]-(n) <b>WITH</b> a, COUNT(n) <b>AS</b> cnt <b>WHERE</b> cnt &gt;= \$nanoServiceCount <b>MATCH</b> p1=(a)-[r:CompositionRelationship]-(n) <b>WITH</b> p1, nodes(p1) <b>AS</b> nodes <b>CALL</b>   ↳ apoc.nodes.cycles(nodes, {maxDepth:   ↳ \$nanoServiceCount}) <b>YIELD</b> path <b>WITH</b> path <b>AS</b> p2, p1 <b>UNWIND</b> [p1,p2] <b>AS</b> p <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [   {     "name": "nanoServiceCount",     "type": "number",     "default": 3,     "description": "Number of nanoservices."   } ] </pre>
Graphical ex-ample	

### 3.1.17 Overgeneralization

The EA Smell *Overgeneralization* originates from the *Ambiguous Interface* Smell [7]. According to Garcia et al. [52] the *Ambiguous Interface* Smell is defined as a component

that handles multiple functions but exposes only a single public service or method. All incoming requests are routed through a single generic entry point, where the component filters and directs them to the appropriate internal services. This issue is particularly prevalent in publish-subscribe patterns [62, 52]. Concealing a component's services behind an ambiguous interface hinders static analysis, making the system more challenging to analyze and comprehend.

In the context of EA, this Smell is relabeled as *Overgeneralization* [7]. In EA design, a component may become overly generalized and adaptive, providing more functionalities than necessary for maximum reusability. Since different functionalities are not exposed via an interface and remain hidden, all requests must pass through a single provided interface. As a result, users are required to perform additional tasks and create uniform request objects, violating the “one and only one” principle[7].

In ArchiMate, detecting this Smell involves identifying components that perform numerous functions and interact with various services but expose only one service through their interface. Table 3.20 provides a graphical example of the *Overgeneralization* EA Smell in the ArchiMate modeling language. An *Application Component* exposes a single service via its interface, yet it executes three internal functions and communicates with two external services. Since ArchiMate is an abstraction language with limitations in representing detailed interface and service information, it cannot specify the number of entry points in an interface or the public methods offered by a service.

The KG query to detect this Smell is outlined in Table 3.20. The detection parameter is based on the sum of the provided functions and services. A default threshold of five is set, meaning that if the total number of functions and services exceeds this threshold, the component will be flagged as a potential candidate for the *Overgeneralization* EA Smell.

Table 3.20: Graphical example, Cypher query and query options for detecting *Overgeneralization* EA Smells

<i>Overgeneralization</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (i)-[r1:CompositionRelationship]-(c) <b>MATCH</b> (c)-[r2:RealizationRelationship]-(s) <b>WITH</b> c, COUNT(s) <b>AS</b> exposedServiceCnt, COUNT(i) <b>AS</b> interfaceCnt <b>WHERE</b> exposedServiceCnt=1 AND interfaceCnt=1 <b>MATCH</b>   ⇨ (c)-[r:AssociationRelationship AssignmentRelationship]-(f) <b>WITH</b> c, COUNT (f) <b>AS</b> functionOrServiceCnt <b>where</b> functionOrServiceCnt &gt; \$functionOrServiceCnt <b>MATCH</b> p = (c)-[r:AssociationRelationship AssignmentRelationship]   ⇨ p RealizationRelationship CompositionRelationship]-(f) <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [{"name": "functionOrServiceCnt",   "type": "number",   "default": 5,   "description": "The number of functions that performed by   ⇨ the overgeneralized componennt." }] </pre>
Graphical ex-ample	

### 3.1.18 Sand Pile

One common approach to implementing Service-Oriented Architecture (SOA) is to have one elementary service per software component [58]. However, this can lead to numerous small components sharing the same data, which leads to the *Sand Pile* Smell [57]. This Smell emerges when a service's atomic capabilities are divided into small, independent components that all access a shared data source.

We translate this Smell into the EA domain and apply the exact definition for its detection. We focus on identifying *Application Component* elements with exposed services that rely on the same data source. An example of the *Sand Pile* EA Smell is illustrated in Table 3.21, where multiple components interact with the same *Data Object* element named “Data Source.” The query for identifying this Smell is presented in Table 3.21, which identifies components with exposed services that access shared data.

Table 3.21: Graphical example and Cypher query for detecting *Sand Pile* EA Smells

<i>Sand Pile</i>	
Property	Description
<i>Cypher query</i>	<pre> MATCH (c)-[r2:AccessRelationship]-(d:DataObject) WITH d, count(c) as cntCmps WHERE cntCmps&gt;1 MATCH p=(s)-[r1:RealizationRelationship]-(c)-[r2:AccessRelationship]-(d:DataObject) WITH c, count(d) as dataCnt where dataCnt=1 MATCH p=(s)-[r1:RealizationRelationship]-(c)-[r2:AccessRelationship]-(d:DataObject) RETURN p           </pre>
<i>Query options</i>	-
<i>Graphical example</i>	

### 3.1.19 Missing Abstraction

In software development, the lack of clear conceptual boundaries for components is a common problem due to the lack of abstraction. Data and behavior that should be highly cohesive are spread across multiple components, which violates the principles of encapsulation and modularization [45]. The lack of the concept of abstraction also causes reusability problems [63]. At the code level, the *Missing Abstraction* Smell arises by either one or combination of two other Smells, *functional composition* and *Primitive Obsession* [7].

In the EA domain, the model should also contain abstract elements to illustrate a more general point of view. The absence of such elements may indicate that the model suffers from *Missing Abstraction* EA Smell. Salentin and Hacks [7] describe the detection of *Missing Abstraction* in the EA model by identifying multiple elements of the same type that are not aggregated into a more general super element. Figure 3.1 illustrates an example of abstraction in the ArchiMate modeling language. The “Insurance Customer Service” is an abstract element of type *Business Service* aggregating four concrete business services. The concrete business services are related and fulfill a more abstract concept called insurance service. Table 3.22 illustrates an example of *Missing Abstraction* EA Smell. As can be seen, numerous services and components exist, but they lack aggregation into a more general element. Consequently, our proposed model is affected by the *Missing Abstraction* EA Smell.

We define three steps for detecting this Smell in the ArchiMate model. First, elements belonging to the same business capability must be grouped together. Second, since *Aggregation* and *Composition* typically link elements of the same type, a certain amount of elements of the same type must exist within a group. Besides, we consider the minimum number of relationships in each group. Groups with a small number of relationships are not considered candidates for Smell detection. Finally, after filtering the candidate groups, in the third step, we check whether *Aggregation* or *Composition* relationships exist between elements of the same group. If these relations are missing, then the model suffers from *Missing Abstraction* EA Smell. If such type of relationships exist then we calculate a ratio by dividing the number of all edges by the number of aggregations per group. If the ratio is below the threshold, then we can assume the EA model suffers from *Missing Abstraction* EA Smell.

For grouping elements based on business capabilities, we can assume there is a high modularity between those elements that belong to the same business capability. Graph clustering is an approach used for grouping elements in the graph. We use a community detection algorithm known as the Louvain method [64] for grouping nodes in the graph. The Louvain algorithm groups elements together with a high modularity ratio, maximizing a modularity score for each community. This score measures how well nodes are assigned to communities based on their modularity.

Neo4j provides a Graph Data Science library (GDS)<sup>2</sup> for using graph algorithms such

<sup>2</sup><https://neo4j.com/docs/graph-data-science/current/>

as centrality, community detection, similarity, etc. We use the GDS library for grouping elements based on the Louvain method. This algorithm adds an extra property called *community* to all nodes. Elements are grouped based on this property. Listing 4 demonstrates two procedures: one for creating the projection and another for executing Louvain method. Both procedures will be called by our detection platform (Chapter 5) before executing the KG query for *Missing Abstraction* EA Smell detection.

```
#Graph projection considering all nodes and relationships

CALL gds.graph.project('ea-model', // graph name for projection
  '*', // All nodes
  '*'); // All relationships

#Execution of the Louvain method to write a new property called "community"
↪ for clustering

CALL gds.louvain.write('myGraph', { writeProperty: 'community' })
YIELD communityCount, modularity, modularities
```

Listing 4: Graph projection and executing the Louvan algorithm

The key aspect lies in defining the query parameters, specifically determining the appropriate group size, the number of aggregation relationships, and the elements of the same type that must exist within the group for it to qualify for Smell detection. These parameters may vary depending on the model size or enterprise concerns. To provide flexibility, we have defined these values as variables that can be adjusted by enterprise architects or users. We provide three query parameters with default threshold values. The first parameter is the number of the elements of the same type within the communities, and the default value is three. The second parameter is about how big the cluster should be. We only consider clusters from a certain number of edges. Otherwise, the cluster is not considered a candidate for Smell detection. The default value is ten. Finally, after filtering the communities based on the first and second parameters, we check whether some aggregation or composition relationships exist between elements. If this is the case, then the query computes the ratio for each community by dividing the number of total edges by the number of aggregations. A threshold value of three is set for this ratio. If the ratio is below the threshold, the community is considered a candidate for the *Missing Abstraction* EA Smell. Table 3.22 demonstrates the query and corresponding parameters for detection.

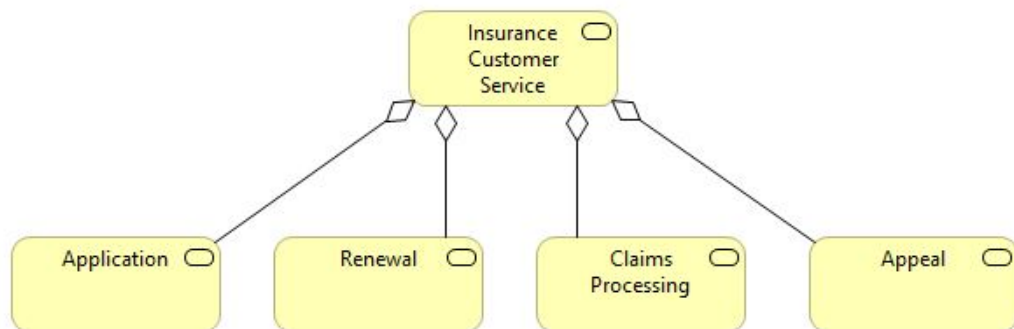


Figure 3.1: An example of an abstract and many aggregated sub elements in an ArchiMate model

Table 3.22: Graphical example, Cypher query and query options for detecting *Missing Abstraction* EA Smells

<i>Missing Abstraction</i>	
Property	Description
Cypher Query	<pre> MATCH (n) with count (n) as nodes, n.ClassName as label, n.community as communities where nodes &gt; \$countOfNodesOfSameType with distinct communitiesv0 as communities MATCH (t)-[st]-&gt;(s) where t.community=s.community and t.community in communities with count (st) as relcount, communities match (a)-[r]-&gt;(b) where relcount &gt; \$relationsCount and a.community=b.community ↔ and b.community in communities optional match p1=(s)-[r:AccessRelationship]-&gt;(t) where s.community=t.community and s.community in communities with count (p1) as aggCnt, relcount, communities match p=(a)-[r1]-&gt;(b) where a.community=b.community and a.community in communities and (aggCnt=0 or relcount/aggCnt &gt; \$ratio) return p </pre>
Continued on next page	

Table 3.22 – continued from previous page

Property	Description
Query options	<pre> "queryParams": [{"name": "countOfNodesOfSameType",   "type": "number",   "default": 3,   "description": "Minumum number of nodes from same type ↔ within a community." }, {"name": "relationsCount",   "type": "number",   "default": 10,   "description": "Minimum number of relations within a ↔ community." }, {"name": "ratio",   "type": "number",   "default": 3,   "description": "Ratio threshold" }] </pre>
Graphical ex-ample	

### 3.1.20 Strict Layer Violation

In software engineering, layers are defined based on conventions rather than language concepts [65]. For example, the *Data Access Object* (DAO) structural pattern isolates the persistence layer from the application and service layers. If the service layer directly communicates with the persistence layer, it indicates the presence of *Strict Layers Violation* Smell.

We can transfer this Smell with a minor modification into the EA domain. In ArchiMate, there are three core layers: *Business*, *Application*, and *Technology*. The ArchiMate specification outlines specific relationships among these core layers, namely *Realization*

and *Serving* relationships [9]. The *Serving* relationship applies when an element from the upper layer utilizes services provided by the lower layer. In contrast, the *Realization* relationship is relevant when a lower-level element realizes or implements a more abstract concept from the upper layer.

Figure 3.2 illustrates the permissible relationships among the various layers. The application and technology layers are required to communicate with the business layer using these two specific types of relations. Although other relation types are not syntactically prohibited, their use is generally discouraged. Table 3.23 provides an example of a *Strict Layers Violation* EA Smell within the ArchiMate model. A *Business Actor* element is directly associated with an *Application Service*. Since the relation between those elements is neither *Serving* nor *Realization* relationship, the model suffers from *Strict Layers Violation* EA Smell.

The KG query designed to detect this Smell verifies the relationships between layers, excluding the *Realization* and *Serving* relationships. If such a path exists, it indicates a *Strict Layers Violation* EA Smell. The KG query and a graphical example illustrating this Smell are presented in Table 3.23.

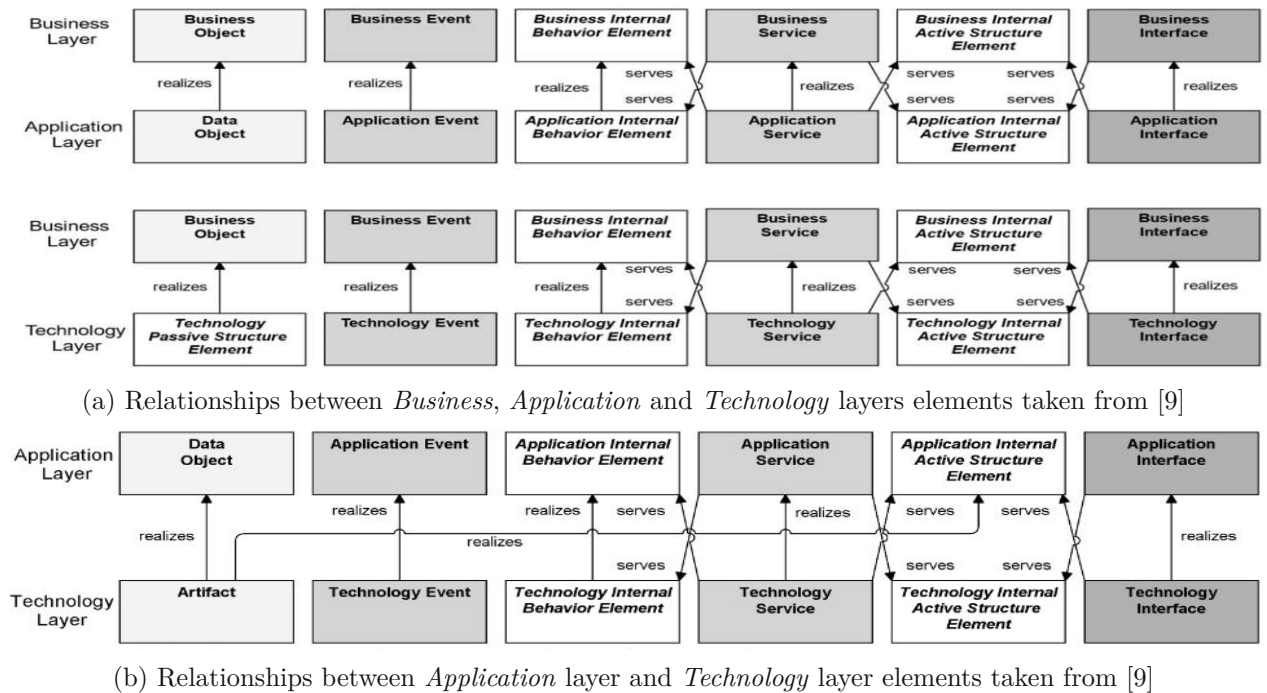
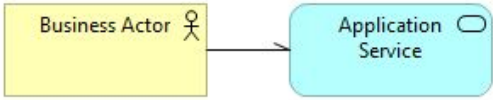


Figure 3.2: Permitted relationships between the core layers of ArchiMate

Table 3.23: Graphical example and Cypher query for detecting *Strict Layer Violation* EA Smells

<i>Strict Layer Violation</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> <b>MATCH</b> p = (a)-[r]-(b) <b>WHERE</b> a.ArchimateLayer &lt;&gt; ↪ b.ArchimateLayer <b>AND NOT</b> (r.Label = 'ServingRelationship' ↪ or r.Label = 'RealizationRelationship') <b>return</b> p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	 <pre> graph LR     BA[Business Actor] --&gt; AS[Application Service] </pre>

### 3.1.21 Message Chain

The *Message Chain* Smell at the code level indicates the sequence of method calls to perform a task [50]. A message chain leads to tight coupling and affects code robustness and testability. This Smell is also known as *Service chain*. At this level, it refers to sequential service calls.

We can adapt this Smell to the EA domain by identifying consecutive service chains within the ArchiMate model. Table 3.24 provides an example of a Service chain with a length of five. We consider any message chain longer than four to be a potential candidate for the *Message Chain* EA Smell.

The KG query designed to detect the *Message Chain* EA Smell is illustrated in Table 3.24. By default, this query considers a relation length between four and ten. The relation length is defined as a query parameter and can be adjusted by enterprise architects. We set an upper limit on the length to avoid performance issues in large, densely connected graphs. Since there is no possibility to specify the lengths of the relationship as a parameter in the Cypher native query language, we utilize a method from the `apoc` library called `apoc.path.expand`.

Table 3.24: Graphical example, Cypher query and query options for detecting *Message Chain* EA Smell

<i>Message Chain</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> MATCH (p) where toLower(p.ClassName) CONTAINS 'service' CALL apoc.path.expand(p, "TriggeringRelationship&gt; FlowRelation  ↳ ship&gt; AssociationRelationship&gt; AssignmentRelationship&gt;", ↳ "+BusinessService +ApplicationService +TechnologyService", ↳ \$minLevel, \$maxLevel) Yield path return path as p </pre>
<i>Query options</i>	<pre> "queryParams": [{"name": "minLevel",   "type": "number",   "default": 4,   "description": "The minimum length that the query considers ↳ for detecting a message chain." }, {"name": "maxLevel",   "type": "number",   "default": 10,   "description": "The maximum length that the query considers ↳ for detecting a message chain." }] </pre>
<i>Graphical example</i>	<pre> graph LR     A[Registration Service] --&gt; B[Information Retrieval]     B --&gt; C[Information Validation]     C --&gt; D[User Registration]     D --&gt; E[User Notification] </pre>

### 3.1.22 Shared Persistency

According to microservice Smells, *Shared Persistency* arises when multiple services access the same data source [46]. This Smell undermines the autonomy of microservices.

To transfer this concept into the EA domain, we examine data sources that are shared among multiple abstractions. In ArchiMate, the elements *Business Object* and *Data Object* represent data sources. The ArchiMate specifications define an *Artifact* element within the technology layer, which can represent files, documents, and database tables. Consequently, an *Artifact* element could also represent a data source.

To detect this Smell in ArchiMate, we check if two or more abstractions access the same data source. Table 3.25 provides an example of the *Shared Persistency* EA Smell in the

ArchiMate modeling language. A *Data Object* element has been accessed by multiple *Application Components*. Table 3.25 outlines the KG query detecting this Smell. The query evaluates whether *Business Object*, *Data Object*, or *Artifact* elements are accessed by multiple abstractions.

