universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

Understanding and Semi-automatically Supporting DDD-based API Design

verfasst von / submitted by

APITCHAKA SINGJAI

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. techn.)

Wien, 2022 / Vienna 2022

Studienkennzahl lt. Studienblatt / Degree programme code as it appears on the student record sheet:	A 786 880
Dissertationsgebiet lt. Studienblatt / Field of study as it appears on the student record sheet:	Informatik / Computer Science
Betreut von / Supervisor:	Univ.-Prof. Dr. UWE ZDUN

Contents

Declaration of Authorship

I, Apitchaka Singjai, declare that this thesis with the title, "Understanding and Semi-automatically Supporting DDD-based API Design" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on works done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

The Domain-Driven Design (DDD) approach, which is becoming increasingly popular, places the domain model at the core of the software development practices. The outwardly apparent interfaces of systems modeled using DDD are frequently Application Programming Interfaces (APIs). As a result, the challenge of how to map DDD models to the API while preserving their key characteristics emerges. If we could better understand the overlapping concepts and relations, we would be able to provide better approaches and tools to support this mapping.

The purpose of this dissertation is to investigate the phenomena of API and DDD, as well as their mapping, with special attention to coupling aspects. We aim to analyze Architectural Design Decisions (ADDs) and synthesize empirical data for pattern mining. We intend to assess conformance to patterns, practices, and ADDs in order to identify violations and enhance the DDD-based API design.

Our research is divided into three parts: establishing empirical evidence, pattern mining, and empirical assessment. Regarding the empirical evidence, we employed the Grounded Theory method to elicit practitioner perspectives on how they perceive the context of the interaction between APIs and DDD and the related context of coupling smells. We analyzed data from the empirical evidence and existing patterns, and then mined patterns in this field. Finally, we developed novel conformance assessment methods to support the automated conformance assessment between APIs and their DDD models.

As a result, we could identify six ADDs with 27 decision options, numerous relations between them, and 27 decision drivers. We also defined factors of coupling smells, as well as smell impacts, trade-offs, linkages with other smells, relationships to practices and patterns, and fix options as perceived by practitioners. We mined two sets of patterns based on the empirical data, namely patterns on deriving APIs and API endpoints from domain model elements and patterns on designing API endpoint operations. For the conformance assessment, we provided detector, parser, and assessor tools to support an automated approach. Multi-case studies were used to evaluate our research results.

Zusammenfassung

Der Ansatz des Domain-Driven Design (DDD), der sich zunehmender Beliebtheit erfreut, stellt das Domänenmodell in den Mittelpunkt der Softwareentwicklungspraktiken. Die nach außen hin sichtbaren Schnittstellen von Systemen, die mit DDD modelliert werden, sind häufig Anwendungsprogrammierschnittstellen (APIs). Daraus ergibt sich die Herausforderung, DDD-Modelle auf die API abzubilden und dabei ihre wichtigsten Merkmale beizubehalten. Wenn wir die sich überschneidenden Konzepte und Beziehungen besser verstehen könnten, wären wir in der Lage, bessere Ansätze und Werkzeuge zur Unterstützung dieser Abbildung bereitzustellen.

Ziel dieser Dissertation ist es, die Phänomene API und DDD sowie deren Abbildung zu untersuchen, mit besonderem Augenmerk auf Kopplungsaspekte. Unser Ziel ist es, architektonische Entwurfsentscheidungen (ADDs) zu analysieren und empirische Daten für das Pattern Mining zu synthetisieren. Wir beabsichtigen, die Konformität mit Mustern, Praktiken und ADDs zu bewerten, um Verstöße zu identifizieren und das DDD-basierte API-Design zu verbessern.

Unsere Forschung gliedert sich in drei Teile: Ermittlung empirischer Belege, Pattern-Mining und empirische Bewertung. Was die empirische Evidenz anbelangt, haben wir die Methode der Grounded Theory angewandt, um die Sichtweise von Praktikern zu eruieren, wie sie den Kontext der Interaktion zwischen APIs und DDD und den damit verbundenen Kontext der Kopplungs-Smells wahrnehmen. Wir analysierten Daten aus der empirischen Evidenz und existierende Muster und untersuchten dann Muster in diesem Bereich. Schließlich entwickelten wir neuartige Konformitätsbewertungsmethoden zur Unterstützung der automatisierten Konformitätsbewertung zwischen APIs und ihren DDD-Modellen.

Als Ergebnis konnten wir sechs ADDs mit 27 Entscheidungsoptionen, zahlreichen Beziehungen zwischen ihnen und 27 Entscheidungstreibern identifizieren. Wir definierten auch Faktoren für die Kopplung von Smells sowie deren Auswirkungen, Kompromisse, Verknüpfungen mit anderen Smells, Beziehungen zu Praktiken und Mustern und Fixierungsoptionen, wie sie von Praktikern wahrgenommen werden. Auf der Grundlage der empirischen Daten haben wir zwei Gruppen von Mustern ermittelt, nämlich Muster für die Ableitung von APIs und API-Endpunkten aus Domänenmodellelementen und Muster für den Entwurf von API-Endpunktoperationen. Für die Konformitätsbewertung haben wir Detektoren, Parser und Bewertungswerkzeuge bereitgestellt, um einen automatisierten Ansatz zu unterstützen. Zur Bewertung unserer Forschungsergebnisse wurden mehrere Fallstudien

durchgeführt.

Acknowledgments

Words cannot express my gratitude to Prof. Uwe Zdun for his invaluable patience and feedback. He always provided continuous support and the right direction at the right time. His direction influenced my scientific reasoning to the point where I was able to produce works that exceeded my expectations.

I also could not have undertaken this journey without my defense committee and my co-advisor, Prof. Gerald Quirchmayr, who generously provided knowledge and expertise. Additionally, this endeavor would not have been possible without the generous support from the College of Art Media and Technology (CAMT), Chiang Mai University, Thailand, which financed my research.

I am also grateful to my co-authors, especially Prof. Olaf Zimmermann, Prof. Cesare Pautasso and Prof. Mirko Stocker, for their work as co-authors, late-night feedback sessions, and moral support. The special thanks goes to Prof. Filipe Correia for shepherding my patterns' publications. Thanks should also go to the members of the Software Architecture Research Group, who impacted, inspired, and helped me get through the pandemic years.

Lastly, I would be remiss in not mentioning my family and friends, especially my mother. Their belief in me has kept my spirits and motivation high during this process. I would also like to thank the Thais who live in Vienna for all the entertainment and emotional support.

Part I

Introduction

Chapter 1

Introduction

As the software industry uses Application Programming Interfaces (APIs) more frequently in modern software development, the number of related studies increases to keep up with the trend. An API is an interface that allows developers to interact with an application through data and functionality. APIs come in many shapes and sizes. API design approaches exist with various API styles (REST, SOAP, GraphQL, gRPC, etc.) and numerous specification formats (OpenAPI, RAML, AsyncAPI). Using APIs as the communication between microservices is twofold for most enterprises, because they can be used to expose an application's data and functionality to third parties. Microservices typically communicate via message-based remote APIs in a loosely coupled fashion. This powerful integration is often for specific purposes, which leads Domain Driven Design (DDD) to the appropriate technique for microservice architecture construction. DDD involves the modeling and shows the big picture of the system. It also supports technical and non-technical parties to improve the model. Consequently, it could prevent conflicts between design documentation and actual implementation.

In 2019, Microservice API Patterns (MAP) [211, 212] have been introduced to the software architecture community. MAP has raised questions about an actual service design rather than the general supporting infrastructure architecture. DDD is an essential approach for building microservices that brings DDD-based API design to our attention. Regarding the literature related to DDD, there is a ten-year gap of shifting from the conceptual level book of Eric Evans in 2003 [33] to the practical level book of Vernon and Millett in 2013 [182]. These years overlap with the timestamp of the last ten years from MAP. As shown in Appendix B, we searched the Google Scholar with three keywords ("Application Programming Interface", "Domain Driven Design", "Microservices") and performed the association among them from Google Scholar from 2003 until 2021. The data covered the year 2003-2021. There are numerous API studies related to quality attributes, but none of them have studied the interrelation between API and DDD yet.

This dissertation seeks to comprehend and support the DDD-based API design with human involvement. Firstly, we intend to study the phenomena of APIs and DDD and their coexistence. This study allows us to examine Architectural Design Decisions (ADDs) based

on the observed occurrences. Secondly, we aim to investigate further the phenomenon of coupling bad smells. The anemic domain model is one of the most commonly mentioned anti-patterns in the field of DDD. This anti-pattern describes a domain model in which the implementation fails to connect data and logic processing. Thirdly, after obtaining the empirical evidence from the preceding aims, we intend to empirically mine patterns pertaining to the relationship between Microservice APIs and DDD. We also relate APIs and DDD practices to the code, especially to relevant code bad smells. Lastly, we aim to measure the conformance to the patterns, practices, and ADDs. ADDs could be used to achieve one or more of a system's qualities. The conformity of ADDs and model elements should be validated in terms of supporting the study of API/DDD combinations.

The contributions of this dissertation are concepts and tools. The grounded theory results became our conceptual/theoretical contributions, including the six ADDs, six coupling smells, and two sets of patterns. With six ADDs, we provide the precise model of the ADDs along with decision options, decision drivers (force), and relations in the field of DDD-based microservice API design. In six coupling smells, we defined their definitions, smell impacts, relationships to other smells, relationships to practices and patterns, and fix options. Those two sets of patterns are 1) patterns on deriving APIs and API endpoints from domain model elements, and 2) patterns on designing API endpoint operations. Besides, providing the automated conformance assessment approach to architectural design decisions, our tool contributions are 1) the detector based on model elements and metrics calculation, 2) the parser and assessor based on API description. Finally, we provide the dataset and source code as long-term open access in Zenodo.

The rest of this chapter is organized as follows: Section 1.1 describes the important terms in more detail. Section 1.2 summarizes the problems that are addressed by the research presented in this dissertation. Section 1.3 lists the research questions which are based on the problems described in Section 1.2. Section 1.4 describes the research methods that were applied in order to answer the research questions. The correlation of these research methods and their applications are explained in 1.5. Section 1.6 briefly draws the overview of each chapter of the dissertation. Section 1.7 provide a list of the journal and conference publications on which this dissertation is based on.

1.1 Terminology of Context

The terms that are frequently used in this dissertation consist of *Architectural Design Decision*, *API Endpoint*, *API Description*, *Bad Smells*, *Conformance Assessment*, *Domain Driven Design*, and *Microservice API*. Each term below contains its definition and its relation to the understanding and semi-automatically supporting DDD-based API Design.

Architectural Design Decision

> "A description of the choice and considered alternatives that (partially) realize one or more requirements. Alternatives consist of a set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements [180]."

Architectural Design Decisions (ADDs) focus on the architecturally significant design decisions of a system, in the sense that a single decision change could significantly affect its entire architecture [82]. Each ADD in this dissertation contains decision options (the choice and considered alternatives), relations (rationale), decision drivers (the design rules, design constraints and additional requirements) .

API Endpoint

> "An *API endpoint* is the provider-side end of a communication channel and a specification of where the *API* endpoints are located so that *APIs* can be accessed by *API clients* (also called *API consumers*). Each endpoint thus must have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), in HTTP-based SOAP, or in RESTful HTTP. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints [212, 90, 165]."

Changes in the API endpoints can significantly affect the architecture of the API clients and of the microservices in the backend serving the API [212]. In the field of microservice APIs and their relations to Domain Models, many central API design problems revolve around API endpoint design.

API Description

> "API Description that defines request and response message structures, error reporting, and other relevant parts of the technical knowledge to be shared between provider and client. In addition to static and structural information, also cover dynamic or behavioral aspects including invocation sequences, pre- and postconditions, and invariants. Complement the syntactical interface description with quality management policies as well as semantic specifications and organizational information. [212]"

API Description [90] has been published as part of the Microservice API Patterns [212] before, and API CONTRACT is just a variant of the INTERFACE DESCRIPTION pattern in the Remoting Patterns [184]. API Descriptions, such as OpenAPI[1], Swagger[2], or RAML[3], are widely used today.

[1] `https://www.openapis.org/`
[2] `https://swagger.io/`
[3] `https://raml.org/`

Bad Smell

> "Code smells are implementation structures that negatively affect system lifecycle properties, such as understandability, testability, extensibility, and reusability; that is, code smells ultimately result in maintainability problems [47]. "
>
> "Design smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality [169]. "

The term Bad Smell (or Smell for short) is used to denote both code and design smells. It is a symptom of poor implementation or design choices that often corresponds to a deeper problem in a software system [1]. One of the key *anti-patterns* discussed in the realm of DDD is the ANEMIC DOMAIN MODEL[4]. This anti-pattern describes a DOMAIN MODEL that fails to combine data with logic processing it in its realization.

Conformance Assessment

> "The conformity assessment is a process of proving that the substantive requirements of the technical regulations relating to the measuring instruments have been met. The following essential requirements that apply in one way or another to the software during measuring instruments conformity assessment are: suitability for use and protection from unauthorized interference [181]."

In general, the conformance relation is defined as the consistency between models [126]. To guarantee the correctness of this consistency relation, an assessment is needed. Conformance assessment is challenging because it concerns the relation between a software system's architecture and its intended architecture [28].

Domain Driven Design

> "Domain-Driven Design is an approach to software development for complex businesses and other domains. DDD tackles that complexity by focusing the team's attention on knowledge of the domain, picking apart the most tricky, intricate problems with models, and shaping the software around those models [35]."

DDD [33, 182] is a design approach that places the (business) domain at the center of software designing and architecting. Design in DDD leads to the rigorous modeling of a DOMAIN MODEL and using it to build a UBIQUITOUS LANGUAGE that enables software development teams to use domain terminology throughout the software systems we build. Evans [33] classifies domain objects into types such as ENTITIES, VALUE OBJECTS, and SERVICES, which are then used to identify larger structures such as AGGREGATES. At the next higher abstraction level, DDD introduces the notion of *Strategic Design* [33, 182] which explains how to structure large domains into a number of BOUNDED CONTEXTS and their relations – modeled in so-called CONTEXT MAPS.

[4] See e.g. `https://martinfowler.com/bliki/AnemicDomainModel.html`.

Microservices Application Programming Interface

> "A microservice is an independently deployable component of bounded scope that support interoperability through message-based communication. Microservice Architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices [103]."
>
> "APIs are used as a key interconnectivity mechanism to access software services over the Internet. Interconnected applications can provide better service at a lower cost to their users [161]. "

Microservices are independently deployable, scalable, and changeable services, each having a single responsibility [203]. The remote APIs as their communication channel can be realized using many technologies, including RESTful HTTP, queue-based messaging, SOAP/HTTP, or remote procedure call technologies such as gRPC. A critical aspect in designing a microservice architecture is API design which includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to represent API messages, and so on [211].

1.2 Problem Statements

This thesis includes four primary problem statements. The first two focus on the relationship between microservice APIs and DDD as it pertains to coupling smells, as well as API and domain model design in practice. One then concentrates on pattern mining and the other on ADD conformance assessment.

PS1: Lack of understanding about the interrelation of microservice APIs and DDD related to coupling smells

The purpose of studying this problem is investigating the phenomena of the combination between API and DDD; and analyzing ADDs associated with the interrelation of microservice APIs and DDD. Microservices themselves are often identified in DDD artifacts. Microservice API design is a critical aspect in crafting a microservice architecture. While quite a number of studies on API design exist, only a few focus on remote APIs. Nonetheless, a number of local API topics are likely transferable to a certain extent to the remote API context. APIs design is not only about a technical aspect but also about a development culture [59]. There are a number of empirical studied on API documentation [101, 153, 113] which emphasize roles such as API documenter and API designer in this context. The human aspects hold the major interest because the perception of stakeholders can reflect the decision and action they take. The scientific literature has studied various API quality attributes, such as accessibility [84], stability [99], compatibility [145, 85], evolvability [65, 190, 83], and usability [124, 198, 13]. Many of the related works focus on local programming APIs, and remote APIs are studied only in a few works yet. Meanwhile API design in general has been studied, the specific relation of API design to design practices and models commonly used

in microservice architectures is yet understudied. In particular, practitioners frequently use Domain-Driven Design (DDD) in their microservice architectures and API designs.

PS2: Lack of understanding on coupling smells

The scientific literature has developed a substantial number of high-quality research studies on code and design smells [143], including metrics to detect code smells [79], smell detection approaches and tools [40, 111, 38, 4, 88], smell fixing approaches and tools [178, 144, 174, 29], taxonomies [94, 98], and a considerable number of empirical studies with human participants [26, 95, 110, 55, 191]. Whereas practitioners participated in many of these empirical studies, this is rarely the case in approaches and tools articles. However, the existing empirical studies indicate that there are gaps between the scientific results and practice. For many current scientific approaches, the actual accuracy of smell detection and fixing in complex coupling situations relevant in practice is either low or unclear. We also observed that only relatively trivial approaches, such as basic definitions and metrics, seem to have been transferred to practitioner views, approaches, and tools yet.

To further illustrate the research problem, we provide a few observations of intriguing phenomena that are inadequately explained in the existing scientific literature. First, a number of empirical studies reveal interesting gaps between the scientific literature and practice. Exemplary phenomena that are not well explained by the current literature on code smell approaches and tools – identified in existing empirical studies on smells: Guo et al. [55] observed that domain-specific tailoring of code smell detection rules and heuristics can greatly improve the human understanding of smells. Furthermore, they found problems in encoding code smells into a tool using metrics from the scientific literature. Mantyla et al. [95] found that the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators, and that metrics and smell evaluations did not correlate. Palomba et al. [110] found that instances of a smell may or may be problematic based on the "intensity" of the problem, and that developer's experience and system's knowledge play an important role in the identification of some code smells. Yamashita and Moonen [191] investigated serious interaction effects between smells. Palomba et al. [110] concluded: "Indeed, there seems to be a gap between theory and practice, i.e., what is believed to be a problem (theory) and what is actually a problem (practice). "Second, for many current scientific approaches the actual accuracy of smell detection and fixing in complex coupling situations relevant in practice is either low or unclear. Third, we observed that only relatively simple approaches, such as basic definitions and metrics, seem to have been transferred to practitioner views, approaches, and tools yet. Fourth, smells which reside at the boundary between code and design quality, such as *coupling smells*, seem to be more prone to these issues than simplistic code smells such as *Long Method*

Despite our knowledge of the interrelation between APIs and DDD, there are insufficient practitioner-focused studies on coupling smells. We are uncertain how to interpret the interrelation of APIs/DDD and coupling smells.

PS3: Lack of study on patterns mining of the deriving domain models to APIs, especially to relevant code bad smells

APIs are often used as the externally visible interfaces of systems modeled with DDD. Thus, the question arises how to derive APIs from DDD models.

If an API is derived from such an ANEMIC DOMAIN MODEL often very basic elements of the DOMAIN MODEL, such as ENTITIES are exposed as e.g. RESTful services. This can lead to shallow API endpoints, where clients need to understand all the complexity in the backend. Transactional or data consistency boundaries between distributed services are missing, often leading to situations where the client needs to take part in ensuring data consistency in the backend services or where consistency management in the backend is challenging to realize well. Such designs lead to chatty APIs with bad performance and scalability. APIs are becoming hard to understand, maintain, or evolve.

A typical symptom of these problems visible at the API operation level. There are many simple and shallow ENTITIES or even VALUE OBJECTS, exposed as API endpoints, with CRUD (Create, Read, Update, Delete) operations only on them. This can lead to all negative consequences mentioned above. However, not every CRUD-based API endpoint is necessarily a bad design either: some such endpoints expose rich domain abstractions well. Or the domain logic is inherently simple, meaning that the simple endpoint is the best possible design.

As it seems natural for a REST resource refers to one or more nouns found in the DOMAIN MODEL, it is possible that, after these initial API designs are then fully specified and implemented, there is the danger that they could result in an ANEMIC DOMAIN MODEL. This happens if the search for finding rich[5] domain abstractions is replaced by simply using abstractions in the DOMAIN MODEL that are easy to realize as REST resources. If a substantial refactoring to API endpoints backed up by rich DOMAIN MODEL abstractions never happens, a shallow API on top of an ANEMIC DOMAIN MODEL is the consequence, with all kinds of negative consequences, such as the ones described above.

To empirically mine patterns related to the interrelation of Microservice APIs and DDD is one of the proposes of this study. In general, the patterns authors rely on their interpretation for establishing the patterns. The data that was gathered from various practitioners rarely be used as the patterns references. The complex topic–deriving domain models to APIs, especially relevant to code bad smells, required rigorous analysis. Perhaps the majority of researchers have examined the data from a particular aspect, but they do not consider the standpoint of practitioners, which could produce a different result. Therefore, the study on recurring patterns in the ADDs and relations to coupling smells is challenging.

[5]Evans [33] uses the term "rich DOMAIN MODEL" to express a DOMAIN MODEL which contains a rich understanding of the processes and rules of a domain. The opposite is an ANEMIC DOMAIN MODEL which represents rather a shallow understanding of the domain. Our patterns, in part, aim to helpt in reflecting such rich domain abstractions in the API.

PS4: The limitation of ADDs' conformance assessment associate to the combination between APIs and DDD

Some studies have already been conducted in closed issues measuring the conformance to the patterns, practices, and ADDs. Researchers just pointed out several points that we do not know, and they remain unclear, for example, the automatic approach of conformance assessment to ADDs, the scoring scheme of judging the practices and so on. These issues have not been addressed yet.

Architectural Design Decisions (ADDs) concentrate on a system's architecturally significant design decisions, in the sense that a single decision change can have a significant impact on its entire architecturee [82]. Many central API design problems in the field of microservice APIs and their relationships to Domain Models revolve around API endpoint design. An *API endpoint* is the provider-side end of a communication channel as well as a specification of where API endpoints are located so that API clients can access APIs [212, 211]. Changes in the API endpoints can significantly affect the architecture of the API clients and of the microservices in the backend serving the API [212].

Many existing practitioner works use DDD to identify and model the microservices which are exposed via APIs [157, 212, 36, 100]. If this is the case, APIs should have a traceable mapping to the Domain Models of the microservices. In each evolution step of either the API or the microservice models, the respective other parts might have to be changed accordingly. To safeguard such evolution steps in practice, it is required to assess whether a given microservice API endpoint design correctly realizes the mapping to its microservice domain model and vice versa. This is especially problematic if such an assessment is needed continuously, e.g. in the context of continuous integration/delivery (CI/CD). Just consider today's large-scale microservice deployments that are deployed with high frequency (e.g. at least daily), such as those of Uber [148], Google [52], or Netflix [105]. To manually assess for each and every service, whether the mapping to the Domain Models and other such constraints imposed by backend or client architectures are still in place and correct, would be an extremely laborious and error-prone manual task.

Based on the empirical study on the interrelation of microservice APIs and DDD related to coupling smells, the automated assessment of conformance to ADD options might decrease time-consuming for researchers interested in this topic. Assessing architecture conformance remains challenging for preventing architecture erosion. Even though the modeling allows us to access the different types of artifacts, the automatic assessment is a challenging task for real-world systems.

1.3 Research Questions

The main research question of this thesis can be broadly stated as "How to better understand and support the DDD-based API design?". More specifically, the following four research questions are elicited from the problem statements in Section 1.2 to support the main research question. Figure 1.1 shows the relationships between four sets of research questions and four problem statements. RQ1, RQ2, RQ3, and RQ4 correspond to PS1, PS2, PS3, and PS4. PS3 is primarily addressed because of RQ3. Furthermore, RQ1 and RQ2 contributions resolve PS3.

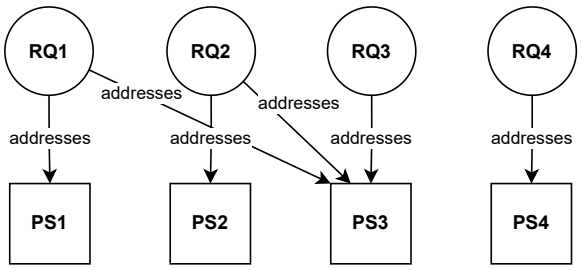

Figure 1.1: The Relations between Research Questions and Problem Statements

To address PS1, we decided to investigate the current practitioner's understanding of the relation of Microservice APIs and DDD (RQ1). Our work is the first to study the relation of DDD and API design in terms of ADD. The design guidance by an ADD model also can help to reduce design efforts and risks. Therefore, we focus on Architectural Design Decisions (ADDs), their decision options, their relations, and decision drivers.

Research Question 1 (RQ1)

How do practitioners understand DDD-based microservice API design?

- **RQ1.1** What are the possible ADDs and corresponding decision options in DDD-based Microservice API design?
- **RQ1.2** What are forces (decision drivers) relevant in those design decisions?
- **RQ1.3** Which relations do the decisions and decision options have?

PS1, PS2 and PS3 have led to the study of RQ2, which aims to investigate the current perception of practitioners on coupling smells. We expand our understanding on the anemic domain model, too. Our prior studies show that coupling issues in APIs and DDD models do not only remain at the model level. They transcend all the way down into the backend services' detailed design and code. We thus explore the current practitioner's understanding of coupling smells, too.

Research Question 2 (RQ2)

How do practitioners understand coupling smells?

- **RQ2.1** How are coupling smells understood by practitioners?
  - **RQ2.1.1** What are the defining factors of coupling smells as perceived by practitioners?
  - **RQ2.1.2** What are the impacts on and trade-offs to be made with regard to code and design quality when considering coupling smells as perceived by practitioners?
  - **RQ2.1.3** What are the relations and interactions among coupling smells, to other related smells, and to other software code and design concepts as perceived by practitioners?
  - **RQ2.1.4** What are the options for fixing coupling smells as perceived by practitioners?
- **RQ2.2** What are gaps in the understanding of coupling smells between the practitioner's view and the scientific literature?
- **RQ2.3** What are opportunities and challenges for future scientific work to address the practitioner concerns on coupling smells well?

After gathering the empirical evidence, we tend to capture the recurring ADDs. While software practitioners realize APIs based on DDD models, clear guidance on how to derive APIs from DDD models is still missing. In RQ3, based on prior in-depth studies of practitioner sources on this and related topics, we have mined patterns to address the design problems in PS3.

Research Question 3 (RQ3)

What are recurring patterns in the ADDs and relations to coupling smells?

- **RQ3.1** How to formally describe the API?
- **RQ3.2** How to derive API resources such as endpoints from Domain Model elements?
- **RQ3.3** How to derive APIs from a Domain Model and its elements?
- **RQ3.4** How to design the operations of an API endpoint in relation to the operations modeled on Domain Model elements?
- **RQ3.5** How to design the operations of an API endpoint in relation to domain events in the domain model?

To address PS4, we aim to automate the assessment of conformance to ADD options in API endpoint designs. Some might argue that APIs are intended to change rarely, and thus such automation is not required. But actually there are many more reasons for API evolution [91], meaning that frequent system changes usually lead to frequent API changes. Also, our approach studies the links of APIs to DDD models. So, if the API stays stable but the DDD model changes, conformance must still be checked. That is why we conduct the study to answer RQ4.

Research Question 4 (RQ4)

How to provide support for assessing how well a microservice API is conforming to a Domain Model it was derived from?

- **RQ4.1** For which decision options in microservice API design decisions can we provide automated support for assessing conformance to the Microservice API's Domain Model?
- **RQ4.2** To which extent can we assess conformance of API design decisions to the Microservice API's Domain Model?
- **RQ4.3** How can we automatically assess conformance to ADD options used in ADDs on deriving API endpoint operations, links, and resource segregation from Domain Models?
- **RQ4.4** How well do such automated measures for assessing ADD conformance perform?
- **RQ4.5** How to provide fully automated conformance assessment of API ADDs?
- **RQ4.6 a)** To which extent can OpenAPI-based parsing of information on APIs provide sufficient information for design option identification in API ADDs? **b)** Which level of accuracy can be expected in fully automated conformance assessment of API ADDs?

1.4 Research Methods

This section introduces and summarizes the research methodologies employed in this dissertation.

Mixed Method

We have applied the definition of the mixed method of Johnson et al. [70] "Mixed methods research is the type of research in which a researcher or team of researchers combines elements of qualitative and quantitative research approaches (e.g., use of qualitative and quantitative viewpoints, data collection, analysis, inference techniques) for the broad pur-

poses of breadth and depth of understanding and corroboration." The mixed methods are emerged because of the limitation of quantitative or qualitative methods. Many research questions cannot be addressed using a singular method. Meanwhile, quantitative methods could tell us "If", the qualitative methods deliver the answer from the question "How" and "Why" [175].

There are a variety of design choices that the mixed method can provide us. These choices involve a range of sequential and concurrent strategies. Creswell et al. [25] classified the types of strategies as follows: sequential explanatory strategy, sequential exploratory strategy, sequential transformative strategy, concurrent triangulation strategy, concurrent embedded strategy, and concurrent transformative strategy. Concerning the integration, we could consider mixing these two traditional methodologies at any point in the study. The ability to deal with diversity and divergence is a power of mixed methods research [147]. This dissertation has adopted mixed methods in every single point of our research methodology.

Case Studies

Yin defines a case study as "an empirical inquiry that investigates a contemporary phenomenon in depth and within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident" [192]. A clear research question on the certain phenomena is a precondition for conducting a case study. Commonly, the case study is used as a research strategy in psychology, sociology, political science, social work, business, and community planning [142]. Nowadays, the case study is used in broader areas, including software engineering. The case study is an answer to understand better how and why certain phenomena occur in the high-level objective. The selection between single case studies and multi-case studies depends on the purpose and context. The important point is that the research be able to describe the theory context and its relationship to the readers [56]. Even though the multi-case studies are time-consuming compare to the single case studies, the multi-case studies are beneficial in terms of reliability and theory convincing with strong evidence. They could clarify the finding with rich information. The case studies can be divided into EXPLORATORY CASE STUDIES and CONFIRMATORY CASE STUDIES [31] . Meanwhile, EXPLORATORY CASE STUDIES are used as the initial investigation, CONFIRMATORY CASE STUDIES are used as the tested artifacts.

This thesis applied multi-case studies approach to evaluate the phenomena associated with the understanding and semi-automatically supporting DDD-based API design.

Prototypical Development

Prototyping development fits several tactics for design studies, including simple starting [149], speed code writing and ready to throw away [149], designing in interview and workshop [22], and deploying the technology probe in the early version [67]. The prototype was utilized to consider how practitioner requirements would evolve through experience with implementing and designing probes. That is why it has become a popular alterna-

tive development methodology. Prototypes are different from mock-up because it is "an information system development methodology based on building and using a model of a system for designing, implementing, testing and installing the system." [78] The form of prototypes is frequently applied at research and development institutes and universities. It serves to clarify problems or elaborate ideas with the purpose of acquiring knowledge [16]. The prototypes could reduce the production time because: applying the intended models in the early stage leads to eliminating the time-consuming revision later, and designing tasks are complete throughout the project [71].

In our work, we have a clear distinction between a model and a prototype. Based on the prototypes, we can illustrate, test, and evaluate our implementation concepts for the conformance assessment of ADDs.

Grounded Theory (GT)

Glaser and Strauss [51] introduced GT as a research method for the discovery of theory from systematically obtained and analyzed data. The method has two unique properties: constant comparison and theoretical sampling [51]. *Constant comparison* means that the method uses an iterative and incremental process of data collection and analysis. *Theoretical sampling* means that researchers should actively find new data sources driven by the results of the data analysis. The researchers roughly follow the GT data analysis and coding steps by Corbin and Strauss [24]. They use three main coding steps: *Open Coding*, *Axial Coding*, *Selective Coding*.

GT ends when *theoretical saturation* is reached. This occurs when no new concepts emerge from new data sources anymore and the theory has been thoroughly validated with the collected data [69].

Grey Literature Study

Grey literature in software engineering can be defined as "any material about software engineering that is not formally peer-reviewed nor formally published [50]." According to Rainer and Williams [132], there are many benefits in using grey literature sources in software engineering research, including that "they provide information on practitioners' contemporary perspectives on important topics relevant to practice and to research" and "promote the voice of practitioners [132]." As our research questions require practitioner views, we decided to focus on grey literature sources stemming from practitioners and mixed groups (practitioners and others). According to Garousi et al. [49], such grey literature sources include blog posts, presentations, Wiki articles, technical reports, audio-video material, practitioner book content, and many other kinds of sources. As we intended to use grey literature sources in a GT study, the actual search process was guided by GT's theoretical sampling concept and stopped when theoretical saturation emerged. We have opted against a Multivocal Literature Review (MLR), where scientific sources are included, as our research goals focus on exploring practitioner views specifically.

Precise Modeling

We used our existing CodeableModels tool[6], a Python implementation for precisely specifying meta-models, models, and model instances in code with an intuitive and lightweight interface. Based on CodeableModels, we specified a meta-model and models for our study, extending both as needed. In addition, we realized automated constraint checkers and PlantUML[7] code generators to generate graphical visualizations of all meta-models and models, as well as a full textual model output generation in Markdown and Latex. Moreover, we also provided the example of the reproduction guideline how to use the CodeableModel in the context of the interrelation of API and DDD as illustrated in Appendix A. The meta-model of our found theory defines the classification (or categories) of model elements, as well as their relations.

1.5 Methodology Overview

Regarding the main research question "How to understand and support the DDD-based API design?" and its sub research questions in Section 1.3, the mixed methods methodology has been conducted. The empirical study has been conducted to perceive the nature of scientific knowledge, whereas, the design science research has been shown in terms of prototyping development. An empirical study in software engineering aims to improve software practice through evidence-driven research that evaluates or develops tools, methods, processes, and other aspects of practice [152]. The Mixed-Methods approach is selected based on guidelines for selecting empirical methods for software engineering research [31].

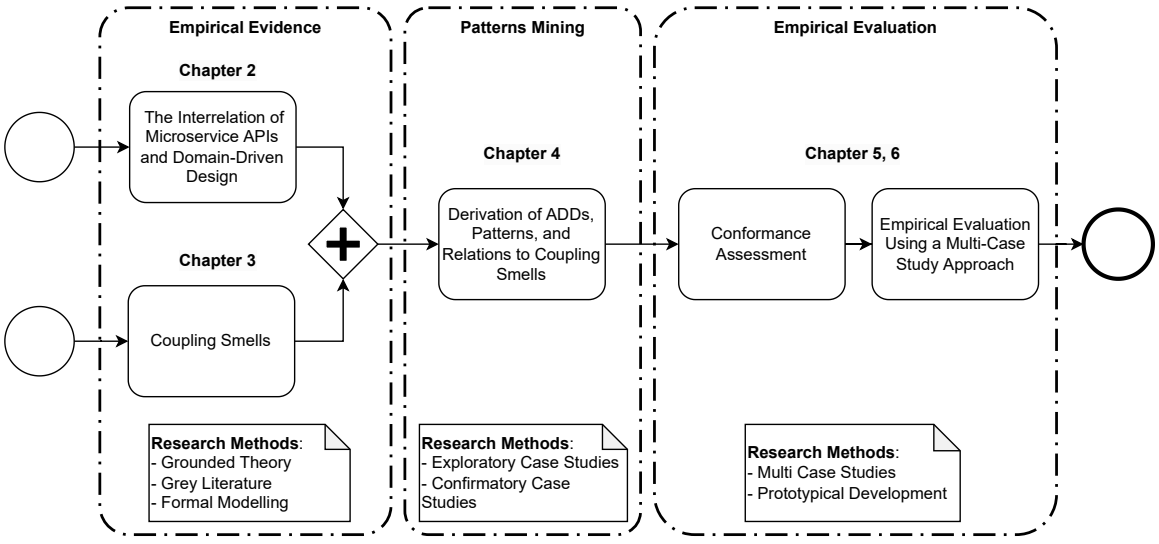

Figure 1.2: Empirical Research Overview

Figure 1.2 illustrates the three major tasks of research study including EMPIRICAL EVIDENCE, PATTERNS MINING and EMPIRICAL EVALUATION. Firstly, EMPIRICAL EVIDENCE,

[6] `https://github.com/uzdun/CodeableModels`
[7] `https://plantuml.com/`

the relation between DDD and coupling smells (Chapter 3) emerged during the investigation of the combination of APIs and DDD phenomena (Chapter 2), which allow us to analyze ADDs related to the interrelation of microservice APIs and DDD. The research Methods used in this task are GROUNDED THEORY, GREY LITERATURE, and PRECISE MODELING Secondly, PATTERNS MINING, concerning both contexts in the APIs/DDD association and the coupling smells, is interesting to observe best practices in these concepts. Therefore, we mainly adopted CASE STUDIES (EXPLORATORY CASE STUDIES and CONFIRMATORY CASE STUDIES) in this task as shown in Chapter 4. Lastly, EMPIRICAL EVALUATION, is about measuring the conformance to the patterns, practices, and ADDs. Moreover, the objectives to relate APIs and DDD practices to the code, especially relevant code bad smells, and suggest improvements in API and Domain Model design are part of the empirical evaluation task. PROTOTYPICAL DEVELOPMENT and MULTI-CASE STUDIES are the primary research methods in this task. It covers Chapter 5 and Chapter 6. Meanwhile, Chapter 5 evaluates the ADDs based on the constructed models, Chapter 6 aims to measure the ADDs based on API descriptions.

Empirical Evidence on the Current State of the Practice

We performed a GT study, in which we used UML-BASED MODELING to precisely encode our findings. GREY LITERATURE is used as the data collection technique for GT. Figure 1.3 shows how to determine empirical evidence using GT. For the artifact, we line-by-line analyzed each source to extract the necessary information, and then created one field note per source. This is a technique known as MEMO WRITING in which we recorded conceptual details, hidden interpretations, and other interesting details about the sources. Through each coding step, we established traceability from source lines to the GT study-derived precise model of our theory. After initial text-based open coding, we used precised UML-based modelling for axial and selective coding, instead of the often-used text-based coding process, in order to develop a precisely defined and consistent theory. The Chapter 2 and the Chapter 3 elaborate this research task in more detail.

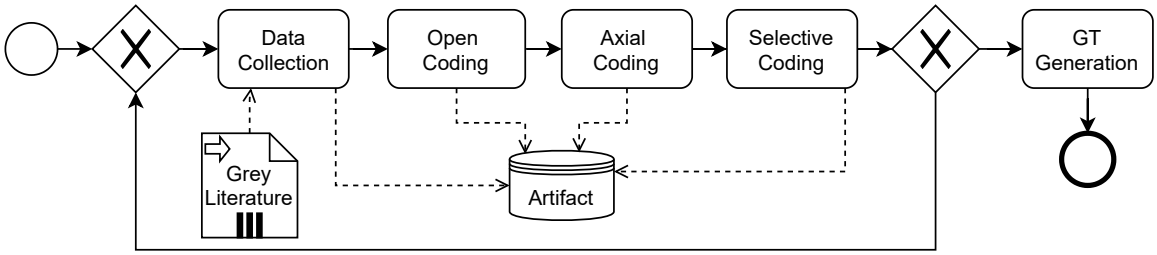

Figure 1.3: Empirical Evidence

Pattern Mining

Figure 1.4 depicts how we extracted patterns from GT study data and existing patterns in the Chapter 4. This research task incrementally improves the previous task (Empirical

Evidence) by conducting pattern analysis and pattern validation on the collected data. In addition, to those detailed studies, we have modeled the DDD to API mappings of 14 system descriptions and open source systems by practitioners as UML models. These system models are also employed to validate our patterns and to describe known uses of the patterns. The target audience consists of software/API developers and architects who are interested in the relationships between DDD and APIs, as well as software engineering researchers who are studying these concepts.

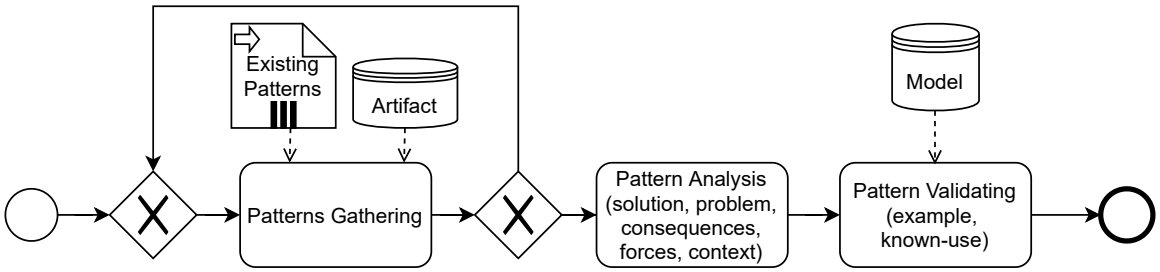

Figure 1.4: Patterns Mining

Modeling-based Conformance Assessment

For the modeling-based conformance assessment, we derived a scoring scheme from ADDs to assess the case studies and evaluate how well the cases match our detector results. After that, we derived "Assessment Detectors" from the ADDs, patterns, and relations to coupling smells. In parallel, we inspected each case study system for conformance to ADDs, patterns, and relations to coupling smells. Finally, we evaluated our methodology empirically through a multiple-case study utilizing the previously prepared and inspected cases. In order to summarize Chapter 5, Figure1.5 demonstrated seven steps used in the empirical evaluation task of the modeling-based conformance assessment.

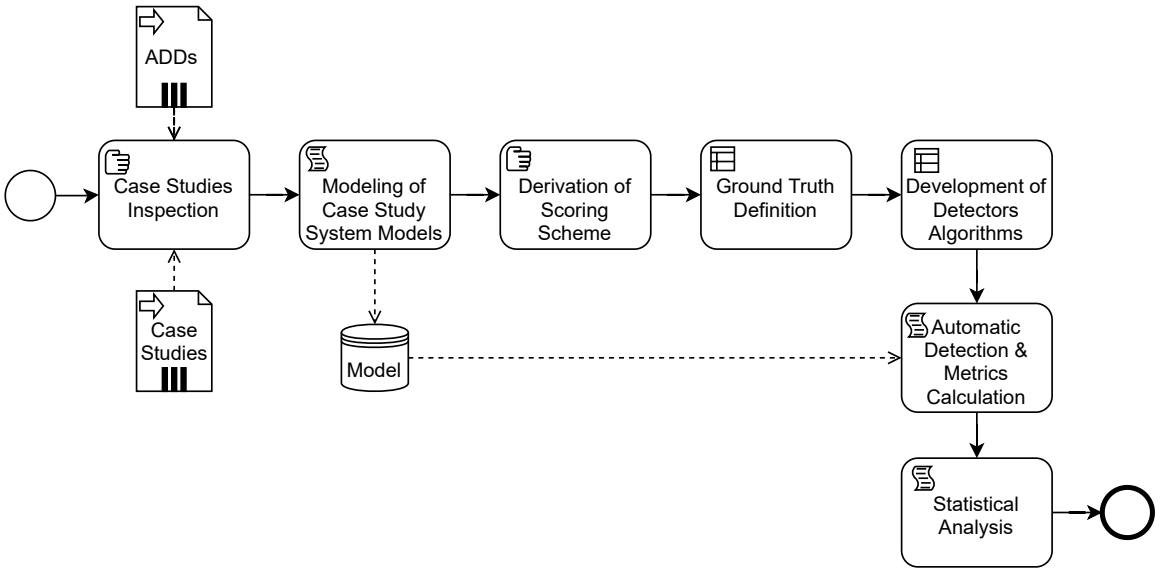

Figure 1.5: Empirical Evaluation: Modeling-based Conformance Assessment

API Description-based Conformance Assessment

In contrast to the modeling-based assessment, the API description-based assessment only needs the OpenAPI code. Therefore, we started with *parsing API description* and followed by *identifying decision options*. Based on the detected information, the approach then *assesses the conformance using the scoring scheme*. Then, *generating assessment results*, we displayed the results to the API developer as shown in 1.6. In addition, Chapter 6 describes the implementation of this research task in detail.

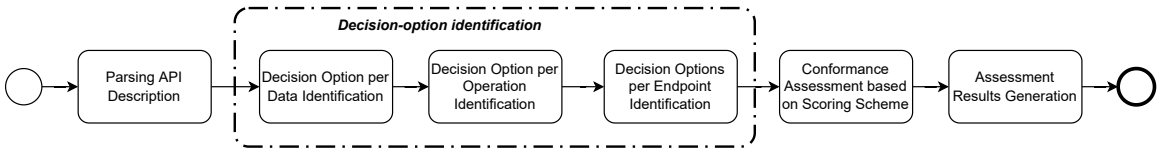

Figure 1.6: Empirical Evaluation: API Description-Based Conformance Assessment

Please note that these steps are to be conducted entirely based on OpenAPI-based API descriptions, i.e. requiring no human effort for modeling creation.

1.6 Thesis Overview

This thesis consists of six chapters. Table 1.1 illustrates the overview of each chapter including objectives, research questions (in Section 1.3), publications (in Section 1.7), and research tasks (in Section 1.5).

Chapter 1 is an introduction which contains context, problem statements, research questions, research methodologies, publications, and an overview of the entire thesis. Firstly, the context covers motivation and thesis subject that the reader should be acknowledged. Then, the problem statements are defined after the realization of what is lacking in the study, what is lacking in the understanding, and what are the limitations for conducting the research related to DDD-based API design. After that, the research questions are identified based on the problem statements. These research problems and research questions frame the research methodology, which is mixed methods. Lastly, the publications related to this thesis are listed.

Chapter 2 offers practitioners' views on the interrelation of microservice API and DDD. This chapter is designed to complement the **Empirical Evidence** research task. We conducted a **GT** study and employed **UML-based modeling** to encode our findings precisely. GT utilizes **Grey Literature** as its data gathering method. We only selected practitioner articles from practitioners targeted at other practitioners. For this, we evaluated whether the sources included themes such as designing APIs for systems based on DDD, how to structure API and DDD-related parts of the system, and whether DDD concepts help to structure and design an API, as well as how to do so. This chapter resulted in the **ICSA'21** publication, which addresses the **RQ1** – how practitioners understand DDD-based microservice API Design.

Table 1.1: Overview of each chapter and its objective, associated research question, publication, and methodology.

Chapter	Objective	Research Question	Publication	Research Stage
Chapter 1	**Introduction:** - identify problem statement, research questions, and research methodology			
Chapter 2	**Practitioner Views on the Interelation of Microservice APIs and DDD:** - develop ADDs, decision options, decision drivers	RQ1: How do practitioners understand DDD-based microservice API design?	ICSA'21	Empirical Evidence
Chapter 3	**On the Practitioners' Understading of Coupling Smells:** - study coupling smells (definitions, trade offs, relationship, fixing option)	RQ2: How do practitioners understand coupling smells?	IST'21	Empirical Evidence
Chapter 4	**Derivation of ADDs, Patterns and Relations to Coupling Smells:** - derive the patterns - mine the patterns	RQ3: What are recurring patterns in the ADDs and relations to coupling smells?	EuroPLoP'21, PLoP'21	Pattern Mining
Chapter 5	**Conformance Assessment of ADDs and Empirical Evaluation:** - provide the automated detectors for conformance assessment - generate the models of case studies	RQ4: How can we automatically assess conformance to ADDs?	JSS'22, ICSA'22	Empirical Evaluation
Chapter 6	**API Description-Based Conformance Assessment of Architectural Design Decision:** - provide the automated parser for retrieving the information from API description - provide the automated assessor for conformance assessment	RQ4: How can we automatically assess conformance to ADDs?	SOSE'22	Empirical Evaluation
Chapter 7	**Conclusion:** - summarize the conducted research - give directions for future work			

Chapter 3 names on the practioners' understanding of coupling smells. This chapter studies the coupling smells in the context of definitions, trade offs, relationship, and fixing options. The purpose of the **IST'21** publication is to address the research question **RQ2** – how practitioners understand coupling smells. This study adopts the technique as same as Chapter 2 which are **GT**, **Grey Literature**, and **Precise Modeling or UML-based Modeling**. This chapter is a part of **Empirical Evidence**. To avoid personal bias in the research, our initial search keywords are taken from "A Taxonomy for Bad Code Smells" [8]. Thus we have initially searched for the following keywords: "Code Smells", ("Couplers" or "Coupling") and "Smells", "Feature Envy", "Inappropriate Intimacy", "Message Chains", and "Middle Man". Our results are defining factors of coupling smells, as well as smell impacts, trade-offs, relationships to other smells, relationships to practices and patterns, and fix options as perceived by practitioners.

Chapter 4 describes derivation of ADDs, their patterns and their connections to coupling smells. This chapter covers the **Pattern Mining** research task which answer the **RQ3** – What are the recurring patterns in the ADDs and relations to coupling smells. **EuroPLoP'21** and **PLoP'21** are associated with this chapter. For endpoint-level patterns,

[8]See `http://mikamantyla.eu/BadCodeSmellsTaxonomy.html` which is part of a scholarly article [96]

we present the domain model facade as api pattern which describes how to derive an API from a Domain Model. To explain further how derive API endpoints constituting the API from Domain Model elements, we present the aggregate roots as api endpoints, domain services as api endpoints, and domain processes as api endpoints patterns. In addition, we relate these patterns to the previously released api description and api contract patterns, which describe how to describe APIs explicitly. For the operation endpoint-level patterns, we present three complementary patterns to derive API operations from the operations of Domain Services and Entities as well as Domain Events, namely Aggregated Domain Operation on API Endpoint, Event-Based API Endpoint Operation, and CRUD-Based API Operation.

Chapter 5 identifies the conformance assessment of ADDs and empirical evaluation. To facilitate in the continuous analysis of mappings of domain model elements to APIs and API endpoints, we intend to automate the assessment of conformance to ADD options. This chapter aims to answer **RQ4**– how can we automatically assess conformance to ADDs. The **Empirical Evaluation** research task performs in terms of modeling 14 cases in multi-case studies as demonstrated in the publications **JSS'22** and **ICSA'22**.

Chapter 6 identifies the API description-based conformance assessment of architectural design decisions. This chapter seeks to accomplish the same objective as that of Chapter 5. It responds to **RQ4** and is included in the **Empirical Evaluation** task. In addition, this chapter's publication is **SOSE'22**.

The Chapter 7 is a summary of the entire thesis and its future work orientation. This chapter discusses revisited research questions, contributions based on research questions, and recommendations for future research.

1.7 Publications

This thesis is based on the content of several scientific journal and conference publications which are either published or currently under a published process. More precisely, the following articles and papers provide the basis for this thesis:

- **ICSA'21:**
  Singjai, Apitchaka, Uwe Zdun, and Olaf Zimmermann. "Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory." *2021 IEEE 18th International Conference on Software Architecture (ICSA)*. IEEE, 2021.

- **IST'21:**
  Singjai, Apitchaka, Georg Simhandl, and Uwe Zdun. "On the practitioners' understanding of coupling smells—A grey literature based Grounded-Theory study." *Information and Software Technology* 134 (2021): 106539.

- **EUROPLOP'21:**
  Singjai, Apitchaka, Uwe Zdun, Olaf Zimmermann, and Cesare Pautasso. "Patterns on Deriving APIs and their Endpoints from Domain Models." *Proceedings of the 26th European Conference on Pattern Languages of Programs*. 2021.

- **PLOP'21:**
  Singjai, Apitchaka, Uwe Zdun, Olaf Zimmermann, Mirko Stocker, and Cesare Pautasso. "Patterns on Designing API Endpoint Operations." *Proceedings of the 28th Conference on Pattern Languages of Programs*. 2021.

- **ICSA'22:**
  Singjai, Apitchaka, and Uwe Zdun. "Conformance Assessment of Architectural Design Decisions on the Mapping of Domain Model Elements to APIs and API Endpoints." *2022 IEEE 18th International Conference on Software Architecture (ICSA)*. IEEE, 2022.

- **JSS'22:**
  Singjai, Apitchaka, and Uwe Zdun. "Conformance Assessment of Architectural Design Decisions on API Endpoint Designs Derived from Domain Models." *Journal of Systems and Software*.

- **SOSE'22:**
  Singjai, Apitchaka, and Uwe Zdun. "API Description-Based Conformance Assessment of Architectural Design Decision." *2022 IEEE 16th International Conference on Service-Oriented System Engineering (SOSE)*. IEEE, 2022.

Part II

Empirical Evidence

Chapter 2

Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory

This chapter presents the literature review on the DDD-based API design. This literature review investigates gray literature as data sources of grounded theory. Consequently, the practitioners' perception on the interrelation of Microservice API and DDD is reviewed. It is the backbone of this dissertation and concerns the **RQ1**. The content of this chapter is based on a peer-reviewed article published in **ICSA'21**.

Microservice API design is a critical aspect in crafting a microservice architecture. While API design in general has been studied, the specific relation of API design to design practices and models commonly used in microservice architectures is yet understudied. In particular, practitioners frequently use Domain-Driven Design (DDD) in their microservice architecture and API designs. We thus decided to study existing Architectural Design Decisions (ADDs), their solutions options, their relations, and the decision drivers in these decisions. Using the Grounded Theory research method we studied grey literature sources. In this study, we identified six ADDs with 27 decision options, numerous relations between them, and 27 decision drivers. The decisions cover mapping domain models to APIs, defining API contracts in relation to domain models, designing API resources based on domain model elements, segregation of API resources, mapping domain model links to the API, and designing the operations of an API resource.

2.1 Introduction

Microservices are independently deployable, scalable, and changeable services, each having a single responsibility [203]. They typically communicate via message-based remote APIs in a loosely coupled fashion. Those remote APIs can be realized using many technologies, including RESTful HTTP, queue-based messaging, SOAP/HTTP, or remote procedure call technologies such as gRPC. A critical aspect in designing a microservice architecture is API design which includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to represent API messages, and so on [211].

Microservices themselves are often identified in Domain-Driven Design (DDD) [33] artifacts. DDD is a design approach where the (business) domain is carefully modeled in software and evolved over time. Microservices and DDD have a synergistic relation. DDD concepts help to design microservices and establish their boundaries. Once established, microservice boundaries provide technical boundaries for separately modeled and evolved parts of the domain model, the *Bounded Contexts* [36]. This helps in enforcing strategic DDD design decisions e.g. modeled using a *Context Map* [36, 182].

We thus decided to investigate the current practitioner's understanding of the *relation of Microservice APIs and DDD*. In this chapter, we describe a Grounded Theory based qualitative study [51, 24] for this purpose. We decided to explore literature sources representing practitioner views on this topic. According to Rainer and Williams [132] there are many benefits in using grey literature sources in software engineering research, as they promote the voice of practitioners and provide information on practitioners' contemporary perspectives on important topics relevant to practice and research. For coding processes in the Grounded Theory study, we applied text-based coding only initially and then applied UML-based modeling instead to develop a precise and consistent theory (i.e. the UML figures shown in this chapter are results generated from our models).

In this chapter, we focus on Architectural Design Decisions (ADDs), their decision options, their relations, and decision drivers. We set out to answer the following research questions:

- **RQ1.1** What are the possible ADDs and corresponding decision options in DDD-based Microservice API design?
- **RQ1.2** What are forces (decision drivers) relevant in those design decisions?
- **RQ1.3** Which relations do the decisions and decision options have?

Our result is a formal model of the ADDs [205] with decision options, forces, and relations in the field of DDD-based Microservice API design. We believe that our results can help scientists to get a better understanding of practitioner concerns in this field. Our work can also help practitioners to get an overview of the current view of other practitioners.

This chapter is structured as follows: First, we discuss the related work in Section 2.2. Next, we explain our research method in Section 2.3. In Section 2.4 we present the detailed results of our grounded theory study, i.e. ADDs, decision options, their relations, forces, and trade-offs in each solution. Section 2.5 discusses the implications of the results for the research questions and threats to validity. Finally, in Section 2.6 we draw conclusions.

2.2 Related Works

While quite a number of studies on API design exist, only a few focus on remote APIs. Nonetheless, a number of local API topics are likely transferable to a certain extent to the remote API context. APIs design is not only about technical aspects but also about the development culture [59]. There are a number of empirical study on API documentation [101, 153, 113] which emphasize roles such as API documenter and API designer in this context. The scientific literature has studied various API quality attributes, such as accessibility [84], stability [99], compatibility [145, 85], evolvability [65, 190, 83], and usability [124, 198, 13]. Many of the related works focus on local programming APIs, and remote APIs are studied only in a few works yet. The following related works relied on the definition as follows: API designers (people who design the APIs), API document writers (people who write the API documentations), API developers (people who develop APIs), API users(people who use the APIs).