Table 3.25: Graphical example and Cypher query for detecting the *Shared Persistency* EA Smells

<i>Shared Persistency</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> <b>MATCH</b> p=(a:BusinessObject DataObject Artifact)-[r:AccessRelationship]-&gt;(b) <b>WITH</b> a, COUNT (r) <b>as</b> cnt <b>WHERE</b> cnt&gt; 1 <b>MATCH</b> p = (a)-[r:AccessRelationship]-&gt;(b) <b>RETURN</b> p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	

### 3.1.23 Chatty Service

When a service requires multiple interactions with other services to complete a single task, it is considered “chatty” and leads to increased overhead [66]. Typically, a chatty service involves numerous fine-grained operations, which can result in maintainability and performance issues [61].

We can directly translate this Smell into the EA domain by identifying services that engage in an excessive number of interactions with other services. A graphical example depicting the *Chatty Service* EA Smell is provided in Table 3.26.

To detect this Smell within the KG, we analyze the relationships between services and set a threshold for an acceptable number of interactions. If the number of interactions exceeds this threshold, it could be a sign of the *Chatty Service* EA Smell in the model. By default, the interaction threshold is set to three. However, enterprise architects can adjust this value to align with their requirements. The KG query for identifying the *Chatty Service* Smell is detailed in Table 3.26. While the KG query offers valuable insights,

it alone is insufficient to determine whether a service is excessively chatty. Dynamic analysis is also necessary to fully evaluate the extent of the issue.

Table 3.26: Graphical example, Cypher query and query options for detecting *Chatty Service* EA Smells

<i>Chatty Service</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> <b>MATCH</b> (a)-[r]-(b) <b>WHERE</b> toLower(a.ClassName) contains 'service' and toLower(b.ClassName) contains 'service' <b>WITH</b> a , COUNT (r) <b>as</b> cnt <b>WHERE</b> cnt&gt; \$interactionCnt <b>MATCH</b> p = (a)-[r1]-(b) <b>WHERE</b> toLower(a.ClassName) contains 'service' and toLower(b.ClassName) contains 'service' <b>return</b> p </pre>
<i>Query options</i>	<pre> "queryParams": [{"name": "interactionCnt",   "type": "number",   "default": 3,   "description": "The permissible number of related services ↔ that the chatty service communicate with." }] </pre>
<i>Graphical example</i>	

### 3.1.24 Infinite Loop

The *Infinite Loop* is a BPM anti-pattern that runs indefinitely because either a termination condition is missing or the condition is never fulfilled [67].

The *Infinite Loop* EA Smell is a specific type of cyclic dependency. The key distinction is that, unlike typical cyclic dependencies, where the process can exit the cycle after a certain number of iterations, the process in this case is trapped in an infinite loop with no opportunity for termination. If the execution is not aborted externally, then it will run forever.

We transfer this Smell into the EA domain by considering two possible cases in the ArchiMate modeling language. In the first case, we can consider the *Infinite Loop* in a KG as a small sub-graph with at least one cyclic path and no exiting edges that escape the loop. In the second case, there could also be an edge that escapes the loop, but the process will never go through that edge for different reasons. For example, the condition is never met. We cannot detect the second case via the KG query, and a dynamic analysis is required. Therefore, the second detection case is beyond the scope of this thesis, and we consider only the first case. Table 3.27 demonstrates an infinite loop in the ArchiMate modeling language. As is evident, no edge can escape the loop. Hence, the process will run forever until it is terminated externally.

Table 3.27 illustrates the KG query for identifying the *Infinite Loop* EA Smell. By default, the query examines cyclic paths with a maximum length of five edges. This upper limit is set to mitigate potential performance issues when querying large, densely connected graphs. However, users or enterprise architects can adjust this parameter as needed. The query is configured to identify cycles of up to five edges, as well as verify whether any edge can escape the loop.

Table 3.27: Graphical example, Cypher query and query options for detecting *Infinite Loop* EA Smells

<i>Infinite Loop</i>	
Property	Description
Cypher Query	<pre> MATCH (a) WITH collect(a) AS nodes CALL ↪ apoc.nodes.cycles(nodes, {maxDepth: \$maxDepth}) YIELD path ↪ WITH path AS p WITH nodes(p) AS loopSet WHERE NOT EXISTS{     MATCH (a)--(b)     WHERE a IN loopSet AND NOT b IN loopSet } Match p=(m)--(n) WHERE m IN loopSet AND n IN loopSet return p </pre>
Query options	<pre> "queryParams": [{"name": "maxDepth",   "type": "number",   "default": 5,   "description": "The length of a cyclic path that that an ↪ infinite loop includes." }] </pre>
Continued on next page	

Table 3.27 – continued from previous page

Property	Description
Graphical ex- ample	

### 3.1.25 The God Object

A *God Object*, or *Multiservice*, refers to a component or service that performs a wide range of operations across different business functions and abstractions. However, the lack of cohesion among these operations can impede reusability and lead to overload. Figure 3.3 provides an example of the *God Object* web service with functions fulfilling various services.

The concept of the *God Object* can also be practically applied and translated into the EA domain. We consider an *Application Component* that implements multiple functions exposed by a single application service. An illustration of the *God Object* in the ArchiMate modeling language is presented in Table 3.28. The “God Object” component offers several capabilities that are accessible via an *Application Interface*.

The KG query for detecting this Smell is depicted in Table 3.28. The query identifies potential *God Objects* by checking whether an *Application Component* executes an excessive number of functions exposed through a single service or interface. The query uses a threshold parameter to define the maximum acceptable number of functions implemented by a single component. If the number of functions exceeds the threshold, the application component may be considered a candidate for the *God Object* EA Smell. The threshold value is configurable, and the default value is set to three. It should be noted that the KG query by itself is insufficient to determine whether the implemented functions are associated with different business capabilities or belong to the same capability. Consequently, in addition to utilizing the query for detection, a human-driven review is essential for identifying the *God Object* EA Smell.

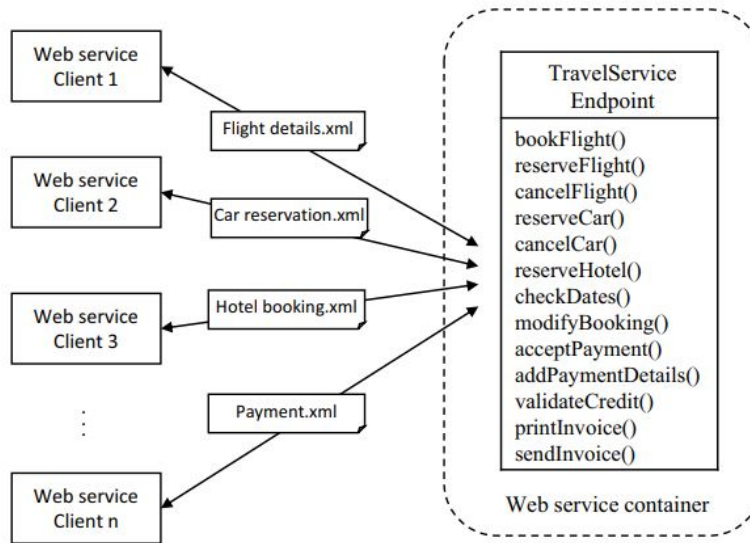


Figure 3.3: Example of a God Object Web Service anti-pattern taken from [68]

Table 3.28: Graphical example, Cypher query and query options for detecting *God Object* EA Smells

<i>God Object</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (c)-[r2:RealizationRelationship]-(s:ApplicationInterface)     ⇔  ApplicationService) <b>WITH</b> c, COUNT(s) <b>AS</b> exposedServiceCnt <b>WHERE</b> exposedServiceCnt=1 <b>WITH</b> c <b>MATCH</b> (c)-[r:RealizationRelationship]-(n) <b>WITH</b> c, COUNT(n) <b>AS</b> functionsCnt <b>MATCH</b> p= (c)-[r:RealizationRelationship]-(n) <b>WHERE</b> functionsCnt &gt; \$functionsCount <b>RETURN</b> p </pre>
Continued on next page	

Table 3.28 – continued from previous page

Property	Description
Query options	<pre> "queryParams": [{"name": "founctionsCount",   "type": "number",   "default": 3,   "description": "The permissible number of functions   ↳ imlemented by an application componnet." }] </pre>
Graphical ex-ample	

### 3.1.26 Lack of Synchronization

The *Lack of Synchronization* is a BPM anti-pattern that occurs when multiple activities are unintentionally triggered due to control flow errors [67].

In the field of EA, the term refers to a situation where concurrent processes are not properly synchronized, leading to unpredictable or inconsistent process flows [26]. In the ArchiMate modeling language, a concurrent process can be represented when a process flow is divided by an *AND Junction*. A Lack of Synchronization occurs when these concurrently executed processes are merged by an *OR Junction*, breaking the synchronized nature of the process.

Table 3.29 provides a graphical example of the *Lack of Synchronization* EA Smell in ArchiMate. Initially, the process flow is split by an *AND Junction* into two concurrent processes: “Evaluate Customer Credit” and “Process Order.” These two processes are interdependent. For instance, if the customer lacks sufficient credit, the order must be canceled. Thus, “Evaluate Customer Credit” and “Process Order” should remain synchronized. However, synchronization is no longer assured when these processes merge via an *OR Junction*, allowing either process to trigger the “End Event,” which can result in an undesirable outcome.

Table 3.29 illustrates the KG query and query options detecting *Lack of Synchronization*

EA Smell. In order to avoid any issues with query performance, a parameter is used to specify the minimum and maximum length of the process chain. By default, the query will retrieve process chains using a length from two to five.

Table 3.29: Graphical example, Cypher query and query options for detecting *Lack of Synchronization* EA Smells

<i>Lack of Synchronization</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (a)-[r]-&gt;(endNode:Junction {type: 'or'}) <b>WITH</b> COUNT(r) <b>AS</b> rCnt, endNode <b>WHERE</b> rCnt&gt;1 <b>WITH</b> endNode <b>CALL</b> (startNode:Junction{type: 'and'}) <b>CALL</b> apoc.path.expandConfig(startNode,{relationshipFilter:"Tri_ ↪ ggeringRelationship&gt; FlowRelationship&gt; AssociationRelation_ ↪ ship&gt;", minLevel: \$minLevel, maxLevel: \$maxLevel, endNode:[endNode] }) <b>YIELD</b> path <b>AS</b> p <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [{"name": "minLevel",   "type": "number",   "default": 2,   "description": "The maximum length of the path for ↪ detection." }, {"name": "maxLevel",   "type": "number",   "default": 5,   "description": "The minimum length of the path for ↪ detection." } ] </pre>
Continued on next page	

Table 3.29 – continued from previous page

Property	Description
Graphical ex-ample	

### 3.1.27 Deadlock

In BPM, the *Deadlock* anti-pattern is a control flow error that halts the execution of business processes [69]. A similar issue can occur in the EA domain, causing an EA Process to be stuck in a stalemate [26]. We can identify and translate this anti-pattern using the ArchiMate modeling language, as it has been outlined for BPM.

To detect the *Deadlock* anti-pattern in ArchiMate, we look for a business or application flow that reaches an *OR* Junction, which then splits into at least two outgoing paths that are later joined by an *AND* Junction. The graphical example in Table 3.30 illustrates a deadlock scenario in ArchiMate. When “Business Process 1” reaches the *OR* Junction, the execution of one of the two process flows is sufficient. However, it later becomes essential for both “Business Process 3” and “Business Process 4” to reach the *AND* Junction. Otherwise, the process will not execute further and can not proceed to “Business Process 5,” resulting in a *Deadlock* EA Smell.

The Cypher query for detecting the *Deadlock* EA Smell is illustrated in Table 3.30. We use a query parameter to define the minimum and maximum lengths of the process chains. By default, the query considers process chains with a maximum length of 10 edges.

Table 3.30: Graphical example, Cypher query and query options for detecting *Deadlock* EA Smells

<i>Deadlock</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (startNode:Junction {type: 'or'}) <b>MATCH</b> (endNode:Junction {type: 'and'}) <b>CALL</b> apoc.path.expandConfig(startNode,{relationshipFilter:"Tri_ ggeringRelationship&gt; FlowRelationship&gt; AssociationRelation_ ship&gt;", ↪ minLevel: \$minLevel, maxLevel: \$maxLevel, terminatorNodes:[endNode] }) <b>YIELD</b> path <b>AS</b> p <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [{"name": "minLevel",   "type": "number",   "default": 1,   "description": "The maximum length of the path for ↪ detection." }, {"name": "maxLevel",   "type": "number",   "default": 10,   "description": "The minimum length of the path for ↪ detection." }] </pre>
Graphical ex-ample	

### 3.1.28 Inconsistent Data

The *Inconsistent Data* is a BPM anti-pattern when there is simultaneous access to a data source without any synchronization mechanism [70, 71].

To address this anti-pattern within the EA domain, it is necessary to understand how concurrent processes arise in ArchiMate. Such a scenario may occur when a process flow is divided by an *AND Junction*, resulting in multiple concurrent processes. If these processes access the same data source, they may do so simultaneously, resulting in inconsistent data [26].

Table 3.31 illustrates a graphical example of the *Inconsistent Data* EA Smell. The “Start event” element initiates a concurrent process. The “Process order” and “Evaluate customer credit” elements operate simultaneously and access the same data source, “Order data.” Consequently, data consistency cannot be ensured, and any modifications made by one element may remain unnoticed by the other.

The KG query provided to identify this Smell is illustrated in Table 3.31. The query checks whether elements that are derived from an *AND Junction* share access to the same data source. A query parameter is employed to specify the minimum and maximum lengths of the process chains. These parameters are essential to prevent potential query performance issues. By default, the query will encompass process chains that have a length of at least two and, at most, ten edges. We must mention that by using the KG query, we can only detect that elements access a data source simultaneously but cannot check whether any synchronization mechanism exists for that simultaneous access. Therefore, a human-driven review is still required to identify this Smell more precisely.

Table 3.31: Graphical example, Cypher query and query options for detecting *Inconsistent Data* EA Smells

<i>Inconsistent Data</i>	
Property	Description
Cypher Query	<pre> MATCH (startNode:Junction {type: 'and'}) MATCH (endNode:BusinessObject DataObject) CALL apoc.path.expandConfig(startNode, {relationshipFilter: "Acc  ↪ essRelationship&gt; TriggeringRelationship&gt; FlowRelationship&gt;  ↪  AssociationRelationship&gt;", minLevel: \$minLevel, maxLevel: \$maxLevel, terminatorNodes: [endNode] }) YIELD path AS p RETURN p </pre>
Continued on next page	

Table 3.31 – continued from previous page

Property	Description
Query options	<pre> "queryParams": [{"name": "minLevel",   "type": "number",   "default": 2,   "description": "The maximum length of the path for ↪ detection." }, {"name": "maxLevel",   "type": "number",   "default": 10,   "description": "The minimum length of the path for ↪ detection." } ] </pre>
Graphical ex- ample	

### 3.1.29 The word Or in element name

The term *The word Or in element name* is derived from a BPM anti-pattern known as “‘or’ between verbs” [72]. This anti-pattern arises when an element’s name includes multiple significant verbs separated by the word “or” [72]. Such a structure can lead to misunderstandings or misinterpretations, as it lacks clarity in the decision-making process regarding which function should be executed. It remains ambiguous whether the verbs linked by “or” are intended to be independent or mutually exclusive

We apply this anti-pattern to the EA domain by examining elements that contain the word “or” between two or more verbs. Table 3.32 presents an example of this Smell within the ArchiMate modeling language. In this case, the element “Execute x or y” is not followed by an *OR Junction* element. Therefore, it might not be clear which path will

be executed under which circumstances. Such ambiguity is an indicator of the presence of this Smell in the model. The KG query detecting *The word Or in element name* is shown in Table 3.32

Table 3.32: Graphical example, Cypher query and query options for detecting *The word Or in element name* EA Smells

<i>The word Or in element name</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> MATCH p = (a)--&gt;(b) WHERE toLower(a.name) contains toLower(" or ") and b.ClassName ↔ &lt;&gt;'Junction' RETURN p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	<pre> graph LR     A[Execute x or y] --&gt; B[Process x]     A --&gt; C[Process y] </pre>

### 3.1.30 Useless Test

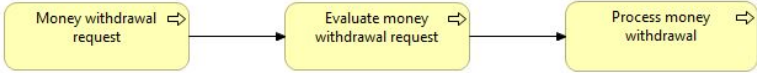
The *Useless Test* EA Smell is derived from a BPM process anti-pattern. It arises when the modeled process takes into account only a subset of possible scenarios and neglects others [26, 72].

We adopt the same approach as described in [72] to identify this Smell within the ArchiMate modeling language. We search for element names that denote a test. Such verbs may include “evaluate,” “verify,” and “validate” [72]. Typically, a test scenario can yield two potential outcomes, either successful or unsuccessful. Therefore, if the test element is not followed by an *OR Junction* or has only one leaving path presenting only one possible outcome of the test, then the model suffers from the *Useless Test* EA Smell.

Table 3.33 shows a graphical example of *Useless Test* EA Smell in ArchiMate. The element “Evaluate money withdrawal request” yields a single possible output, “Process money withdrawal,” while the negative case is not taken into account in the model. Table 3.33 presents the KG query along with default parameters for detecting the *Useless*

*Test* EA Smell. The verbs signifying the test are given as query options and can be modified by enterprise architects. By default, we use the verbs “check,” “test” and “verify” as provided by Ralf Laue et al. [72].

Table 3.33: Graphical example, Cypher query and query options for detecting *Useless Test* EA Smells

<i>Useless Test</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> <b>MATCH</b> p = (m)--&gt;(n) <b>WHERE ANY</b> (name <b>in</b> \$testVerbs <b>where</b> toLower(m.name) CONTAINS     ↳ toLower(name)) <b>and NOT EXISTS</b> ((m)--&gt;(:Junction {type: "or"})) <b>RETURN</b> p <b>UNION</b> <b>MATCH</b> (m)--&gt;(Junction {type: 'or'})--&gt;(n) <b>WITH</b> m, COUNT (n) <b>as</b> testResults <b>WHERE</b> testResults=1 <b>MATCH</b> p=(m)--&gt;(Junction {type: 'or'})--&gt;(n) <b>WHERE any</b> (name <b>in</b> \$testVerbs <b>where</b> toLower(m.name) CONTAINS     ↳ toLower(name)) <b>RETURN</b> p                     </pre>
<i>Query options</i>	<pre> "queryParams": [   {     "name": "testVerbs",     "type": "list",     "default": ["test", "verify", "execute", "evaluate"],     "description": "List of names that indicates a test or     ↳ verification."   } ]                     </pre>
<i>Graphical example</i>	

#### 3.1.31 The word And in element name

*The word And in element name* originates from a BPM anti-pattern referred to “‘and’ between verbs” [72]. This pattern occurs when an element’s name contains at least two verbs connected by the word “and” [72]. A conjunction can be interpreted in two ways. It may take place either in temporal order or simultaneously [72]. For instance, in the task “produce and ship product,” it is evident that producing the product should occur

before shipping. However, ambiguity arises in cases where the process flow is unclear. Table 3.34 illustrates an example of *The word And in element name* EA Smell. The element “Update account and pay for the services” is ambiguous because it is unclear if the updating of the account precedes payment or if both actions are intended to be performed in parallel. Such naming may lead to misinterpretation and requires extra information to clarify the behavior of the element. Furthermore, encapsulating multiple concepts within a single element complicates the model’s maintenance.

We can transfer this anti-pattern into ArchiMate directly by looking at elements containing the word “and” between two or more verbs followed by two or more elements.

The KG query that detects *The word And in element name* is illustrated in Table 3.34. Here, we check if an element contains the word “and” between two verbs or phrases and has any outgoing edges. However, the KG query cannot determine the exact meaning of the word “and” in the element without further analysis tools, such as NLP or Process Manager.

Table 3.34: Graphical example and Cypher query for detecting *The word And in element name* EA Smells

<i>The word And in element name</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> MATCH p = (a)--&gt;(b) WHERE toLower(a.name) contains toLower(" and ") RETURN p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	<pre> graph LR     A[Update account and pay for the services] --&gt; B[Update accounts]     A --&gt; C[Payment] </pre>

### 3.1.32 Start Event Missing

The *Start Event Missing* is a BPM anti-pattern. If a process chain does not have a start event, then it might not be clear where and when it has to be started, or it might not be carried out at all [73]. Specifying the start event is not always necessary, especially in a simple or short EA process. However, it adds some comprehensibility to the modeled process.

In contrast to BPMN, ArchiMate lacks a start event as a specific element type, but an event element at the outset of the EA process can be viewed as a start event. We can transform this Smell into the EA domain by checking process chains of a certain length that have no event element at the beginning of the chain. Table 3.35 illustrates an example of *Start Event Missing* EA Smell in the ArchiMate modeling language. As is obvious, a business chain with a degree of five does not have an event element at the beginning. In other words, the element “Receive Order” is not triggered by an event.

Table 3.35 also depicts the Cypher query for detecting the *Start Event Missing* EA Smell. Process flows are modeled by the relationships *Flow*, *Triggering* and *Association*, which connect the elements *Business Process*, *Application Process* and *Technology Process*. The KG query aims to find process chains that do not contain an event at the beginning of the process. If such a process exists, then it may indicate that the model suffers from the *Start Event Missing* EA Smell. Since there is no precise definition of a simple and complex EA process, a query parameter is set that specifies the minimum and maximum lengths of the process chain. Additionally, specifying an upper limit on path length can prevent query performance issues with large, densely connected data. By default, the query looks for process chains containing a minimum of five and a maximum of twenty edges.

Table 3.35: Graphical example, Cypher query and query options for detecting *Start Event Missing* EA Smells

<i>Start Event Missing</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (startNode:BusinessProcess ApplicationProcess TechnologyProcess) <b>WHERE NOT</b> EXISTS((startNode)&lt;--()) <b>MATCH</b> (endNode:BusinessProcess ApplicationProcess TechnologyProcess) <b>CALL</b> apoc.path.expandConfig(startNode,{relationshipFilter:"TriggeringRelationship FlowRelationship AssociationRelationship", minLevel: \$minLevel, maxLevel: \$maxLevel, endNode: endNode }) <b>YIELD</b> path <b>AS</b> p <b>RETURN</b> p </pre>
Continued on next page	

Table 3.35 – continued from previous page

Property	Description
Query options	<pre> "queryParams": [   { "name": "minLevel",     "type": "number",     "default": 5,     "description": "The maximum length of the path for     ↳ detection."   },   { "name": "maxLevel",     "type": "number",     "default": 20,     "description": "The minimum length of the path for     ↳ detection."   } ] </pre>
Graphical ex-ample	

### 3.1.33 End Event Missing

The *End Event Missing* is a BPM anti-pattern. If a process chain does not contain an end event, it might not be clear where it is terminated [73].

Unlike BPMN, Archimate does not have a special element type for displaying end events, but we can assume an event element at the end of the EA process as an end event. We can transform this Smell into the EA domain by checking process chains of a certain length that have no event element at the end of the chain. Table 3.36 depicts the graphical example of the *End Event Missing* EA Smell in ArchiMate. A process chain with length of five has an element named “Handle Decision” at the end of the process flow which is not followed by an event element. Therefore, we cannot be quite sure whether the process has been completed after this step. Although an end event is not necessarily required in a short or straightforward process chain.

The Cypher query for detecting *End Event Missing* EA Smell with the corresponding query parameter is shown in Table 3.36. The relationships *Flow*, *Triggering* and *Association* alongside *Business Process*, *Application Process* and *Technology Process* elements are mainly used to model process flows. Consequently, the KG query attempts to identify a process chain that includes these elements and relationships, which does not contain an end event at the end of the process. As there is no exact definition of what constitutes

a simple or complex EA process, a query parameter has been defined to indicate the minimum and maximum lengths of the process chain. Furthermore, setting an upper limit on the path length can prevent query performance issues when dealing with large, densely connected data. By default, the query checks for process chains with at least five and a maximum of twenty edges. Enterprise architects may modify the length of the process chain.

Table 3.36: Graphical example, Cypher query and query options for detecting *End Event Missing* EA Smells

<i>End Event Missing</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (startNode:BusinessProcess ApplicationProcess TechnologyProcess) ↪ hnologyProcess) <b>MATCH</b> (endNode:BusinessProcess ApplicationProcess TechnologyProcess) ↪ ologyProcess) <b>where not</b> exists ((endNode)--&gt;()) <b>CALL</b> apoc.path.expandConfig(startNode,{relationshipFilter: ↪ er:"TriggeringRelationship&gt; FlowRelationship&gt; AssociationRelationship&gt;", ↪ iationRelationship&gt;"), minLevel: \$minLevel, maxLevel: \$maxLevel, terminatorNodes:[endNode] }) <b>YIELD</b> path <b>AS</b> p <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [   {"name": "minLevel",    "type": "number",    "default": 5,    "description": "The minimum length of the path for ↪ detection."   },   {"name": "maxLevel",    "type": "number",    "default": 20,    "description": "The maximum length of the path for ↪ detection."   } ] </pre>
Continued on next page	

Table 3.36 – continued from previous page

Property	Description
Graphical ex-ample	<pre> graph LR     A[Recieve Order] --&gt; B[Verify Order]     B --&gt; C(( ))     C --&gt; D[Approved]     C --&gt; E[Reject]     D --&gt; F(( ))     E --&gt; F     F --&gt; G[Handle Decision]   </pre>

### 3.1.34 Missing negative case

The *Missing negative case* is a BPM process anti-pattern. It occurs when a model only considers positive cases while neglecting the negative ones [72]. This anti-pattern is transferred to the EA domain under the same name in the EA Smell catalog.

We can detect the *Missing negative case* EA Smell in ArchiMate, analogous to BPM by searching for elements with positive phrases that lack corresponding negative cases included in the model.

Table 3.37 shows an example of *Missing negative case* EA Smell in the ArchiMate modeling language. In the model, there is an element “Product ordered successfully,” but no corresponding element, that addresses the opposing case. As a result, the model suffers from the *Missing negative case* EA Smell. Additionally, Table 3.37 provides the KG query and its parameters for detection. The query evaluates elements that contain positive expressions, provided they are not separated by an *OR junction*, or the *OR junction* only has a single outgoing path leading to this positive case. We provide positive phrases as query parameters. Ralf Laue et al. [72] provided standard phrases for specifying the positive cases in BPM. These are “successfully,” “without errors” and “in time.” By default, we use the same phrases as Ralf Laue et al. [72].

Table 3.37: Graphical example, Cypher query and query options for detecting *Missing negative case* EA Smells

<i>Missing negative case</i>	
Properties	Descriptions
Cypher Query	<pre> <b>MATCH</b> p=(m)&lt;-[r:TriggeringRelationship FlowRelationship]-(j) <b>WHERE</b> <b>any</b> (name <b>in</b> \$positivePhrases <b>where</b> toLower(m.name) ↪ CONTAINS toLower(name)) <b>AND</b> <b>not</b> EXISTS ((m)&lt;--(:Junction {type: "or"})) <b>RETURN</b> p <b>UNION</b> <b>MATCH</b> p= (m)&lt;-[r:TriggeringRelationship FlowRelationship]-(j:J) ↪ unction {type: "or"}) <b>WHERE</b> <b>any</b> (name <b>in</b> \$positivePhrases <b>where</b> toLower(m.name) ↪ CONTAINS toLower(name)) <b>WITH</b> p, m, j <b>WHERE</b> <b>not</b> exists { <b>MATCH</b> (j)--&gt;(t) <b>WHERE</b> m&lt;&gt;t } <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [ {   "name": "positivePhrases",   "type": "list",   "default": ["successfully", "without errors", "in Time", ↪ "Develop"],   "description": "positive phrase(s) for specifying the ↪ positive cases." } ] </pre>
Graphical ex-ample	

### 3.1.35 Missing Data

The *Missing Data* is identified as a data flow anti-pattern, which occurs when a process attempts to access a data element that does not exist at that point [71]. Figure 3.4 depicts the original example describing this anti-pattern in a Workflow nets with Data (WFD-nets) with a start and end point and a set of places and transitions. The properties

$r$ ,  $w$ , and  $d$  stand for *read*, *write* and *delete* operations. Two errors of type *Missing Data* are noticeable here. First, in  $t_1$  an attempt is made to read the variable  $a$ , but  $a$  is created later in  $t_2$ , therefore reading the variable  $a$  in  $t_1$  will fail. The second case is about the variable  $b$ . It has been created in  $t_3$  but destroyed in  $t_4$ , therefore reading  $b$  in  $t_7$  shall fail.

ArchiMate is an abstract modeling language, and typically, such low-level details are not included in the model. In addition, reading data from a data source without first writing it does not necessarily mean that the data does not exist, or has not been previously retained. Nonetheless, transferring this anti-pattern to the ArchiMate model is straightforward. We can check the process flows, if accessing specific data happens before writing or after deletion, then it may be an indicator for *Missing Data* EA Smell. We use, by default, the same properties ( $r$ ,  $w$ , and  $d$ ) as provided by the underlying BPM anti-pattern and check if the specific data no longer exists or did not exist from the beginning.

Table 3.38 illustrates an example displaying the *Missing Data* EA Smell in the ArchiMate modeling language. As it is evident, the “Business Process k” accesses the data source to retrieve the data  $x$ , but the data is deleted several steps earlier by “Business Process i.”

The KG query detecting this Smell checks the element’s properties in a process flow to determine whether data is retrieved at a time when it does not exist yet, or no longer exists. The default properties can be modified by enterprise architects.

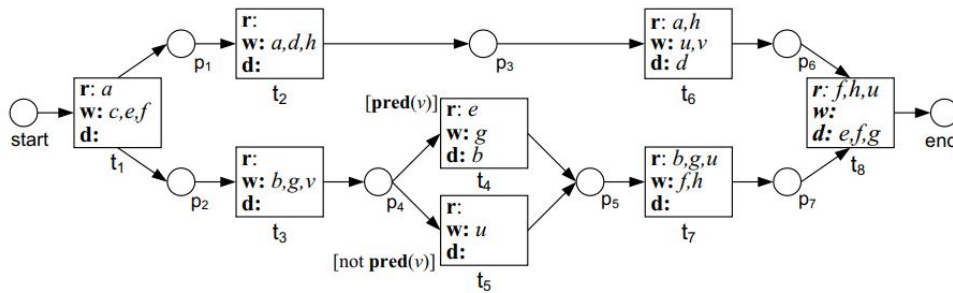


Figure 3.4: Data flow errors in WFD-nets taken from [71]

Table 3.38: Graphical example, Cypher query and query options for detecting *Missing Data* EA Smells

<i>Missing Data</i>	
Property	Description
Cypher Query	<pre> <b>MATCH</b> (startNode:BusinessProcess) <b>where any</b> (x <b>in</b> keys ↪ (startNode) <b>where</b> x <b>in</b> [\$delete]) <b>WITH</b> startNode <b>MATCH</b> (endNode:BusinessProcess) <b>WHERE ANY</b> (x <b>in</b> keys(endNode) <b>where</b> x <b>in</b> [\$read]) and ↪ startNode[\$delete] = endNode[\$read] <b>WITH</b> startNode, endNode <b>WITH</b> startNode, endNode <b>CALL</b> apoc.path.expandConfig(startNode,{relationshipFilter:"Tri_ _ggeringRelationship&gt; FlowRelationship&gt; AssociationRelation_ _ship&gt;", ↪ minLevel: \$minLevel, ↪ maxLevel: \$maxLevel, ↪ terminatorNodes:[endNode] }) <b>YIELD</b> path <b>AS</b> p <b>RETURN</b> p </pre>
Continued on next page	

Table 3.38 – continued from previous page

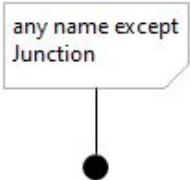
Property	Description
Query options	<pre> "queryParams": [   {     "name": "delete",     "type": "string",     "default": "d",     "description": "Property that specifies the deletion of ↔ certain data."   },   {     "name": "read",     "type": "string",     "default": "r",     "description": "Property that specifies the retrieval of ↔ certain data."   },   { "name": "minLevel",     "type": "number",     "default": 2,     "description": "The maximum length of the path for ↔ detection."   },   { "name": "maxLevel",     "type": "number",     "default": 5,     "description": "The minimum length of the path for ↔ detection."   } ] </pre>
Graphical ex-ample	

### 3.1.36 Junction named as element

The *Junction named as element* EA Smell occurs when the name of a junction represents a function or a task. In ArchiMate, a Junction is a model element that demonstrates a conjunction or a disjunction between elements and should not describe a task that

involves decision-making. If a Junction is named as a task, it can be overlooked by enterprise architects or cause misunderstanding. Furthermore, it can negatively impact the long-term maintainability of the model. Ideally, a Junction should be named as “junction” or remain unnamed completely. We can detect this Smell by checking the Junction element that has a name other than “Junction.” Table 3.39 illustrates a graphical example and the KG query for detecting this Smell.

Table 3.39: Graphical example, Cypher query and query options for detecting *Junction named as element* EA Smells

<i>Junction named as element</i>	
Properties	Descriptions
<i>Cypher query</i>	<pre> <b>MATCH</b> p=(n:Junction) <b>WHERE</b> <b>NOT</b> isEmpty(n.name) <b>AND</b> <b>NOT</b> tolower(n.name) ↪ CONTAINS "junction" <b>RETURN</b> p </pre>
<i>Query options</i>	-
<i>Graphical example</i>	

## 3.2 Undetectable EA Smells

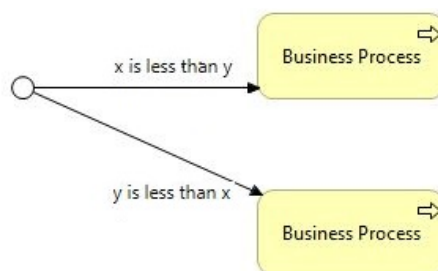
During our analysis, we encountered some EA Smells that cannot be detected either in ArchiMate or via KG queries. We classify the undetectable Smells into three categories. The first category includes the Smells that are not detectable in the ArchiMate modeling language, and therefore are also undetectable via a KG query. The second category consists of those that are detectable in ArchiMate models, but not through a KG query. The third group contains those Smells whose detection requires a comparison between different versions of the model (*diff*). Since the focus of this thesis is the detection of EA Smells in a single graph-based ArchiMate model, detecting EA Smells that require more than one ArchiMate model is beyond the scope of this thesis. In this Chapter we explain the EA Smells that are not detectable via KG query. Table 3.40 shows the undetectable EA Smells with their categories.

**Undefined junction condition** The *Undefined junction condition* EA Smell is derived

Not detectable EA Smells		
KG query	ArchiMate	Out of scope
<i>Architecture by Implication</i> <i>Business Process Forever</i> <i>Contradiction in Input</i> <i>Dead Element</i> <i>Deficient Names</i> <i>Golden Hammer</i> <i>Nothing New</i> <i>Language Deficit</i> <i>Layout Deficit</i> <i>Shiny Nickel</i> <i>Undefined junction condition</i>	<i>Ambiguous Viewpoint</i> <i>Connector Envy</i> <i>Jumble</i>	<i>API Versioning</i> <i>Big Bang</i> <i>Data-Driven Migration</i> <i>Incomplete Node or Collaboration</i> <i>Inconsistent Versioning</i> <i>Temporary Solution</i>

Table 3.40: Undetectable and out of scope EA Smells

from a BPM anti-pattern named *Forgotten Edge Case* [72]. It occurs when the paths leaving from a *Junction* do not cover all possible outcomes in the model. Figure 3.5 illustrates a scenario, where  $x$  is greater than  $y$  and vice versa. However, the case, where  $x$  equals  $y$ , is omitted. As it can be seen, checking expressions and statements requires additional tools and cannot be detected via KG query.

Figure 3.5: *Undefined junction condition* EA Smell in ArchiMate

**Contradiction in Input** The *Contradiction in input* EA Smell arises when a contradictory condition impedes the process of execution [26, 74]. For example, there is a statement that becomes true only if the two contradictory rules are valid, which is not possible. For example the statement “ $x < y$  AND  $x = y$ ” cannot be valid at the same time and prevent the execution process. This Smell always leads to another Smell named *Dead Element*.

**Dead Element** The *Dead Element* is an element that, due to some miscalculation, will never be executed because the process never reaches or passes through that element, although it is integrated into the model and has relations with other elements. An example is an *OR Junction* that has two or more outgoing edges, but the logic behind the junction always leads to one path, and the other will never be reached. This Smell could be identified by Process managers or enterprise architects.

**Architecture by Implication** This anti-pattern occurs when there are no architecture specifications for a system in development [75]. Architects may assume documentation is unnecessary based on their experience with previous systems. Detecting this Smell is not possible using a KG query.

**Layout Deficit** The *Layout Deficit* is originally an anti-pattern from BPM, which addresses the understandable problems in the layout, such as spacing, overlapping, and position of elements in the model [67]. In a graph-based ArchiMate model, we cannot check the positions or reading directions of elements via KG query. Therefore, identifying this Smell requires the involvement of a process manager or an enterprise architect.

**Language Deficit** The *Language Deficit* EA Smell originates from the BPM anti-pattern. It addresses the issues of the vague or unsuited text labels for describing elements that do not follow the naming conventions [67]. The ArchiMate specifications suggest some naming rules for elements. For example, the name of a *Business Object* should be preferably a noun, and the name of *Business Process* should be a verb or a verb-noun combination. By using a KG query, we are not able to verify whether an element has an appropriate name. In this case, enterprise architects can manually check the Smell, or some Natural Language Processing (NLP) tools are required to check the naming conventions automatically.

**Deficient Names** *Deficient Names* EA Smell is derived from *Ambiguous Name* [76], also known as *Ambiguous Service*. This anti-pattern arises when the name chosen for components is meaningless and does not make sense semantically and syntactically [77, 76]. *Deficient Names* in ArchiMate cannot be detected through a KG query. It could be identified by a process manager or an enterprise architect.