Robillard [138] raised the question what makes APIs hard to learn from the developer's perspective. In a survey with 80 experienced developers he shows that the most popular learning strategy is reading documentation. He developed a survey with 13 questions (3 open-ended questions on learning obstacle, 7 questions on specific API learning, and 3 open-ended questions on learning strategies). His survey included 80 experienced developers who working at Microsoft's Redmond, Wash., campus. The survey result has shown that the most popular learning strategy is reading documentation. The majority of participants also mentioned resources category as the main obstacle among other API learning obstacles category. In conclusion, the API users barrier is their special scenario, similarly, API documentation guidelines are the problem for API designers and API document writers. This work has implication for the fact that API document writers and API developers in Microsoft are separated, resulted in high level design understanding from API users is required. In this case, domain driven design concept might help to design a better API. This chapter aims to convince the DDD possibility in API development. Murphy et al. [102] studied the usability aspect from API designers' perspective. They interviewed 24 professional designers from the industry with 36 structured questions related to the core processes. They would like to understand the API designers' point of view during the process like design, implement, release and maintain. As a result, they implied the information to support the well-designed APIs. The data sources of this work are from interviews, meanwhile, our data sources are from various grey literature reviews where the practitioners share their knowledge online. They found, among other things, that many

designers learned API design on the job, that it is hard for them to discern which potential use cases of the API users will value most, and that they lack tools to gather aggregated feedback from the deployed API. Surwase [168] compared REST API modeling Languages with A developer's perspective. He provided general information what is a REST API, what is not a REST API, and the comparison between three different modeling languages. He recommends API developers to create an API specification with the tool. This work could help the API documentation writers to select the modeling language for their REST API. Our work also focuses on API designers and API developers perspective. Our work tends to provide the abstract level on design decision rather than the detail implementation or the tool preference. Goski et al. [53] studied on security warning for cryptographic APIs. They would like to get the feedback from software developers who are the API users. They conducted four focus groups with 25 participants. In conclusion, it turned out that only two design goal of six end-user security warning guidelines by Bauer et al. [9] can be applied directly. It also means acquiring the perception from end-user who is not developer failed in the context of security warning. This work really emphasized how importance developers perspective are. Meanwhile, their work mainly focuses on security warning, our work fits in general APIs.

In comparison to those other works, our work is the first to study the relation of DDD and API design. Our work structures the resulting design space systematically, in terms of design problems via ADDs, their decision options, their relations, and the API decision drivers (forces) linked to them. Finally, many of the related works focus on local programming APIs. As mentioned, remote APIs are studied only in a few works yet, and it is not clear if and which properties of local APIs can be transfered to remote APIs.

Design patterns are an essential concept offering decision options in our ADDs, and they offer a systematic way to organize API design knowledge. The Microservice API Patterns [211, 195][1] collect a number of API patterns in the realm of remote APIs. Context Mapper and its MDSL generator implement parts of the design advice and options that our literature review in this chapter reports [74], especially those options related to the Microservice API Patterns and API contracts. Richardson [136] describes some patterns with relevance to API design such as API Gateway or Command Query Responsibility Segregation (CQRS). Gosrki and Wojtach [54] propose the Use Case API pattern, an architectural pattern for use case based interfaces exposed e.g. via RESTful services. All three works apply DDD and explain relations to DDD patterns. However, none of these works has yet empirically investigated the relation of microservice API design and DDD. Being aware of design patterns in APIs development are infrequent but existed.

Gosrki and Wojtach [54] purpose use case API approach for the Data sharing and opening API which is a read-only API. They demonstrate their approach with the specific use cases. In conclusion, they can keep the domain knowledge in the server-side. They claim that DDD is their design approach and they also adopt domain adapter concept. In the future they plan to evaluate architecture of the API server-side. This work contribute the

[1] See also: `https://microservice-api-patterns.org/`

approach and evaluate by the usability aspect, but our work has different contribution. They concentrated on their approach evaluation. We concentrate on design decision elicitation from the APIs developer's perspective.

Ellist et al., [32] compare a factory pattern and a constructor in the context of usability evaluation. They conduct the experiment with 12 participants. These participants have to deal with five JAVA programming tasks which are notepad email task, eclipse email task, thinkies task, PIUtils task and Sockets task. For the participants, the factory pattern is harder than a constructor, regardless of context. This work has shown how design pattern fit in APIs development. Our work also take design patterns into consideration and we also see some opportunity to apply design patterns in the design decisions.

Iyer and Wyner [68] developed design science research to evaluate APIs. They applied stakeholder analysis in the scope identification step, and kernel theory in the objective identification step. They also explored some of GOF design pattern (Singleton pattern) and Active Record design pattern of Martin Fowler. Same as above, this work also investigate the use of gof design pattern. The main characteristic of gof design pattern is it is interpreted from the requirements. Our work did not concentrate on requirements but architecture point of view.

However , the work of Tulanch [179] concluded that the design patterns for object-oriented application are not necessary for API development. Bloch [10] also mentioned misuse of a wrapper in APIs design can develop the inefficient APIs. Beside the design patterns, the maturity levels of Richardson are used to define the API paradigm [168].

2.3 Research Method

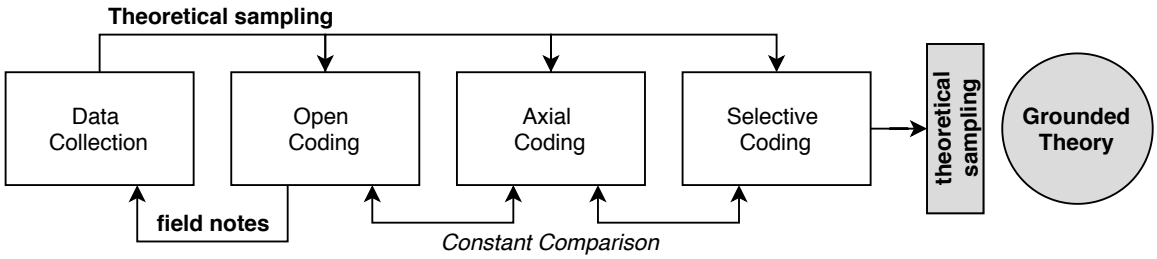

Figure 2.1: Research Method Overview

In this chapter we conducted a grounded theory study based on the grey literature [50]. We used formal modeling to precisely encode our findings (similar to [195]). *Grounded Theory (GT)* [51, 24] is a systematic research method for discovery of theory from data. GT has two unique properties: *constant comparison* and *theoretical sampling* [51] and three main data analysis and coding steps: *open coding*, *axial coding*, and *selective coding* [24]. Moreover, We also applied *memo writing* technique [76] to acquire conceptual detail on the data. GT [69] ends when *theoretical saturation* is reached. The *grey literature* [50, 132] is the main data source in our work. In software engineering, grey literature can be defined as "any material about software engineering that is not formally peer-reviewed nor formally

Table 2.1: List of Knowledge Sources Included in the chapter

ID	Title	Tiny URL	Source Type	Example	Source Code
s1	*Bounded Context in APIs (1/2)*	tinyurl.com/api-ddd-s1	Practitioner Audience Article	No	No
s2	*DDD & REST Domain Driven Apis for the Web*	tinyurl.com/api-ddd-s2	Slides	Yes	Yes
s3	*Why the domain model should not be used as resources in REST API?*	tinyurl.com/api-ddd-s3	Discussion Forum Post	Yes	Yes
s4	*How to clearly define boundaries of a bounded context*	tinyurl.com/api-ddd-s4	Discussion Forum Post	No	Yes
s5	*REST API Design - Resource Modeling*	tinyurl.com/api-ddd-s5	Practitioner Audience Article	Yes	No
s6	*Rest API and DDD*	tinyurl.com/api-ddd-s6	Discussion Forum Post	Yes	Yes
s7	*Introduction to DDD Lite: When microservices in Go are not enough*	tinyurl.com/api-ddd-s7	Practitioner Audience Article	Yes	Yes
s8	*REST and DDD: incompatible?*	tinyurl.com/api-ddd-s8	Practitioner Audience Article	Yes	No
s9	*Domain Driven Design - External Data API as Respository or Service*	tinyurl.com/api-ddd-s9	Discussion Forum Post	Yes	No
s10	*Conceptual mismatch between DDD Application Services and REST API*	tinyurl.com/api-ddd-s10	Discussion Forum Post	Yes	Yes
s11	*Microservices: Overview, Misinterpretations and Misuses*	tinyurl.com/api-ddd-s11	Practitioner Audience Article	No	No
s12	*Design a DDD-oriented microservice*	tinyurl.com/api-ddd-s12	Practitioner Audience Article	Yes	No
s13	*Apply Domain-Driven Design to microservices architecture*	tinyurl.com/api-ddd-s13	Practitioner Audience Article	No	No
s14	*Designing APIs and Microservices Using Domain-Driven Design*	tinyurl.com/api-ddd-s14	Slides	Yes	No
s15	*REST Service and CQRS*	tinyurl.com/api-ddd-s15	Discussion Forum Post	Yes	Yes
s16	*Exposing CQRS Through a RESTful API*	tinyurl.com/api-ddd-s16	Practitioner Audience Article	Yes	Yes
s17	*REST-first design is Imperative, DDD is Declarative [Comparison] - DDD w/ TypeScript*	tinyurl.com/api-ddd-s17	Practitioner Audience Article	Yes	Yes
s18	*Designing APIs for microservices*	tinyurl.com/api-ddd-s18	Practitioner Audience Article	Yes	Yes
s19	*Moving Towards Domain Driven Design in Go*	tinyurl.com/api-ddd-s19	Practitioner Audience Article	Yes	Yes
s20	*Microservices, Apache Kafka, and Domain-Driven Design*	tinyurl.com/api-ddd-s20	Practitioner Audience Article	Yes	No
s21	*API & Domain Driven Design*	tinyurl.com/api-ddd-s21	Slides	Yes	Yes
s22	*Implementing Domain-Driven Design for Microservice Architecture*	tinyurl.com/api-ddd-s22	Practitioner Audience Article	Yes	No
s23	*Pattern: Decompose by subdomain Context*	tinyurl.com/api-ddd-s23	Practitioner Audience Article	Yes	No
s24	*Building Microservices with Event Sourcing/CQRS in Go using gRPC, NATS Streaming and CockroachDB*	tinyurl.com/api-ddd-s24	Practitioner Audience Article	Yes	Yes
s25	*Aggregate Oriented Microservices*	tinyurl.com/api-ddd-s25	Practitioner Audience Article	Yes	Yes
s26	*Designing a Serverless Application with Domain Driven Design*	tinyurl.com/api-ddd-s26	Slides	Yes	No
s27	*Bounded Contexts With Axon*	tinyurl.com/api-ddd-s27	Practitioner Audience Article	Yes	No
s28	*Building Real-Time Web Applications using wolkenkit*	tinyurl.com/api-ddd-s28	Practitioner Audience Article	Yes	Yes
s29	*Uncovering API Implementation*	tinyurl.com/api-ddd-s29	Practitioner Audience Article	Yes	No
s30	*Implementing an API-First Design Methodology*	tinyurl.com/api-ddd-s30	Practitioner Audience Article	No	No
s31	*API First Development*	tinyurl.com/api-ddd-s31	Practitioner Audience Article	Yes	Yes
s32	*The API Design Process*	tinyurl.com/api-ddd-s32	Practitioner Audience Article	No	No

published" [50]. We decided to study grey literature sources representing acknowledged practitioners' views on *the interrelation of distributed APIs and DDD*. According to Rainer and Williams [132] there are many benefits in using grey literature sources in software engineering research, as they promote the voice of practitioners and provide information on practitioners' contemporary perspectives on important topics relevant to practice and research. These sources are then used as unbiased descriptions of established practices in the further analysis.

We studied each knowledge source in depth, followed GT's coding process, as well as a constant comparison procedure to derive a model, as illustrated in Figure 2.1. In contrast to classic GT, the research begins with an initial research question, as in Charmaz's constructivist GT [19]. Whereas GT typically uses textual analysis, uses textual codes only initially and then transfers them into formal software models (hence it is model-based). The data sources studied in this chapter (see Table 2.1) include *Discussion Forum Post*, *Practitioner Audience Article*, and Practitioner *Slides*. We searched for API and DDD related keywords in search engine like Bing or Google to find the initial data sources. From many obtained sources, we selected relevant sources by skimming and reading. We only selected practitioner articles from practitioners targeted at other practitioners. For this, the authors judged whether the sources focused on topics such as designing APIs for systems based on DDD, how to structure API and DDD-related parts of the system, or which DDD concepts help to structure and design an API and how. We reviewed each source in the author team. We excluded sources that seemed to sell a product or a service. As GT is mainly concerned with phenomena that have specifically been observed to exist [202], it is only necessary to find enough source that are relevant with regard to the phenomena being study. It is not necessary to find all possibly relevant sources (as it would be for instance in a systematic literature study).

The knowledge source acquisition is applied in many iterations. We excluded sources that seemed to sell a product or a service. This is an essential property of GT called *theoretical sampling* [51], which means that results of each data analysis step are used for the following data collection. The researchers should actively find new data sources driven by the results of the data analysis. In GT, the coding and data selection cycle stops, when *theoretical saturation* [51] is reached. We stopped our analysis when five to seven additional knowledge sources did not add anything new to our understanding of the research topic. As a result of this very conservative operationalization of theoretical saturation, we studied a rather large number of sources in depth (32 in total, summarized in Table 2.1).

The authors analyzed every source line by line to elicit the required information, and then created one field note per source. This is a memo writing technique where we noted down the conceptual details, hidden interpretations, and other interesting details on the sources. We established traceability from lines in the sources, over each coding step, to the formal model of our theory derived during the GT study. Next, we followed Corbin's and Strauss' method for GT coding [24]: First, we applied *Open Coding* with the goal to transform conceptual details into conceptual labeling. Next, during *Axial Coding* we

identified categories in the concepts, e.g., by identifying concepts that reappear in the data, synonymous concepts, related concepts, and so on. The identification helps to find out the relations between the concepts. Finally, during *Selective Coding* we carved out main ideas of the theory, i.e., understanding the big picture by reflecting on the data and analysis results. As mentioned, after initial text-based open coding, we used formal UML-based modelling for axial and selective coding, instead of the often-used text-based coding process, in order to develop a precisely defined and consistent theory. We used the Python tool CodeableModels[2] for this. Formal modelling also eased establishing an audit trail of the research, and thus enable repeatability of the study. In addition, we provide public access to the original data as well as the derived models[3]

2.4 Architectural Design Decisions

In this section, we present the results of our study in form of ADDs that appear in the discussions of the practitioners in our grey literature sources. Figure 2.2 shows an overview of the decisions identified and their relations (explained in detail below). For space reasons, we cannot give detailed examples. Please note that Table 2.1[4] shows which sources provide examples (with or without source code).

Forces Codes/Sources (Table. 2.2)

f1:Brittle Interfaces [s1, s23], **f2**:Avoid Exposing Domain Model Details in API [s1, s2, s3, s4, s5, s7, s10, s17, s18, s29, s32], **f3**:Chatty API [s1, s5, s11, s18], **f4**:Minimize API calls [s1, s2], **f5**:Performance [s1, s11, s16, s18], **f6**:Scalability [s1, s4, s5, s8, s11, s16, s18, s20, s24, s28, s29], **f7**:API Complexity [s1, s5, s6, s11, s12, s15, s18, s25, s29], **f8**:API Usability [s1, s5, s14], **f9**:API Evolvability [s1, s5, s18, s31], **f10**:API Modifiability [s1, s3, s4, s5, s17, s18, s23, s30, s31], **f11**:Data Consistency [s1, s2, s5, s10, s12, s16, s18, s20, s21, s22, s25, s27, s28, s29], **f12**:Message Size [s1], **f13**:Coupling of Clients to Server [s2, s5, s6, s8, s10, s14, s18, s20, s22, s23, s27], **f14**:Protocol Complexity in Client [s2, s8, s10], **f15**:Design and Implementation Effort [s3, s21, s23, s29, s30], **f16**:Interface Design Limits Domain Model Design [s3, s10], **f17**:Clients Need to Manage Crossing Model Boundaries [s3, s4], **f18**:Maintainability of API and API Consumers [s5, s6, s7, s10, s17, s29, s30], **f19**:Reliability [s4, s5], **f20**:Eventual Consistency Support [s5, s15, s16, s24, s25, s27, s28], **f21**:Separation of API Contract and Domain Concerns [s1, s3, s5, s10, s13, s24, s29, s30, s32], **f22**:API Understandability [s6, s7, s15, s16, s17, s30, s31, s32], **f23**:Can Lead to Anemic Domain Model Anti-Pattern [s8, s17], **f24**:API Stability [s10, s23, s30, s31], **f25**:Domain Model Flexibility [s10], **f26**:Initial Effort Required [s17, s30, s31], **f27**:Support for External or Public Clients [s18]

Domain Model Mapping Decision

The core result of a DDD modeling effort is the *Domain Model*, which defines the *Ubiquitous Language*. This term is used by Evans to describe the language shared by the whole team, including developers, domain experts, and other participants [33]. For larger domains, it is usually not possible to model the whole *Ubiquitous Language* in a single unified model.

[2] `https://github.com/uzdun/CodeableModels`

[3] We provide all open and axial coding files, derived formal models in Python, and generated models (in UML, Markdown, and Latex) as a replication package for download on `https://doi.org/10.5281/zenodo.4569578`.

[4] In the table we use the following color coding to visualize the frequency of the evidences: $< 5\%$, $< 10\%$, $< 20\%$, $< 35\%$, $< 50\%$, $< 70\%$, $\geq 70\%$.

Table 2.2: Study Results: Overview of Design Decisions, Decision Options, Evidences and Related Forces

Design Decision	**#**	**Solution**	**Evidences**	**Forces**
How to Map a Domain Model and its Elements to an API?	16	1.Expose the Whole Domain Model in 1:1 Relation as API	s1, s3, s9	f1(--), f2(-), f17(+), f7(--), f8(-), f4(-), f10(-), f15(+)
		2.Expose Domain Model Subset as API	s3, s6, s9	f1(o), f2(o), f17(+), f7(o), f8(o), f4(-), f10(-), f15(+)
		3.Expose Each Bounded Context as an API	s1, s4, s13, s14, s20, s27, s29, s32	f1(-), f2(-), f17(-), f7(-), f8(-), f4(-), f10(-), f15(+)
		4.Expose Selected Bounded Contexts as APIs	s1, s3, s4, s13, s14, s20, s21, s22, s23, s27, s29, s32	f1(o), f2(o), f17(-), f7(o), f8(o), f4(-), f10(o), f15(+)
		5.Introduce and Expose Interface Bounded Context as an API	s1, s3, s23	f1(+), f2(+), f17(++), f7(+), f8(+), f4(+), f10(+), f15(-)
		6.Expose a Shared Kernel between Client and Server as an API	s1, s3	f1(+), f2(+), f17(++), f7(+), f8(+), f4(+), f10(+), f15(-)
Which Approach is Chosen for Defining the API Contract in Relation to the Domain Model?	15	1.Explicitly Specify the API Contract	s3, s8, s10, s13, s17, s18, s21, s30, s31, s32	f21(+), f24(+), f25(+), f26(o), f10(o), f18(o), f23(o), f27(++)
		2.Extract API Contract from Domain Model	s3, s10, s13, s17, s18	f21(+), f24(+), f25(+), f26(o), f10(o), f18(o), f23(o), f27(+)
		3.Domain Model Defines API Contract	s3, s10, s13, s18	f21(-), f24(--), f25(--), f26(o), f10(o), f18(o), f23(o), f27(o)
		4.Bounded Context Defines API Contract	s3, s13, s27	f21(-), f24(--), f25(--), f26(o), f10(o), f18(o), f23(o), f27(o)
		5.Write API Code First which Defines the Contract	s17, s30	f21(--), f24(--), f25(--), f26(++), f10(-), f18(-), f23(--), f27(--)
Which Domain Model Elements Should be Offered as Resources or Endpoints in an API?	21	1.Entities as API Resources	s1, s2, s5, s6, s12, s13, s14, s16, s17, s18, s19, s21, s32	f2(--), f11(--), f3(-), f5(-), f6(-), f7(-), f18(-), f19(-), f13(-)
		2.Domain Services as API Resources	s9, s13, s19	f2(o), f11(+), f3(o), f5(++), f6(++), f7(+), f18(+), f19(+), f13(+)
		3.Aggregate Roots as API Resources	s1, s2, s5, s6, s10, s12, s13, s17, s18, s19, s25, s26, s27, s28	f2(+), f11(++), f3(+), f5(+), f6(+), f7(+), f18(+), f19(+), f13(+)
		4.Bounded Contexts as API Resources	s1, s2, s5, s20, s25, s26, s29	f2(+), f11(o), f3(+), f5(+), f6(+), f7(-), f18(o), f19(o), f13(o)
		5.Domain or Business Processes as API Resources	s5, s10, s20	f2(+), f11(++), f3(+), f5(+), f6(+), f7(+), f18(+), f19(+), f13(+)
Segregate Resources for Reading and Updating Information in an API?	11	1.Expose Segregated Command and Query Resources in API	s5, s6, s8, s11, s12, s15, s16, s24, s25, s27, s28	f7(-), f11(-), f6(+), f20(+)
		2.Do Not Segregate Queries and Commands in an API	s5, s6, s8, s11, s12, s15, s16, s24, s25, s27, s28	f7(+), f11(+), f6(-), f20(-)
How to Design the Operations of a Resource?	26	1.CRUD-Style Operations on Resources	s1, s2, s3, s5, s6, s7, s8, s10, s11, s12, s14, s15, s16, s17, s18, s19, s20, s21, s29	f2(-), f3(--), f5(--), f6(--), f16(-), f18(-), f11(-), f19(-), f13(-), f22(+)
		2.Domain Operations on Resources	s1, s2, s3, s4, s5, s6, s7, s9, s10, s11, s12, s15, s16, s18, s20, s25	f2(+), f3(+), f5(+), f6(+), f16(+), f18(+), f11(+), f19(+), f13(+), f22(+)
		3.Encode Operations as Commands in the Payload	s6, s15, s16	f2(+), f3(+), f5(+), f6(+), f16(+), f18(o), f11(+), f19(+), f13(+), f22(-)
		4.Expose Domain Events as State Transitions	s2, s3, s4, s5, s9, s11, s12, s14, s16, s20, s24, s25, s26, s27, s28, s29	f2(++), f3(+), f5(+), f6(++), f16(+), f18(+), f11(+), f19(+), f13(+), f22(-)
		5.Expose Domain Events via Feeds or Pub/Sub	s2, s20, s24, s27, s28	f2(++), f3(+), f5(+), f6(++), f16(+), f18(+), f11(+), f19(+), f13(+), f22(-)
How to Map Links between Domain Model Elements to the API?	8	1.None	s1, s2, s31	
		2.Use Distributed or Hypermedia Links in the Payload	s1, s2, s3, s8, s10, s18, s31	f11(+), f9(+), f10(+), f12(++), f2(++), f13(++), f14(-), f4(-), f5(-), f6(-)
		3.Pass Object Identifiers in the Payload	s1, s2, s18, s31	f11(+), f9(+), f10(+), f12(++), f2(-), f13(-), f14(+), f4(-), f5(-), f6(-)
		4.Embed Linked Data in the Payload	s1, s2	f11(-), f9(-), f10(-), f12(-), f2(o), f13(+), f14(+), f4(+), f5(+), f6(+)

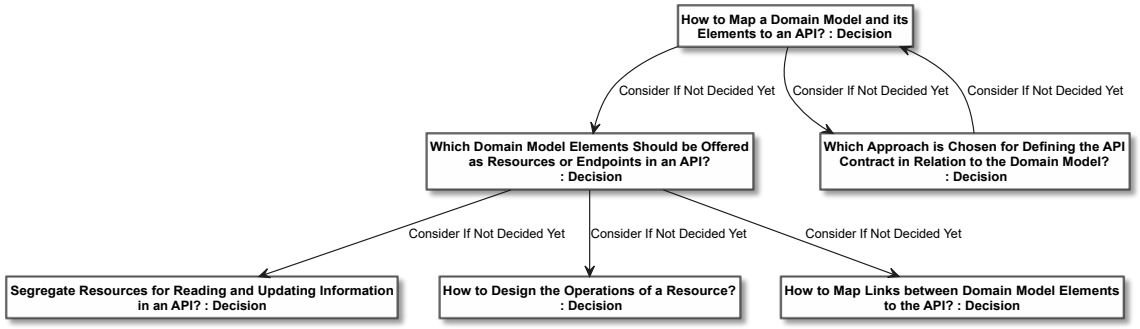

Figure 2.2: ADDs and ADD Relations: Overview

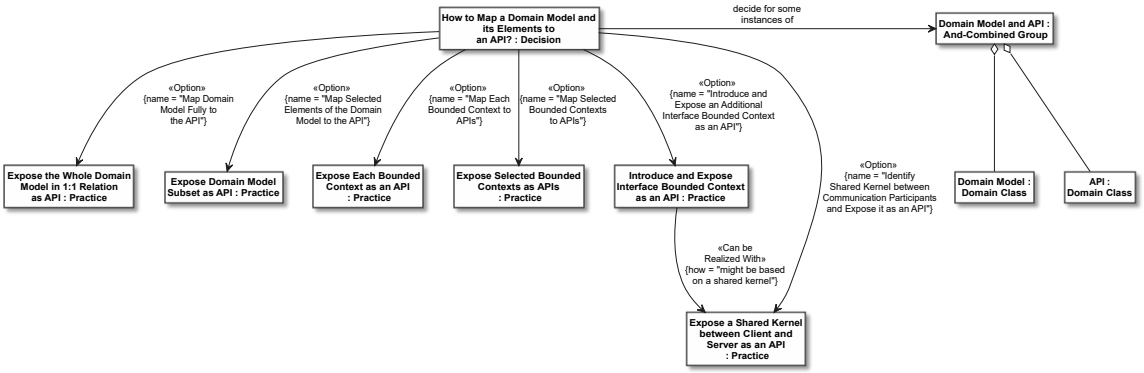

Figure 2.3: Domain Model Mapping Decision

Instead DDD divides larger domains into different *Bounded Contexts* and explicitly models their relationships, usually with the help of *Context Maps* [33, 182].

In the sources analyzed in our study, the practitioners frequently discuss the ADD *How to Map the Domain Model and its Elements to an API?* (overall 16 source discuss this ADD) as summarized in Table 2.2. We found evidence for seven possible design options as solutions for this ADD as illustrated in Figure 2.3. A simple solution is to *Expose the Whole Domain Model in 1:1 Relation as API* but this is seen as working only for small examples, as it leads to negative impacts on coupling and maintainability forces such as *Brittle Interfaces*, *API Complexity*, and *Avoiding Exposing Domain Model Details in API*. A usually better working solution is *Expose Domain Model Subset as API* which partly improves on the negative force impacts. Both solutions have benefits like little required *Design and Implementation Effort*. For complex domains, it can be advisable to consider the *Bounded Contexts* as well. Then there are the options to *Expose Each Bounded Context as an API* or *Expose Selected Bounded Contexts as APIs*, both leading to a sub-division of the API along the *Bounded Context* boundaries. In most large domains, the latter solution is seen as being better suited to avoid *Brittle Interfaces*, *Exposing of Domain Model Details in the API*, and *API Complexity*, and improve *API Usability* and *API Modifiability*. Both solutions offer positive impact on *Design and Implementation Effort*, but require more effort than the first two solutions. The downside of those solutions is that *Clients Need to Manage Crossing Model Boundaries*, i.e., the boundaries between the *Bounded Contexts*. One suggested solution to this problem is to *Introduce and Expose Interface Bounded Context as an API*. That is, a

new special *Bounded Context* that represent the API interface is exposed. This solution is neutral or positive on all so far mentioned forces, except the *Design and Implementation Effort* where it leads to additional effort compared to all other so far mentioned solutions.

In some cases, it might make sense to consider *Expose a Shared Kernel between Client and Server as an API*, which can be seen as an option how to realize the solution *Introduce and Expose Interface Bounded Context as an API*. *Shared Kernel* is a DDD relation between two *Bounded Contexts* in which some subset of the domain model is shared between the two teams developing the contexts. For example, it is often implemented as a local code library available to API client and server. Here, the client and server contexts would share such a *Shared Kernel*. This is a good solution with similar force impacts as *Introduce and Expose Interface Bounded Context as an API* in cases, where close interaction between the teams developing client and server is acceptable.

Please note that the two options explained before *Expose Each Bounded Context as an API* or *Expose Selected Bounded Contexts as APIs* also use *Bounded Context* relations to determine what is exposed as a remote API: They would use *Bounded Context* relations that are more frequently leading to remote interconnections between services in the implementation such as *Open Host Service*, *Customer/Supplier* or *Anti-Corruption Layer*. Implementation-wise such solutions lead to use of the *Service Layer* pattern [42] for the services exposing the API, with each service being the *Remote Facade* [42] for the elements shielded by the API.

As can be seen in Figure 2.2, this decision has a number of direct and indirect follow-on decisions that should be considered as well. It is also suggested to first consider the API contract related ADD in Section 2.4 and then consider the domain model mapping ADD. Many of the DDD elements in both decisions are sometimes using *Event Storming*; while not directly related to API design, such techniques have an indirect influence on API design which is interesting to explore in future work.

API as Contract Decision

The *API Contract* is an API-related concept in which the API is defined in some formal language, e.g., based on Open API or RAML for RESTful APIs, WSDL for SOAP APIs, some special Domain-specific Language for *API Contract* definition, and so on. *API Contracts* can support the decoupling between API consumers and API providers. The contracts are essential in terms of mutual understanding between those parties. Ideally, the *API Contract* is more stable than the backend systems, i.e., it is only changed rarely, whereas the backend systems often constantly evolve.

The relation between DDD and *API Contract* leads to the ADD *Which Approach is Chosen for Defining the API Contract in Relation to the Domain Model?* This ADD is mentioned in 15 sources (see Table 2.2). We have identified 5 decision options in the sources. The first option *Explicitly Specify the API Contract* describes the practice to design the *API Contract* before or relatively independent from the domain model. A specific

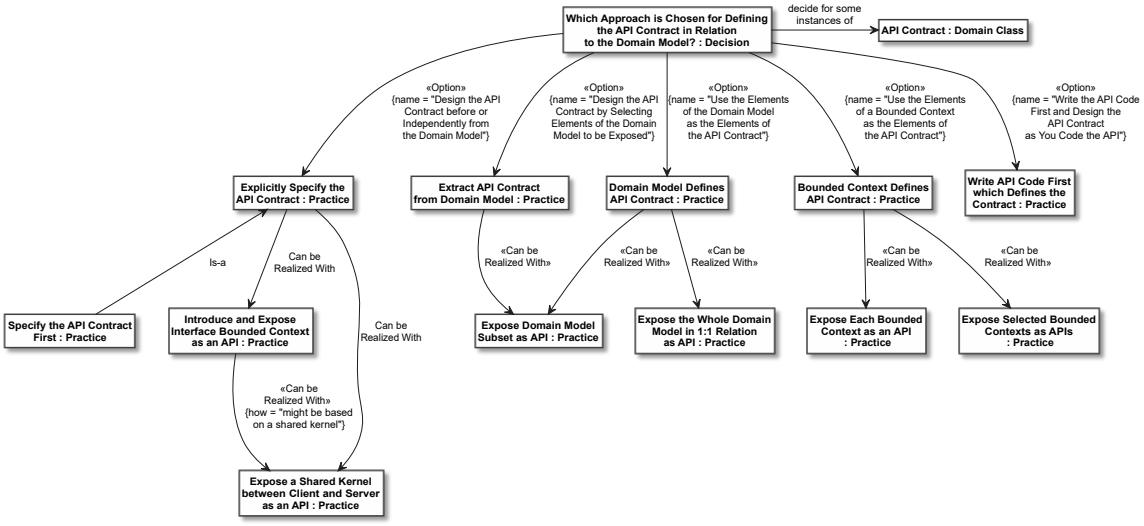

Figure 2.4: API as Contract Decision

variant of this, sometimes mentioned, is *Specify the API Contract First*; usually it meant as continuous improvement of a contract specification, starting from a specification, not a rigid, pre-defined contract. *Explicitly Specify the API Contract* can for instance be achieved using the *Expose a Shared Kernel between Client and Server as an API* or *Introduce and Expose Interface Bounded Context as an API* options from the previous decision. Another option is *Extract API Contract from Domain Model*, i.e., the practice of selecting elements of the domain model to be exposed in the contract. Both are practices that help in reaching *Separation of API Contract and Domain Concerns*, are positive for *API Stability*, and enable *Domain Model Flexibility*.

Not all possible solutions focus on *Separation of API Contract and Domain Concerns*: *Domain Model Defines API Contract* and *Bounded Context Defines API Contract* use the elements of the domain model or a *Bounded Context* as the elements of the *API Contract*. This leads to a worse *Separation of API Contract and Domain Concerns* as the API elements are tightly coupled to the faster evolving domain concept, which in turn is negative for either *API Stability* or *Domain Model Flexibility*. *Domain Model Defines API Contract* can be realized with the *Expose the whole Domain Model in 1:1 Relation as API* or *Expose Domain Model Subset as API* options of the previous decision. *Bounded Context Defines API Contract* can be realized with the *Expose Each Bounded Context as an API* or *Expose Selected Bounded Contexts as API* options of the previous decision.

The option *Write API Code First Which Defines the Contract* describes an API first approach which is seen, in contrast to the prior options, rather negatively: It can lead to bad *Separation of API Contract and Domain Concerns* as the domain model is based on premature, often low-level API design decisions, which *Can Lead to Anemic Domain Model Anti-pattern*, i.e., a domain model which contains little deep domain knowledge. This results in bad *API Stability* or *Domain Model Flexibility*, as well as bad *API Modifiability* and other issues in *Maintainability of API and API Consumers*.

Finally, there is the force *Support for External or Public Clients* to be considered. It is

important to distinguish between two types of APIs: public APIs that client applications call, and backend APIs that are used for communication between services. For external clients or public clients the maintainability, stability, and modifiability forces are usually of much higher importance than for internal APIs where a certain level of control of changes is possible. Thus for *Support for External or Public clients*, the option *Write API Code First which Defines the Contract* performs worst. *Domain Model Defines API Contract* and *Bounded Context Defines API Contract* can be less positive here than *Explicitly Specify the API Contract* and *Extract API Contract from Domain Model*, as they lead to a less strict separation of API and domain model, which might impede stability. The API as Contract decision has a strong connection to the domain model mapping decision and vice versa.

Designing API Resources Decision

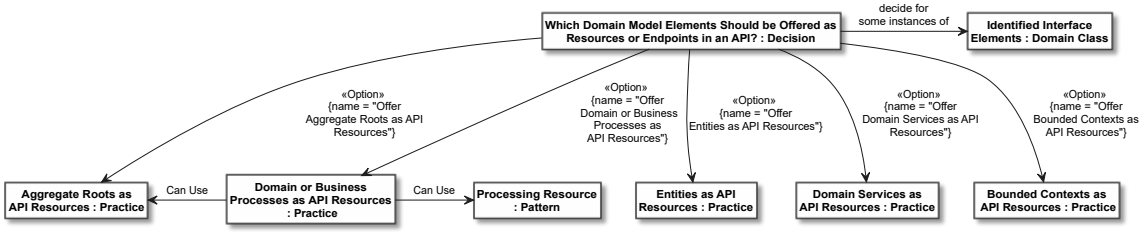

Figure 2.5: Designing API Resources Decision

We use the term *API Resource* for a set of related interface elements exposed in an API. Sometimes the term *API Endpoint* is used instead in a similar sense, even though endpoint usually denotes a remote location of a resource such as a URI. Please note that we use *API Resource* for any API technology, not limited to the RESTful resources. On the other hand, the decision outcomes of the two previous decisions determine the scope in which *Interface Elements* can be identified in the *Domain Model* and/or *API contract*. These need to be mapped to *API Resources*, which is the purpose of this ADD.

The decision on *Support for External or Public clients Which Domain Model Elements Should be Offered as Resources or Endpoints in an API?* (21 evidences as summarized in Table 2.2) consist of five options namely *Entities*, *Aggregate Roots*, *Domain Services*, *Bounded Contexts*, and *Domain or Business Process as API Resources* (typically realized as a *Processing Resource* [211]). That is, the basic abstractions *Entity*, *Aggregate*, and *Service* in tactical DDD [33] are discussed as possible sources for *API Resources*, as well as the two "broader" concepts *Bounded Contexts* and *Processes* (the later is applicable only in process-based system designs). Interestingly *Bounded Contexts* have already been used as API scopes in some options of the previous two ADDs, and are rather a concept of strategic design in DDD: *Strategic Design* shapes the big picture, while *Tactical Design* dives into design details [182]. This shows that this ADD is located at the border of these two central DDD design levels.

Entities as API Resources has the highest number of evidences (13 sources) as shown in Table 2.2. While being an obvious option, a number of practitioners in our sources

advise strongly against using *Entities* as foundations for *API Resource*. With 10 sources, *Aggregate Roots as API Resources* has the second highest number of evidences, and seems to be the most often recommended option how to start looking for *API Resources*. As an *Aggregate* abstracts the implementation details of a number of related *Entities* and other DDD model elements, it naturally serves as an *Identified Interface Element*. Practitioners agree that *Aggregate Roots* are a good starting point for *API Resources*, but many other options exist and often deliberate, incremental design is needed to find good *API Resources*. Some practitioners suggest *Domain Services as API Resources*. For instance, they can lead to stateless *API Resources* in addition to the usually stateful resources based on *Aggregates*. Both *Bounded Contexts* and *Processes* bundle a number of related DDD model elements and can thus, both according to a few practitioners, be candidates for defining *API Resources*. Some sources suggest to consider first *Aggregates*, and then the other options.

Main drawbacks of exposing *Entities* are issues related to *Avoiding Exposing Domain Model Details in API*, *Data Consistency*, and *Chatty APIs*. This can lead to bad *Performance*, *Scalability* issues, and high *API Complexity*. Other issues are possibly *Coupling of Clients and Server* and other issues in *Maintainability of API and API Consumers*. *Aggregate Roots as API Resources* and where applicable *Domain Services as API Resources* can help to avoid most of these issues. *Domain or Business Processes as API Resources* can have a similar positive impact on the forces, if a process-based abstraction makes sense in the domain context. Certain *Bounded Contexts* can work well for many of the forces, but there is a risk of higher *API Complexity* due the size of the *Bounded Contexts*. Also *Data Consistency* can be more natural to manage on an *Aggregate* than on a *Bounded Context* and low *Coupling of Clients and Server* can be harder to reach.

Resource Segregation Decision

After *API Resources* have been identified, e.g. using the options from the previous decision, there sometimes is the option to decide whether or not to *Segregate Resources for Reading and Updating Information in an API*. This is closely related to the *Command Query Responsibility Segregation (CQRS)* Pattern [136]. The idea of CQRS is to use a different model to update data than the model that is used to read data. A common implementation technique in the context of event-based microservices is *Event Sourcing* [136]. As querying the corresponding event store can be hard to realize efficiently, often CQRS is used in this context.

If CQRS is used e.g. for a microservice design in the backend, often it makes sense to offer the segregated commands and queries as two resources in an API which this ADD is about (see Figure 2.6). Practitioners agree that CQRS is a complex pattern and it should only be chosen if it offers a substantial benefit. It can be chosen in the backend and not exposed in the API, too. It is possible to use *Encoding Operations as Command in Payload* in the operations design decision in Section 2.4 as a practice to realize the segregation option. The segregation option would enable *Eventual Consistency* as a practice, not only

in the backend, but covering the clients.

Regarding forces, this option has the benefit of possibly improving *Scalability* and enabling *Eventual Consistency Support* where it is needed, e.g. in long running transactions. Downsides are higher *API Complexity* and less *Data Consistency*.

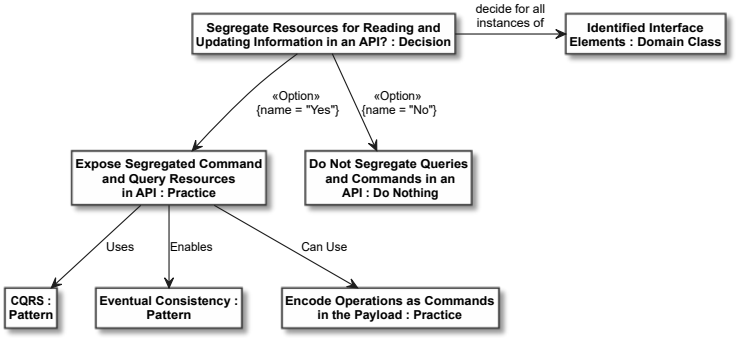

Figure 2.6: Resource Segregation Decision

Link Mapping Decision

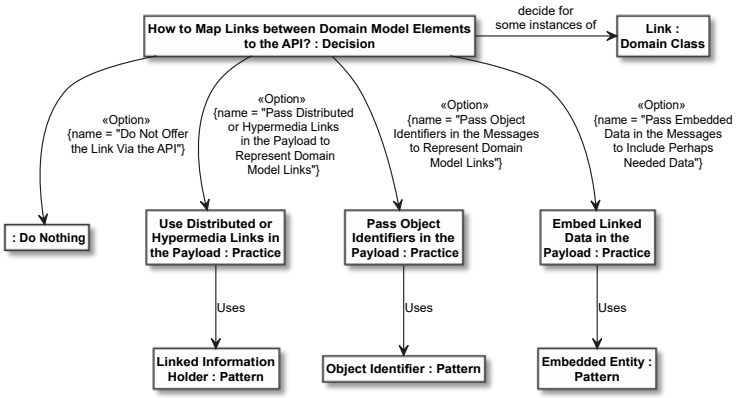

Figure 2.7: Link Mapping Decision

Another decision following the API resource decision is on link mapping. In APIs the links between *API Resources* play a central role. In DDD the links between model elements have a similar role. Obviously not all links in the DDD model are candidates to be exposed in an API, but only those between the model elements offered as *API Resources*, e.g. following the ADD from Section 2.4.

Figure 2.7 shows the link mapping decision. Many links are not mapped, as explained. The *Use Distributed or Hypermedia Links in the Payload* option means to use standard distributed/hypermedia links, such as URIs in RESTful HTTP or URIs and HAL/JSON-LDs in JSON. This conforms to using the *Linked Information Holder* pattern [211], which describes a linked API Resource. An alternative is *Embed Linked Data in the Payload* which follows the *Embedded Entity* pattern [211] where the content (or a part of it) is added to the message payload instead of linking to it. Finally, there is the option to *Pass Object Identifiers in the Payload*, i.e. to follow the *Object Identifier* pattern [184]. This means to

send an *Object Identifier* that is meaningful (only) in the server context, but contains no remote location information such as a distributed link.

Compared to the embedding option, use of distributed links is beneficial for *Data Consistency* as the link is always up-to-date, and thus *API Evolvability* and *API Modifiability* are positively influenced. It leads also to smaller *Message Sizes*. Links however lead to higher *Protocol Complexity*, make it harder to *Minimize API Calls*, and can have a worse *Performance* and *Scalability* because of many resulting distributed calls. The *Object Identifier* based option is very similar in its effects to the distributed links based option. In direct comparison, it has the disadvantages of possibly *Exposing Domain Model Details in API* as well as higher *Coupling of Clients to Server*.

Operation Design Decision

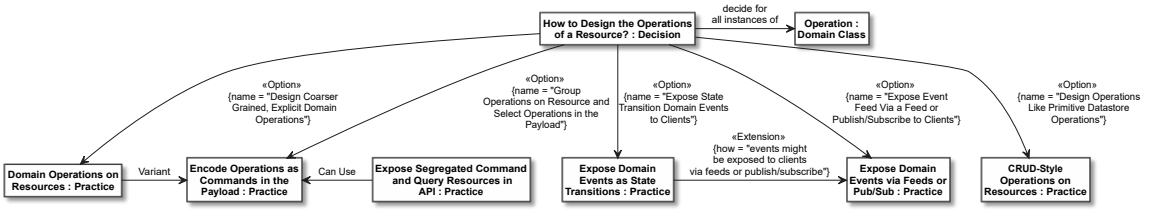

Figure 2.8: Operation Design Decision

The ADD *How to Design the Operations of a Resource?* shown in Figure 2.8 appeared in 26 investigated sources (see Table 2.2).

A simplistic option, especially in a RESTful context, are *CRUD-style Operations on Resources* which are discussed in 16 sources. *CRUD-style Operations on Resources* designs operations like primitive data store operations. This practice, while commonly used, is seen negatively for many forces. In particular, in contrast to the options below it is negative for the *Avoiding Exposing Domain Model Details in API* force, and can lead to *Chatty APIs* with bad *Performance* and *Scalability*. It is argued that it can lead to *Interface Design Limits Domain Model Design*, various *Maintainability* issues including *Coupling* issues, *Reliability* problems, and *Data Consistency* problems. On the positive side, *CRUD-style Operations on Resources* are simple and good for *API Understandability*.

The option to expose *Domain Operations on Resources* has the second highest in a number of evidences (15 sources). *Domain Operations on Resources* designs are focused on coarser-grained, explicit domain operations. This option is usually positive on the mentioned forces, but needs more design work when mapped to a RESTful API. A variant of it is *Encode Operations as Commands in the Payload* which can be harder to understand than domain operations exposed via other means such as protocol features (if supported by the protocol). It is often used e.g. when CQRS is exposed in the API, as then the operation commands can be encoded in a similar way as the queries.

Many microservice systems are event-based systems. In such systems, another option is to *Expose Domain Events as State Transitions* (or its variant *Expose Domain Events via Feeds or Pub/Sub*). These options are also positive for most of the mentioned forces. They

can even lead to a better solution for *Exposing Domain Model in API*, if events are used to model the domain, and/or improving *Scalability*. Events can be harder to understand and thus these options also can have some issues with regard to *API Understandability*.

2.5 Discussion

How Practitioners Understand DDD-Based Microservice API Design

In the previous section, we have presented detailed findings on each of our research questions. **RQ1.1** investigates what the possible architectural design decisions and corresponding decision options in DDD-based microservice API design are. We have found evidence for six principal ADDs with 27 decision options. It is possible to classify those ADDs based on their context and DDD design level. The *Domain Model Mapping Decision* in Section 2.4 and the *API as Contract Decision* in Section 2.4 clearly focus on DDD elements of Strategic Design, such as *Domain Model*, *Bounded Context*, and *Shared Kernel*. On the API side, they focus on coarse-grained API concepts such as the whole *API* or the *API Contract*. The next set of decisions, i.e., the *Designing API Resources Decision*, and the *Resource Segregation Decision*, focus at the transition from Strategic to Tactical Design considering atomic model elements such as *Entities* but also composite structures such as *Aggregates* or *Processes*, as well as *Bounded Contexts*. Finally, the *Link Mapping Decision* and the *Operation Design Decision* focus on detailed mapping options. That is, it is considered how links are mapped to messages and how DDD operations are mapped to API operations.

RQ1.2 considers what the relevant decision drivers in those ADDs are. We have observed 27 relevant decision drivers as shown in Table 2.2 From this detailed list we can generalize a number of main concerns API developers care about. Please note that GT aims to explain phenomena that have been observed to exist [202]; that is the number of evidences listed in Table 2.2 can at most provide a rough indication of force importance. More interestingly, the forces can be categorized into a number of recurring themes:

- A number of forces focus on a *loosely coupled relation of API/its clients and the Domain Model* (f2, f16, f17, f21, f23) which is related to *independent modifiability and evolvability of the API* (f9, f10) as well as *API stability* (f24, f1) in relation to Domain Model changes (f25).
- Various forces concern *maintainability aspects of the API e.g. complexity and understandability* (f18, f7, f14, f22).
- Some forces consider the *consistency of data* (f11, f20).
- A number of forces are related to *runtime properties of the API* including performance and scalability (f3, f4, f5, f6), bandwidth use (f12), and reliability (f19).
- The *relation of API client and server (API provider)* is also important; it should be loosely coupled (f13), usable (f8), and consider different types of API clients (f27).

- *Costs and effort* are reappearing as forces (f15, f26).

RQ1.3 investigates which relations the decisions and decision options have. Here, the decisions define most of the relations via their options in our model. While the first two decisions *Domain Model Mapping Decision* and the *API as Contract Decision* mainly need to be decided once per API/Domain Model, lower-level decisions need to be made repeatedly. For example, the *Designing API Resources Decision* needs to be made for each *Identified Interface Element* in the *Domain Model* and those *Identified Interface Elements* can change based on prior decisions. Thus there are *no unequivocal recommendations* in the practitioner literature, such as "*Aggregate Roots as API Resources* work best in the most common design situation and the other work well in certain niches". Rather a deliberate, incremental, and iterative human (often collaborative) design approach required. Always decisions depend on project context, goals, and requirements. For instance, for the *Designing API Resources Decision* (as an example), the advice is given to start off considering *Aggregate Roots as API Resources*. If they do not provide a well working abstraction, *Domain Services* or *Processes* can be considered, if applicable. If those do not work well either, then maybe smaller-scale *Entities* or coarser *Bounded Contexts* might provide a well working solution. Similar considerations need to be made in the contexts of the *Resource Segregation Decision*, *Link Mapping Decision* and *Operation Design Decision*. Such a deliberate, incremental design process to map domain models to technical realizations is commonly suggested in the practitioner literature. For example, in the related area of identifying microservice boundaries, one of our sources (s18) suggests a very similar process of first analyzing *Bounded Contexts*, then considering *Aggregates*, then analyzing *Domain Services*, and finally considering non-functional requirements[5]. In this context, it is also interesting to observe that the recommended practices for microservice identification and API or API resource identification differ. Just exposing all microservices in system in an API without a deliberate, incremental API design effort, as explained above, might lead to bad API designs.

In addition to these main relations, the two decisions *Domain Model Mapping Decision* and *API as Contract Decision* have options that are closely related, as shown in Figure 2.4. Finally, a couple of decision options from the designing API resources, link mapping, and resource segregation decisions have relations to well-established software patterns such as *CQRS*, *Processing Resource*, *Linked Information Holder*, and so on, as detailed above.

Our resulting formal model describes a theory with a precise classification (or categorization) of model elements, as design decision, related context, decision options, decision driver, and related decisions and options. The model reveals interesting insights on which conceptual elements are present in practitioner discussions. Our work can help scientists to gain insights into current practitioner views, whereas practitioners can get a consolidated view integrating many practitioner views based on empirical research methods.

[5]See: `https://docs.microsoft.com/en-us/azure/architecture/microservices/model/domain-analysis`

Threats to Validity

As GT is mainly concerned with phenomena that have specifically been observed to exist, threats to the results' validity are mainly restricted to inappropriate conceptualization [202]. There is a risk that generalizing from some of our results might be misleading, but as we do not claim completeness and use a rather high number of sources (i.e., more than needed for theoretical saturation), it is likely that generalization is valid to at least some extent. At any rate, our results are only valid in our set scope.

To increase internal validity or credibility, we decided to use practitioner reports that were produced independently from our study. This avoids bias, e.g. compared to interviews in which the practitioners would have known that their answers are used in a study. However, this introduces a different threat: Some important information might be missing in the reports, which would have been revealed in interviews. We tried to mitigate this threat by looking at more sources than needed to reach theoretical saturation, as it is unlikely that all different sources miss the same important information. Different members of the author team have cross-checked all models independently to minimize researcher bias. The threat to validity that the researcher team is biased in some sense remains, however. The same applies to our coding procedure and the formal modeling: Other researchers might have coded or modeled differently. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study.

The experience and search-based procedure for finding knowledge sources may have introduced some kind of bias as well. However, this threat is mitigated to a large extent by the chosen research method, which requires just additional sources corresponding to the inclusion and exclusion criteria, not a specific distribution of sources. Note that our procedure is in this regard rather similar to how interview partners are typically found in qualitative research studies in software engineering today. However, the threat remains that our procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches.

2.6 Conclusions

We have studied the relation of DDD practices and microservice API design. We have identified six ADDs with 27 decision options concerning the mapping of domain models to APIs, defining API contracts in relation to domain models, designing API resources based on domain model elements, segregation of API resources, mapping domain model links to the API, and designing the operations of an API resource. In addition, we identified 27 decision drivers (forces) considered by practitioners in those decisions, which can be broadly categorized into forces on *loosely coupled relation of API/its clients and the Domain Model*, *maintainability aspects of the API such as complexity and understandability*, *consistency*

of data, *runtime properties*, the *relation of API client and server (API provider)*, and *costs and effort* (see Section 2.5). Finally, we identified numerous decision-option relations and other relations. Those usually require a deliberate, incremental human design effort. Our results can help scientists to get a better understanding of practitioner concerns, and practitioners to get an overview of the current view of other practitioners on the interrelation of microservice APIs and DDD. Moreover, the design guidance by our ADD model also can help to reduce design efforts and risks.

Chapter 3

On the Practitioners' Understanding of Coupling Smells: A Grey Literature Based Grounded-Theory Study

This chapter aims to extend the enhancement on the interrelation of Microservice API and DDD by observing the practitioners' understanding of the coupling smells. As a consequence, the practitioners' perspective on coupling smells could provide eight lessons. This study concerns **RQ2** and applies the same research methodology as chapter 2. Anyhow, a grey literature-based grounded theory is elaborated in this chapter. The content of this chapter is based on a peer-reviewed journal published in **IST'21**.

Code and design smells, such as the coupling smells examined in this chapter, are widely studied. Existing empirical studies reveal gaps between the scientific theory and practice, not yet explained by the scientific literature. Only basic coupling smell detection approaches and metrics seem to have been transferred to practice so far. This chapter aims to study the current practitioner's understanding of coupling smells. Based on grey literature sources containing practitioner views on coupling smells, we performed a Grounded Theory (GT) study. We used UML-based modeling to precisely encode our findings and performed a rigorous analysis of our codes and models. Our results are defining factors of coupling smells, as well as smell impacts, trade-offs, relationships to other smells, relationships to practices and patterns, and fix options as perceived by practitioners. We further identified gaps in the understanding of coupling smells between science and practice, and derived opportunities and challenges for future scientific work. Eight lessons are presented as opportunities and challenges for future research. Our results can help scientists to get a better understanding of practitioner concerns, and practitioners to get an overview of the current perception of other practitioners on coupling smells.

3.1 Introduction

A *Code or Design Smell* is a symptom of poor implementation or design choices that often corresponds to a deeper problem in a software system [1]. The term *Coupling Smell* refers to those kinds of smells that contribute to a specific kind of design problem: excessive coupling between classes. As an example of a coupling smell consider a Method m of a Class A that uses two methods of A itself, but 10 methods of Class B. This method likely suffers from the *Feature Envy* coupling smell because it uses the features of B excessively. Coupling could be substantially reduced, if m is moved to B or – even better – the part of m that uses B could be extracted and then moved to B.

The scientific literature has developed a substantial number of high-quality research studies on code and design smells [143], including metrics to detect code smells [79], smell detection approaches and tools [40, 111, 38, 4, 88], smell fixing approaches and tools [178, 144, 174, 29], taxonomies [94, 98], and a considerable number of empirical studies with human participants [26, 95, 110, 55, 191]. Whereas practitioners participated in many of these empirical studies, this is rarely the case in approaches and tools articles. However, the existing empirical studies indicate that there are gaps between the scientific results and practice. For many current scientific approaches, the actual accuracy of smell detection and fixing in complex coupling situations relevant in practice is either low or unclear (see Section 3.2). We also observed that only relatively trivial approaches, such as basic definitions and metrics, seem to have been transferred to practitioner views, approaches, and tools yet. Finally, smells which reside at the boundary between code and design quality, such as *coupling smells*, seem to be more prone to these issues than simplistic code smells such as *Long Method*. These problems have led to this study, which aims to investigate the current practitioner's understanding of coupling smells.

To illustrate the research problem further, let us give a few examples of interesting phenomena not well explained by the current scientific literature Our work is motivated by a couple of observations. First, a number of empirical studies reveal interesting gaps between the scientific literature and practice. Exemplary phenomena that are not well explained by the current literature on code smell approaches and tools – identified in existing empirical studies on smells: Guo et al. [55] observed that domain-specific tailoring of code smell detection rules and heuristics can greatly improve the human understanding of smells. Furthermore, they found problems in encoding code smells into a tool using metrics from the scientific literature. Mantyla et al. [95] found that the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators, and that metrics and smell evaluations did not correlate. Palomba et al. [110] found that instances of a smell may or may be problematic based on the "intensity" of the problem, and that developer's experience and system's knowledge play an important role in the identification of some code smells. Yamashita and Moonen [191] investigated serious interaction effects between smells. Palomba et al. [110] concluded: "Indeed, there seems to be a gap between theory and practice, i.e., what is believed to be a problem (theory) and what is actually a problem

(practice)."