### 3.3 Out of Scope

There is a group of EA Smells whose detection requires checking more than one EA model. Namely, we have to go through different versions of the same EA model in order to be able to detect those Smells. For example, we consider the *Inconsistent Versioning* EA Smell. This Smell derives from *API Versioning* Smell [46]. It arises when the correct versioning of EA models does not follow the versioning standards and has an inconsistent and poor versioning. In order to check whether the versioning is semantically correct, we have to compare the different versions of the model. However, this thesis evaluates only one EA model at a time. Therefore, detecting such Smells is beyond the scope of this thesis. Another EA Smell that requires different versions of the model is *Data-Driven Migration*. This anti-pattern is relevant when migrating to a microservices architecture from a monolithic application [78]. In order to check whether the functionality is migrated first and then the data, we are required to compare different versions of the model. *Big Bang* is also a migration process when the whole new model is built at once. We can only check whether there are many changes in the new model by comparing it with previous

ones. The same applies to *Temporary Solution* in this case, we need to examine several versions of the model to determine if a temporary solution has been removed at any point.

## 3.4 Summary

In this chapter, we analyzed all EA Smells from the catalog. We described how we transformed the concepts from underlying Code Smells and BPM anti-patterns to the EA Smells. We provided more accurate and appropriate definitions suitable for the EA domain. We were able to provide queries with corresponding parameters for thirty-six EA Smells. As mentioned at the beginning of this chapter, there were some KG queries for a small set of EA Smells in [12]. We have also improved and redefined some of these queries where there was room for improvement. However, we did not include all of those queries in this work as they required no intervention. In the next chapter, we will extend the catalog by incorporating our findings from this effort. In particular, we will update the catalog with provided queries and parameters, and will revise some Smell descriptions to bring more clarity and integrity to the catalog.

## CHAPTER 4

# Extension of the Enterprise Architecture Smells catalog

This chapter describes the process of extending and revising the current EA Smell catalog [8], which builds on the findings of the previous chapter.

The current EA Smells catalog necessitates further clarifications as the existing definitions are overly abstract and lack a precise mechanism for detecting these Smells within the EA domain. Additionally, the structure and meta-models used to describe EA Smells in the catalog are not uniform, with varying fields for different Smells. Therefore, unification is required to have the same information structure for all EA Smells. In addition, some descriptions and examples of EA Smells are either too abstract or taken directly from Code Smells and anti-patterns without being adequately adapted to the EA domain. Therefore, a revision is underway to refine the information on EA Smells. This will involve adding new properties to the catalog schema and, where necessary, revising the definitions based on the analysis from the previous section.

In the following sections, we begin by outlining the two schemes that represent EA Smells in the catalog, and we demonstrate how we consolidate them by providing a unified structure for all EA Smells. In the second step, we revise the existing definitions for EA Smells where required, aiming to make them more precise and suitable in the EA domain. We also extend the scheme with new properties for KG queries, query parameters, thresholds, and graphical examples based on our analysis and findings from Chapter 3. We demonstrate the revision process, including unification and extension for some EA Smells. Accordingly, the final result is updated in the EA Smells catalog website [8].

## 4.1 Unifying the schema of EA Smells

The catalog contains 63 EA Smells. Generally, every Smell in the catalog consists of *name*, *description*, *consequences*, *detection*, *solution* and others. However, some Smells have more properties than others. Table 4.1 presents the two schemes used to describe the EA Smells in the catalog. The first scheme presented by Salentin and Hacks [7] has twelve properties. This scheme has been provided for representing 45 EA Smells derived from the Code Smells and anti-patterns. The second scheme provided by Lehman et al. [26] has seven properties and represents the 18 EA Smells derived from BPM anti-patterns. The scheme provided by Salentin and Hacks [7] includes all the properties from the second scheme, except the field *Graphical Definition*. We rename the property to *Graphical Example* and adjust the second scheme to match the first one by adding the missing properties from the first scheme, creating a standardized format for describing all Smells in the catalog. As an example, we explain the unification process for the *Useless Test* EA Smell in Section 4.2. A unified scheme for all EA Smells will ease the maintenance and automation process of detection which we discuss in Chapter 5.

Table 4.1: Two schemes for representing EA Smells

1-Scheme provided by Salentin and Hacks [7]	
Property	Description
<i>Name</i>	Name of the EA Smell
<i>Also known as</i>	Other aliases in the literature
<i>Description</i>	Describing the Smell and how it can arise
<i>Context</i>	Underlying Smell, which the EA Smell is derived from
<i>Consequences</i>	Possible negative impacts
<i>Cause</i>	explains the reasons for an EA Smell
<i>Detection</i>	How to detect the Smell
<i>Solution</i>	A suggestion for refactoring
<i>Example</i>	Example for better understanding of the Smell
<i>Evidence</i>	how often it occurs
<i>Related Items</i>	Relatedness with other Smells
<i>Tags</i>	Categories related to a Smell
2-Scheme provided by Lehman et al [26]	
Property	Description
<i>Name</i>	Name of the EA Smell
<i>Description</i>	Describing the Smell and how it can arise
<i>Consequences</i>	Possible negative impacts
<i>Solution</i>	A suggestion for refactoring
<i>Graphical Definition</i>	Graphical representation
<i>Related Items</i>	Relatedness with other Smells
<i>Tags</i>	Categories related to a Smell

## 4.2 Extending the schema and revising the EA Smells catalog

Apart from unification, we further extend the scheme to include the Cypher queries, query parameters, and graphical examples based on our analysis from Chapter 3. The new properties are *Cypher query*, *Query options*, and *Graphical example*. The *Cypher query* field references the KG query for detecting the corresponding EA Smell. If the query has some parameters, these parameters and their data types are listed in *Query options*. Furthermore, every query option has a default value as a threshold. Depending on the enterprise characteristics, it can be changed by users or enterprise architects. The *Graphical example* field depicts an image of the Smell in ArchiMate to facilitate a better understanding of the EA Smell and to avoid misinterpretations. These fields are optional, because as mentioned in Section 3.2, some EA Smells cannot be detected through a KG using KG query or in the ArchiMate modeling language. Concurrently, we revise some of the information describing EA Smells to make it more precise and relevant in the EA domain.

In order to describe the processes of the scheme extension and revision of the catalog, we demonstrate these processes for *Useless Test*, *Deficient Encapsulation*, and *Multifaceted Abstraction* EA Smells. We first look at the existing definitions of the Smell, and then we add new fields and revise the descriptions where required. For the *Useless Test* EA Smell, apart from extension and revision, a unification is also required.

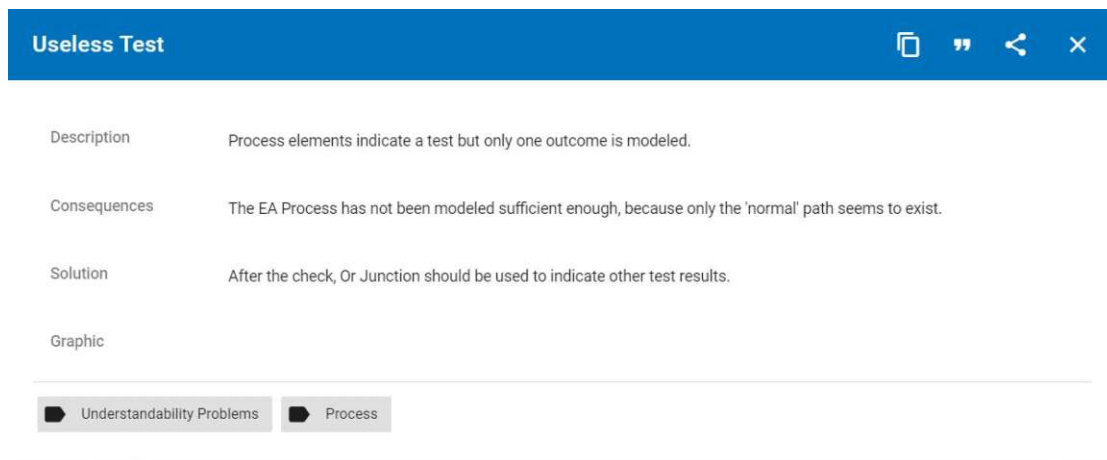


Figure 4.1: Current representation of the *Useless Test* EA Smells in the catalog

**Useless Test:** The current representation of the *Useless Test* EA Smell in the catalog is illustrated in Figure 4.1. As it can be seen, the meta-model describing this EA Smell has fewer properties than other Smells. Thus, unification is required by adding the missing properties as described in Section 4.1. Besides, the information pieces describing this Smell are vague and might be misunderstood. Therefore, a revision is also necessary to make them more understandable and suitable for the EA domain. Finally, we extend

#### 4. EXTENSION OF THE ENTERPRISE ARCHITECTURE SMELLS CATALOG

the scheme by adding the new properties *Cypher query*, *Query Options*, and *Graphical example* based on our analysis from Section 3.1.30. Table 4.2 illustrates the unified, revised version of existing definitions and extended properties.

Table 4.2: *Useless Test* EA Smell after unification, extension and revision

<i>Useless Test</i>	
Property	Description
<i>Description</i>	The <i>Useless Test</i> EA Smell is derived from a BPM process anti-pattern. It arises when the modeled process takes into account only a subset of possible scenarios and neglects others.
<i>Example</i>	The element “Evaluate money withdrawal request” has only one possible output “Process money withdrawal,” but a negative case is not considered in the model.
<i>Consequences</i>	Not considering all possible outcomes in models hinders enterprise architects to analyze other outcomes and handle the unexpected behavior.
<i>Detection</i>	We search for element names that denote a test. Such verbs may include “evaluate,” “verify,” and “validate”. Typically, a test scenario can yield two potential outcomes, either successful or unsuccessful. Therefore, if the test element is not followed by an <i>OR Junction</i> or has only one leaving path presenting only one possible outcome of the test, then the model suffers from the <i>Useless Test</i> EA Smell.
<i>Cypher query</i>	<pre> MATCH p = (m)--&gt;(n) WHERE ANY (name in \$testVerbs where toLower(m.name) CONTAINS ↪ toLower(name)) and NOT EXISTS ((m)--&gt;(:Junction {type: "or"})) RETURN p UNION MATCH (m)--&gt;(Junction {type: 'or'})--&gt;(n) WITH m, COUNT (n) as testResults WHERE testResults=1 MATCH p=(m)--&gt;(Junction {type: 'or'})--&gt;(n) WHERE any (name in \$testVerbs where toLower(m.name) CONTAINS ↪ toLower(name)) RETURN p </pre>
Continued on next page	

Table 4.2 – continued from previous page

Property	Description
<i>Query options</i>	<pre> "queryParams": [{   "name": "testVerbs",   "type": "list",   "default": ["test", "verify", "execute", "evaluate"],   "description": "List of names that indicates a test or ↪ verification." }] </pre>
<i>Graphical example</i>	<pre> graph LR     A[Money withdrawal request] &lt;--&gt; B[Evaluate money withdrawal request]     B &lt;--&gt; C[Process money withdrawal] </pre>

**Deficient Encapsulation:** Figure 4.2 shows the current scheme and information describing this Smell. As it is seen, the information describing this Smell is generally taken from Code Smell, and most of it is not valid for the EA domain, especially the ArchiMate language. For example, the detection’s text fits only the underlying Code Smell, and the description is too general. We rewrite these definitions and also extend the current scheme with new properties based on our analysis from Section 3.1.8. Table 4.3 shows the revised version of the *Deficient Encapsulation* EA Smell.

Table 4.3: *Deficient Encapsulation* EA Smell after extension and revision

<i>Deficient Encapsulation</i>	
Property	Description
<i>Description</i>	This Smell occurs when an abstraction makes its members accessible to other abstractions in a more permissive way, which is not necessarily required. In the EA domain, particularly in the ArchiMate modeling language there are no access modifiers. As a result, we need to slightly adjust the definition to apply this concept to ArchiMate. Instead of modifiers, we try to identify sources of information that contain sensitive data while the rules for accessing those data are more lenient than necessary.
<i>Consequences</i>	Not restricted sensitive data can be retrieved and modified by other abstractions unintentionally or imperceptibly.
Continued on next page	

Table 4.3 – continued from previous page

Property	Description
<i>Detection</i>	In ArchiMate, Business Objects, Data Objects and Artifacts serve as data source representations. However, there is no distinction made between sensitive and non-sensitive data sources. To address this, we can explore identifying a property that designates a data source as confidential. Additionally, we can establish a threshold for permissible access to sensitive data. If the number of accesses exceeds this threshold, we may interpret it as an indication of a Deficient Encapsulation EA Smell.
<i>Cypher Query</i>	<pre> <b>MATCH</b> (n) <b>WHERE</b> n.ClassName = 'DataObject' or ↪ n.ClassName = 'BusinessObject' or n.ClassName = 'Artifact' <b>WITH</b> n <b>WHERE ANY</b> (x <b>in</b> keys(n) <b>WHERE</b> n[x] <b>IN</b> \$sensitiveProperties OR ↪ toLower(x) <b>IN</b> \$sensitiveProperties) <b>WITH</b> n <b>MATCH</b> (n)-[r:AccessRelationship]-(m) <b>WITH</b> n, count(r) <b>as</b> relCnt <b>MATCH</b> p=(n)-[r:AccessRelationship]-(m) <b>WHERE</b> relCnt &gt; \$permissibleCount <b>RETURN</b> p </pre>
Query options	<pre> "queryParams": [   {     "name": "sensitiveProperties",     "type": "list",     "default": ["confident", "classified", "sensitive"],     "description": "The list of property values specifying ↪ an element as confidential."   },   {     "name": "permissibleCount",     "type": "number",     "default": 2,     "description": "The permissible number of elements for ↪ accessing a confidential data source."   } ] </pre>

Continued on next page

Table 4.3 – continued from previous page

Property	Description
Graphical ex-ample	<pre> graph TD     BO[Business Object]     BA1[Business Actor]     BA2[Business Actor]     BA3[Business Actor]     BO -.-&gt; BA1     BO -.-&gt; BA2     BO -.-&gt; BA3 </pre> <p>Default properties specifying a data source as confidential: confident : true sensible : true</p>

**Multifaceted Abstraction:** The current representation of the *Multifaceted Abstraction* EA Smell in the catalog is illustrated in Figure 4.3. The field *Description* is too general. The text for *Detection* and *Example* is mostly applicable to the underlying Code Smell, rather than EA domain. It is also noticeable that a concrete approach for detecting this EA Smell in ArchiMate is missing. We extend and revise this Smell based on our analysis from section 3.1.3. Table 4.4 illustrates the revised version of the EA Smell and the new properties *Cypher query*, *Query options* and *Graphical example*. Since the KG query has no query parameter, the *Query options* remain empty.

For the rest of EA Smells, we use the same approach for unification, extension and revision.

Table 4.4: *Multifaceted Abstraction* EA Smell after extension and revision

<i>Multifaceted Abstraction</i>	
Property	Description
<i>Description</i>	A good abstraction should only be responsible for one specific functionality, and when it encapsulates more than one responsibility, it signifies the the <i>Multifaceted Abstraction</i> Smell. This Smell violates the single responsibility principle and has a low cohesion, which also violates the principle of modularization.
<i>Example</i>	A business actor performing more than one business processes, or an Application component realizing two different Services.
<i>Consequences</i>	The <i>Multifaceted Abstraction</i> Smell violates the single responsibility principle, which also violates the principle of modularization. The maintainability of such a service may be reduced. It may be more difficult to analyze and grasp all provided service functionality. The business logic is not reusable, because it contains service-specific implementation. Low cohesion and possibly low availability with high response times.
Continued on next page	

Table 4.4 – continued from previous page

Property	Description
<i>Detection</i>	We can transfer this Smell in the EA domain by examining an active structure performing or realizing more than one behavior. According to the ArchiMate specification, the Assignment Relationship links active structures to behaviors and states for responsibility, execution, and performing of a behavior. The Realization Relationship signifies that an element implements and supplies the abstract element.
<i>Cypher Query</i>	<b>MATCH</b> $\hookrightarrow (m) - [r:AssignmentRelationship RealizationRelationship] -> (n)$ <b>WITH</b> m, COUNT(r) <b>AS</b> rCount <b>MATCH</b> p = (m) - [r:AssignmentRelationship RealizationRelationship] _ $\hookrightarrow -> (n)$ <b>WHERE</b> rCount > 1 <b>RETURN</b> p
Query options	-
Graphical ex-ample	

## 4.2. Extending the schema and revising the EA Smells catalog

Deficient Encapsulation	
Also known as	Too Much Information
Description	This smell occurs when the accessibility of one or more members of an abstraction is more permissive than actually required.
Context	Derived from: "Deficient Encapsulation" Components have too deep insight into unimportant details of other elements, making their work harder.
Consequences	Confusing information for other components with impact on understandability.
Cause	Keep a close eye on components that spend too much time together. Good elements should know as little about each other as possible. Such elements are easier to maintain and reuse.
Detection	Count the global and public elements.
Solution	Try to hide as much unneeded information as possible. Integrate functionality into components that use that functionality. This is only possible if the first component truly does not need these parts.
Example	
Evidence	not set
IEEE LNCS APA HARVARD VANCOUVER BIBTEX	
soa microservices business application technology The Couplers between elements	

Figure 4.2: Current representation of the *Deficient Encapsulation* EA Smell in the catalog

Multifaceted Abstraction	
Also known as	Divergent Change, Large Component, Low Cohesive Operations
Description	Occurs when one component is commonly changed in different ways for different reasons, because of multiple responsibilities. Separating these divergent responsibilities decreases the chance that one change could affect another and lowers maintenance costs.
Context	Derived from: "Multifaceted Abstraction" High cohesion is one of the most important design principles. Services should provide operations that are closely related to each other. Functionality that is likely to change together should be in the same service.
Consequences	The maintainability of such a service may be reduced. It may be more difficult to analyze and grasp all provided service functionality. The business logic is not reusable, because it contains service specific implementation. Low cohesion and possibly low availability with high response times.
Cause	Often these divergent modifications are due to poor architectural structure or "cloning".
Detection	Processing tasks exist that also implement business logic.
Solution	Split up the behavior of the component. These components should dispatch requests to the underlying business logic.
Example	A service endpoint implements business logic, but also deals with XML processing.
Evidence	not set

Figure 4.3: Current representation of the *Multifaceted Abstraction* EA Smell in the catalog

## 4.3 Summary

This chapter described unification, extension, and revision of EA Smells from the catalog. We demonstrated these processes for three EA Smells and applied the same approach to the rest of EA Smells. We have extended and revised forty-four EA Smells from the catalog. The updated information is now available on the EA Smells catalog website [8].



# Integrating the EA Smells catalog into a Knowledge Graph-based detection platform

This chapter provides an overview of the development of the graph-based detection platform used to automatically identify the EA Smells.

The analysis of EA Smells and the KG queries is detailed in Chapter 3. Following that, Chapter 4 expands the EA smell catalog to include those queries. This chapter introduces an automated graph-based detection tool specifically designed to identify EA Smells within graphs that represent ArchiMate models. The Section 5.1 describes the platform's architecture and its interaction with the EA Smells catalog and the CM2KG service. Sections 5.2.1 and 5.2.2 describe how our detection platform achieves an interoperability between EA Smell catalog and the CM2KG Platform 5.2.2. Finally, Section 5.2.4 describes the User Interface of our detection platform in more details.

## 5.1 General Overview

The main purpose of our detection platform is to identify EA Smells using KG queries in graph-based ArchiMate's models. To accomplish this, we use the CM2KG platform, an external service for model transformation. It transforms ArchiMate files from ArchiMate Open Group Exchange FILE Format into GraphML, which contains all the information about ArchiMate elements, relations, types, and properties. The detection platform then imports the transformed model into Neo4j database. Additionally, the detection platform retrieves EA Smells, including KG queries, query parameters and descriptions from the catalog. At this stage, the detection platform is capable of identifying potential EA Smells in KG through queries obtained from the catalog.