Second, for many current scientific approaches the actual accuracy of smell detection and fixing in complex coupling situations relevant in practice is either low or unclear (see Section 3.2). Third, we observed that only relatively simple approaches, such as basic definitions and metrics, seem to have been transferred to practitioner views, approaches, and tools yet. Fourth, smells which reside at the boundary between code and design quality, such as *coupling smells*, seem to be more prone to these issues than simplistic code smells such as *Long Method*

We thus explore the current practitioner's understanding of *coupling smells*. In this chapter, we describe a *Grounded Theory (GT)* [51, 24] based qualitative study for the above mentioned purpose. We decided to perform a Grey Literature Study (GLS) of acknowledged practitioners' views on *coupling smells*. According to Rainer and Williams [132] there are many benefits of GLS in software engineering research, as they promote the voice of practitioners and provide information on practitioners' contemporary perspectives on important topics relevant to practice and research. In our GT study we used, after initial text-based open coding, formal UML-based modeling for axial and selective coding, instead of the often-used text-based coding process, in order to develop a precisely defined and consistent theory. We have successfully used similar research methods in a couple of prior studies [195, 194, 93, 109]. A secondary contribution of this chapter is that the description of this method in Section 3.3 can be used as a definition of this GT variant for later use in other research studies, and the coupling smell study in this context can be seen as a detailed example. We set out to answer the following research questions:

- **RQ2.1** How are coupling smells understood by practitioners?
  - **RQ2.1.1** What are the defining factors of coupling smells as perceived by practitioners?
  - **RQ2.1.2** What are the impacts on and trade-offs to be made with regard to code and design quality when considering coupling smells as perceived by practitioners?
  - **RQ2.1.3** What are the relations and interactions among coupling smells, to other related smells, and to other software code and design concepts as perceived by practitioners?
  - **RQ2.1.4** What are the options for fixing coupling smells as perceived by practitioners?
- **RQ2.2** What are gaps in the understanding of coupling smells between the practitioner's view and the scientific literature?
- **RQ2.3** What are opportunities and challenges for future scientific work to address the practitioner concerns on coupling smells well?

Our results comprise a set of relevant coupling smells described with a detailed meta-model and a set of definitions describing the defining factors of coupling smells. For each smell, we found many possible liabilities and violations of software design principles that explain their impacts on code and design quality. Our model contains detailed relations among the smells, as well as to existing practices and patterns. Finally, for each smell the model includes detailed options for fixing the smell. We compared our results to previous studies in this field. This revealed interesting gaps, from which we derived eight lessons that represent opportunities and challenges for future research.

We believe that our results can help scientists better understand practitioner concerns regarding code smells. This can help in better explaining the mentioned empirical results and target future research towards approaches that likely have a high chance of adoption in practice.Our work can also help practitioners get an overview of other practitioners' current views on coupling smells.

This chapter is structured as follows: First, we discuss the related work in Section 3.2. Next, we explain our research method in Section 3.3. In Section 3.4 we present the detailed results of our grounded theory, i.e. the resulting meta-model and the detailed results for the coupling smells. Section 3.5 discusses the implications for the research questions, which includes an in-depth comparison of our results to the related work, and a discussion of threats to validity. In Section 3.6 we draw conclusions.

3.2 Related Work

Research works on coupling and related smells detection and repair

Many studies on coupling smells are about engineering detection and/or smell fixing approaches and tools. Most existing works use metrics or other measures on structural or behavioral features of the code to detect smells. In a recent systematic literature review [143] static source code analysis (such as behavioral, empirical, algorithm-based, methodology-based, and linguistic source code analyses) and dynamic source code analysis (such as dynamic threshold adaptation or genetic algorithms), are found as the primary methods for smell detection. Refactorings, such as those from Fowler's book [1], are often used or suggested as fixes for the smells.

Lanza and Marinescu [79] suggest metrics to identify a large number of code smells. For instance, for *Feature Envy* detection they suggest the Access To Foreign Data, Locality of Attribute Accesses, and Foreign Data Provider metrics, as well as metrics thresholds. The book also covers *Data Class* and other smells, as well as situations of intensive coupling underlying many smells discussed in our work. It suggests detection strategies, i.e. metrics-based rules, to detect the smells in the code. Lanza and Marinescu [79] suggest metrics and thresholds to identify a large number of code smells. The book covers *Feature Envy*, *Data Class* and other smells, as well as situations of intensive coupling underlying many smells discussed in our work. The authors recommend detection strategies, i.e. metrics-based

rules, to detect the smells in the code.

JDeodorant [40] is a tool for detecting smells such as *Feature Envy*. Chatzigeoriou and Manakos [20] adopt JDeodorant to investigate three code smells (*Long method*, *Feature Envy*, and *State Checking*) in two open source projects. They recommend resolving *Feature Envy* by the move method and move field refactorings [1], but the study is limited to the refactoring options offered by the tool. In another work, Tsantalis and Chatzigeorgiou [178] suggest distance metrics as a way to identify move method refactoring opportunities to remove *Feature Envy*. Palomba et al. [111] suggest HIST, a history-based smell detection approach. They compare HIST to JDeodorant for detecting *Feature Envy*, (they also compare to other tools for non-coupling smells). They conclude that the overall F-measure for HIST is 77% and for JDeodorant 68%. For this evaluation, they used a manually produced oracle from 20 open source systems. This work led to the Landfill open data set [112], one of the few manual oracles in the field of code smell. A summary of research related to JDeodorant can be found in [177].

Distance metrics are often applied in the context of move method refactoring [1], which is one of the most commonly proposed resolution strategies for coupling smells (in the scientific literature and also in the results of our study). Here, Metrics-based identification helps to identify an opportunity for refactoring. JMove [144, 174] is based on an approach for move method refactoring recommendation that uses static dependency sets. The precision and recall for JMove [144, 174] were evaluated on synthesized versions of open source systems, in which the authors moved random methods to random classes. It is assumed that a tool should suggest the opposite of the randomly moved methods. JMove's precision ranges from 21% to 32% and its recall ranges from 21% to 60%. While the evaluation approach might evaluate well some other goals of JMove, we have strong doubts that this synthesized evaluation resembles in any way the rich understanding of coupling smells that practitioners have (and which is carved out in our chapter).

Dipongkor et al. [29] propose another move method refactoring recommendation approach based on the frequency of coupled methods and attributes. Rahman et al. [130] recommend to move methods based on coupling, cohesion, and contextual similarities.

Fontana et al. [41] exploit code smell relations to assess software architecture quality. They identify code smells by recurring anomalies and metrics. Their code smells include among other smells, the *Data Class* and *Message Chain* coupling smells. They evaluate 74 systems to figure out the relationships among the code smells. Most of these smells have some impact on coupling aspects.

Vidal et al. [183] present, prioritize, and evaluate criteria to identify architectural problems using their JSpIRIT tool. 23 versions of four systems are evaluated to reveal smells including *Feature Envy*. For smell detection, they simply use the catalog by Lanza and Marinescu [79] and assume correct identification.

Fard and Mesbah [38] propose automated JavaScript code smells detection in their Jsnose tool which combines a metrics-based approach with static and dynamic analysis. They gather 13 code smells including *Message Chain*. They use rather simple metrics to

identify the smells, but most of their smells are simple in nature, too.

Some authors experiment with machine learning approaches for smell detection. Fontana et al. [4] compare 16 machine learning algorithms on four code smells, including *Data Class* and *Feature Envy*. They use a data set of 1986 manually validated code smell samples from 74 software systems, and They report a highest accuracy of 96%. However, their training sets are generated by tools using simple metrics and heuristics. That is, the 96% measure how well the machine learning algorithms can find the problematic simple understandings from the other tools, not the semantically more richer interpretations of smells by practitioners (which are revealed in our chapter). Liu et al. [88] suggest deep learning based *Feature Envy* detection based on textual features of the source code. They compare the accuracy precision, recall, and F1-measure of their approach to JDeodorant and JMove. They claim an F1-measure of 18.66% for JDeodorant and 17.27% for JMove, whereas their own approach offers an F1-measure of 52.98%. However, such numbers need to be considered with great care, as the training data is generated based on distance metrics; that is, one of the approaches the authors wanted to improve with their approach is used for training data generation.

Our study investigates among many other research problems whether the rather narrow definitions of coupling smells and the rather isolated studies of certain smells and fixing options reflect the current concerns of practitioners with regard to coupling smells. Our study also reflects on the assumption that coupling smells can be understood and then detected by rather simple heuristics, metrics, static or dynamic analyses, or metrics-based rules, made by many of the approaches summarized above. We also investigate whether the smells can be fixed by rather limited refactoring options. We further point out directions in which scientific approaches differ from the understanding of practitioners and how future research could be adapted to be more relevant to practitioner concerns. Systematics qualitative studies such as the one presented in our chapter can help in those directions by pointing out clearly what the practitioners' problems and understandings of coupling smell phenomena are.

Research works on improving smell detection and repair with additional knowledge

Several studies try to improve smell detection and repair them with some kind of domain or otherwise specialized additional knowledge. As our study results indicate this as a promising future research direction, we highlight different kinds of related approaches in this section below.

Concerning architectural issues, some approaches study the impact of coupling smells on architectural technical debt [41, 183]. A more promising approach might be to study architecture coupling smells directly; e.g. Garcia et al. [47] introduce four architectural smells directly based on architecture concepts (Connector Envy, Scattered Functionality, Ambiguous Interfaces, and Extraneous Connector). Connector Envy represents the coupling

problem when a component utilizes functions from another component or other components excessively. focusing on architectural component interactions. Another even more domain-specific example is the study of Taibi et al. on microservice smells [171] which uses architecture structures and specifics of microservices-based architectures to define the smells precisely.

Using architecture-specific knowledge is just one way to improve coupling smell identification. Other approaches focus on other specialized knowledge. Ratiu et al. [133] use historical information to increase the accuracy of automatic detection. They consider the *God Class* and *Data Class* smells. Another history-based approach is suggested by Palomba et al. [111]. Marinescu [97] shows that detection accuracy of the *Data Class* and *Feature Envy* smells can be improved by considering aspects of the studied enterprise applications. Fontana et al. [3] suggest filters to remove false positives, e.g. a filter on test class methods is defined for *Message Chains*, and for *Data Classes* filters such as Serializable Class, Test Class, and Logger Class are defined. Guo et al. [55] present an approach to tailor the detection heuristics to take domain-specific factors into account.

The many studies aiming at defining more targeted or filtered smell detection approaches confirm our research motivations, as they would not be attempted if the generic smell detection would already be accurate enough. Authors also acknowledge that different smell detection tools yield very different results and that there are issues with subjective interpretation of the smells by different authors [3]. Our study results confirm that domain-specific or more targeted smell detection should be considered as a future research direction.

Systematic studies of coupling and related smells

There are a number of studies on taxonomy and empirical studies of coupling smells. Also many smell detection and fixing approaches are studied empirically. Some studies mentioned above contained empirical evaluations. Sabir et al. [143] perform a systematic literature review to study approaches for detection of smells and their evolution in object-oriented and service-oriented systems during 2000-2017. In their coupling category they identify the *Schizophrenic Class*, *Message Chain*, *Middle Man*, *Incomplete Library Class*, *Feature Envy*, *Inappropriate Intimacy*, *Intensive Coupling*, *Extensive Coupling*, and *Unnamed Coupling* smells as coupling smells. *Message Chain*, *Middle Man*, *Feature Envy*, *Inappropriate Intimacy* have also been identified in our study as coupling smells, but we have identified a few other smells than discussed in the scientific literature and some of the smells from the literature study are not or only implicitly discussed by practitioners in our study. As in our study, *Feature Envy* is considered most often in the scientific literature. The survey identified various static and dynamic source code analyses as the primary methods for smell detection (summarized above). deletedHere, our study reveals a great discrepancy: so far only a few of these techniques are applied in practice. In addition, our study results indicate that the understanding of smells in practice, as we will discuss in detail below, often makes these techniques hard to apply.

Mantyla et al. [94] present a subjective taxonomy and provide an initial empirical study on bad smells in code by performing a survey with developers of a small Finnish software company. They categorized 22 smells into 7 categories: Bloaters, Object-Orientation Abusers, Change Preventers, Dispensables, Encapsulators, Couplers, and Others. They suggest that grouping similar smells can increase their comprehensibility and reveal meaningful relationships among the smells. For the Couplers, they categorize *Feature Envy* and *Inappropriate Intimacy*. Our study of practitioner views include these two smells in this category, too, but reveals that practitioners see many other smells as coupling smells. Marticorena et al. [98] aim to extend the taxonomy of Mantyla et al. [94] with metrics. Their Couplers category comprise of *Feature Envy*, *Inappropriate Intimacy*, *Message Chain*, and *Middle Man*, i.e., a larger fraction of those smells identified in this category in our study. In our work, we used these two works as a foundation for defining the original search strategy in our GLS. Our study extends those studies with a broader view on the coupling smell category and detailed evidence on the practitioner views on this smell category.

Carneiro et al. [26] investigate in an empirical study whether multiple views of a concern such as package-class-method structure, inheritance structure, dependency, and dependency weight views influence code smell detection. They investigate the *Feature Envy*, *God Class*, and *Divergent Change* smells. This exploratory study is performed with only 5 participants, so the results are not highly reproducible. The recall and precision for the *Feature Envy* smell were only 0.4 and 0.2, respectively, which confirms points made below related to difficulties related to detecting *Feature Envy* by practitioners.

Mantyla et al. [95] describe the results of a case study in a Finnish software company, where they studied the evaluator effect when subjectively evaluating the existence of smells in code modules. This study is interesting, as it is one of the few studies that actually consider subjective impressions of practitioners. The study considers the *Feature Envy* and *Middle Man* smells. Mantyla et al. found that the use of smells for code evaluation purposes is hard due to conflicting perceptions of different evaluators, a notion strongly confirmed by our study. Secondly, they applied source code metrics for identifying three smells and compared the results to the practitioner evaluations. They concluded that the metrics and smell evaluations did not correlate. Mantyla et al. find this result surprising; according to the results of our study, it can be seen as the expected result.

Palomba et al. [110] report about an empirical study on the developers' perception of 12 bad smells. They have shown code entities (that are affected by the smells or not) from three open source projects (ArgoUML, Eclipse, JEdit) to original developers, invited industrial developers, to developers and master students. Regarding the coupling smells discussed in our chapter, *Feature Envy*, *Inappropriate Intimacy*, and *Middle Man* are considered in their study. The authors found that some smells are generally not perceived by developers as design problems, including *Middle Man* and *Inappropriate Intimacy*. Those smells are among various others *Middle Man* and *Inappropriate Intimacy*, also included in our chapter. They also found that instances of a smell may or may not represent a problem based on the "intensity" of the problem. This is confirmed in our results. Finally, they found that

developer's experience and system's knowledge play an important role in the identification of some smells. They claim smells related to possible misuses of OO principles, such as *Feature Envy*, are among those. Our study shows that practitioners see relations to OO principles in all coupling smells. Furthermore our results indicate that interactions with other smells, the relations to patterns and practices, the system context, the impacts on design and code quality, and other factors might need to be taken into account to judge the smells. Our results thus would provide another explanation for some smells not being perceived as design problems; maybe in a very smelly system context with rich interactions and a complex domain design, the participants would have analyzed the smells differently.

Guo et al. [55] present an approach to tailor the metrics and detection rules to take domain-specific factors into account. Input for these domain-specific heuristics is derived from an iterative empirical field study in a software maintenance project. Among others, they consider the *Data Class* smell. They found that the code smell definitions, as proposed by Lanza and Marinescu [79], were judged to be accurate and actionable by a panel of practitioners, but they required tailoring to be applicable in the project. They found that simple tailorings can improve the results. Problems in encoding code smells into a tool called CodeVizard are reported, too, especially that some semantic factors could not be encoded into the tool.

Yamashita and Moonen [191] studied to which extent problems in maintenance projects can be predicted by code smell presence. In a multiple case study six developers working on four different Java systems were observed. Code smells were detected using tools before the study. In-depth examination of quantitative and qualitative data was conducted to determine if the observed problems could be explained by the detected smells (including *Data Class* and *Feature Envy*). Roughly 30% percent of the problems investigated were somehow related to files containing the found code smells. They observed interaction effects amongst code smells, and between code smells and other code characteristics, and these effects led to severe problems during maintenance. They conclude that the role of code smells in the context of the overall system maintainability is relatively minor, and thus they alone cannot predict maintainability issues. In any such analysis, interaction effects amongst collocated smells and coupled smells should be taken into account. These findings are generally confirmed by some of our results , and our results would predict that the tool results without manual review might not lead to a set of smells that practitioners would accept as being relevant.

Related grey and multivocal literature studies

In recent years, the grey literature is increasingly used as a foundation for studies in evidence-based software engineering, for example, with the goal of unfolding the state of the practice. In the social media era, software developers and other practitioners change the way how they communicate and how they share their knowledge. It is argued that researchers cannot ignore this data source. Garousi et al. [50] provide an overview and how

to get the best benefit out of grey literature studies in software engineering. Our work applies grey literature as a data sources of a qualitative study. Rainer and Williams [131] explore the opportunities for using blog articles in software engineering research, which are heavily used in our study. While their research only focuses on blog articles, our research also considers other online grey literature data sources. Ma et al. [92] investigate the reliability of pervasive web content in evidence-based software engineering. They apply the systematic literature review method in the deep learning domain. They confirm the value of the information in Web content and a strong correlation among different Web content sources. They report on difficulties of data extraction from web content, and state that modeling is a powerful tool. Our work also derives the data from Web content sources, and puts additional emphasis on data extraction and analysis via the grounded theory method. Garousi et al. [48] study the gap between systematic literature reviews and grey literature studies. They argue for the need of multivocal literature reviews, i.e. studies combining the two methods. In multivocal literature reviews, grey literature adds value by bringing practitioner experience and opinions into the literature review. Our work also contrasts the grey literature and the scientific literature, but in contrast to multivocal literature reviews we study the grey literature in a qualitative, exploratory study in order to understand discrepancies between scientific and practitioner views, as well as reasons for adopting or not adopting certain scientific concepts in practice. The work by Fard and Mesbah [38] on the Jsnose tool, discussed above, gathers 13 in part Javascript-specific code smells from various Web development resources, i.e., the grey literature. Like this work, our study also gathers textual data from various grey literature sources but in more systematic way.

Related grounded theory and other qualitative studies

We analyze the data in our study in a qualitative fashion using the Grounded Theory research method, explained in detail in Section 3.3. In the information system, Grounded Theory is applied to manage the knowledge because textual data can be analyzed in several ways. Wolfswinkel et al. [189] use grounded theory as a method for rigorously reviewing literature in the manner that is relevant to information system. They present five stages (define, search, select, analyze, present) as the guidelines inspired by the work of Webster and Watson [186]. Our work also adopts their roadmap to conduct the coupling smell ground theory based on grey literature. Numbers of papers present how to use ground theory on software engineering in practice, for example Adolph et al. [2] study how people manage software development process with the grounded theory methodology. Their study consists of three phases which are a first report, a pilot study and a main study. In stead of producing a grounded theory, they provided the lesson learned for grounded theory evaluation. Regarding their big concern about the fundamental practices of grounded theory, our research confirm to utilize grounded theory methodology to generate the practitioners based theory. Hussain et al. [66] apply grounded theory to investigate Agile User-Centered Design in practice. They collect data from different Europe-based professionals who work

in this field by semi-structured interview, observation, and their documents. In conclusion, the usability is an importance non-functional requirement that agile team members realize. Even though, they collect data from various sources but five participants. Our work include different types of data sources from more than 40 participants. Siau and Tian [185] propose open source software development process model through an in-depth grounded theory analysis. Their data source is from a basic communication tool called QuantLib. They used 820 messages to identify the factors of open source software development project and proposed a Phase-Role-Skill-Responsibility process model. They study one particular open source project with five years data which is quite precise in scope. Our work is not limited to project, it could bring several emerging directions in the future. Bang et al. [7] use the grounded theory to derive knowledge, skill, and abilities for DevOps on modern web applications. They start with tools in modern web applications, then the skills for using tools, after that the knowledge from skills, finally the knowledge that are related to the DevOps properties. Our work applies grounded theory to derive knowledge about the definition, characteristic and relationship of coupling smells in opened coding phase. Some researches claim that they build up the grounded theory. Sousa et al. [162] build the grounded theory to identify design problems in the source code. They conducted the experiment with diverse experienced developers in five different software companies from North and Northeast Brazil. Four activities in their experiment consist of subjects characterization, training, problem identification, and follow up. They also conclude that the knowledge about design problem identification in practice is limited.Their work is very well organized. They can report the very useful results and they also show us how difficult it is to deal with people. Our work studies the developers' perception in textual form. Each textual data generate an individual theoretical saturation of each data source. The certain amounts of the individual knowledge could ensure the theoretical saturation in the big picture. Waterman et al. [185] present grounded theory of Agile Architecture. The theory consists of five strategies and six forces to answer how much upfront architecture design effort is enough. Their data source is from face-to-face semi-structured interviews with 44 agile practitioners. Consequently, the trade-off between a full up-front architecture design and a totally emergent design is required a proper decision to maximize agility. Our work gathers the data from grey literature which widely use as a first source of developer's knowledge.

3.3 Research Method

This chapter aims to systematically study practitioner knowledge on the established practical knowledge and practices in the field of design and code coupling smells based on a GLS focused on practitioner views on coupling smells. As outlined in the introduction, we focused on coupling smells because they reside at the boundary between code and design quality. For coupling smells we observed in an initial skim review of the literature (both scientific and grey literature) that it is interesting to study practitioner views on them in-

depth. In particular, in contrast to simple code smells such as *Long Method*, the following aspects motivated us to study practitioner views on coupling smells: both practitioners and scientists seem to agree that they have a high importance for design quality, but there seems to be a gap between scientific and practitioner views on them, the actual accuracy of smell detection and fixing in complex coupling situations relevant in practice is either low or unclear, and only relatively simple approaches, such as basic definitions and metrics, seem to have been transferred to practitioner views, approaches, and tools yet.

We performed a GT study in which we used UML-based modeling to precisely encode our findings. GLS is used as the data collection technique for GT. A number of methods have been suggested to study established practices. A classical method is pattern mining (see e.g. [23]) which starts with the authors' own experiences, searches systematically for other known uses in real-life systems, and then applies a series of feedback loops to improve the pattern. A number of techniques have been suggested for improving this research method. Hentrich et al. [61] define a pattern mining method as a form of qualitative research resembling methods like Grounded Theory (GT) explained in Section 3.3. In this chapter, we describe a GT study based on grey literature texts as data sources – a research method that has been developed based on our experiences in pattern mining, applying GT, and architectural design decision modeling. In the grounded theory study we used, after initial text-based open coding, formal UML-based modeling for axial and selective coding, instead of the often-used text-based coding process. UML-based modeling is introduced with the goal to develop a precisely defined and consistent theory. We have successfully used this and very similar research methods in a couple of prior studies [195, 194, 93, 109]. As it is a rather new combination of research method elements, we explain it and its motivation in this section in detail. A secondary contribution of our study is that this section can be seen as a definition of this research method for later use in other research studies, and the coupling smell study presented below can – in this context – be seen as a detailed example study.

Grounded Theory

Glaser and Strauss [51] introduced *Grounded Theory (GT)* as research method for discovery of theory from systematically obtained and analyzed data. The method has two unique properties: constant comparison and theoretical sampling [51]. *Constant comparison* means that the method uses an iterative and incremental process of data collection and analysis. GT does not completely collect the data before it starts the data analysis, but the data collection and analysis are performed in each iterative research step , i.e. almost in parallel. GT derives concepts from the data instances (called data incidents in [24]) and then categories from the derived concepts. Comparisons of data instances with other instances need to be made during data analysis [24], as well as comparisons to concepts and categories. *Theoretical sampling* is the term used for the data collection process for this iterative and incremental constant comparison [51]. In it, means that results of each data analysis step

are used for the following data collection. The researchers should actively find new data sources driven by the results of the data analysis. Researchers need to be theoretically sensitive [24]; that means, they should think effectively about what types of data need to be collected and which aspects of the data already collected are most important for the theory [69].

Different incarnations of GT in the social sciences follow a different underlying philosophy and use different data analysis and coding steps [76]. In this chapter, we roughly follow the GT data analysis and coding steps by Corbin and Strauss [24] who designed a highly systematic and rigorous coding structure to create (rather than to discover) a rigorous theory which closely corresponds to the data [76]. They use three main coding steps:

- *Open coding* is performed first after data collection. It aims to examine the data and identify discrete elements in the data. Open coding identifies concepts in the data.
- During *axial coding* the researchers identify categories in the concepts, e.g. by identifying concepts that reappear in the data, synonymous concepts, related concepts, etc. That is, axial coding performs classification of concepts and identification of their relationships.
- *Selective coding* is about carving out main ideas of story lines of the theory, i.e. understanding the big picture by reflecting on the data and analysis results. It also involves double checking that no mistakes were made. An important step is the reduction of concept and categories. Parsimony and simplicity are important goals to achieve.

GT ends when *theoretical saturation* is reached. This occurs when no new concepts emerge from new data sources anymore and the theory has been thoroughly validated with the collected data [69].

Grey Literature Study

Grey literature in software engineering can be defined as "any material about software engineering that is not formally peer-reviewed nor formally published [50]." According to Rainer and Williams [132] there are many benefits in using grey literature sources in software engineering research, including that "they provide information on practitioners' contemporary perspectives on important topics relevant to practice and to research" and "promote the voice of practitioners [132]." software engineering research: In general, "they provide information on practitioners' contemporary perspectives on important topics relevant to practice and to research" and "promote the voice of practitioners [132]." The specific kind of practitioner-oriented posts and articles, our work focuses on, provide "(access to) information on the practitioner's experience and inexperience of theirs' and others' software practice; motivations for that practice; values relating to that practice; emotions relating to that practice; beliefs about software practice; empirical data from their practice; and

explanations of that practice [132]." As our research questions require practitioner views, we have opted against a Multivocal Literature Review (MLR) where scientific sources are included, too. Instead we decided to focus on grey literature sources stemming from practitioners and mixed groups (practitioners and others). According to Garousi et al. [49] such grey literature sources includes blog posts, presentations, Wiki articles, technical reports, audio-video material, practitioner book content, and many other kinds of sources. So-called black literature, i.e., mere concepts, ideas, and thoughts, has been excluded. Garousi et al. [50] also stated that "grey literature sources can be classified according to the two dimensions: expertise and outlet control." Expertise is the extent to which the authority and knowledge of the producer of the content can be determined. Outlet control is the extent to which content is produced, moderated or edited in conformance with explicit and transparent knowledge-creation criteria." In our work we are interested in (technical) practitioner knowledge targeted at other practitioners. Please note that many practitioners self-publish such grey literature and do not care much for outlet control. We thus reviewed each source in the author team and only if all authors agreed that the content contains knowledge by acknowledged practitioners we included the source. We excluded sources that seemed to sell a product or a service. As we intended to use grey literature sources in a GT study, the actual search process was guided by GT's theoretical sampling concept and stopped when theoretical saturation emerged. Assuming that the GT literature is right about theoretical saturation, if the set of sources is large enough, a valid theory will emerge regardless of the particular selected set of sources or their order. We will discuss risks of this process in threats to validity.

In social science, interviews are often used as data sources for GT, but it is clearly described in the literature that GT can be used with any kind of data [51, 24]. Given that we are not studying some general topic which more or less any software developer can speak about without preparation, and as expertise for coupling code smells is rather rare, searching for interview candidates could have led to substantial bias in the selection process. These are the main reasons for using grey literature instead of interviews. Rainer and Williams [132] summarize some other reasons that have been important for this decision, namely: grey literature has compensated for the unavailability of other sources of evidence. We were able to access harder-to-access practitioners, and we could gather information for the research in a non-invasive way. We were able to scale-up the research to larger number of samples, and complement and triangulate them with, other sources of data. Finally, it was easier to provide an audit trail of the research, and thus enable repeatability of the study through public access to original data[1].

Using a search engine to find grey literature results in various types of data [57]. One major concern about search engines in such research is their search algorithm because the results are dependent on the user [123]. To avoid personal bias in the research, our initial

[1]We provide all open and axial coding files, derived coded models in Python, and generated models (in UML, Markdown, and Latex) as a replication package for download in the Zenodo long-term open access archive [155].

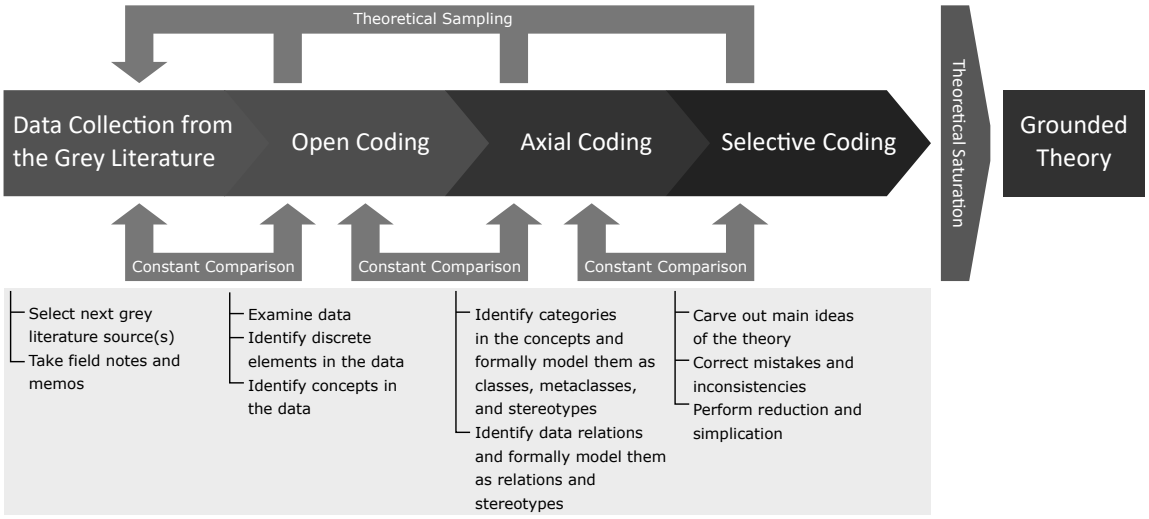

Figure 3.1: Research Method Steps

search keywords are taken from "A Taxonomy for Bad Code Smells" [2]. Thus we have initially searched for the following keywords: "Code Smells", ("Couplers" or "Coupling") and "Smells", "Feature Envy", "Inappropriate Intimacy", "Message Chains", and "Middle Man." We used major search engines (e.g., Google, Bing, DuckDuckGo) and topic portals (e.g., InfoQ, DZone) to find relevant practitioner texts. We have opted against a Multivocal Literature Review (MLR) where scientific sources are included, as our research goals focus on exploring practitioner views specifically. Our initial data are from the keywords above. After the coding process, further data sources emerged in terms of new keywords or new referenced sources. For example, we added synonyms such as "Object Orgy" or new coupling smell candidates such as "Data Class" after they appeared the first time. The data collection process is repeatable with scanning and skimming techniques. During open and axial coding we studied each included source line by line in-depth during open coding – for most sources in many iterations. We chose this method over manually browsing selected grey literature initially because it is replicable [166]. In our open access appendix, each step in the open, axial, and selective coding, including the emergence of new concepts, is documented. This comes along with traces which parts of the sources led to which codes in all coding stages.

Methodology overview

Our application of the research method happened in many iterations. That is, we first searched for a few new knowledge sources (such as practitioner articles, forum posts, slides) using major search engines and topic portals. Then we applied the GT coding process (modified with UML-based modeling) to identify candidate categories and compared with the so-far-designed model continuously. We improved this model incrementally. A crucial question in GT is when to stop this process; here, theoretical saturation [69], explained in Section 3.3, has attained widespread acceptance in qualitative research: We stopped our analysis when 10-15 additional knowledge sources did not add anything new to our

[2] See `http://mikamantyla.eu/BadCodeSmellsTaxonomy.html` which is part of a scholarly article [96]

understanding of the research topic. As a result of this very conservative operationalization of theoretical saturation, we studied a rather large number of knowledge sources in depth (48 in total, summarized in Table 3.1 which is explained in Section 3.4), whereas most qualitative research often saturates with a much lower number of knowledge sources. We included knowledge sources, if they were advanced practitioner reports on their experiences or knowledge about coupling smells. We checked that the source texts fulfilled a certain minimum quality level, as described above.

Figure 3.1 illustrates the GT research method steps as explained in Section 3.3 in the upper half of the figure. In the grey box beneath the steps, we detail our specific research methodology steps. In particular, it can be seen that in the data collection step, we used grey literature as input for the data collection. The open coding step has been performed very closely to how it would be performed without UML-based modeling. In the axial coding step, we have then formalized the found concepts and categorized them using classes, meta-classes, and stereotypes, as well as their formal relations and relation stereotypes. Constantly, during the selective coding step, we have compared the results of the axial coding with the big picture to carve out main ideas of the theory, to correct mistakes and inconsistencies, and to perform reduction and simplification. Those steps have been repeated for each data source.

For modeling, we used our existing CodeableModels tool[3], a Python implementation for precisely specifying meta-models, models, and model instances in code with an intuitive and lightweight interface. Based on CodeableModels, we specified a meta-model and models for our study, extending both as needed. In addition, we realized automated constraint checkers and PlantUML code generators to generate graphical visualizations of all meta-models and models, as well as a full textual model output generation in Markdown and Latex.

[3] `https://github.com/uzdun/CodeableModels`

ID	Title	Archive URL	Author Type	Source Type	Exampl	Source Code
S1	What is the difference between Inappropriate Intimacy and Feature Envy?	https://bit.ly/3bFyKQT	Practitioner	Discussion Forum Post	False	False
S2	Code Smell	https://bit.ly/2S6Ca7K	Mixed	General Audience Article	False	False
S3	Code Smells	https://bit.ly/354s905	Practitioner	Practitioner Audience Article	False	False
S4	Code Smells	https://bit.ly/2VVaELq	Practitioner	Practitioner Audience Article	False	False
S5	Smells to Refactorings Cheatsheet	https://bit.ly/2VBtqbE	Practitioner	Practitioner Audience Article	False	False
S6	Feature Envy	https://bit.ly/2VApxE4	Practitioner	Tool Documentation	True	True
S7	Data Class and Feature Envy	https://bit.ly/2yO90mP	Practitioner	Practitioner Audience Article	False	False
S8	Code Smell: Data Class	https://bit.ly/2znerJN	Practitioner	Practitioner Audience Article	True	False
S9	Feature Envy Smell	https://bit.ly/2KCO3hi	Practitioner	Practitioner Audience Article	False	False
S10	Spotting Feature Envy and Refactoring	https://bit.ly/355ugAz	Practitioner	Practitioner Audience Article	True	True
S11	Resolving Feature Envy in the Domain	https://bit.ly/3cLhkm1	Practitioner	Practitioner Audience Article	True	True
S12	Object Orgy	https://bit.ly/2Y4XM8k	Practitioner	Practitioner Audience Article	False	False
S13	Object Orgy	https://bit.ly/3bD3u5j	Practitioner	Practitioner Audience Article	False	False
S14	Code Smells	https://bit.ly/2yFH6cO	Mixed	General Audience Article	True	True
S15	Refactoring Couplers	https://bit.ly/354O72Z	Practitioner	Practitioner Audience Article	True	True
S16	When to Start Refactoring Code and When to Stop	https://bit.ly/2xdpc0M	Practitioner	Practitioner Audience Article	True	False
S17	Improving Application Design with a Rich Domain Model	https://bit.ly/2VAqT1C	Practitioner	Slides	True	True
S18	Code Smell: Feature Envy or Data Envy	https://bit.ly/2W0Q5Nx	Practitioner	Practitioner Audience Article	True	True
S19	Practical PHP Refactoring: Remove Middle Man	https://bit.ly/2S78tTY	Practitioner	Practitioner Audience Article	True	True
S20	Rich Domain Model with DDD/TDD Reviewed	https://bit.ly/2VDK36P	Practitioner	Practitioner Audience Article	True	False
S21	Feature Envy in Component Design	https://bit.ly/2VD99mc	Practitioner	Practitioner Audience Article	True	True
S22	Write clean code and get rid of code smells with real life examples	https://bit.ly/3aCg57r	Practitioner	Practitioner Audience Article	True	True
S23	Middle Man Code Smell Resolution with examples	https://bit.ly/3aE1lVA	Practitioner	Practitioner Audience Article	True	True
S24	Bad smell in Code Inappropriate Intimacy	https://bit.ly/3bF2Yng	Practitioner	Practitioner Audience Article	True	True
S25	Inappropriate Intimacy Code Smell Resolution	https://bit.ly/2Y7fCav	Practitioner	Practitioner Audience Article	True	True
S26	Patterns in Practice Cohesion And Coupling	https://bit.ly/3bF3GRs	Practitioner	Practitioner Audience Article	True	True
S27	Feature envy	https://bit.ly/2xQZuQ2	Practitioner	Practitioner Audience Article	True	True
S28	Refactoring a Feature Envy Code	https://bit.ly/2KLOqqh	Practitioner	Practitioner Audience Article	True	True
S29	Feature Envy - Code Smell	https://bit.ly/353QLpP	Practitioner	Practitioner Audience Article	True	True
S30	Class: Reek::Smells::FeatureEnvy	https://bit.ly/2KzyC9Q	Practitioner	Tool Documentation	True	True
S31	Feature Envy Code Smell Resolution with examples	https://bit.ly/2VDr7Vz	Practitioner	Practitioner Audience Article	True	True
S32	Why are data classes considered a code smell?	https://bit.ly/2VQ4vAa	Practitioner	Discussion Forum Post	True	False
S33	Coding Best Practices: Clean Code, Refactoring and TDD	https://bit.ly/2W0Dlqy	Practitioner	Practitioner Audience Article	False	False
S34	Data Class - Is It Really A Smell?	https://bit.ly/2KIuggR	Practitioner	Discussion Forum Post	False	False
S35	Refactoring: Code Smells	https://bit.ly/3bC44jw	Practitioner	Slides	False	False
S36	Identifying Code Smells In Java	https://bit.ly/2VQQAtE	Practitioner	Practitioner Audience Article	True	True
S37	How to identify a Data Class using NDepend	https://bit.ly/2S5q6n2	Practitioner	Tool Documentation	True	True
S38	Sharpen your sense of code smell	https://bit.ly/3cNOotY	Practitioner	Practitioner Audience Article	False	False
S39	How to find the code smell AND do not let it go bad, part 2	https://bit.ly/2VWobT6	Practitioner	Practitioner Audience Article	True	False
S40	Everything you need to know about Code Smells	https://bit.ly/2znK9Xj	Practitioner	Practitioner Audience Article	False	False
S41	How to identify a Data Class using NDepend	https://bit.ly/2Ygi26W	Practitioner	Tool Documentation	True	True
S42	thoughts on feature envy	https://bit.ly/3aGIg5a	Practitioner	Practitioner Audience Article	False	False
S43	Feature envy	https://bit.ly/2YfTA5D	Practitioner	Practitioner Audience Article	True	True
S44	Inappropriate Intimacy	https://bit.ly/3f1i3Bv	Practitioner	Practitioner Audience Article	False	False
S45	Feature Envy	https://bit.ly/2KItgJB	Practitioner	Tool Documentation	True	False
S46	Exploring Smelly Code	https://bit.ly/2YigCJ8	Practitioner	Practitioner Audience Article	True	True
S47	Code Smell: Feature Envy	https://bit.ly/35k6UY7	Practitioner	Practitioner Audience Article	True	True
S48	Inappropriate Intimacy	https://bit.ly/3d16DMe	Practitioner	Practitioner Audience Article	False	False

Table 3.1: Overview of studied knowledge sources

Lines from the knowledge source (Example from S5):

Sometimes classes become far too intimate and spend too much time delving into each others' private parts. We may not be prudes when it comes to people, but we think our classes should follow strict, puritan rules. Over-intimate classes need to be broken up as lovers were in ancient days. [F 85]

Additions during Open Coding:

Definition for Inappropriate Intimacy that implies that two or more classes depend on each other's private features excessively.

Additions during Axial Coding:

Added new definition class:

```
ii_def_classes_depend_on_each_others_private_features_excessively = \
CClass(definition, "Two or more classes depend on each other's private features
    excessively",
values={
    "origins": "classes",
    "targets": "private features of classes"
    })
```

Linked S5 to `inappropriate_intimacy` and `ii_def_classes_depend_on_each_others_private_features_excessively` codes:

```
add_links({S5: [..., inappropriate_intimacy,
    ii_def_classes_depend_on_each_others_private_features_excessively,
...]},role_name="contained_code")
```

Changes and Additions during Selective Coding:

This definition can be seen as special case of the two other definitions `ii_def_classes_depend_on_each_other_excessively`, `ii_def_class_uses_other_class_private_features_excessively`. Added them as superclasses to the definition class:

```
ii_def_classes_depend_on_each_others_private_features_excessively = \
CClass(definition, "Two or more classes depend on each other's private features
    excessively",
superclasses=[ii_def_classes_depend_on_each_other_excessively,
ii_def_class_uses_other_class_private_features_excessively], ...)
```

To illustrate the coding process, above we show an excerpt of the coding for the knowledge source *S5* as an example, which just covers some lines on a definition of *Inappropriate Intimacy*. Like this, all relevant lines in the knowledge source are coded. The open coding provides our initial interpretation of the source lines. The axial coding then provides the details on how we have categorized and formalized this knowledge, first in text and then as code. The code excerpts shown here provides the traceability to our Python source code. Finally, selective coding was applied later to find errors and perform improvements: Here, we improved the categorization by modeling the superclass relations to similar definitions we have found in other classes.

3.4 Grounded Theory on Coupling Code Smells

Meta-model

The meta-model of our found theory defines the classification (or categories) of model elements, as well as their relations, necessary to model the design space of coupling code

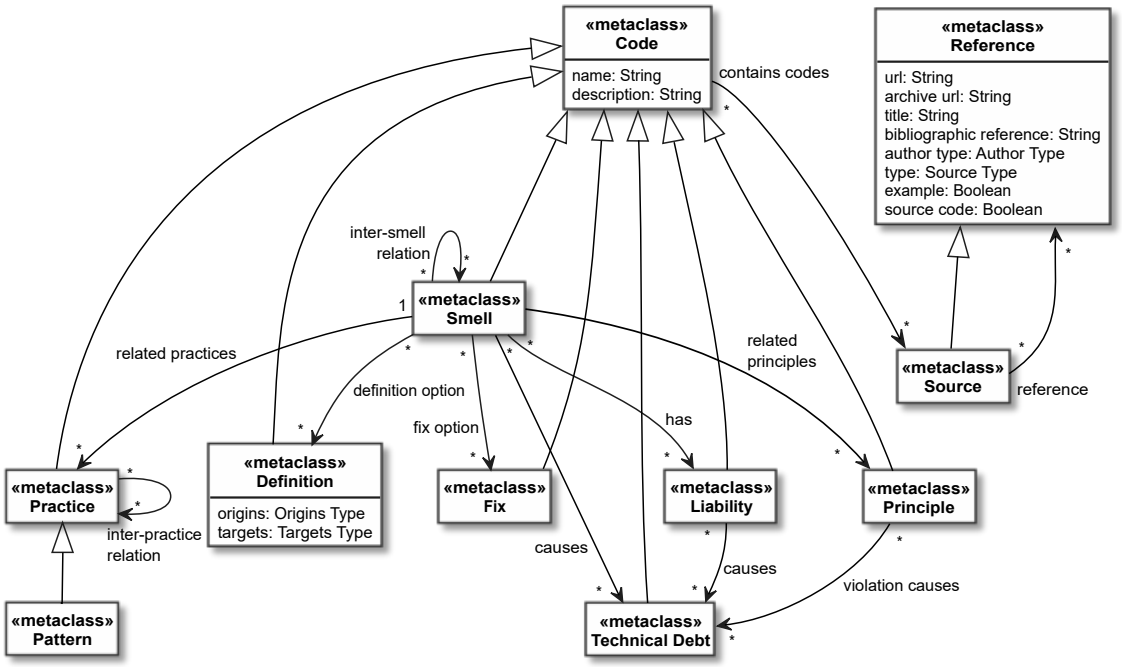

Figure 3.2: Meta-model main meta-classes

smells fully. Of course, many alternative ways to model that design space are possible, such as different naming of model elements or using more meta-classes instead of stereotypes. Nonetheless, the meta-model reveals interesting insights of which conceptual elements that are present in practitioner discussions of coupling code smells.

We first present the basic meta-classes derived in the GT study in Figure 3.2[4]. The basis of our study are *Sources* which are texts from which we extracted the GT *Codes*. *Sources* are specific kinds of *References* used as knowledge sources in the study; other references are used to model links to study-external sources (such as a book or scientific paper cited by the practitioners). We describe each *Reference* using a couple of meta-data attributes: the title, a URL, an archive URL linking to the Web archive version of the source (to enable reproducibility of our results), and an optional additional bibliographic reference is used to identify the source. The author type is an enumeration with possible values *Practitioner*, *Academic*, *Mixed*, and *Unknown*; in our study we only included sources with *Practitioner* and *Mixed* author types. The type of the source describes which kind of grey literature is used. In our study we included *Discussion Forum Post*, *General Audience Article*, *Practitioner Audience Article*, *Tool Documentation*, *Blog Post*, and *Slides* as source types. Further, we indicate whether an example was provided, and whether the example, if present, includes source code. Please note that the properties explained in this paragraph are reported as columns in Table 3.1 for our included knowledge sources.

Each *Code* is described with a name and an optional description. The core code is the *Smell*, which can be linked to other smells. The smell has a *Definition*, which for coupling smells is in detail specified with the origin and target types in the coupling relation.

[4]Please note that the UML figures used in this chapter are directly generated from our coded models in our tool using PlantUML. The figures are only touched to optimize the layout for the chapter slightly.

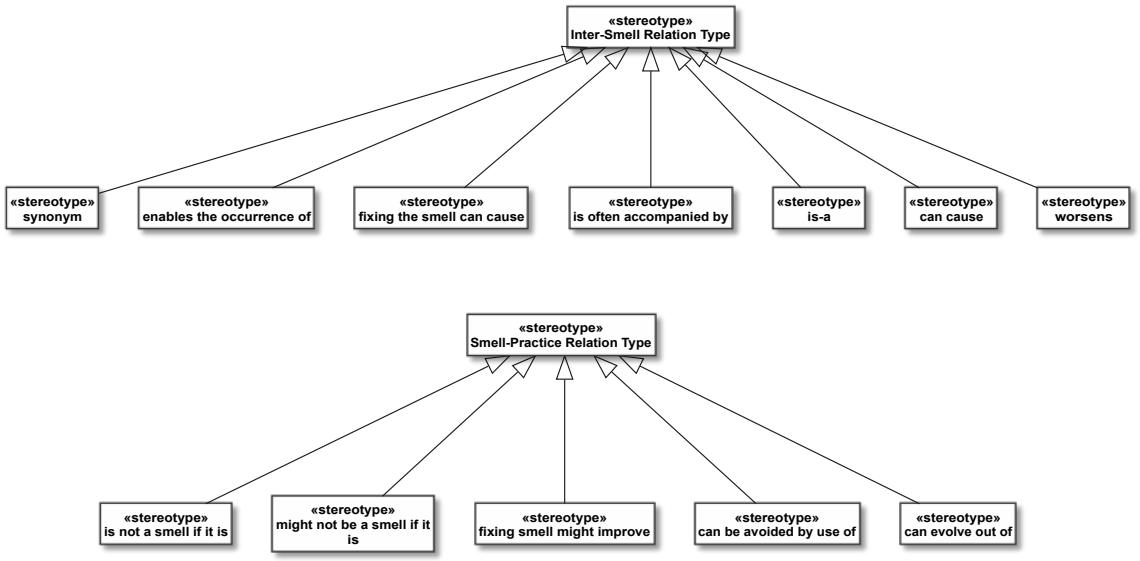

Figure 3.3: Meta-model stereotypes (extending inter-smell relation and inter-practice relation in Figure 3.2)

These are two additional enumerations. In our study we have found evidence for *method*, *class*, *classes*, or *code fragment* as origins of the coupling, and *a class's methods*, *a class's public features*, *a class's features*, *a class's private features*, *a class's implementation details*, *classes*, *a class's data*, *methods of classes*, *private features of classes*, and *clients* are possible targets of the coupling. Moreover, a smell can have *Fixes*. It has a number of *Liabilities* associated which make it a bad smell in code or design that should be resolved. Similar are violations to *Principles* the smell can cause. The *Smell*, as well as its *Liabilities* and its *Principle* violations can cause *Technical Debt*.

The *Association* meta-class of the inter-smell relationship is in our model extended by the stereotype *Inter-Smell Relation Type*. We have found evidence for the kinds of inter-smell relations depicted at the top of Figure 3.3. *Smells* can be associated to certain (best or common) *Practices* in our field. Many such best practices are described in the literature as *Patterns*. We have found evidence for the kinds of smell-practice relations depicted at the bottom of Figure 3.3.

Coupling smells and their relationships

Figure 3.4 gives an overview of the coupling smells and their relations. As can be seen we identified *Feature Envy*, *Inappropriate Intimacy*, *Data Class*, *Indecent Exposure*, *Message Chain*, and *Middle Man* as immediate sub-classes (is-a relation) of coupling smell. Those major coupling smells will each be discussed in their own subsection below. *Data Envy* is a special kind of *Feature Envy* and will be discussed together with it below. As can be seen some smells that are not coupling smells have also been found in our study. Those are only included in our study if there was clear evidence that a coupling smell has some kind of strong relation to the other smell. We only referenced those other smells, but did not study them in detail. The coupling smells and their inter-smell relationships are discussed

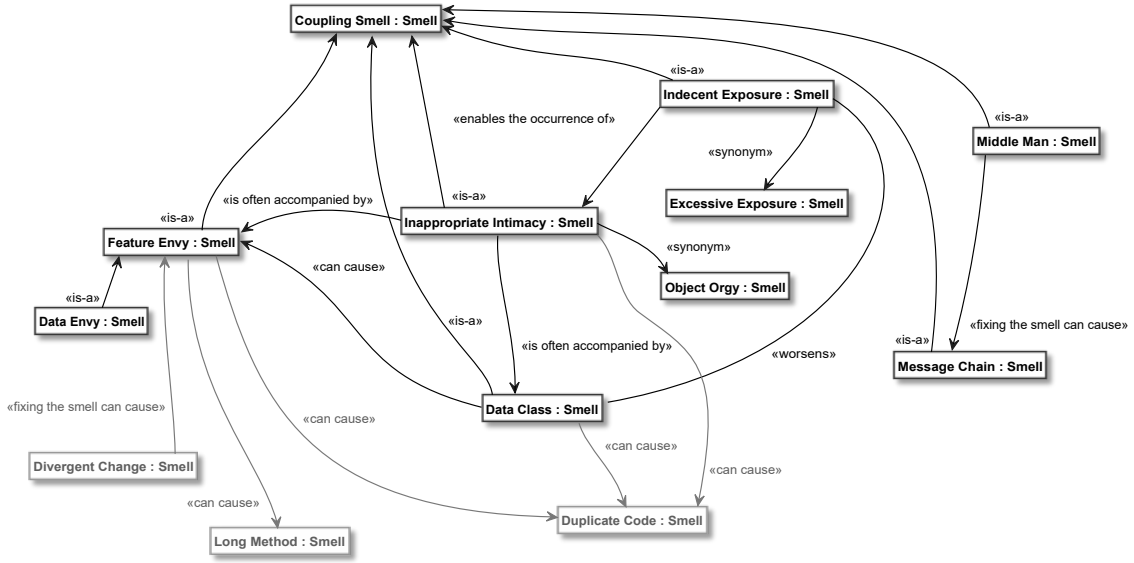

Figure 3.4: Coupling smells and their relations (non-coupling smells are rendered in grey)

separately per smell in the following sections.

Table 3.2 presents all smells and color coding[5] in the second row indicates how often the smell appears in our total of 48 sources. All relations are listed twice for their source and target smell, and the different color coding in the fourth row indicates how often the relation appears in the total number of sources of the smell (from the second row).

Feature Envy

Definitions

Feature Envy is a very commonly discussed coupling smell that is defined in different sources differently. With 33 evidences for the smell it occurs most frequently in our sources. With a total of 9 different definition variants shown in Table 3.3, it also has the largest number of possible interpretations. There are two common relations between coupling origin (i.e. the place where *Feature Envy* can be spotted) and the coupling targets (i.e. the places the origin is envious of): 1. the origin makes excessive use of the target; 2. the origin uses specific target features more often than its own. That is, we can classify Feature Envy definitions in the table into two main variants: Definition Variant 1 (covering the 3 Feature Envy definitions in Table 3.3 containing the word "excessively") is the broader definition, whereas Variant 2 (covering the definitions not containing the word "excessively") is often used in sources (or derived from other original source) where automated detection or similar operationalization is a goal or at least considered to be a goal. That is, "excessive use" is less specific and thus hard to judge automatically, whereas "more features of a specific kind" can be easily counted and compared. Note that Variant 2 is problematic in the sense that it is easy to construct examples which a human reviewer usually would not accept as *Feature Envy*, but which would conform to the definition.

[5] In the tables we use the following color coding to visualize the frequency of the evidences: $< 5\%$, $< 10\%$, $< 20\%$, $< 35\%$, $< 50\%$, $< 70\%$, $>= 70\%$

Smell	All Smell Evidences	Relations	Relation Evidences
Feature Envy	33	Data Envy *is-a* Feature Envy	S9, S18
		Feature Envy *can cause* Duplicate Code	S4, S10, S42, S46
		Inappropriate Intimacy *is often accompanied by* Feature Envy	S15
		Feature Envy *can cause* Long Method	S17
		Data Class *can cause* Feature Envy	S7, S17, S28, S39
		Divergent Change *fixing the smell can cause* Feature Envy	S14, S16
Inappropriate Intimacy	21	Inappropriate Intimacy *synonym* Object Orgy	S2, S12, S13
		Indecent Exposure *enables the occurrence of* Inappropriate Intimacy	S3, S5, S15
		Inappropriate Intimacy *is often accompanied by* Data Class	S12
		Inappropriate Intimacy *can cause* Duplicate Code	S15
		Inappropriate Intimacy *is often accompanied by* Feature Envy	S15
Data Class	16	Inappropriate Intimacy *is often accompanied by* Data Class	S12
		Data Class *can cause* Duplicate Code	S4, S36
		Indecent Exposure *worsens* Data Class	S8
		Data Class *can cause* Feature Envy	S7, S17, S28, S39
Indecent Exposure	6	Indecent Exposure *enables the occurrence of* Inappropriate Intimacy	S3, S5, S15
		Indecent Exposure *worsens* Data Class	S8
		Indecent Exposure *synonym* Excessive Exposure	S15
Message Chain	12	Message Chain *fixing the smell can cause* Middle Man	S4, S39
Middle Man	13	Message Chain *fixing the smell can cause* Middle Man	S4, S39
Data Envy	3	Data Envy *is-a* Feature Envy	S9, S18
Object Orgy	3	Inappropriate Intimacy *synonym* Object Orgy	S2, S12, S13
Excessive Exposure	1	Indecent Exposure *synonym* Excessive Exposure	S15
Duplicate Code	4	Feature Envy *can cause* Duplicate Code	S4, S10, S42, S46
		Data Class *can cause* Duplicate Code	S4, S36
		Inappropriate Intimacy *can cause* Duplicate Code	S15
Long Method	1	Feature Envy *can cause* Long Method	S17
Divergent Change	2	Divergent Change *fixing the smell can cause* Feature Envy	S14, S16

Table 3.2: Overview of coupling smells, their relations, and evidences

The sources are also split among different options where *Feature Envy* occurs (i.e., the origin): Most see a single method (or operation in non-object-oriented languages) as the typical *Feature Envy* origin, but some see a whole class or a code fragment (e.g. in a method) as places to spot *Feature Envy*. The class interpretation can be difficult to judge, as some methods might be envious while many others are not. Also, it is easy to create long methods with code fragments in it that are envious, where the rest of the long method is not. Thus the code fragment interpretation sounds appealing in that sense, but would also make judgments whether it is *Feature Envy* or not more difficult. Note that an envious part in a longer method can also be spotted and then the *Extract Method* refactoring is suggested as a solution to first isolate (and later improve) the envious parts.

For the coupling target, (i.e. the places an origin is envious of), the sources agree that it is a class, but they see envy occur with different kinds of features, namely all the class's features, its public features, its data, or its methods. Methods and all features are the most frequent options. Note that if all data is accessed via method (i.e., no *Indecent Exposure* of data occurs), the variety in targets is rather limited.