## 5. INTEGRATING THE EA SMELLS CATALOG INTO A KNOWLEDGE GRAPH-BASED DETECTION PLATFORM

The component diagram in Figure 5.1 illustrates the abstract view of the architecture. Our detection platform, shown in turquoise, along with the extended parts of the catalog, indicated in blue, represents our contributions to this thesis. The yellow components, which include the CM2KG platform and the Neo4j Database, are not our contributions; we utilize them as services.

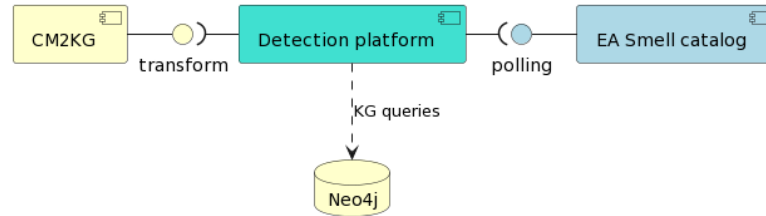


Figure 5.1: General overview of the EA Smell detection platform

### 5.2 Implementation of the Detection platform

The detection platform interacts with the CM2KG platform, the EA Smell catalog, and the Neo4j graph database, and relies on these systems being up and running. Their absence affects the functionality of the detection platform. Additionally, the platform provides a User Interface that allows users to filter out Smells and adjust the thresholds. The results can be displayed in both tabular and graphical forms. We use Java 17 and the Spring Boot Framework 3.4 to implement the detection platform. Additionally, we use Spring Data Neo4j to simplify access to Neo4j graph databases.

#### 5.2.1 Catalog Integration

The EA Smell catalog has been updated in Chapter 4 to include detection queries and parameters. The EA Smells are stored as a JSON object in the catalog. Listing 5 provides an example of an EA Smell in JSON format.

To integrate the EA Smell catalog into our platform, we do not store them in our detection platform, but retrieve all properties and information about EA Smells as JSON arrays from the catalog. We keep the EA Smells detection logic in the catalog and outside of our platform, so that any changes in the catalog do not affect the platform, and vice versa. This ensures a long-term maintainability. Communication between the detection platform and the EA Smell catalog occurs through the HTTP call. We initiate an HTTP request and receive a list of JSON objects in response. The structure of the HTTP request and response from the EA Smells catalog is detailed in Table 5.1.

Detection platform has numerous interactions with the catalog. Each time we refresh the UI page or filter Smells by tags or names, a request is sent to EA Smell Catalog to retrieve the Smells. Since the catalog is not often updated, we cache the EA Smells into the detection platform to reduce the number of requests sent to the EA Smell catalog.

Table 5.1: *REST API* for retrieving EA Smells from the catalog

<i>Method</i>	GET
<i>URI</i>	https://swc-public.pages.rwth-aachen.de/smells/ea-smells/assets/result.json
<i>Request parameter</i>	-
<i>Response</i>	JSON array A single JSON object representing an EA Smell is depicted in Listing 5

Figure 5.2 illustrates a sequence diagram which outlines the process interactions between the detection platform and the catalog. When the EA Smells are fetched from the catalog they still remain in cache for a period of the configured time. Once the cache expires, the EA Smells are automatically removed from the cache and will have to be retrieved from the catalog again.

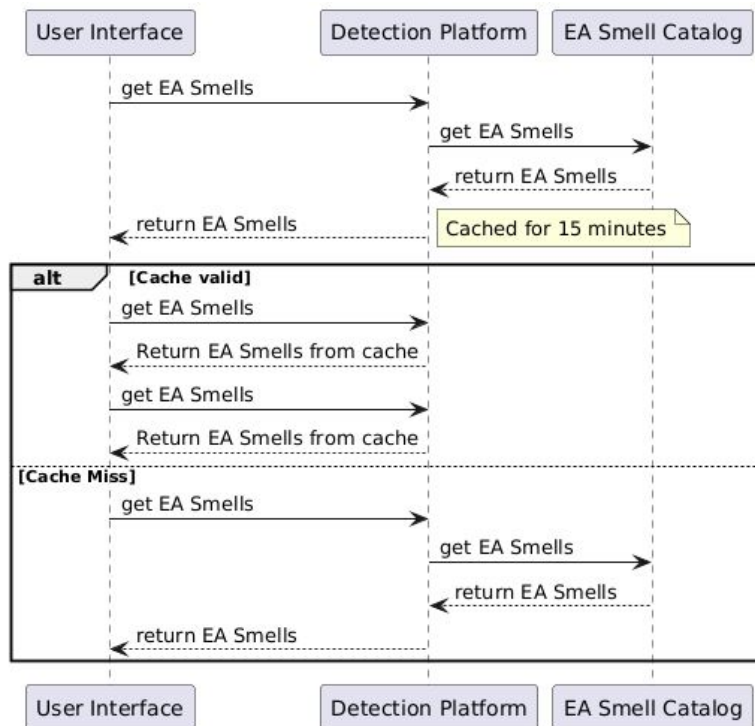


Figure 5.2: Interactions between the detection platform and the EA Smell catalog

## 5. INTEGRATING THE EA SMELLS CATALOG INTO A KNOWLEDGE GRAPH-BASED DETECTION PLATFORM

```
{
  "name": "Deficient Encapsulation",
  "aliases": ["Too Much Information"],
  "description": "This smell occurs when the accessibility of one or more
  ↪ members of an abstraction is more permissive than actually required.",
  "context": "...",
  "detection": "ArchiMate does not differentiate between sensitive and
  ↪ non-sensitive data sources. Therefore, we can search for a property that
  ↪ specifies the data source as sensitive. In addition to that, we set a
  ↪ threshold to specify a permissible number for accessing sensitive data.
  ↪ If the number of accesses exceeds the given number, then we can
  ↪ interpret it as a sign of Deficient Encapsulation EA Smell.",
  "consequences": "...",
  "cause": "...",
  "solution": "...",
  "example": "",
  "cypher": "MATCH (n) <br /> WHERE n.ClassName='DataObject' or
  ↪ n.ClassName='BusinessObject' UNWIND keys(n) as nkeys WITH nkeys, n <br
  ↪ /> WHERE ANY (regex in $sensitiveProperty where n[nkeys] contains regex)
  ↪ <br /> WITH n <br /> MATCH (m)-[r:AccessRelationship]-(n) <br /> WITH
  ↪ n,m, COUNT (r) as accessCount <br /> WHERE accessCount>
  ↪ $permissibleCount <br /> RETURN m, n",
  "sources": [],
  "tags": ["soa", "microservices", "business", "application", "technology"],
  "queryParams": [
    {
      "name": "sensitiveProperty",
      "type": "list",
      "default": ["confident", "sensible"],
      "description": "The property values specifying an element as
      ↪ confidential."
    },
    {
      "name": "permissibleCount",
      "type": "number",
      "default": 2,
      "description": "The permissible number of elements for accessing a
      ↪ confidential data
      ↪ source."
    }
  ]
}
```

Listing 5: Structure of JSON object for representing single EA Smell

### 5.2.2 CM2KG Platform

The CM2KG acronym stands for “From Conceptual Models to Knowledge Graphs” [13]. This platform can convert conceptual models such as EMF, ADOxx meta-modeling, and Ecore-based modeling platforms into KG [11]. In the context of this thesis, we utilize the CM2KG platform to convert Archi exchange files into Knowledge Graphs. Figure 5.3

Once the Archi models have been converted into GraphML files, the next step is to initialize the Neo4j Database. We use an *apoc* procedure called `apoc.import.graphml`<sup>3</sup> to import the GraphML files into the database. Before each import is made, we make sure that the previous schema is deleted from the database. After importing the graph



Figure 5.3: CM2KG model transformation taken from [13]

### 5.2.3 Initializing the Neo4j database

Once the Archi models have been converted into GraphML files, the next step is to initialize the Neo4j Database. We use an *apoc* procedure called `apoc.import.graphml`<sup>3</sup> to import the GraphML files into the database. Before each import is made, we make sure that the previous schema is deleted from the database. After importing the graph

<sup>1</sup><https://www.opengroup.org/xsd/archimate/>

<sup>2</sup><http://graphml.graphdrawing.org/>

<sup>3</sup><https://neo4j.com/docs/apoc/current/import/import-graphml/>

## 5. INTEGRATING THE EA SMELLS CATALOG INTO A KNOWLEDGE GRAPH-BASED DETECTION PLATFORM

Table 5.2: *REST API* for transforming Archi exchange files to GraphML

<i>Method</i>	GET
<i>URI</i>	/api/transformation/archi
<i>Request parameter</i>	Multipart file <ArchiMate Exchange File xml file content>
<i>Response</i>	"id": <uuid> "transformedGraph": <GraphML xml file content>

into the database, we also make some minor adjustments by adding additional labels to elements and relationships. We ensure that element and relation types are stored as Neo4j labels. Finally, during initialization, we invoke two procedures for graph projection and the *Louvain* algorithm, which we have defined in Chapter 3, Listing 4). Clustering is required for detecting *Missing Abstraction* EA Smell (Section 3.1.19).

### 5.2.4 UI

The user interface of the detection platform is a single-page application (SPA). Since the entire UI is not clearly visible in a single image, we have split it into two images, Figure 5.4 and Figure 5.5. Figure 5.4 shows three tabs: Home, Model Transformation and Smell Detection. The home tab provides information about the platform and its interconnected components. In the Model Transformation tab, we upload the selected file to the CM2KG platform and import the transformed result into a Neo4j Database. The central part of our application is in the detection section, which is also illustrated in Figure 5.4. In the detection part, users can filter the EA Smells by categories located on the top left side. The EA Smells are displayed in the center based on the selected categories. Each EA Smell has a turquoise button that displays the corresponding information from the catalog in a pop-up window. Additionally, if a Smell has query options, a blue button will appear. Clicking on the query parameter button for a specified EA Smell displays one or more dynamic fields on the right side, showing the possible query options with their default values and descriptions. Users can update the default query options and set appropriate parameter values based on their enterprise-specific requirements or other concerns.

Upon clicking the detect button (Figure 5.5), the names of selected EA Smells, along with their parameters, are sent to our detection server. The server then associates the Smells names with the queries, sets the parameters and executes them. It is important to note that users do not have any access to the queries and cannot modify them. They only have access to the query parameters. This approach serves two purposes. First, users need not be familiar with query syntax and logic. Second, it prevents users from altering the queries within the detection platform. Thus, the queries are not exchanged between the frontend and the backend. Instead, they always remain in the backend and are linked with the name of the Smells sent by the request. Figure 5.5 demonstrates the results of

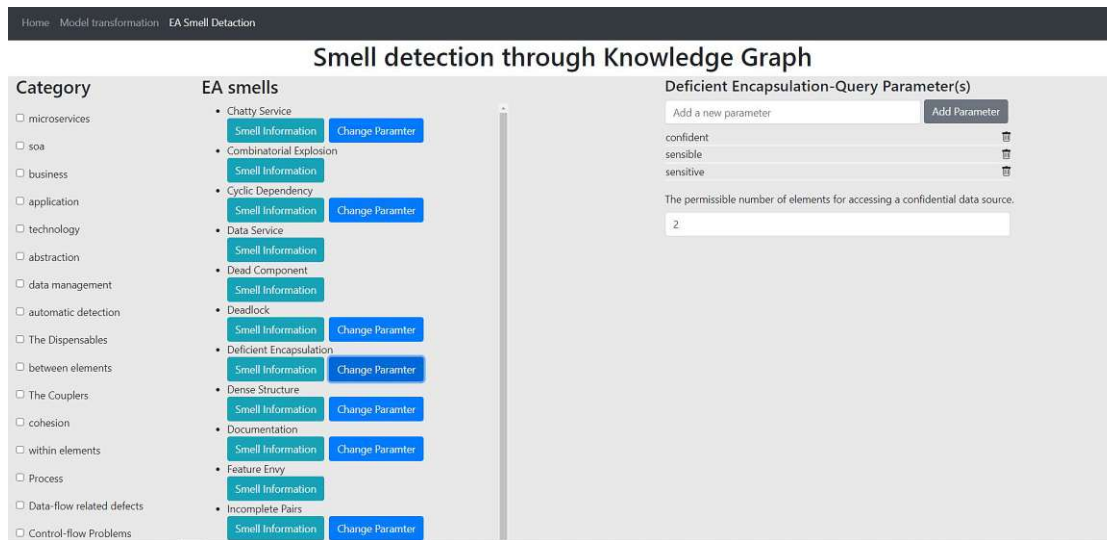


Figure 5.4: General overview of the EA Smell detection platform

the detection request. The list of identified EA Smells is presented on the lower left side. Clicking on a detected Smell will display the EA Smell in a graph form. The ArchiMate elements are depicted as nodes, while their relationships are shown as edges. This visual representation of a specific Smell provides users with additional insights into the model's flaws and drawbacks. Clicking on a node allows users to view all node properties on the lower right side.

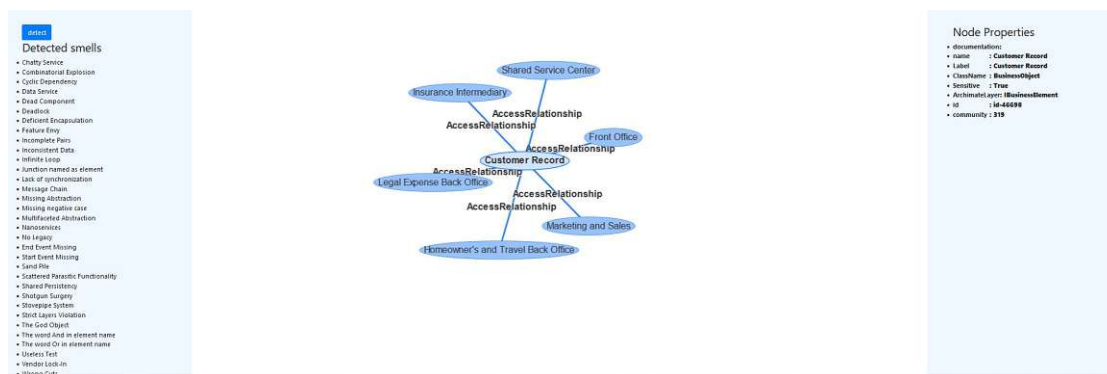


Figure 5.5: Detection section, illustrating detected EA Smells in textual and visual forms.

## 5.3 Summary

In this section, we introduced our detection platform and its integration with the EA Smell catalog and the CM2KG platform. Our platform has the capability to perform automated detection, filter Smells, adjust query options and present results in both

## 5. INTEGRATING THE EA SMELLS CATALOG INTO A KNOWLEDGE GRAPH-BASED DETECTION PLATFORM

---

textual and graphical formats. Section 6 will evaluate the correctness of our approach and platform from both conceptual and technological standpoints.

# CHAPTER 6

## Evaluation

This chapter evaluates our approach described in Section 3 and the detection platform provided in Section 5. Our evaluation aims to assess the quality and accuracy of our approach from both technological and conceptual standpoints. In our evaluation, we answer the following research questions.

**RQ.1 - Feasibility:** is our detection platform feasible to achieve interoperability between EA Smell catalog and the CMKG platform to perform automation for EA Smells detection?

**RQ.2 - Correctness:** How accurate are the provided KG queries in detecting EA Smells in graphs?

### 6.1 Methodologies and Metrics

We utilize *Software Testing* and *Empirical Experiment* methodologies for evaluating RQ.1, while *Precision and Recall* serve as correctness metrics for RQ.2.

**Software Testing:** In order to ensure that our platform is working correctly, we conduct tests to check the transfer of data and interaction between various components (Section 6.2). This test serves two purposes. First, it aims to detect a set of EA Smells in a single EA Model. Second, it aims to confirm that the platform interacts correctly with different components (CM2KG, EA Smell catalog, Neo4j database)

**Empirical experiment:** To assess how our approach performs in real-world scenarios, we test our platform using a large number of real EA models (Section 6.3)

**Precision and Recall:** To evaluate the correctness of our approach, we calculate the precision and recall ratios in section 6.4 to measure the correctness of our KG

queries. Hereafter, we discuss the expected and actual results of our detections and justify the deviations (Section 6.4.2).

## 6.2 Software Testing

We have conducted integration tests to assess the feasibility of our platform and its interaction with the extended catalog and the CM2KG platform. Salentin and Hacks [43] provided an ArchiMate model `smellexample.xml` for their prototype demonstrating 14 EA Smells. The model initially consisted of 47 elements and 88 relationships. We have extended this model by incorporating all graphical examples provided in Section 3.1, resulting in 302 elements and 652 relationships. The updated model <sup>1</sup> now contains all detectable EA Smells via KG queries.

We tested the extended `smellexample.xml` model against our detection platform. For each EA Smell we wrote an integration test. Each test consists of the following steps:

1. The platform fetches the EA Smells from the catalog.
2. Transforms the `smellexample.xml` to GraphML file using the CM2KG platform.
3. Imports the transformed model into the Neo4j database.
4. The KG query fetched from the catalog should detect the specified EA Smell.
5. Changes the query parameters (if applicable) and expects a different detection result.

In total, 36 integration tests were written and successfully performed to confirm the desired behavior of our detection platform and its interaction with the CM2KG and the EA Smell catalog.

## 6.3 Empirical Experiment

To evaluate the effectiveness of our approach in real-world scenarios, we conducted tests on our detection platform using a large set of EA models derived from real-life applications.

Initially, we converted these models from the ArchiMate Open Group Exchange Format to GraphML using the CM2KG platform. Next, for each converted model, we imported it into a Neo4j database and executed 36 KG queries with their default parameters, as outlined in Section 3.

<sup>1</sup><https://github.com/big-thesis/Alexanian.EASmells/blob/main/graph-based.ea-smells.detection/graph-based.ea-smells.detection/src/main/resources/SmellExample.xml>

### 6.3.1 Experiment Setup

We experiment with a large set of EA models from the FAIR repository [79]. It currently contains 979 EA models <sup>2</sup> with 104,222 elements, 136,567 relations, and 715 views. Table 6.1 provides further details about the model set. The models are stored in ArchiMate Open Group Exchange Format which is the required format for graph transformation via the CM2KG platform. We utilize a sandbox <sup>3</sup> instance of Neo4j, an online platform that allows for exploration and experimentation with graph databases without the need to set up a local Neo4j environment. Importing GraphML files involves using a procedure called “apoc.import.graphML.” Each GraphML file is imported by specifying its file location with a public URL. To streamline this process, we have uploaded all the files to a git project. The detection platform operates on a laptop. Table 6.2 shows the specifications of the computer utilized for hosting detection platform. It is important to note that our local environment should not impact the model transformation or query execution times, as both the CM2KG platform and Neo4j Sandbox are hosted and accessible on the internet.

Table 6.1: Metadata of the real-world sets of EA Models from the FAIR repository [79]

Models	979
All elements	104,222
Business layer	39,877
Application layer	28,504
Technology layer	11,692
Motivation layer	11,936
Strategy layer	5,427
Implementation/Migration layer	160
Other	5,186
Relations	136,567
Views	715
Duplicates	242

Table 6.2: Local environment hosting the detection platform

Name	HP ProBook 450 G1
Operating System	Windows 10 Professional (x64)
RAM	16 GB (DDR3)
Hard drive	250 GB (SSD)

In this assessment, our goal is not only to determine the presence of different Smells in a model but also the number of occurrence of each Smell per model. However, the queries outlined in Section 3 have limitations when it comes to detecting the number

<sup>2</sup><https://me-big-tuwien-ac-at.github.io/EAModelSet/home>

<sup>3</sup><https://neo4j.com/sandbox>

of each Smell per model. These queries identify the Smells in a model and display the results in graph. Since a Smell may consist of a single or multiple elements, automatically determining the number of detections per Smell is not feasible. As a result, we had to slightly modify the existing queries in order to create new ones that count the occurrences of each detected Smell in the model. We had to redefine almost all queries, except for *Cyclic Dependency* and *Infinite Loop*, because in those cases, the correct number cannot be retrieved using Cypher queries only. Finally, we added the newly defined queries to the EA Smell catalog by introducing a new attribute called *cypher\_count*. However, these new queries will not replace the ones provided in Section 3.

### 6.3.2 Experiment Results

The CM2KG platform was able to transform 947 of 977 models. Few models had encoding issues or the source model in the Exchange File format was not well formed. Furthermore, 19 out of the 947 transformed models could not be imported into the Neo4J Database, reducing the total number of models for evaluation down to 928. For these 19 transformed models, the import process took forever and did not return a response or an exception. It should be borne in mind that analyzing the failed models and verifying the behavior of the CM2KG platform is beyond the scope of our thesis. We use it as an external service for model transformation and assume that it performs correctly. Figure 6.1 demonstrates the detection results for 928 EA models. We found 33 out of the 36 EA Smells provided in Chapter 3. We were able to detect a total of 35,327 EA Smells from 928 models.

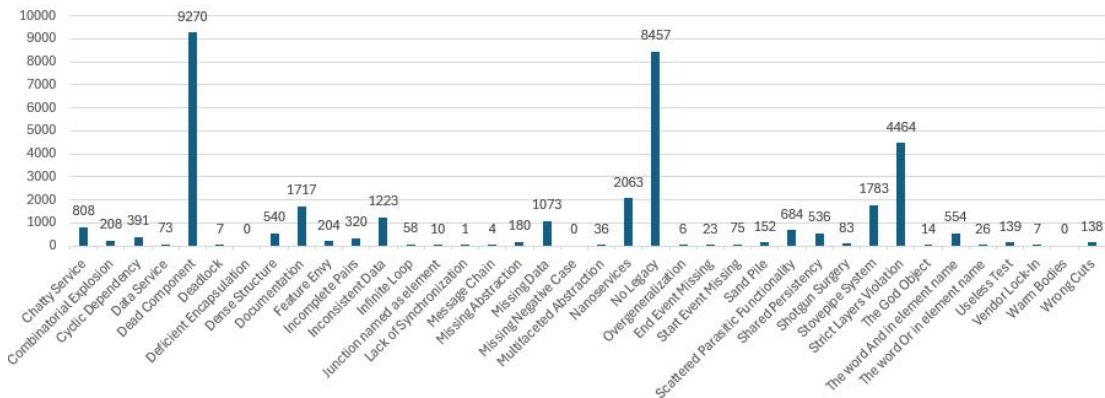


Figure 6.1: Number of detected EA Smells from a set of EA models from FAIR repository

The majority of identified Smells originate from the business and application layers, followed by the technology and motivation layers. Figure 6.2 illustrates the distribution of layers in Smells expressed as percentage. The “other” category represents elements that do not belong to any specific layer, such as junctions or grouping elements.

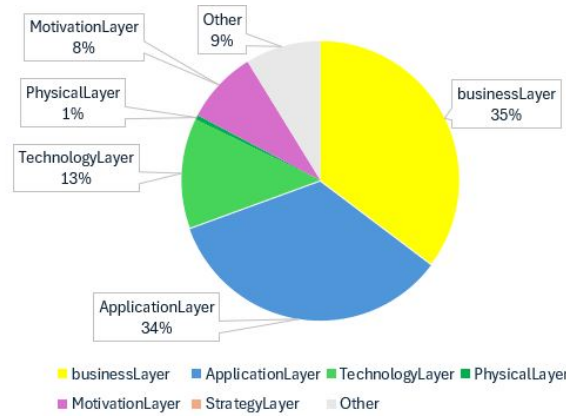


Figure 6.2: Involvement of ArchiMate layers in detected Smells

We have computed the total query execution time for 928 models. Figure 6.3 presents the time taken for each model. The x-axis is the query execution time and the y-axis is the number of elements (vertices and edges) in graph. The query execution time may vary based on the graph's size and structure, as well as the query's complexity and parameter. As shown in Figure 6.3, the execution time of the query generally increases with increasing number of elements. Especially for graphs with a high density, the query requires more time to traverse. Our findings reveal that the longest execution time was 47 seconds for a KG having 1,400 elements, whereas the shortest time was 9 seconds for a KG with 11 elements. Almost 80 percent of the models had a query execution time of less than 20 seconds. EA models with more than 1,000 elements occur rarely in the real world [12]. Given this, we can conclude that the most optimal range of query execution on average took approximately 14-15 seconds (min. 9.5 - max. 47.5) as the majority of real-world cases involve graphs with fewer than 1,000 elements. These results were obtained using our default parameters outlined in Section 3.

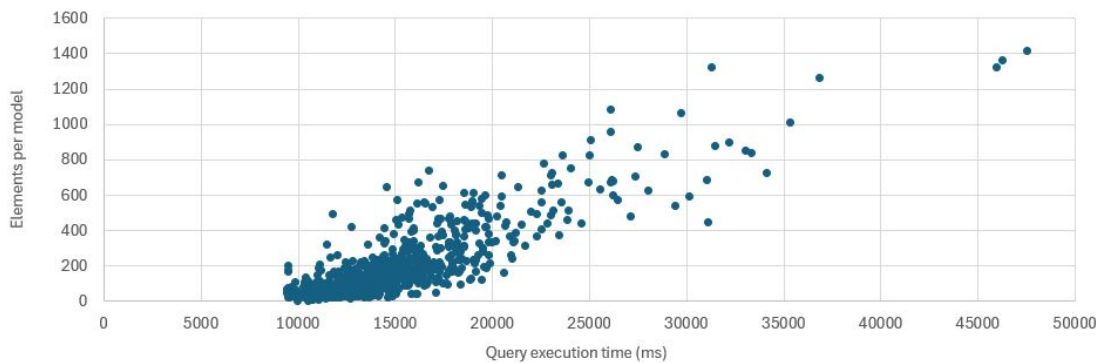


Figure 6.3: KG queries execution time per number of elements in the model

## 6.4 Precision and Recall

In order to address RQ2, we utilized *Precision and Recall* metrics. We were given an ArchiMate model called `ArchiSurance3.1CaseStudy(revised).xml`<sup>4</sup> which had been enhanced by an EA expert with various EA Smells. We transformed the expert model into a GraphML file<sup>5</sup>, imported it into a Neo4j database, and ran all Smell queries using default parameters. Upon sharing our detection results, we received the expected results from the EA expert for comparison.

### 6.4.1 Result

Our Platform has detected 825 EA Smells, of which 19 were false positive and 12 were false negative. In total, the correct number of existing EA Smells in the model were 819. These include both the existing ones and new ones added by the EA expert.

Table 6.3 presents the results detected by our platform alongside the expected results. The comparison between the detected and expected results for each EA Smell is displayed. Smells marked with an asterisk in Table 6.3 mean there is a deviation between the expected and actual results detected by the platform.

Once we collected all the necessary values, we proceeded to calculate the *Precision and Recall* for our platform. Table 6.4 presents the key values for calculating *Precision and Recall* metrics. True positives (TP) refer to the correct number of detected EA Smells on our platform. False positives (FP) indicate the number of EA Smells that our platform incorrectly identified. True negatives (TN) represent instances where model elements that do not belong to any EA Smell are correctly identified. However, we do not factor in TN because our platform is designed solely to detect Smells and does not identify elements that are unaffected by EA Smells. False negatives (FN) are the instances of EA Smells that exist within the model but are not detected by our platform.

Table 6.4: Values of True Positive (TP), False Positive (FP) and False Negative (FN) calculated from Table 6.3

TP= 807	FP= 19
FN= 12	TN= 0

We used the following formula to calculate the *Precision* of our model.

$$\frac{TP}{(TP + FP)} \Rightarrow \frac{807}{(807 + 19)} = 0.97$$

<sup>4</sup>[https://github.com/big-thesis/Alexanian.EASmells/blob/main/graph-based.ea-smells.detection/graph-based.ea-smells.detection/src/main/resources/ArchiSurance3.1CaseStudy\(revised\).xml](https://github.com/big-thesis/Alexanian.EASmells/blob/main/graph-based.ea-smells.detection/graph-based.ea-smells.detection/src/main/resources/ArchiSurance3.1CaseStudy(revised).xml)

<sup>5</sup><https://github.com/big-thesis/Alexanian.EASmells/blob/main/graph-based.ea-smells.detection/graph-based.ea-smells.detection/src/main/resources/ExpertModelgraphMLv1.xml>

Table 6.3: Actual results detected by platform vs expected results given by the EA expert

EA Smells	Detected by platform	Expected
*And in element name	16	1
Chatty Service	7	7
Combinatorial Explosion	3	3
Cyclic dependency	2	2
Data Service	3	3
Dead Component	610	610
Deadlock	1	1
Deficient Encapsulation	1	1
End Event Missing	4	4
Feature Envy	2	2
Incomplete Pairs	8	8
Inconsistent Data	1	1
Infinite Loop	1	1
Junction named as element	1	1
Lack of Synchronization	2	2
Message Chain	6	6
Missing Abstraction	1	1
Missing Data	0	0
Missing Negative Case	4	4
Multifaceted Abstraction	41	41
Nanoservices	63	63
No Legacy	1	1
Or in element name	1	1
*Overgeneralization	0	1
Sand Pile	2	2
Scattered Parasitic Functionality	2	2
Shared Persistency	5	5
*Shotgun Surgery	4	1
Start Event Missing	3	3
Stovepipe System	2	2
*Strict layer Violation	12	13
The God Object	1	1
Useless Test	3	3
*Vendor Lock-In	9	17
*Warm Bodies	1	2
Wrong Cuts	2	2

Consequently, we calculated the *Recall* with following formula:

$$\frac{TP}{(TP + FN)} \Rightarrow \frac{807}{(807 + 12)} = 0.98$$

Despite the absence of TN, we also calculated the accuracy as follows:

$$\frac{TP + TN}{(TP + TN + FP + FN)} \Rightarrow \frac{807 + 0}{(807 + 0 + 19 + 12)} = 0.96$$

*Precision* measures the fraction of detected EA Smells that were actually EA Smells. *Recall* is the ratio of all actual detected EA Smells that were correctly classified as positive. *Accuracy* refers to the ratio of all correct classifications, encompassing both positive and negative outcomes. The lower the values of false negatives (FN) and false positives (FP), the more accurate the results are. In our evaluation, we achieved a precision of 0.97, a recall of 0.98 and 0.96 for accuracy. All three metrics are close to 1, demonstrating that our approach effectively identifies EA Smells.

#### 6.4.2 Discussion

This section discusses the differences in identifying EA Smells between our detection platform and the expected EA Smells from the expert model. Table 6.3 compares the actual results with the expected results, with deviations marked by asterisks.

##### False Positives

The number of FP was 19. It occurred by detection of *The word And in element name* and *Shotgun Surgery* EA Smells. Our platform detected 16 instance of *The word And in element name* EA Smell. However, in reality there was only one instance of this Smell in the Model. The number of false positives (FP) for the *Word And in Element Name* EA Smell was 15. Table 6.5 illustrates the detected instances by our platform. The element “Do C and D” was added by the EA expert, while the rest were already part of the model. In the demonstrated results, some element names contain “And” in between but should be considered as single units, such as “Home and Away Policy” or “Policy and Claim Management.” These should not be classified as Smell candidates. Additionally, when examining the names and verbs as element names, the order becomes evident. For example, “Sales and Distribution” implies that Sales occurs before Distribution, which makes it an incorrect instance of the Smell. The same reasoning applies to “Marketing and Sales,” as traditionally, Marketing precedes Sales. Therefore, as described in Section 3.1.31, they are not considered Smell candidates since the temporal order is clear and visible.

We understood that detecting this Smell solely via KG may not be sufficient. Using an alternative approach, such as Natural Language Processing (NLP), could yield more precise results.

The second Smell resulting in false positives was the *Shotgun Surgery*. According to the definition of the *Shotgun Surgery*, multiple abstractions depend on a single abstraction, meaning any change to a single abstraction may force changes to other abstractions as well. Thus, we have considered the dependency relationships in ArchiMate with a given threshold to define the *Shotgun Surgery* EA Smell (Section 3.1.6). Our platform detected 4 instances of this Smell. However, after comparing the result with the expert representation of *Shotgun Surgery* (Figure 6.4), we have noticed that this instance remained undetected since the EA expert utilized numerous bidirectional *Flow* relationships between abstractions to demonstrate the *Shotgun Surgery* EA Smell. Our defined KG query does not consider *Flow* relationships for detection. Consequently, we concluded that the four instances we detected were false positives and the undetected one was a false negative.

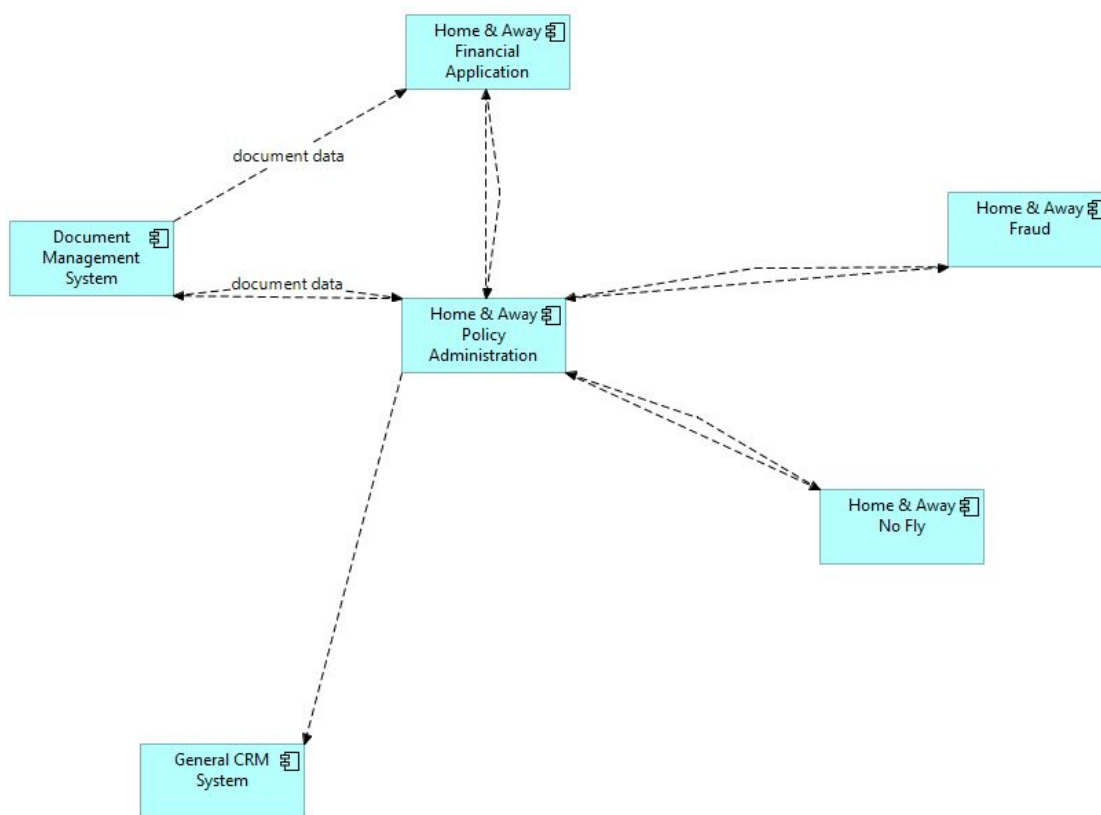


Figure 6.4: The *Shotgun Surgery* EA Smell provided by the expert in the model

### False Negatives

As mentioned earlier, the undetected instance of the *Shotgun Surgery* EA Smell was considered as a false negative. The second variation relates to *Overgeneralization*. Figure 6.5 illustrates the EA Smell proposed by the expert. Our platform failed to detect this EA

Table 6.5: Instances of *The word And in element name* EA Smell identified by detection platform

"Policy and Claim Management"
"Market and Sell Products"
"Manage Policies and Claims"
"Sales And Distribution"
"Marketing and Sales"
"Home and Away Policy Administration"
"Home and Away Financial Application"
"Home and Away LAN"
"Target: CRM, Back Office and Data Warehouse Operational"
"Project: Data Warehousing and BI"
"Homeowners and Travel Back Office"
"Home and Away Headquarters"
"Home and Away Fraud"
"Home and Away No Fly"
"Do C and D"

Smell, because our defined query only considered *Assignment* and *Association* relationships, and not *Realization* relationships. We interpret an overgeneralized component as one responsible for performing and executing various functions and services (Section 3.1.17). In contrast, the expert model represented the *Overgeneralization* Smell as one that implements numerous functions. After discussing the discrepancies with our expert, we concluded that both approaches allow for valid interpretation. However, we regard it as an undetected Smell by our platform.

The third Smell was *Strict Layer Violation* EA. Our platform detected twelve instances of this Smell, excluding the one highlighted by the expert. Figure 6.6 shows the example provided by the expert, which involves an element from the technology layer that realizes an element from the business layer. Referring to a demonstration in Table 3.2b, we considered the *Realization* and *Serving* relationships generally acceptable between layers (Section 3.1.20). However, realizing a *Business Object* with an *Artifact* is syntactically permissible but not advisable, making it a potential Smell candidate.

The last two Smells were *Vendor Lock-in* and *Warm Bodies*. Our detection platform was able to detect 9 out of 17 instances of the *Vendor Lock-in* EA Smell. Upon reviewing the results, we found that detection outcomes varied based on the query parameters used. For instance, the elements represented in the expert models included properties such as “vendor: IBM,” “vendor: Microsoft.” By incorporating these values into our query parameters, we were able to detect all instances of this Smell. Regarding *Warm Bodies*, our platform can detect one of two cases of that Smell. However, by adjusting the query parameters, our platform can also identify the second Smell, which was not possible with the default query parameters.