The following code shows a simple *Feature Envy* example[6]. *getTotalPrice()* is a method

[6]Note that we give a few simplistic example to illustrate some of the smells. While they conform to the smell definitions, they alone might not be seen as harmful by many practitioners. According to our results smells become harmful, when their "intensity" is high enough.

envious of the features in *Item* according to both definitions, as it uses 4 methods of *Item* and 0 methods of*Basket*.

```
class Item { .. }
class Basket {
  // ..
  float getTotalPrice(Item i) {
    float price = i.getPrice() + i.getTax();
    if (i.isOnSale())
    price = price - i.getSaleDiscount() * price;
    return price;
  }
}
```

The simple example can also be used to explain the issues with the Variant 2 definition ("more features of a specific kind") and code fragment vs. method as origin easily. Consider we add a statement that uses 4 *Basket* features to the method: Then the code fragment in the first three lines is just as envious as before, but now it is not *Feature Envy* according to the Variant 2 definition anymore (assuming *method* as origin).

```
  float getTotalPrice(Item i) {
    float price = i.getPrice() + i.getTax();
    if (i.isOnSale())
    price = price - i.getSaleDiscount() * price;
    if (basketContainsGroupDiscountItems() || isLargeBasketDiscount() ||
        isPriceDiscount())
    price = price - basketDiscounts();
    return price;
  }
}
```

The discussion in this section might suggest that there are quite different views on *Feature Envy* in the practitioner literature, but when reflecting on the practitioner texts more deeply, this is not really the case. In cases where the practitioners do not just take over a definition from another source, but discuss it in more detail, it becomes clear that even if they use Definition Variant 2, they rather interpret it loosely. In contrast, a strict interpretation has the mostly *excessive use of another class's features from one or more places (e.g. one or more methods)* is meant. The other definitions are mostly narrowing this definition down for easier explanation or operationalization e.g. in a tool. Also note that this discussion should show that there is a great risk of producing a lot of false positives or negatives in automated tools. , when any of the more narrow definition options is chosen.

Relationships to other smells

As shown in Figure 3.4, sometimes *Feature Envy* is discussed under the term *Data Envy*, which thus has an is-a relation to *Feature Envy*. It occurred relatively seldom (3 sources) and

was defined using the same two definitions types as observed for *Feature Envy* ("excessive use" and "more features of a specific kind").

Feature Envy can be caused by *Data Classes*. That is, other classes might be envious to the data access features offered by the *Data Class*. It is often accompanied by *Inappropriate Intimacy*, e.g. if the envied class also has an intimate relation to the envious class or if the envious class uses implementation details of the envied class.

Patterns such as *Strategy* and *Visitor* are used to fix the *Divergent Change* smell, which refers to making unrelated changes in the same location. Fixing *Divergent Change* with means other than such established patterns might thus accidentally lead to *Feature Envy* , or implementations of those patterns might be identified as *Feature Envy* (which they are not). *Feature Envy* means many accesses to another class, e.g., in one method, which might lead to a *Long Method*. As often the same envious code is used in multiple places, *Feature Envy* can cause *Duplicate Code*.

Liabilities and principle violations

Practitioners often list either liabilities of a smell or principle violations to show why one should care about the smell (summarized in Table 3.4). Sometimes liabilities listed here are derived from positive aspects occurring when the smell is fixed. , which is the opposite of the liability. For *Feature Envy* a large fraction of the practitioners see *high coupling* (and its companion *low cohesion*) and *low code comprehensibility* as typical liabilities. Many practitioners see also various *maintainability issues* as liabilities such as *reduced changeability*, *missing encapsulation*, *misplaced responsibilities*, *reduced testability*, *reduced reusability*, and *high complexity caused by code duplication*. (which is also related to the *Duplicate Code* smell).

Feature Envy can lead to violations of the *Single Responsibility Principle* when a method or class takes over responsibilities of another class (or its features) it is envious of. *Feature Envy* can lead to a violation of the *Tell Don't Ask Principle* which means that objects should not ask for details or internals, such as envied data features, but rather tell other objects what to do. *Feature Envy* can lead to a violation of that principle, but this is only mentioned in a single source.

Note that the liabilities have dependencies, too, such as one liability can cause another one. We report the liabilities in this chapter in the way we found them in the sources, without considering such dependencies, as those dependencies are not discussed in our grey literature sources.

Related practices and patterns

As shown in Table 3.5, *Feature Envy* has numerous relations to various common practices and patterns. Many of them are additional reasons why (automated) detection and repair of *Feature Envy* can be very hard, as they describe situations that look like *Feature Envy* but are actually places in the code where *behavior is kept separate from its data for a clear*

Definition	Coupling origin(s)	Coupling target(s)	Definition Evidences
Feature Envy			
Method uses class excessively	a method	a class's methods	S1, S3, S5, S14, S15, S22, S26, S31, S33, S47
Method uses features of another class more often than its own features	a method	a class's features	S6, S7, S9, S10, S11, S14, S16, S17, S21, S27, S35, S36, S40, S41, S43, S45, S46
Class uses features of another class more often than its own features	a class	a class's features	S31
Method uses public features of another class more often than its own features	a method	a class's public features	S1
Class uses class excessively	a class	a class's methods	S2, S14, S18, S31
Method uses more data of other class more than its own class's data	a method	a class's data	S4, S29, S31, S38, S39
Class uses more data of other class more than its own data	a class	a class's data	S20
Class uses data of another class excessively	a class	a class's data	S28
Code fragment uses features of another class more often than its own features	a code fragment	a class's features	S30
Inappropriate Intimacy			
Class uses other class's private features excessively	a class	a class's private features	S1, S24, S35
Class uses other class's implementation details	a class	a class's implementation details	S1, S2, S4, S12, S13, S22, S25, S26, S38, S40
Two or more classes depend on each other excessively	classes	classes	S3, S15, S33, S36
Two or more classes depend on each other's private features excessively	classes	private features of classes	S5, S14
Method uses other class's implementation details	a method	a class's implementation details	S38, S44, S48
Data Class			
Class only has data, getters, and setters	a class	classes	S3, S4, S5, S7, S8, S12, S14, S16, S17, S32, S33, S35, S36, S37
Indecent Exposure			
Class exposes internal details	a class	clients	S3, S5, S15, S33, S38
Message Chain			
Long sequence of method calls	a class	methods of classes	S3, S4, S5, S14, S15, S16, S22, S33, S35, S36, S39, S40
Middle Man			
Class only delegates to other classes	a class	classes	S3, S4, S5, S15, S16, S22, S23, S39
Class is merely a wrapper over the other class	a class	classes	S3, S33
Class delegates the majority of the work to other classes	a class	classes	S14, S33, S35, S40
Method is merely a wrapper hiding delegates	a method	classes	S19, S38

Table 3.3: Definitions of coupling smells and evidences. Color coding indicates how many evidences of a smell containing a definition contain the specific definition.

purpose. Examples of this are the *Visitor* [46] and *Strategy* [46] patterns, and practices of *self delegation*. Delegation classes or *Wrappers* such as in the *Adapter* [46], *Decorator* [46], or *Facade* [46] patterns are also places in the code where *Feature Envy* can be wrongly identified. Some sources mention *utility/helper functions or classes* as code that is similar to *Feature Envy* but is not *Feature Envy*. Note that some see *utility/helper functions or classes* as a code smell, too, but a different one than *Feature Envy*; definitively it is less of a coupling smell. While none of these relations is mentioned by many sources, overall a lot of sources identify common coding practices and patterns that can be identical in appearance to *Feature Envy* but are actually best practices. The relation of *Feature Envy* to inheritance

Smell	Liability/Principle Violation	Liability/Principle Violation Evidences
Feature Envy Number of evidences discussing liabilities/principles option: 26	High coupling	S1, S6, S7, S15, S18, S20, S26, S28, S30, S31, S43, S47
	Low cohesion	S6, S7, S26, S30, S43
	Low code comprehensibility	S4, S6, S11, S26, S29, S30, S31, S43, S47
	High complexity caused by code duplication	S4, S10, S43
	Reduced changeability	S6, S15, S20, S21, S26, S30, S31, S46
	Missing encapsulation	S10, S17, S18, S26, S27, S28, S29, S31
	Reduced testability	S10, S11, S28
	Reduced reusability	S11, S21, S26, S46
	Misplaced responsibility	S21, S27, S30, S31, S39, S43, S46
	Maintainability issues	S31, S38, S40
	Can cause violation of Single Responsibility Principle	S7, S15, S28, S29, S31
	Can cause violation of Tell Don't Ask Principle	S28
Inappropriate Intimacy Number of evidences discussing liabilities/principles option: 20	Relying on internal details can lead to defects	S1
	Maintainability issues	S2, S25, S31, S36, S38, S40
	High complexity	S2, S13
	Low code comprehensibility	S4, S13, S25, S26, S31, S36
	Hinders code reuse	S4
	Missing encapsulation	S2, S12, S13, S24, S26, S31
	Reduced changeability	S15, S24, S26
	High complexity caused by code duplication	S15
	High coupling	S15, S24, S25, S26, S31, S36
	Low cohesion	S24
	Reduced reusability	S26
	Can cause violation of Single Responsibility Principle	S15, S25, S26
	Can cause violation of Law of Demeter: Only talk to your immediate friends	S26, S44
Data Class Number of evidences discussing liabilities/principles option: 13	Low code comprehensibility	S4, S34
	High complexity caused by code duplication	S4
	High coupling	S8
	Low cohesion	S8
	Misplaced responsibility	S16, S32
	Reduced changeability	S34
	Can cause violation of Single Responsibility Principle	S7, S32, S34
	Can cause violation of Tell Don't Ask Principle	S7, S32
Indecent Exposure Number of evidences discussing liabilities/principles option: 11	High complexity	S5
	Low code comprehensibility	S5
	High coupling	S15
	Reduced changeability	S15
	Maintainability issues	S38
	Can cause violation of Single Responsibility Principle	S15
Message Chain Number of evidences discussing liabilities/principles option: 15	Low code comprehensibility	S4, S15
	High coupling	S4, S14, S15, S39
	Reduced changeability	S15, S36
	Maintainability issues	S40
	Can cause violation of Single Responsibility Principle	S15
	Can cause violation of Law of Demeter: Only talk to your immediate friends	S36, S39
Middle Man Number of evidences discussing liabilities/principles option: 13	Low code comprehensibility	S4, S15, S23
	High complexity	S15
	High coupling	S15, S23
	Reduced changeability	S15
	Missing encapsulation	S23
	Maintainability issues	S38, S40
	Can cause violation of Single Responsibility Principle	S15

Table 3.4: Liabilities/principle violations suggested per smell. Color coding indicates how many evidences of a smell containing liability/principle violations descriptions contain a specific liability/principle violations.

Smell	Relation	Practice/Pattern	Smell-Practice Relation Evidences
Feature Envy	is not a smell if it is	Behavior is kept separate from its data for a clear purpose (sub practices/patterns: Visitor, Strategy, Self Delegation)	S4, S14, S16, S27
	is not a smell if it is	Visitor	S4, S14, S16, S27
	is not a smell if it is	Strategy	S4, S14, S16, S27
	is not a smell if it is	Self Delegation	S16
	might not be a smell if it is	Use of superclass features	S9
	is not a smell if it is	Wrapper	S9
	is not a smell if it is	Adapter	S42, S46
	is not a smell if it is	Decorator	S46
	is not a smell if it is	Facade	S42, S47
	might not be a smell if it is	Utility/Helper Function or Class	S6, S42
	can be avoided by use of	Template Method	S27
	fixing smell might improve	Domain Model	S10, S11, S17, S20, S21, S28
	fixing smell might improve	Entity	S10, S20
	fixing smell might improve	Service	S20
	fixing smell might improve	Extension Methods	S20
	fixing smell might improve	Value Object	S10, S11
	fixing smell might improve	Ubiquitous Language	S11
Inappropriate Intimacy	can be avoided by use of	Chain of Responsibility	S13
	might not be a smell if it is	Use of superclass features	S14
Data Class	is not a smell if it is	Data Transfer Object	S7, S8, S32, S34
	is not a smell if it is	Parameter Object	S7
	is not a smell if it is	Data Access Object	S32
	is not a smell if it is	Immutable Data Object	S16, S32
	fixing smell might improve	Domain Model	S17, S34
	fixing smell might improve	Service	S17
Middle Man	is not a smell if it is	Delegation class with clear purpose (sub practices/patterns: Adapter, Wrapper, Proxy, Facade, Decorator)	S3, S15, S22
	is not a smell if it is	Facade	S14, S15, S22
	is not a smell if it is	Wrapper	S3
	is not a smell if it is	Proxy	S15, S39
	is not a smell if it is	Adapter	S3, S15
	can evolve out of	Mediator	S14

Table 3.5: Practice and pattern relations suggested per smell. Color coding indicates how many evidences of a smell contain a specific practice or pattern.

is seen controversially; one source mentions that the *use of superclass features* might lead to a wrong identification of *Feature Envy*.

Quite a number of sources relate *Feature Envy* to Domain-Driven Design (DDD) [34]. Most of them mention the general *Domain Model* pattern [43] but some are mentioning more specific DDD patterns such as *Entity* [34], *Service* [34], *Value Object* [34], and *Ubiquitous Language* [34] that make up or represent the *Domain Model*. For DDD to have a positive effect on the code and throughout a software's evolution, it must be represented well in the code. *Feature Envy* can lead to the *Anemic Domain Model* anti-pattern[7], i.e., a *Domain Model* which does not combine data and process together in its realization. Thus fixing *Feature Envy* can greatly improve the realization of those patterns in the code.

In one source it is mentioned that a similar positive effect can also be achieved for the *Extension Methods* practice (i.e. methods added to existing types), and another one explains how the *Template Method* pattern [46] can be used to avoid *Feature Envy*.

[7]See e.g. `https://martinfowler.com/bliki/AnemicDomainModel.html`.

Fix Options

As shown in Table 3.6 overall 28 sources discuss or list possible fixes for *Feature Envy*. Most of those are well-known refactorings. The most often suggested fix is the *move method* refactoring. In cases like the second example above before considering to move a method, the envious part of a method might have to be extracted with the *extract method* refactoring, which is also often suggested. Sometimes it makes sense to extract multiple parts at once and apply the *extract class* refactoring. E.g. when moving a method, *move field* might be needed as well.

It is a bit unclear from the rather generic definitions of the smell if *Feature Envy* can occur within an inheritance hierarchy. If this is considered, either *add higher-level or more abstract feature* or *subclass and extend* can be considered as fix options. Other options mentioned only by one source are *hide delegate*, *bundle multiple methods into a single method*, *hide internal details*, or to first *use OO metrics to detect smell* before applying other fix options.

The variety of fix options presented and the fact that *extract method* and other changes to a class might be needed before *move method* can be applied underline the points from above: While it can be easy for humans to spot *Feature Envy* in a method, automation of detection and fixing is hard. Just consider an example like the second *getTotalPrice()* example above, which is a *Long Method* with alternating and intertwined code fragments from the own class and multiple other classes the method is envious of. Refactoring in this situation requires multiple steps combining many of the fix options listed here in a creative way. Automatically detecting such situations is also hard, and if achieved, still might be impractical, as many false positives and negatives might lead to large manual efforts for sorting them out (this issue is also the case for the special OO metrics suggested only by one practitioner source).

Inappropriate Intimacy

Definitions

Inappropriate Intimacy has the second highest number of evidences (21 sources) with five different definition variants in Table 3.3. Overall, it mainly focuses on improper information hiding. Almost a half of the sources define *Inappropriate Intimacy* as a class using another class's implementation details, some interpret implementation details only as private features, and some others view only a single method as the coupling origin. Another type of definition sees *Inappropriate Intimacy* as two or more classes excessively depending on each other. Two of those sources again limit the definition to private features.

The following example briefly outlines how a class can make use of another class's implementation details. Here the method *getAccomodationType()* of the class *Tenant* uses the internal implementation detail of the class *Accomodation*. If *Accomodation* would make similar use of *Tenant's* features, the more narrow "two or more classes excessively depending on each other" kind of definition would be fulfilled, too. If used in a very few places, as in

Smell	Fix Option	Fix Option Evidences
Feature Envy Number of evidences discussing fixes option: 28	Move method	S1, S3, S4, S5, S9, S10, S11, S14, S15, S16, S17, S18, S20, S22, S27, S28, S29, S31, S35, S39, S42, S43, S45, S46, S47
	Add higher-level or more abstract feature	S1, S21, S27
	Extract method	S4, S5, S6, S9, S10, S11, S14, S16, S27, S28, S29, S31, S35, S39, S43, S45, S46, S47
	Extract class	S9, S14
	Move field	S5, S31, S35
	Hide delegate	S11
	Bundle multiple methods into a single method	S15
	Hide internal details	S20
	Use OO metrics to detect smell	S41
	Subclass and extend	S10, S39
Inappropriate Intimacy Number of evidences discussing fixes option: 14	Change target class to only offer public features if possible	S1
	Change calling class to only use public features if possible	S1
	Move method	S4, S5, S14, S22, S24, S25, S26, S35, S44, S48
	Move field	S4, S5, S14, S24, S25, S35, S44, S48
	Extract class	S4, S5, S24, S25, S44, S48
	Hide delegate	S4, S5, S24, S35, S48
	Replace delegation with inheritance	S4, S5, S24, S35, S44, S48
	Change Bidirectional Association to Unidirectional Association	S4, S5, S24, S25, S35, S48
	Use reflection with caution	S13
	Encapsulate field	S13, S15, S26
	Encapsulate collection	S13, S26
	Introduce extra class between intimate classes	S15
	Extract method	S22, S25, S26
	Subclass and extend	S10
Data Class Number of evidences discussing fixes option: 11	Move method	S4, S5, S7, S8, S14, S16, S17, S35
	Encapsulate field	S4, S5, S8, S14, S16, S35
	Encapsulate collection	S4, S5, S14, S16, S35
	Extract method	S4, S7, S14, S16, S17
	Move data-using methods to data class	S4, S8, S17, S32
	Remove getters and setters from data class and make fields private	S14, S16
	Make immutable data object	S32
	Use OO metrics to detect smell	S37
	Subclass and extend	S10
Indecent Exposure Number of evidences discussing fixes option: 1	Encapsulate field	S15
	Encapsulate collection	S15
	Hide behind method	S15
	Hide behind abstract class or interface	S15
Message Chain Number of evidences discussing fixes option: 9	Hide delegate	S4, S5, S14, S16, S22, S35, S39
	Extract method	S4, S5, S14, S16, S39
	Move method	S4, S5, S14, S16
	Ignore, if refactoring leads to middle man	S4
	Hide behind method	S15
	Ignore, if it is a small chain which causes little coupling (is harmless)	S14
	Subclass and extend	S10
Middle Man Number of evidences discussing fixes option: 12	Remove middle man	S4, S5, S14, S15, S16, S22, S23, S33, S35, S39
	Ignore, if middle man is used to reduce inter-class dependencies	S4
	Inline method	S5, S16, S35
	Replace delegation with inheritance	S5, S35
	Remove middle man method	S19
	Subclass and extend	S10

Table 3.6: Fix options suggested per smell. Color coding indicates how many evidences of a smell containing fix option suggestions contain a specific fix option.

this example, this code is likely not very harmful for the software qualities outlined below, but if this kind of class interaction, just inelegant code. If such intimacy is used heavily in many places of the code, many software qualities can be harmed.

```
class Tenant {
  private String name;
  private Accomodation accomodation;
  public String tenantAccomodation = accomodation.type;
  // ..
  public String getAccomodationType(String newtype){
    accomodation.setType(newtype);
    if (tenantAccomodation.equals(accomodation.getType()))
    return tenantAccomodation;
    else
    return accomodation.getType();
  }
}
```

Relationships to other smells

As shown in Figure 3.4, *Object Orgy* is a synonym of *Inappropriate Intimacy* often used in the Perl community. We have confirmed this by studying a number of sources on *Object Orgy*. The *Object Orgy* perspective could help to explain the examples of *Inappropriate Intimacy*. It is another helping key in coupling smell classification because it involved the directed use of attributes across the classes. *Indecent Exposure* is very similar to *Inappropriate Intimacy* and fosters its occurrence. Sometimes, an object initiation requires many implementation details. When it used other implementation details, it is easily trigger the *Inappropriate Intimacy* symptom. A couple of possible relations are mentioned only by one source: *Inappropriate Intimacy* can cause the *Duplicate Code* smell. Only one source identify this relationship, but it is nice to pick it up because our example above also shows how duplicated code bring the unnecessary complexity to the source code. *Inappropriate Intimacy* might be accompanied by *Feature Envy* as many intimate relations, as in our example above, might lead to classes using the other class excessively. Such intimate or excessive use can often be observed with *Data Classes*, which also can accompany *Inappropriate Intimacy*.

The feature excessively usage overlaps with the definition of *Feature Envy*. With this reason, many practitioners get confuse to identify whether it is *Feature Envy* or *Inappropriate Intimacy*. Some practitioners do not differentiate these two smells al all. When the internal implementation details from one class is used by another class excessively, this symptom could refer to both smells. For the *Data class*, it is more about information hiding through the encapsulation concept. In some cases, when the *Inappropriate Intimacy* is resolved, *Data class* is resolved as well. the *Inappropriate Intimacy* often points out the internal features from *Data class* and use them inappropriately in its class.

Liabilities and principle violations

As shown in Table 3.4, motivations for avoiding *Inappropriate Intimacy* are very often *maintainability issues* in general or more specific ones such as *low code comprehensibility*, *missing encapsulation*, *high coupling*, *high complexity* (maybe caused by code duplication), *reduced changeability*, *low cohesion*, and *reuse issues*.

Inappropriate Intimacy often violates the *Single Responsibility Principle* when responsibilities of another class are not delegated to the other class, but performed locally. It also violates the *Law of Demeter* ("only talk with your immediately friend") if classes make excessive use of many classes' implementation details and thus have more knowledge about other classes than necessary.

Related practices and patterns

As shown in Table 3.5, *Inappropriate Intimacy* has only a few typical links to practices and patterns. One source brings up the *Chain of Responsibility* pattern [46]. Its multiple handlers in a chain with clear interfaces can help to avoid coupling between the classes. Another source sees the use of superclass features as a harmless symptom only looking like *Inappropriate Intimacy*.

Fix Options

Inappropriate Intimacy has the widest range of fix options with 14 fix options from 14 sources as shown in Table 3.6. Strongly recommended fixes are *move method*, *move field*, *extract class*, *hide delegate*, *replace delegation with inheritance*, and *change bidirectional association to unidirectional association*. It is interesting to see that many of those also appear for *Feature Envy*. Again, the practitioner recommendation often describes fixing as a process in which many of those are applied to resolve a complex design situation. A couple of sources mention specific fixes for *Inappropriate Intimacy*, such as recommendation of increasing *encapsulation*, focus on *public features only*, *cautious use of reflection*, or *introducing an extra class between intimate classes*.

Data Class

Definitions

For *Data Class* all 16 sources covering the smell conform to the same kind of definition (as shown in Table 3.3): It is a class that only offers data. It might have methods but these are only getters and setters for the data. The following example shows such a pure *Data Class*:

```
public class Student {
  private int id;
  private String name;

  public Student(int id){
    this.id = id;
```

```
  }
  public void setId(int id){
    this.id = id;
  }
  public void setName(int name){
    this.name = name
  }
  public int getId(){
    return id;
  }
  public String getName(){
    return name;
  }
}
```

Relationships to other smells

As shown in Figure 3.4, sometimes *Data Class* is accompanied by *Inappropriate Intimacy*, for instance, if other classes use a *Data Class's* private features or implementation details excessively. *Data Classes* can cause *Feature Envy* as classes using *Data Classes* might be envious to the data access features. If *Data Classes* are accessed via private or other internal features, they also suffer from *Indecent Exposure*, which can worsen the coupling of the *Data Class*. Using *Data Classes* entails the danger of causing *Duplicate Code*, if the functionality related to the data in the *Data Classes* is handled in several places across the code base.

Liabilities and principle violations

As shown in Table 3.4, *Data Classes* are harmful because they *lower the code comprehensibility*. They are seen as a *misplaced responsibility* if code manipulating the data is located elsewhere than the data. This is also the reason why *Data Classes* might violate the *Single Responsibility* and Tell Don't Ask Principles. Other liabilities often mentioned are other maintainability issues such as *high coupling*, *high complexity*, *low cohesion*, and *reduced changeability*.

Related practices and patterns

Data Class seems rather simple to detect because of its obvious symptoms. But this is deceiving. Practitioners point out many structurally and behaviorally more or less identical solutions, which are seen as best practices and not as harmful *Data Classes*. The most common of those is the *Data Transfer Object* [43] which is an object that carries data between processes, e.g. often used in distributed systems. Immutability is an important principle of functional programming, and thus *Immutable Data Objects* are used e.g. where objects and functional programming are combined. Complex parameters can be simplified with *Parameter Objects*. *Data Access Objects* are usually offering data access behavior as well, but some practitioners point out that they can be confused with *Data Classes*.

As explained above in Section 3.4, Domain-driven Design aims for rich *Domain Models* with data and behavior. *Data Class* can be seen as a symptom of the *Anemic Domain Model* anti-pattern, and fixing them can improve the *Domain Model* as well as the *Services* relying on domain model classes.

Fix Options

As shown in Table 3.6, reworking the various classes that use the *Data Class'* data by *move method*, *move data-using methods to data class*, and *extract method* are very often suggested fixes. In addition, information hiding fix option such as *encapsulate field*, *encapsulate collection* and *remove getters and setters from data class and make fields private* are often suggested, too. *Making data objects immutable* (i.e., turning then into *Immutable Data Objects*), *subclass and extend*, and *using OO metrics to detect the smell* before other fixes are applied are each suggested by one source.

Indecent Exposure

Definitions

For *Indecent Exposure* all 5 sources covering the smell agree about the definition (as shown in Table 3.3): It is a class that exposes internal detail. One typical option considered are classes offering public variables, e.g. in languages like Java. In other languages such as Python this might be seen as less problematic, but still internal details can be exposed. Finally, many other language options exist, e.g. reflection or pointer manipulations, that might lead to *Indecent Exposure*.

Relationships to other smells

As shown in Figure 3.4, *Excessive Exposure* is another name of *Indecent Exposure*. *Indecent Exposure* is similar to *Inappropriate Intimacy* and can enable its occurrence. *Indecent Exposure* makes *Data Classes* worse because it reveals their data without any restrictions.

Liabilities and principle violations

For *Indecent Exposure maintainability issues* such as *high complexity*, *low code comprehensibility*, *high coupling*, and *reduced changeability* are discussed as typical liabilities. *Indecent Exposure* might lead to violations of the *Single Responsibility Principle*, if other classes use the exposed features to realize responsibilities belonging to those features.

Fix Options

Table 3.6 shows 4 different fix options for *Indecent Exposure*. They are all about adding more encapsulation or information hiding: *encapsulate field*, *encapsulate collection*, *hide behind method*, and *hide behind abstract class or interface*.

Message Chain

Definitions

For *Message Chain* our 12 sources agree on a single definition: long sequence of method calls. They often distinguish a call chain, as in the following example, or reaching them same with other means, e.g. by using temporary variables.

```
int id = obj.getDepartment().getSubDepartment().getHOD().getId();
```

Relationships to other smells

Fixing *Message Chain* can cause the *Middle Man* smell, if an object is introduced to only coordinate the calls in the chain.

Liabilities and principle violations

For *Message Chain* various *maintainability issues* such as *high coupling*, *low code comprehensibility*, and *reduced changeability* are discussed as typical liabilities. In a *Message Chain* calls might combine various responsibilities in one place, violating the *Single Responsibility Principle*. *Message Chains* might lead to more linked objects through calls, i.e. a violation of the *Law of Demeter*.

Fix Options

The majority of fix options are about step-wise reworking the *Message Chain*, i.e. *hide delegate*, *extract method*, *move method*, *hide behind method*, and *subclass and extend*. It is advised to ignore the smell, if it leads to introduction of a *Middle Man* or if it is just a *small chain causing little coupling*.

Middle Man

Definitions

Middle Man appears in 13 sources shown in Table 3.3. More than a half of the sources define *Middle Man* as a class that only delegates to other classes. Similarly, another definition sees the *Middle Man* as a Wrapper. Some practitioners mention that a *Middle Man* might do other work than only delegating. This is reflected in one definition variant by stating that the majority of the work is delegated. In another one it is introduced by making a single method the origin of the coupling, i.e. this way other methods of the class can perform other tasks than pure delegation.

Relationships to other smells

As shown in Table 3.2, *Middle Man* can result after resolving *Message Chain*.

Liabilities and principle violations

For *Middle Man* various *maintainability issues* such as *low code comprehensibility*, *high complexity*, *high coupling*, *reduced changeability*, and *missing encapsulations*, are discussed. A *Middle Man* gathers responsibilities that likely belong to the wrapped classes, i.e. it often violates the *Single Responsibility Principle*.