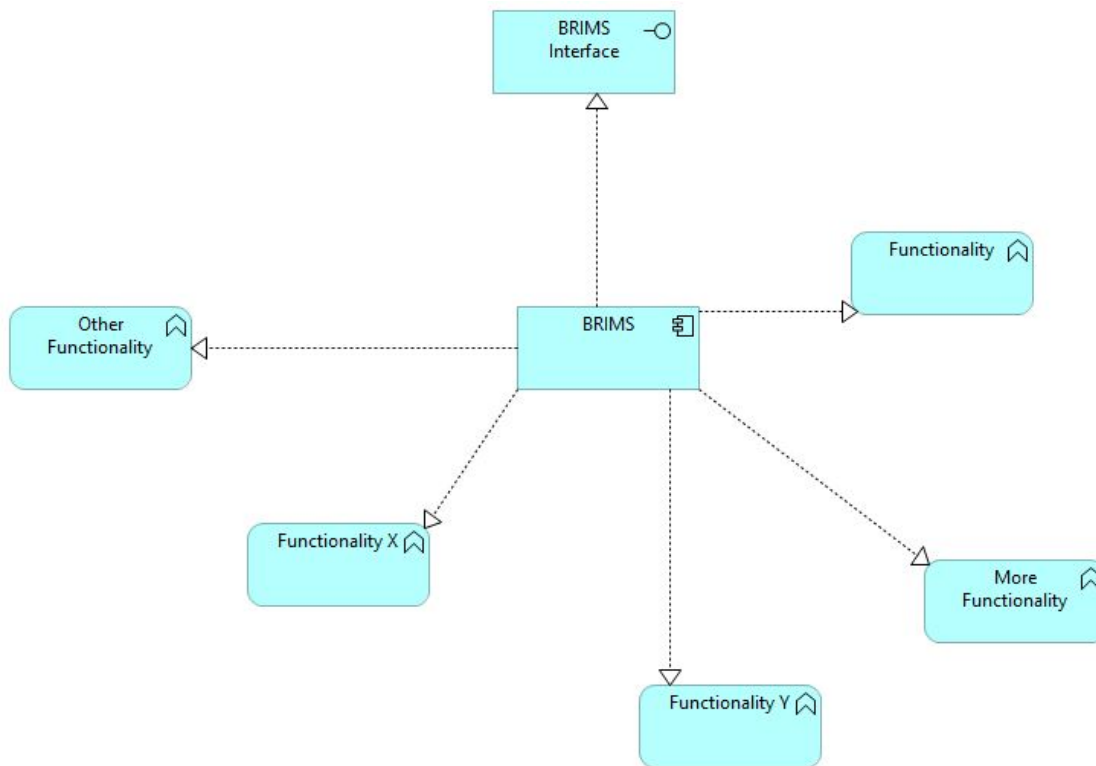


Figure 6.5: The *Overgeneralization* EA Smell provided by the expert in the model

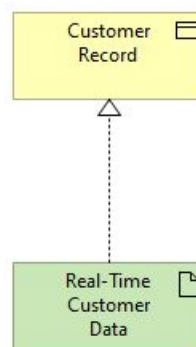


Figure 6.6: The *Strict Layer Violation* EA Smell provided by the expert in the model

### 6.5 Summary

We conducted three different tests to evaluate our provided approach. We verified our detection platform against 36 integration tests to answer the RQ.1. Regarding RQ.2 we used a large set of EA models to test the behavior of our approach in real-world EA models. Using our provided queries, our platform was able to detect 35,327 EA Smells across 928 models. The query execution time was also acceptable. For nearly 80 percent of models, the query execution time was under 20 seconds. Finally, we verified our detection platform with a different set of tests. We achieved a precision of 0.97 and a recall of 0.98, which is considered a satisfactory outcome. Additionally, following a discussion about *Precision and Recall* results with our expert, we can confidently assert that our provided queries exhibit high precision in detecting EA Smells in ArchiMate models.

# Related Work

The research area of the Enterprise Architecture Debt (EAD) [2] and EA Smells is relatively new, and there are currently many research activities in progress. In contrast, there has been extensive research on the TD and the associated Code Smells. Several Code Smell detection tools have been developed, such as *Jdeodorant* [80], *inFusion* <sup>1</sup> [81], *PMD* <sup>2</sup>, and *JSpIRIT* <sup>3</sup> [82]. These tools are limited to the technical domain and do not cover business aspects. As for EA analysis, there are numerous graph-based approaches for analyzing EA models (see Section 2.5). Some analysis and visualization tools have been provided so far [83, 84, 85, 86, 87]. However, these approaches are neither automated, nor used in the context of EAD and EA Smell detection.

**Detection Platform** - Salentin and Hacks [8] were the first to attempt automated detection of EA Smells. They created a prototype in Java [43] that took an ArchiMate Exchange File as input, analyzed 14 EA Smells from the catalog, and printed the detected EA Smells in the console. Benny Tieu [88] extended this prototype for his research [27]. This approach has limitations. It is not scalable for large EA models, and the program needs to be recompiled whenever an EA Smell is added or removed. Additionally, the captured knowledge inside the Java program makes maintainability challenging in the long term. Furthermore, any change in the semantics of EA Smell detection requires modifying the program.

The second prototype used a Knowledge Graph KG-based approach for detecting EA Smells. It was developed by Smajevic, Hacks, and Bork [12] by transforming Conceptual Models to KG (CM2KG) [11]. The CM2KG converts ArchiMate models to GraphML format. KG queries were used to evaluate the same 14 EA Smells as

<sup>1</sup><https://www.intooitus.com/products/infusion/>

<sup>2</sup><https://pmd.github.io/>

<sup>3</sup><https://sites.google.com/site/santiagoavidal/projects/jspirit>

in the previous approach. According to their analysis, KG-based solutions are three to five times faster than the previous approach in detecting EA Smells. However, some queries provided in this work need improvement to enhance accuracy and precision in Smell detection. Our thesis is an extension of the work by Smajevic, Hacks and Bork [12], however, we do not maintain the KG queries directly in the detection platform. Instead, we store them in the EA Smell catalog. We separate the detection logic from the detection platform. With this approach, we can ensure long-term maintainability. Furthermore, we can easily replace our detection platform, written in Java, with another programming language without modifying the detection logic. If we need to change some KG queries, those changes remain transparent to the users. Furthermore, in our approach, users do not need to have any knowledge of KG queries.

Code Smell detection approaches are categorized into five methods, metric, history, rules/heuristics, machine learning and optimization [89]. For EA Smell detection we have thus far focused on metric and graph-based approaches. We use both approaches in our thesis: we use graph queries and at the same time define some threshold for detection.

**EA Smell Catalog** - Hacks et al. [2] introduced the concept of EAD as a metaphor, but their definition did not provide a method for identifying potential debts in EA. In response, Salentin and Hacks [7] introduced the idea of EA Smell by adapting the prominent Code Smells to the domain of EAD. They analyzed and selected Code Smells that are applicable in the EA domain and compiled a catalog [8]. Lehman et al. [26] examined anti-patterns in the field of Business Process Modeling (BPM) and expanded the catalog by adding eighteen new EA Smells. Finally, Tieu and Hacks [27] extended this catalog further by introducing additional EA Smells derived from software architecture Smells. As discussed in Sections 2.3 and 4, the current catalog lacks specific definitions and a clear approach to identifying EA Smells. Most definitions are either too vague or copied from Code Smells or anti-patterns directly without being adjusted for the EA domain. In our thesis, we aim to expand the current catalog, introducing new fields and revising the descriptions as needed to adapt them to the EA domain.

## CHAPTER 8

# Conclusion

### 8.1 Summary

We studied and analyzed the EA Smells from the catalog and provided thirty-six KG queries. To define the queries, we briefly described each Smell at the underlying level and explained how it can arise in the EA domain. We also explained some EA Smells that are not detectable via KG query or in the ArchiMate modeling language. We extended the catalog by incorporating our findings and redefined some definitions in the catalog, having made the descriptions more precise. We also provided graphical examples in the catalog to facilitate understanding. We implemented a platform to automate detection and visualize the results as a graph. In order to ensure long-term maintainability, we kept the detection logic separate from the platform, ensuring that any changes in either platform or catalog do not impact others. We evaluated our solution in both functional and conceptual aspects. The results are promising and provide some hints for improvements and future work.

It is important to note that identifying EA Smells in models does not always reduce the level of EAD. The presence of a Smell in an EA model does not necessarily imply its existence in the real world. However, the ability to detect, understand and distinguish between different Smells can increase awareness of their potential impact on the quality of EA in the short and long term.

Like the metaphor of TD, having debt in EA is sometimes necessary as a trade-off between quality and an acceptable outcome, as long as the debt is paid off in the future.

### 8.2 Future Work

From our evaluation results in Section 6.4 we learned that the evaluation outcome may strongly depend on the thresholds and the query parameters. Modifying thresholds

and parameters will also incur changes in false negatives and false positives. The detection results consisted of positives, false positives and false negatives. Adjusting the thresholds and query parameters was observed to affect those results. Therefore, a comprehensive measurement for defining thresholds and detection parameters would be required. Furthermore, interviewing EA practitioners could provide valuable insights into the impact of EA Smells on EA.

Additionally, another approach for detecting EA Smells could be proposed, and the results could be compared with our work. A concrete example would be using machine learning algorithms for pattern recognition to identify or predict potential EA Smells in models. Furthermore, some EA Smells, such as *Undefined Junction Condition*, *Layout Deficits*, *Language Deficits* and *Deficient Naming*, which are not detectable through KG queries, could potentially be identified using pattern recognition and NLP techniques.

## APPENDIX A

# ArchiMate Elements and Relationships

## A. ARCHIMATE ELEMENTS AND RELATIONSHIPS









Elements								
	Active Structures		Behavior					Passive Structures
								
	Collaboration	Interface	Interaction	Function	Process	Event	Service	Object
BUSINESS LAYER	The collective effort of multiple rules working on a single goal	Makes a business service available to the environment	A business process performed by multiple roles	Business behavior based on resources rather than flow	Business behaviors that achieve a specific outcome, as in BPMN	A state change that triggers or interrupts other business behaviors	Exposes other business elements' functionality	A passive element that has relevance from a business perspective
Examples	Pre-Sales Activity	Service Hotline/ Web Chat	Verify Claim (by Supervisor & Sales Agent)	Assign Claim to Accountant	Pay Invoice	New Invoice Arrived	Payment Service	Invoice
APPLICATION LAYER	Specifies which components are needed to perform some collaborative task	Specifies the way a component's functionality is exposed to the environment	Describes the collective behavior of a connected collaboration	Represents automated, internal behavior of an application component	Internal behavior performed by an application component to achieve a specific outcome	Denotes a state change that may trigger/ interrupt other application behavior	Represents exposed application behavior/ functionality	Contains some piece of information in the form of an object type
Examples	Transaction Administration API (Billing and Accounting)	Transaction API	Communication Pattern	Accounting or Billing	Invoice Creation	Request for Transaction	Transaction Processing	Billing Information
TECHNOLOGY LAYER	Multiple nodes cooperating to perform a collaborative task	Specifies how the functionality of a node can be accessed	Behavior of multiple nodes working together in a technology collaboration	The internal behavior of a node can be described by a Technology Function	Similar to the Technology Function but more focussed on sequences	Instantaneous element that can trigger/ interrupt other technology components	Exposes the functionality of a node and makes it accessible	Represents the type of physical Objects that can be manipulated by the Technology Layer
Examples	Access Control (User and Data Management)	Data Formats	Communication Pattern	Grant Data Access	Authorize User	Request for Data Access	Data Management	Artifact

Figure A.1: ArchiMate elements categorized by active, passive and behaviour elements [90]

Structural Relationships		Notation	Role Names
Composition	Represents that an element consists of one or more other concepts.		→ composed of ← composed in
Aggregation	Represents that an element combines one or more other concepts.		→ aggregates ← aggregated in
Assignment	Represents the allocation of responsibility, performance of behavior, storage, or execution.		→ assigned to ← has assigned
Realization	Represents that an element plays a critical role in the creation, achievement, sustenance, or operation of a more abstract element.		→ realizes ← realized by
Dependency Relationships		Notation	Role Names
Serving	Represents that an element provides its functionality to another element.		→ serves ← served by
Access	Represents the ability of behavior and active structure elements to observe or act upon passive structure elements.		→ accesses ← accessed by
Influence	Represents that an element affects the implementation or achievement of some motivation element.		→ influences ← influenced by
Association	Represents an unspecified relationship, or one that is not represented by another ArchiMate relationship.		associated with → associated to ← associated from
Dynamic Relationships		Notation	Role Names
Triggering	Represents a temporal or causal relationship between elements.		→ triggers ← triggered by
Flow	Represents transfer from one element to another.		→ flows to ← flows from
Other Relationships		Notation	Role Names
Specialization	Represents that an element is a particular kind of another element.		→ specializes ← specialized by
Relationship Connectors		Notation	Role Names
Junction	Used to connect relationships of the same type.	 (And) Junction Or Junction	

Figure A.2: Different types of relationships in ArchiMate [9]



# List of Figures

2.1	Technical Debt Quadrant [21] . . . . .	8
2.2	Placing of TD within the concept of EAD [2] . . . . .	9
2.3	Current representation of the <i>Combinatorial Explosion</i> EA Smell in the catalog	10
2.4	The ArchiMate Full Framework taken from [9] . . . . .	11
2.5	ArchiMate Assignment Relationship . . . . .	12
2.6	ArchiMate Composition Relationship . . . . .	12
2.7	ArchiMate Aggregation Relationship . . . . .	12
2.8	ArchiMate Realization Relationship . . . . .	13
2.9	ArchiMate Access and Triggering Relationships . . . . .	13
2.10	ArchiMate Influence relationship . . . . .	13
2.11	ArchiMate Serving Relationship . . . . .	14
2.12	ArchiMate Association Relationship . . . . .	14
2.13	ArchiMate Triggering Relationship . . . . .	14
2.14	ArchiMate Flow Relationship . . . . .	14
2.15	ArchiMate Specialization Relationship . . . . .	15
2.16	CM2KG, a generic platform for transforming models into graphs [11] . . .	16
3.1	An example of an abstract and many aggregated sub elements in an ArchiMate model . . . . .	52
3.2	Permitted relationships between the core layers of ArchiMate . . . . .	54
3.3	Example of a God Object Web Service anti-pattern taken from [68] . . . . .	61
3.4	Data flow errors in WFD-nets taken from [71] . . . . .	76
3.5	<i>Undefined junction condition</i> EA Smell in ArchiMate . . . . .	80
4.1	Current representation of the <i>Useless Test</i> EA Smells in the catalog . . . .	85
4.2	Current representation of the <i>Deficient Encapsulation</i> EA Smell in the catalog	91
4.3	Current representation of the <i>Multifaceted Abstraction</i> EA Smell in the catalog	92
5.1	General overview of the EA Smell detection platform . . . . .	96
5.2	Interactions between the detection platform and the EA Smell catalog . .	97
5.3	CM2KG model transformation taken from [13] . . . . .	99
5.4	General overview of the EA Smell detection platform . . . . .	101
5.5	Detection section, illustrating detected EA Smells in textual and visual forms.	101
6.1	Number of detected EA Smells from a set of EA models from FAIR repository	106
		123

6.2	Involvement of ArchiMate layers in detected Smells . . . . .	107
6.3	KG queries execution time per number of elements in the model . . . . .	107
6.4	The <i>Shotgun Surgery</i> EA Smell provided by the expert in the model . . . . .	111
6.5	The <i>Overygeneralization</i> EA Smell provided by the expert in the model . . . . .	113
6.6	The <i>Strict Layer Violation</i> EA Smell provided by the expert in the model . . . . .	113
A.1	ArchiMate elements categorized by active, passive and behaviour elements [90] . . . . .	120
A.2	Different types of relationships in ArchiMate [9] . . . . .	121

## List of Tables

3.1	Possible parameters for <code>apoc.nodes.cycles</code> procedure . . . . .	21
3.2	Possible parameters for <code>apoc.path.expand</code> procedure . . . . .	22
3.3	Graphical example, Cypher query and query options for detecting <i>Missing Abstraction</i> EA Smells . . . . .	23
3.4	Symmetric pairs taken from [45, p. 37] . . . . .	24
3.5	Graphical example, Cypher query and query options for detecting <i>Incomplete Pairs</i> EA Smells . . . . .	25
3.6	Graphical example and Cypher query for detecting <i>Multifaceted Abstraction</i> EA Smells . . . . .	26
3.7	Graphical example and Cypher query for detecting <i>Data Service</i> EA Smells . . . . .	28
3.8	Graphical example and Cypher query for detecting <i>Feature Envy</i> EA Smells . . . . .	29
3.9	Graphical example, Cypher query and query options for detecting <i>Shotgun Surgery</i> EA Smells . . . . .	30
3.10	Graphical example and the Cypher query for detecting <i>Scattered Parasitic Functionality</i> EA Smells . . . . .	31
3.11	Graphical example, Cypher query and query options for detecting <i>Deficient Encapsulation</i> EA Smells . . . . .	33
3.12	Graphical example and Cypher query for detecting <i>Wrong Cuts</i> EA Smells . . . . .	35
3.13	Graphical example and Cypher query for detecting <i>Dead Component</i> EA Smells . . . . .	36
3.14	Graphical example, Cypher query and query options for detecting <i>Vendor Lock-In</i> EA Smells . . . . .	37
3.15	Graphical example, Cypher query and query options for detecting <i>No Legacy</i> EA Smells . . . . .	39
3.16	Graphical example, Cypher query and query options for detecting <i>Warm Bodies</i> EA Smells . . . . .	41
3.17	Graphical example, Cypher query and query options for detecting <i>Combinatorial Explosion</i> EA Smells . . . . .	42
3.18	Graphical example, Cypher query and query options for detecting <i>Stovepipe System</i> EA Smells . . . . .	44
3.19	Graphical example, Cypher query and query options for detecting <i>Nanoservices</i> EA Smells . . . . .	46
3.20	Graphical example, Cypher query and query options for detecting <i>Overgeneralization</i> EA Smells . . . . .	48
		125

3.21	Graphical example and Cypher query for detecting <i>Sand Pile</i> EA Smells .	49
3.22	Graphical example, Cypher query and query options for detecting <i>Missing Abstraction</i> EA Smells . . . . .	52
3.23	Graphical example and Cypher query for detecting <i>Strict Layer Violation</i> EA Smells . . . . .	55
3.24	Graphical example, Cypher query and query options for detecting <i>Message Chain</i> EA Smell . . . . .	56
3.25	Graphical example and Cypher query for detecting the <i>Shared Persistency</i> EA Smells . . . . .	57
3.26	Graphical example, Cypher query and query options for detecting <i>Chatty Service</i> EA Smells . . . . .	58
3.27	Graphical example, Cypher query and query options for detecting <i>Infinite Loop</i> EA Smells . . . . .	59
3.28	Graphical example, Cypher query and query options for detecting <i>God Object</i> EA Smells . . . . .	61
3.29	Graphical example, Cypher query and query options for detecting <i>Lack of Synchronization</i> EA Smells . . . . .	63
3.30	Graphical example, Cypher query and query options for detecting <i>Deadlock</i> EA Smells . . . . .	65
3.31	Graphical example, Cypher query and query options for detecting <i>Inconsistent Data</i> EA Smells . . . . .	66
3.32	Graphical example, Cypher query and query options for detecting <i>The word Or in element name</i> EA Smells . . . . .	68
3.33	Graphical example, Cypher query and query options for detecting <i>Useless Test</i> EA Smells . . . . .	69
3.34	Graphical example and Cypher query for detecting <i>The word And in element name</i> EA Smells . . . . .	70
3.35	Graphical example, Cypher query and query options for detecting <i>Start Event Missing</i> EA Smells . . . . .	71
3.36	Graphical example, Cypher query and query options for detecting <i>End Event Missing</i> EA Smells . . . . .	73
3.37	Graphical example, Cypher query and query options for detecting <i>Missing negative case</i> EA Smells . . . . .	75
3.38	Graphical example, Cypher query and query options for detecting <i>Missing Data</i> EA Smells . . . . .	77
3.39	Graphical example, Cypher query and query options for detecting <i>Junction named as element</i> EA Smells . . . . .	79
3.40	Undetectable and out of scope EA Smells . . . . .	80
4.1	Two schemes for representing EA Smells . . . . .	84
4.2	<i>Useless Test</i> EA Smell after unification, extension and revision . . . . .	86
4.3	<i>Deficient Encapsulation</i> EA Smell after extension and revision . . . . .	87
4.4	<i>Multifaceted Abstraction</i> EA Smell after extension and revision . . . . .	89

5.1	<i>REST API</i> for retrieving EA Smells from the catalog . . . . .	97
5.2	<i>REST API</i> for transforming Archi exchange files to GraphML . . . . .	100
6.1	Metadata of the real-world sets of EA Models from the FAIR repository [79]	105
6.2	Local environment hosting the detection platform . . . . .	105
6.4	Values of True Positive (TP), False Positive (FP) and False Negative (FN) calculated from Table 6.3 . . . . .	108
6.3	Actual results detected by platform vs expected results given by the EA expert	109
6.5	Instances of <i>The word And in element name</i> EA Smell identified by detection platform . . . . .	112



# Acronyms

- AI** Artificial Intelligence. 15
- BPM** Business Process Modeling. 2, 58, 62, 64, 65, 67–70, 72, 74, 76, 80–82, 84, 86, 116
- BPMN** Business Process Modeling Notation. 71, 72
- CM2KG** Conceptual Model to Knowledge Graph. 3–5, 7, 16, 19, 95, 96, 98–101, 103–106, 115, 123
- DSR** Design Science Research. 3
- EA** Enterprise Architecture. ix–xiii, 1–5, 7–10, 15–17, 19–89, 91–93, 95–98, 100–106, 108–118, 123–127
- EAD** Enterprise Architecture Debt. ix, xi, 1, 2, 4, 5, 7–9, 16, 115–117, 123
- HMMs** Hidden Markov Models. 15
- IoT** Internet of Things. 15
- KG** Knowledge Graph. ix–xi, 2–5, 7, 15–17, 19–21, 23, 24, 26–31, 33–36, 39–42, 44, 45, 47, 51, 54, 55, 57, 59, 60, 62, 66, 68, 70–72, 74, 76, 79–81, 83, 85, 89, 95, 103, 104, 107, 110, 111, 115–118, 124
- ML** Machine Learning. 15
- NLP** Natural Language Processing. 70, 81, 110, 118
- SOA** Service-Oriented Architecture. 27, 49
- TD** Technical Debt. ix, xi, 1, 2, 4, 5, 7–9, 16, 115, 123
- WFD-nets** Workflow nets with Data. 75, 76, 123



# Bibliography

- [1] Karl E Kurbel. Developing information systems. *The making of information systems: Software engineering and management in a globalized world*, pages 155–234, 2008.
- [2] Simon Hacks, Hendrik Höfert, Johannes Salentin, Yoon Chow Yeong, and Horst Lichter. Towards the definition of enterprise architecture debts. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 9–16. IEEE, 2019.
- [3] Muhamed Smajevic and Dominik Bork. Towards graph-based analysis of enterprise architecture models. In *International Conference on Conceptual Modeling*, pages 199–209. Springer, 2021.
- [4] Amanda Barbosa, Alixandre Santana, Simon Hacks, and Niels von Stein. A taxonomy for enterprise architecture analysis research. In *21st International Conference on Enterprise Information Systems*, volume 2, pages 493–504. SciTePress, 2019.
- [5] Maria-Eugenia Iacob and Henk Jonkers. Quantitative analysis of enterprise architectures. In *Interoperability of enterprise software and applications*, pages 239–252. Springer, 2006.
- [6] Edith Tom, Aybüke Aurum, and Richard Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.
- [7] Johannes Salentin and Simon Hacks. Towards a catalog of enterprise architecture smells. In *Wirtschaftsinformatik (Community Tracks)*, pages 276–290, 2020.
- [8] Salentin J., Lehmann B., Hacks S., and Alexander P. Enterprise architecture smells catalog (2021). <https://swc-public.pages.rwth-aachen.de/smells/ea-smells/>.
- [9] Omg: Archimate® 3.2 specification. the open group (2023). <https://pubs.opengroup.org/architecture/archimate3-doc>.
- [10] Hector Florez, Mario Sánchez, and Jorge Villalobos. A catalog of automated analysis methods for enterprise models. *SpringerPlus*, 5(1):1–24, 2016.

- [11] Muhamed Smajevic and Dominik Bork. From conceptual models to knowledge graphs: A generic model transformation platform. In *ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion, MODELS 2021 Companion, Fukuoka, Japan, October 10-15, 2021*, pages 610–614. IEEE, 2021.
- [12] Muhamed Smajevic, Simon Hacks, and Dominik Bork. Using knowledge graphs to detect enterprise architecture smells. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 48–63. Springer, 2021.
- [13] Conceptual model to knowledge graph (cm2kg) platform. <https://github.com/borkdominik/CM2KG>.
- [14] Alan R Hevner, Salvatore T March, Jinsoo Park, and Sudha Ram. Design science in information systems research. *MIS quarterly*, pages 75–105, 2004.
- [15] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. A design science research methodology for information systems research. *Journal of management information systems*, 24(3):45–77, 2007.
- [16] Archimate® model exchange file format for the archimate 3.1 modeling language. <https://www.opengroup.org/xsd/archimate/>.
- [17] Ward Cunningham. The wycash portfolio management system. *ACM Sigplan Oops Messenger*, 4(2):29–30, 1992.
- [18] Carolyn Seaman, Yuepu Guo, Nico Zazworka, Forrest Shull, Clemente Izurieta, Yuanfang Cai, and Antonio Vetrò. Using technical debt data in decision making: Potential decision approaches. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pages 45–48. IEEE, 2012.
- [19] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52, 2010.
- [20] RC Martin. A mess is not a technical debt, 2009.
- [21] Martin Fowler. Technical debt quadrant, 2009. URL: <http://martinfowler.com/bliki/TechnicalDebtQuadrant.html>, 2009. (last accessed on 2024-01-31).
- [22] Patrick Saint-Louis, Marcklyvens C Morency, and James Lapalme. Defining enterprise architecture: A systematic literature review. In *2017 IEEE 21st international enterprise distributed object computing workshop (EDOCW)*, pages 41–49. IEEE, 2017.
- [23] Leon Kappelman, Tom McGinnis, Alex Pettite, and Anna Sidorova. Enterprise architecture: Charting the territory for academic research. 2008.

- [24] Togaf as an enterprise architecture framework, 2006. <https://pubs.opengroup.org/architecture/togaf8-doc/arch/chap01.html> [Accessed: (11.11.2024)].
- [25] Marc Lankhorst. *Enterprise Architecture at Work: Modelling, Communication and Analysis*. Springer, 2017.
- [26] Barry-Detlef Lehmann, Peter Alexander, Horst Lichter, Simon Hacks, S Aydin, T Sunetnanta, and T Anwar. Towards the identification of process anti-patterns in enterprise architecture models. In *QuASoQ@ APSEC*, pages 47–54, 2020.
- [27] Benny Tieu and Simon Hacks. Determining enterprise architecture smells from software architecture smells. In *2021 IEEE 23rd Conference on Business Informatics (CBI)*, volume 2, pages 134–142. IEEE, 2021.
- [28] Stephan Aier. How clustering enterprise architectures helps to design service oriented architectures. In *2006 IEEE International Conference on Services Computing (SCC'06)*, pages 269–272. IEEE, 2006.
- [29] Pontus Johnson, Mathias Ekstedt, and Robert Lagerstrom. Automatic probabilistic enterprise it architecture modeling: a dynamic bayesian networks approach. In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 123–129. IEEE, 2016.
- [30] Björn Bebensee and Simon Hacks. Applying dynamic bayesian networks for automated modeling in archimate: a realization study. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Workshop (EDOCW)*, pages 17–24. IEEE, 2019.
- [31] Simon Hacks and Horst Lichter. A probabilistic enterprise architecture model evolution. In *2018 IEEE 22nd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 51–57. IEEE, 2018.
- [32] Vassilis Giakoumakis, Daniel Krob, Leo Liberti, and Fabio Roda. Technological architecture evolutions of information systems: Trade-off and optimization. *Concurrent Engineering*, 20(2):127–147, 2012.
- [33] Ulrik Franke, Oliver Holschke, Markus Buschle, Per Narman, and Jannis Rake-Revelant. It consolidation: an optimization approach. In *2010 14th IEEE International Enterprise Distributed Object Computing Conference Workshops*, pages 21–26. IEEE, 2010.
- [34] Alan D Maccormack, Robert Lagerstrom, and Carliss Young Baldwin. A methodology for operationalizing enterprise architecture and evaluating enterprise it flexibility. *Harvard Business School working paper series# 15-060*, 2015.

- [35] Muhamed Smajevic, Syed Juned Ali, and Dominik Bork. Cm2kg<sup>cloud</sup> - an open web-based platform to transform conceptual models into knowledge graphs. *Sci. Comput. Program.*, 231:103007, 2024.
- [36] Luigi Bellomarini, Daniele Fakhoury, Georg Gottlob, and Emanuel Sallinger. Knowledge graphs and enterprise ai: the promise of an enabling technology. In *2019 IEEE 35th international conference on data engineering (ICDE)*, pages 26–37. IEEE, 2019.
- [37] Peter Mika, Abraham Bernstein, Chris Welty, Craig Knoblock, Denny Vrandečić, Paul Groth, Natasha Noy, Krzysztof Janowicz, and Carole Goble. *The Semantic Web-ISWC 2014: 13th International Semantic Web Conference, Riva del Garda, Italy, October 19-23, 2014. Proceedings, Part II*, volume 8797. Springer, 2014.
- [38] L Ehrlinger and W Wofi. Towards a definition of knowledge graphs. semantics. In *Proceedings of 12th International Conference on Semantic Systems SEMANTiCS 2016, CEUR Workshop Proceedings*, volume 1695, 2016.
- [39] Viktor Beneš and Miroslav Svítek. Knowledge graphs for smart cities. In *2022 Smart City Symposium Prague (SCSP)*, pages 1–6. IEEE, 2022.
- [40] Fatima Zohra Smaili, Xin Gao, and Robert Hoehndorf. Opa2vec: combining formal and informal content of biomedical ontologies to improve similarity-based prediction. *Bioinformatics*, 35(12):2133–2140, 2019.
- [41] Aditya Garg, Rick Kazman, and Hong-Mei Chen. Interface descriptions for enterprise architecture. *science of Computer Programming*, 61(1):4–15, 2006.
- [42] Alixandre Santana, Kai Fischbach, and Hermano Moura. Enterprise architecture analysis and network thinking: A literature review. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 4566–4575. IEEE, 2016.
- [43] Salentin J. and Hacks S. Enterprise architecture smells prototype (2020). <https://git.rwth-aachen.de/ba-ea-smells/program>.
- [44] Parameters - Cypher Manual — neo4j.com. <https://neo4j.com/docs/cypher-manual/current/syntax/parameters/>. [Accessed 20-12-2023].
- [45] Girish Suryanarayana, Ganesh Samarthayam, and Tushar Sharma. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.
- [46] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE software*, 35(3):56–62, 2018.
- [47] Francis Palma and Naouel Mohay. A study on the taxonomy of service antipatterns. In *2015 IEEE 2nd International Workshop on Patterns Promotion and Anti-patterns Prevention (PPAP)*, pages 5–8. IEEE, 2015.

- [48] DM Hutton. Clean code: a handbook of agile software craftsmanship. *Kybernetes*, 38(6):1035–1035, 2009.
- [49] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- [50] Mika V Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11:395–431, 2006.
- [51] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2014.
- [52] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258. IEEE, 2009.
- [53] Martin Fowler James Lewis. Microservices, a definition of this new architectural term). <https://martinfowler.com/articles/microservices.html>.
- [54] Rafik Tighilt, Manel Abdellatif, Naouel Moha, Hamed Mili, Ghizlane El Boussaidi, Jean Privat, and Yann-Gaël Guéhéneuc. On the study of microservices antipatterns: A catalog proposal. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, pages 1–13, 2020.
- [55] William J Brown, Raphael C Malveau, Hays W McCormick III, and Thomas J Mowbray. *Refactoring software, architectures, and projects in crisis*. 1998.
- [56] Lukas Liss, Henrik Kämmerling, Peter Alexander, and Horst Lichter. Towards a catalog of refactoring solutions for enterprise architecture smells. In *SEED/QuASoQ@APSEC*, pages 60–69, 2021.
- [57] Jaroslav Kral and Michal Zemlicka. The most important service-oriented antipatterns. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 29–29. IEEE, 2007.
- [58] Jaroslav Král and Michal Zemlicka. Crucial service-oriented antipatterns. *International Academy, Research and Industry Association (IARIA)*, pages 160–171, 2008.
- [59] Jeri Edwards and D. Devoe. "10 tips for three tier success, d.o.c. magazine". pages 39–42, 1997.
- [60] William E Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. 1990.

- [61] Francis Palma, Naouel Moha, Guy Tremblay, and Yann-Gaël Guéhéneuc. Specification and detection of soa antipatterns in web services. In *European Conference on Software Architecture*, pages 58–73. Springer, 2014.
- [62] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In *Architectures for Adaptive Software Systems: 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24-26, 2009 Proceedings 5*, pages 146–162. Springer, 2009.
- [63] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. Does your configuration code smell? In *Proceedings of the 13th International Conference on Mining Software Repositories*, pages 189–200, 2016.
- [64] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.
- [65] Martin Lippert and Stefan Roock. Refactorings in large software projects: How to successfully execute complex restructurings, 2006.
- [66] Mathieu Nayrolles, Naouel Moha, and Petko Valtchev. Improving soa antipatterns detection in service based systems by mining execution traces. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 321–330. IEEE, 2013.
- [67] Agnes Koschmider, Ralf Laue, and Michael Fellmann. Business process model anti-patterns: a bibliography and taxonomy of published work. 2019.
- [68] Hanzhang Wang, Ali Ouni, Marouane Kessentini, Bruce Maxim, and William I Grosky. Identification of web service refactoring opportunities as a multi-objective problem. In *2016 IEEE International Conference on Web Services (ICWS)*, pages 586–593. IEEE, 2016.
- [69] Ralf Laue and Ahmed Awad. Visualization of business process modeling anti patterns. *Electronic Communications of the EASST*, 25, 2010.
- [70] Silvia Von Stackelberg, Susanne Putze, Jutta Mülle, and Klemens Böhm. Detecting data-flow errors in bpmn 2.0. *Open Journal of Information Systems (OJIS)*, 1(2):1–19, 2014.
- [71] Nikola Trčka, Wil MP Van der Aalst, and Natalia Sidorova. Data-flow anti-patterns: Discovering data-flow errors in workflows. In *International Conference on Advanced Information Systems Engineering*, pages 425–439. Springer, 2009.
- [72] Ralf Laue, Wilhelm Koop, and Volker Gruhn. Indicators for open issues in business process models. In *International Working Conference on Requirements Engineering: Foundation for Software Quality*, pages 102–116. Springer, 2016.

- [73] Tomislav Rozman, Gregor Polancic, and Romana Vajde Horvat. Analysis of most common process modeling mistakes in bpmn process models. *Eur SPI'2007*, 2008.
- [74] Markus Döhning and Steffen Heublein. Anomalies in rule-adapted workflows-a taxonomy and solutions for vbpmn. In *2012 16th European Conference on Software Maintenance and Reengineering*, pages 117–126. IEEE, 2012.
- [75] Design Patterns and Refactoring — sourcemaking.com. <https://sourcemaking.com/antipatterns/architecture-by-implication>. [Accessed 13-12-2023].
- [76] Juan Manuel Rodriguez, Marco Crasso, Alejandro Zunino, and Marcelo Campo. Automatically detecting opportunities for web service descriptions improvement. In *Software Services for e-World: 10th IFIP WG 6.11 Conference on e-Business, e-Services, and e-Society, I3E 2010, Buenos Aires, Argentina, November 3-5, 2010. Proceedings 10*, pages 139–150. Springer, 2010.
- [77] Ali Ouni, Raula Gaikovina Kula, Marouane Kessentini, and Katsuro Inoue. Web service antipatterns detection using genetic programming. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 1351–1358, 2015.
- [78] Mark Richards. *Microservices antipatterns and pitfalls*. O'Reilly Media, Incorporated, 2016.
- [79] Philipp-Lorenz Glaser, Emanuel Sallinger, and Dominik Bork. Ea modelset—a fair dataset for machine learning in enterprise modeling. In *IFIP Working Conference on The Practice of Enterprise Modeling*, pages 19–36. Springer, 2023.
- [80] Nikolaos Tsantalis, Theodoros Chaikalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. In *2008 12th European conference on software maintenance and reengineering*, pages 329–331. IEEE, 2008.
- [81] Thanis Paiva, Amanda Damasceno, Eduardo Figueiredo, and Cláudio Sant'Anna. On the evaluation of code smells and detection tools. *Journal of Software Engineering Research and Development*, 5:1–28, 2017.
- [82] Santiago Vidal, Hernan Vazquez, J Andres Diaz-Pace, Claudia Marcos, Alessandro Garcia, and Willian Oizumi. Jspirit: a flexible tool for the analysis of code smells. In *2015 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6. IEEE, 2015.
- [83] Dominik Bork, AURONA Gerber, Elena-Teodora Miron, Phil van Deventer, Alta Van der Merwe, Dimitris Karagiannis, Sunet Eybers, and Anna Sumeder. Requirements engineering for model-based enterprise architecture management with archimate. In *Enterprise and Organizational Modeling and Simulation: 14th International Workshop, EOMAS 2018, Held at CAiSE 2018, Tallinn, Estonia, June 11–12, 2018, Selected Papers 14*, pages 16–30. Springer, 2018.

- [84] Fabian Gampfer, Andreas Jürgens, Markus Müller, and Rüdiger Buchkremer. Past, current and future trends in enterprise architecture—a view beyond the horizon. *Computers in Industry*, 100:70–84, 2018.
- [85] Knut Hinkelmann, AURORA Gerber, Dimitris Karagiannis, Barbara Thoenssen, Alta Van der Merwe, and Robert Woitsch. A new paradigm for the continuous alignment of business and it: Combining enterprise architecture modelling and enterprise ontology. *Computers in Industry*, 79:77–86, 2016.
- [86] Dierk Jugel. An integrative method for decision-making in ea management. *Architecting the Digital Transformation: Digital Business, Technology, Decision Support, Management*, pages 289–307, 2021.
- [87] Ben Roelens, Wout Steenacker, and Geert Poels. Realizing strategic fit within the business architecture: the design of a process-goal alignment modeling and analysis technique. *Software & Systems Modeling*, 18:631–662, 2019.
- [88] Benny Tieu. Detecting enterprise architecture smells based on softwarearchitecture smells prototype (2021). <https://github.com/bennytieu/ea-smell-detector-prototype>.
- [89] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.
- [90] ArchiMate® Quick Reference Guide - SAP Signavio — signavio.com. <https://signavio.com/downloads/short-reads/archimate-quick-reference-guide/>. [Accessed 12-12-2023].