Related practices and patterns

There are structurally and behaviorally identical situations to *Middle Man* which are not considered as a smell. They occur if it is a *delegation class with a clear purpose*. Examples might be patterns and practices such as *Facade* [46], *Wrapper*, *Proxy* [46], and *Adapter* [46]. A *Mediator* [46] is also structurally and behaviorally similar, and if its mediation responsibilities shrink during evolution, it might turn into a mere *Middle Man*.

Fix Options

Most practitioners agree to resolve the *Middle Man* smell by a simple *remove middle man* fix (*remove middle man method* is similar to this). Three sources suggest to fix it by *inline method*. Two sources suggest to fix it by reworking the calls into the inheritance hierarchy: *replace delegation with inheritance* and *subclass and extend*. One source suggests *ignore if middle man is used to reduce interclass dependency*.

3.5 Discussion

Discussion of how practitioners understand coupling smells

In the previous section, we have detailed our finding regarding **RQ2.1**, i.e., our derived model how coupling smells are understood by practitioners. In particular, by describing our meta-model, depicted in Figure 3.2, we have outlined some of our findings of **RQ2.1.1**, i.e. what the relevant defining factors of coupling smells are. This is detailed in Figure 3.3 with possible relationship types. Besides, we have categorized the definitions of the smells used by practitioners for each of the coupling smells, as well as the coupling origins and targets in those definitions (see Table 3.3). Here, it is interesting to observe that many practitioners have a rather broad and hard to formalize understanding of the smells (such as "excessive use" based definitions). This means, it requires design expertise to judge whether or not a certain situation is a smell, making automated smell detection hard. While most coupling smells have only small variations in their definitions, the scope of *Feature Envy* is more controversial , as discussed in Section 3.4. Here, it is interesting to observe that one practitioner source explicitly sees code fragments as being potentially envious and many mention such code fragments in method or class contexts, e.g. in *Long Methods*. This is interesting because it also makes the coupling smell understanding of many practitioners hard to detect and fix without considering human design expertise.

It is worth to note that the practitioners sometimes confuse the various coupling smells. For instances, a confusion between *Feature Envy* and *Inappropriate Intimacy* is mentioned in S1. Some *Feature Envy* definitions actually describe *Data Envy* (e.g. in S28). *Feature Envy* can also be confused with more simple structures, e.g. utility functions (see e.g. S6, S42). Some practitioners see a difference between the notions of *Indecent Exposure* and *Inappropriate Intimacy*, for others it is the same smell.

RQ2.1.2 considers what the relevant quality impacts and trade-offs are. We have mainly found the stated liabilities and principle violations (see Table 3.4). Across the different smells, *low code comprehensibility*, i.e. impact on understandability qualities, is often mentioned, as well as various kinds of *maintainability issues* affecting software qualities such as understandability, changeability, evolvability, testability, and reusability. Violations to *Single Responsibility*, *Tell Don't Ask*, and *Law of Demeter* principles are often reported, too (and closely related to the mentioned liabilities). In a very few cases, impact on *Code* and *Design Technical Debt* are discussed, as well. We have found no evidence for concrete ways to spot, calculate, or pay back technical debt other than smell definitions, fixes, and two sources mentioning simple OO metrics for detection (S37, S41).

RQ2.1.3 investigates what the relevant relations among smells and to other software concepts are. We have found relations between smells (see Table 3.2), as well as relations to patterns and practices (see Table 3.5). For many coupling smells structurally and behaviorally identical solutions exist, which are considered in the literature as best practices. For example, for *Data Class* various exceptions to the definition are discussed, which are actually best practices, that some practitioners doubt whether *Data Class* is actually a code smell (see e.g. S32). For this reason, *Data Class* is, even though structurally very simple, rather hard to automatically detect without considering human design expertise.

A remarkable observation is that many practitioners relate Domain-Driven-Design to coupling smells. It seems a good *Domain Model* design can help to avoid coupling smells, and existing coupling smells might also need to be considered at the *Domain Model* level to be fixed well.

An overarching aspect concerning **RQ2.1.2** and **RQ2.1.3** is that simple examples of the smells, such as those used for illustration in this chapter, might be not critical enough to be considered harmful. Rather with enough *intensity of the smell*, a harmful negative effect on the qualities can arise that requires fixing. For instance, if multiple smells occur together, the intensity of each single smell might increase. For that reason, understanding the relations of smells is important.

RQ2.1.4 studies what the relevant options for fixing coupling smells are. We have found the detailed lists of fixes outlined in the previous section and summarized in Table 3.6. Here, it is interesting to observe that for situations where the smells occur in enough intensity to be harmful, the situations will likely also be too complex to be fixed by a single fix and often there are many options available for fixing. Complex interdependencies to other places in the code need to be considered for fixing the smells by applying a cascade of refactorings or other fix options.

References	Smell Name	Definition	Relations to Other Smells	Liabilities / Principle Violations	Related Patterns	Fix Options
[79]	Feature Envy	Yes	Yes	Yes	No	Yes
[79]	Data Class	Yes	Yes	Yes	No	Yes
[40]	Feature Envy	Yes	No	Yes	No	Yes
[111]	Feature Envy	Yes	No	No	No	Yes
[41]	Data Class	Yes	Yes	Yes	No	Yes
[41]	Message Chains	Yes	Yes	Yes	No	No
[38]	Long message chain	Yes	No	Yes	No	Yes
[88]	Feature Envy	Yes	No	Yes	No	Yes
[133]	Data Class	Yes	No	Yes	No	Yes
[97]	Feature Envy	Yes	Yes	Yes	Yes	Yes
[97]	Data Class	Yes	Yes	Yes	Yes	No
[26]	Feature Envy	Yes	No	No	No	No
[95]	Middle Man	Yes	No	No	No	No
[95]	Feature Envy	Yes	No	No	No	No
[110]	Feature Envy	Yes	No	No	No	No
[110]	Inappropriate Intimacy	Yes	No	Yes	No	No
[110]	Middle Man	Yes	No	No	No	No
[191]	Data Class	Yes	Yes	Yes	No	No
[191]	Feature Envy	Yes	Yes	Yes	No	No

Table 3.7: Summary: Smells in the Scientific Literature

Overall, the relation of many smells to inheritance practices remains unclear to a certain extent. Also, if inheritance can or should be used to fix the smells remains unclear. Smells might differ in different programming languages. For example, practices considered as *Indecent Exposure* in Java might be acceptable to Python developers. The developer's perception is also dependent on organization standard and project size. For example, S40 states that: "Code smell differs from project to project and developer to developer, according to the design standards that have been set by an organization." Our results indicate that understanding the smells' contexts, such as programming languages, project details, and design methods, could help understanding the implications of smells better. More research is needed to understand the impact of the smells' contexts.

For all of RQ2.1, it is interesting that practitioners sources often relate to only a few core sources, especially Fowler's refactoring book [1],

Interesting gaps to the scientific literature and future research opportunities and challenges

In this section we discuss the results in the relation to **RQ2.2**, i.e., what the gaps in the understanding of coupling smells between the practitioner's view and the scientific literature are. From each discussed aspect we derive lessons as answers to **RQ2.3**, i.e., what the resulting opportunities and challenges for future scientific work are. To illustrate our comparison further, we summarize in Table 3.7 how smells, smell definitions, relations, liabilities, related patterns, and fix options are covered in the scientific literature.

Coupling smell definitions and their detection

At first glance, the definitions of coupling smells used in research literature and those in the practitioner texts seem to match well. At the closer inspection performed in our study, this is not the case. Let us illustrate this using the example of *Feature Envy*: The scientific detection literature summarized in Section 3.2 mostly uses a definition such as "a method (class) uses more features (data) of another class than its own" in a rigorous sense, as this is easily measurable e.g. with distance metrics. As outlined, many practitioners rather use broader, pragmatic definitions. In addition, practitioners who use the "more feature" type definitions rather interpret them in their texts rather loosely. Practitioners seem to assume a review for the liabilities and principle violations in Section 3.4, and some code can be classified as *Feature Envy* only if a substantial harmful impact on desired design or code qualities is present. The relations to practices and patterns found in our study reveal many exceptions that must be made to the definitions of coupling smells. Overall, researchers rarely take patterns and practices into account [143] and only by other simple heuristics. For example, there is no structural or behavioral clue if some code detected as having a *Feature Envy* like structure and/or behavior is not really a *Visitor* or *Strategy* pattern. In response, some approaches tailor their detection heuristics by looking for terms like "visitor" and "strategy" in class and package names, but this approach is futile: Firstly, there is no guarantee that a visitor instance has "visitor" in its name. Secondly, these patterns are only used as examples by practitioners for situations where *behavior is kept separate from its data for a clear purpose*. There are many other such cases that look like *Feature Envy*, but are neither *Feature Envy* nor *Visitor* or *Strategy*. The found relations to domain-driven design practices reveal many additional situations where *Feature Envy* can be harmful; thus for detecting *Feature Envy* well, often a deep understanding of the system's *domain model* is required, too. Finally, often *Feature Envy* is seen as being harmful only if it occurs intensely enough in relation with other smells. The empirical study by Palomba et al. [110] confirms this finding. In other words, an approach to detect *Feature Envy* could consider all rich relations to the other smells that *Feature Envy* has (see Figure 3.4) into account.

While *Feature Envy* is the most complex smell in our study, the situation is more or less the same for all other coupling smells. As a consequence, many of the smell detection results found by current scientific tools would be considered not being the smell or non-harmful occurrence of the smell by practitioners, for one or the other of the reasons given above. Note that a couple of the empirical studies with human participants in Section 3.2 report similar findings (e.g. [95, 26, 55, 191]). For example, Carneiro et al.'s study [26] confirms our result that practitioners see coupling smell detection as a complex problem, as opposed to the tools which only use simple heuristics. Mantyla et al.'s study [95] confirms this view, too, e.g. as they found that conflicting perceptions of different evaluators play a large role. In Mantyla et al.'s study, metrics-based and practitioner identification of smells do not correlate, a result which our study would predict as a possible result. Guo et al. [55] found that semantic factors of smells could not be encoded into their tool, which can

also be explained using our results. Yamashita and Moonen [191] found that interaction effects amongst collocated smells and coupled smells should be taken into account. This is confirmed by our study, revealing many inter-smell relations. This study has also found that smells alone cannot predict maintainability issues. This is not surprising according to our results, which indicate that a considerable number of other aspects need to be taken into account when judging smells.

Lesson 1. *The understanding of coupling smells used in scientific studies could benefit from being broadened: When classifying some code as a coupling smell, it would make sense to consider a qualitative assessment of the code's impact on code and design qualities, the relations and interactions with other smells, and the relations to practices, patterns, and the system's domain model that sometimes lead to exceptions or changed quality impacts.*

As outlined in Section 3.2, many of the current coupling smell detection approaches assume that coupling smells can be detected by rather simple heuristics such as metrics, static or dynamic analyses, or metrics-based rules. Broader understandings of coupling smells, as suggested above, mean that no heuristic technique can ever take all situations covered by those definitions into account. That is, heuristics can help to find suspicious places in the code, but cannot solidly decide if it is a code smell or not.

Lesson 2. *Scientific studies could stronger focus on providing hints to developers and getting human design and domain expertise into the loop before classifying some code as being a coupling smell.*

Given the typical work pressures of industrial software developers, we speculate that using localized approaches to provide such hints for smell candidates directly during their work on the code, e.g. in the IDE or code editor, is advisable, instead of tools providing long lists of potential code issues (which contain a lot of false positives).

Coupling smell fix options

A similar situation as for the detection can actually be observed for the fixes of coupling smells. As can be seen (see e.g. Table 3.6), practitioners consider a much broader variety of fix options than usually considered in the scientific literature discussed in Section 3.2. Also they often consider doing many more changes than just one single refactoring. For example, whereas the scientific literature sometimes seems to imply typical harmful *Feature Envy* can be easily fixed by moving a method, very often a much more complex procedure is needed. Consider a complex, central class of a system envious to many other classes, but intertwined in many ways with those other classes. According to the various practitioner sources many steps for fixing might be needed such as the following might be needed. For instance, first a careful understanding of how the system's domain model needs to be changed is need or what the impacts of a change on system properties such its service API model are. Then various options for fixing the smell need to be considered, including complex refactorings such as multiple extract method refactorings, followed by moving some methods and fields,

bundling of methods, changing of the inheritance hierarchy, and so on. Before applying the fix, impacts on other smells (are new smells created? are other smells worsened by the change?), impacts on practices, patterns, and the domain model, and impacts on the overall code and design qualities need to be considered. In summary, Our GLS results indicate that fixing a complex coupling situation can be a complex multi-criteria optimization or decision problem that needs deep domain and design experience to be solved. Practitioners mainly care for understanding and getting help in these extremely complex design situations. It is thus explainable that empirical studies where human participants review small code entities lead to the result that practitioners do not see the smell identified by the researchers as being problematic (as e.g. in [110]).

Lesson 3. *Scientific studies could take into account a broad range of fixing options for a coupling smell and the trade-offs between various fixing options. They could consider the impacts on other smells, practices, patterns, the system's domain model, relevant code and design qualities, and so on. Approaches could consider getting human design and domain expertise into the loop. They could see fixing a highly coupled design situation as a complex multi-criteria optimization or decision problem.*

Domain-specific and other specializations for coupling smells and related approaches

It is interesting to observe that practitioners explicitly or implicitly discuss the relation to domain modeling, domain-driven design, and related semantically rich areas of software design quite often. In the scientific literature discussed in Section 3.2, these aspects play a very minor role. To explore such relations could lead to a better understanding of the rich semantic understanding of coupling smells revealed in our GLS.

Lesson 4. *Scientific studies could start to study the relations of domain modeling, domain-driven design, and related areas to coupling smells more intensely.*

Domain modeling and domain-driven design are areas, where more knowledge about the system or domain can help to improve smell detection and fixing. We have summarized a number of scientific approaches already going into that direction in Section 3.2. This is an interesting research direction that could be intensified. Many aspects identified in our study as being rather broad in the view of the practitioners, such as impact of liabilities and principle violations, relations to practices and patterns, inter-smell relations, and fix options, might be just that broad because the coupling smells cover every possible domain and every kind of system. Limiting the view e.g. to software architecture, microservice-based systems, domain modeling aspects, or enterprise architectures might greatly reduce the breadth of options and at the same time lead to more specific smells. Guo et al.'s empirical study results [55] confirm this and suggest to tailor the generic heuristics usually used for detection to take domain-specific factors into account.

Lesson 5. *Scientific studies could consider more specialized coupling smells and related approaches, for instance, in certain application or technology domains, or consider the smells only in a specific (technical) context.*

Note that covering more application or technology domains, and other such specializations, requires broader views on coupling smells (as suggested by Lesson 1). Very narrow understandings, e.g. in automated tools, might miss application- or technology-specific smells and fix options.

Experiences, Challenges, and Lessons Learned in the Grey Literature Study

A secondary contribution of our work is the integration of GT with GLS, using the grey literature as a data source for GT. As explained, we have successfully used similar research methods in a couple of prior studies [195, 194, 93, 109]. In our experience, this combination works well for studying existing phenomena in practice and their relations. For instance, it is very well applicable for studying which smells practitioners are concerned with, and their relations to existing patterns and practices, quality attributes (liabilities), principles, inter-smell relations, and so on.

Using grey literature compared to interviews avoids the interview situation context and the possibility to unintentionally leading the interviewees in the expected direction of interviewers. A challenge compared to interviews is that practitioner sources are not written with the goal to fuel a GLS. To address this challenge we opted to report opportunities and challenges for research, rather than criticizing existing research directions. Stronger statements would require additional research, e.g. quantitative studies or surveys, showing that the observed phenomena are relevant for practitioners at a broader scale. Multi-vocal literature reviews might also be a better option for research in such directions. All those research methods, however, would lack the explorative nature of our study. We also identified recommendations for the practitioners, including gaps to the research literature where practitioner approaches could benefit from the current state of the art in research, as well as many insights on the coupling smells offering a broader view than a few knowledge sources could offer.

In other GLSs and especially in multi-vocal literature reviews, the search strategy for the grey literature plays a central role. This is minimized in GT-based research, as GT does not aim to study phenomena holistically, but rather phenomena that have specifically been observed to exist [202]. This however means that reporting the literature search strategy in an overview is harder than for other GLS. To enable following our coding in detail, we provide all data and code in an open access repository. This downside of GT has a big advantage nonetheless: In most other GLSs it is hard to define exclusion criteria to avoid having to include sources of low quality. For example, in our study we could exclude otherwise well written sources that aimed to sell a product or service by a qualitative review

performed by the authors. That is, the selection process is very similar to interview studies and other such qualitative research – and has the same threats to validity discussed below.

We combined the GLS with UML-based, rigorous modeling for coding because it is highly beneficial to precisely represent the results in our experience. A major challenge is that the grey literature usually consists of rather informal writings and thus any categorization or classification performed by the researchers has a subjective element to it. Yet, this is the case for all axial coding in GT; it is not a specific property of our study. In contrast to text-based axial coding, the Python implementation keeps track of relations between codes, auto-layouts visualizations of those relations, and yields an error for many coding mistakes researchers might make. Such errors might go undetected in text-based coding.

Another big difference to many systematic literature studies is that each included source was not only scanned for knowledge, but instead analysed in-depth, line-by-line, usually many times. Whenever a later studied source reveals new insights on a phenomenon studied before, we re-analyzed the related source studied before. Here, our experience shows that precisely coded trace links can help a lot in guiding which sources provided evidence for some phenomenon and thus might need to be inspected again. To enable replicability of such studies, it is important to fully document each coding stage for each source – as provided in our open access replication package.

Adoption in practice, accuracy measures, oracles, and empirical studies

Given the very rich research literature on coupling smell detection and fixing, discussed in Section 3.2, it is interesting to observe that there are almost no signs of adoption of proposed techniques and methods in the practitioner grey literature we have studied (neither explicit by referencing the research work or implicit by using a techniques or method from the research literature without reference). Only two sources by the same author (S37, S41) use very basic metrics from the research literature. Assuming that average practitioners search for knowledge about coupling smells in a similar way like we did (i.e., by entering search terms into major search engines), it is unlikely that their search will yield much different results than those suggested in our study. We think for wider adoption of research results, expert practitioners who care for coupling smells, i.e. practitioners like the authors of the grey literature sources we have found, are essential to bridge the gap between more foundational science and practice. In our study, we have found subtle seeming but in the end substantial discrepancies between the view on coupling smells in the practitioner grey literature and the scientific literature. We think our study results can help adapt future research efforts in coupling smell detection and fixing to what practitioners actually need.

Lesson 6. *Currently there seems to be only low adoption of scientific results on coupling smells by practitioner. Adapting future research to practitioner needs, e.g. by following the other lessons outlined above, is necessary.*

One issue with many of the studies listed in Section 3.2 is that it is unclear for various reasons how accurate they are. First of all, many studies do not contain an evaluation of

their accuracy at all [143], in some cases for a good reason. For example, in some studies accuracy cannot easily be measured because these studies focus on cause-effect relationships (such as impact of smells on the developers' maintenance effort or change proneness) [143]. In addition, we reported a few findings in Section 3.3 where accuracies very roughly range from about 20% to about 75% for the different approaches. Here, it must be noted that even in the very best cases, according to our study results, they will contain many false positives and likely also false negatives with regard to what practitioners consider as harmful coupling smell instances.

In addition, many oracles used for such evaluations are questionable. For example, some cases we discussed in Section 3.2 use training data generated from the metrics they have set out to improve or generated synthesized issues in systems. The best case we have found is the Landfill open data set [112]. However, even here, the oracle was not produced by experts, but by two master students of the authors' university who reviewed overall a large set of open source systems. Only *Feature Envy* is considered as a coupling smell (i.e., no interactions or relations are considered), and a very narrow definition of Feature Envy is considered, easy to be found by tools, whereas our results indicate many practitioner seem to favor a broad definition with many possible, hard to find exceptions. Only 86 instances of *Feature Envy* were found in all these systems, 28 are in single system, and in many systems no *Feature Envy* was found. There is no indication that the many exceptions that practitioners report according to our study results were considered by the master students. If the students would have understood well the much broader views on *Feature Envy* found in our study, they might have ruled out certain cases and added a few others. Supporting such efforts is one of the key goals of our study. However, if our results are taken serious, one conclusion must be that only software design experts with many years of experience in the field can judge harmful coupling code smells well.

Lesson 7. *Scientific studies should use accuracy measures and oracles which consider the more semantically rich definitions and understandings of coupling smells we found in the practitioner literature to allow practitioners to more easily relate to the scientific results.*

Our discussion of empirical studies in Section 3.2 reveals a number of studies with human participants mostly in line with our results. Carneiro et al.'s study [26] confirms our result that practitioners see coupling smell detection as a complex problem, as opposed to the tools which only use simple heuristics. Mantyla et al.'s study [95] confirms this view, too, e.g. as they found that conflicting perceptions of different evaluators play a large role. In Mantyla et al.'s study, metrics-based and practitioner identification of smells do not correlate, a result which our study would predict as a possible result. Palomba et al. [110] find that "intensity" of the problem, as well developer's experience and system's knowledge play an important role in the identification of some smells by participants. The gaps between scientific and practitioner understanding found in our study can provide an explanation for these results. Guo et al. [55] found that semantic factors of smells could not

be encoded into their tool, which can also be explained using our results. Yamashita and Moonen [191] found that interaction effects amongst collocated smells and coupled smells should be taken into account. This is confirmed by our study, revealing many inter-smell relations. This study has also found that smells alone cannot predict maintainability issues. This is not surprising according to our results, which indicate that a considerable number of other aspects need to be taken into account when judging smells. It is good that such empirical finding with practitioners are mostly explainable by our Grounded Theory, on the one hand. On the other hand, more such studies are needed to confirm or refute our other findings, not covered in these studies yet.

Lesson 8. *More empirical work that considers a broader understanding of coupling smells and that is based on data from human participants is needed.*

Threats to validity

It is important to consider the threats to validity during the design of a study to increase its validity. As GT is mainly concerned with phenomena that have specifically been observed to exist, threats to the results' validity are mainly restricted to inappropriate conceptualization [202]. As we carefully added practitioner articles until theoretical saturation was reached, and then carefully reviewed and coded each source in multiple iterations by all authors, we believe the validity of our results as likely to be high. We do not claim that all practitioners in our data sources are experts for coupling smells. Our study has a certain mono-method bias as it uses grounded theory for analyzing grey literature only. To mitigate the threat we contrasted our finding in-depth to the scientific literature.

To increase internal validity or credibility we decided to use practitioner reports that were produced independent of our study. This avoids bias, e.g. compared to interviews in which the practitioners would have known that their answers are used in a study. However, this introduces a different threat: Some important information might be missing in the reports, which would have been revealed in an interview. We tried to mitigate this threat by looking at many more sources than needed to reach theoretical saturation, as it is unlikely that all different sources miss the same important information. A major challenge is that identifying gaps between scientific and practitioner understandings is always prone to a certain level of interpretation by the researchers.

The different members of the author team have cross-checked all models independently to minimize researcher bias. The threat to validity that the researcher team is biased in some sense remains, however. The same applies to our coding procedure and the UML-based modeling: Other researchers might have coded or modeled differently, leading to different models. As our goal was only to find one model that can specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study.

The experience and search-based procedure for finding knowledge sources may have introduced some kind of bias as well. However, this threat is mitigated to a large extent by the chosen research method, which requires just additional sources corresponding to

the inclusion and exclusion criteria, not a specific distribution of sources. Note that our procedure is in this regard rather similar to how interview partners are typically found in qualitative research studies in software engineering today. However, the threat remains that our procedures introduced some kind of unconscious exclusion of certain sources; we mitigated this by assembling an author team with many years of experience in the field, and performing very general and broad searches.

GT aims to explain phenomena that exist, and it neither claims to capture all such phenomena nor to quantify their frequency or distribution [202]. Therefore, even results derived from a very few sources will be valid and can be relevant [202]. There is a risk that generalizing from some of our results might be misleading, but as we do not claim completeness and use a rather high number of sources (many more than needed for theoretical saturation), it is likely that generalization is valid to at least some extent. Note that we have provided the number of sources mentioning a phenomenon to give an impression of the data we gathered; this should not be misinterpreted as quantification of frequency or distribution.

3.6 Conclusions

Based on a Grounded Theory study of grey literature sources containing practitioner views on coupling smells, this chapter has found defining factors of coupling smells and relevant impacts on and trade-offs to be made with regard to code and design quality. We identified Feature Envy, Inappropriate Intimacy, Data Class, Indecent Exposure, Message Chain, and Middle Man as coupling smells widely discussed by practitioners. We further studied the relations among those coupling smells, to other related smells, and to other software code and design concepts, as well as options for fixing coupling smells. Based on this knowledge, we identified gaps in the understanding of coupling smells between science and practice. Finally, we derived opportunities and challenges for future scientific work. In particular, we have derived eight lessons that represent opportunities and challenges for future research. From those we can conclude that the definitions of coupling smells used in approaches and tools need to be broadened to the various practitioner concerns identified in this chapter. Coupling smells detection tools and approaches could aim to better include human design expertise in the detection and fixing decisions on coupling smells. Coupling smell fixing often could be understood as a complex multi-criteria optimization or decision problem, not simply applying one or a few refactorings. The interaction of coupling smells and domain modeling could be studied more intensely. In general, the information related to the context, the domain, the size, and the design of the analyzed system is usually not considered. Scientific accuracy measures and oracles should consider these insights to allow practitioners to more easily relate to the scientific results.

Part III

Patterns Mining

Chapter 4

Derivation of ADDs, Patterns and Relations to Coupling Smells

This chapter presents the patterns mining research stage as same as addresses **RQ3**. After the study on the recurring patterns, two sets of patterns are revealed: (1.) patterns on deriving APIs and their endpoints from domain models (2.) patterns on designing API endpoint operation. The content of this chapter is based peer-reviewed article published in **EuroPloP'21** and **PloP'21**. Both are conferences on patterns and pattern languages which allow the authors experience with shepherding process and writing workshop.

Domain-Driven Design (DDD) places the domain model at the center of all software development practices. Remote API design is crucial for developing distributed systems including, for example, microservice-based systems. While software practitioners realize APIs based on DDD models, clear guidance on how to derive APIs and API endpoints from domain model elements is still missing. Based on prior in-depth studies of practitioner sources on this and related topics, we have mined patterns to address these design problems. In particular, we present the DOMAIN MODEL FACADE AS API pattern which describes how to derive an API from a Domain Model. To explain further how derive API endpoints constituting the API from Domain Model elements, we present the AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, and DOMAIN PROCESSES AS API ENDPOINTS patterns. In addition, we relate these patterns to the previously published patterns API DESCRIPTION and API CONTRACT, both explaining how to describe APIs formally. After we described endpoint-level patterns for this context. We present three complementary patterns, namely Aggregated Domain Operation on API Endpoint, Event-Based API Endpoint Operation, and CRUD-Based API Operation. These patterns aim to derive API operations from the operations of Domain Services and Entities as well as Domain Events. We also discuss variants of these patterns, such as their combination with the patterns Command Query Responsibility Segregation (CQRS) and Publish/Subscribe. Our pattern mining work is based on a data set from an empirical study.

4.1 Introduction

In Domain-Driven Design (DDD) [33, 182] the (business) domain is placed at the center of software designing and architecting by rigorously crafting and specifying a DOMAIN MODEL [42]. This DOMAIN MODEL is then used to build a UBIQUITOUS LANGUAGE that enables software development teams to use domain terms throughout the development process for the software system. Evans [33] classifies domain objects into types such as ENTITIES, VALUE OBJECTS, and SERVICES, which are then used to identify larger structures such as AGGREGATES or BOUNDED CONTEXTS.

Microservices are independently deployable, scalable, and changeable services, each having a single responsibility [203]. They are often identified based on DDD models [104, 157]. Microservices typically communicate via *APIs* in a loosely coupled fashion. These remote APIs can be realized using many technologies, including RESTful HTTP, queue-based messaging, SOAP/HTTP, or gRPC. A critical aspect in designing a microservice architecture is API design which includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to represent API messages, and so on [211, 212].

In this chapter, we use the following definitions (from the Microservice API Patterns [212, 90, 165]): "An *API endpoint* is the provider-side end of a communication channel and a specification of where the *API* endpoints are located so that *APIs* can be accessed by *API clients* (also called *API consumers*). Each endpoint thus must have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), in HTTP-based SOAP, or in RESTful HTTP. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints".

APIs are often used as the externally visible interfaces of systems modeled with DDD. Thus, the question arises how to derive APIs from DDD models. This design issue has been addressed in our prior pattern study [158]. In other prior works, we investigated the interrelation of microservice API design and DDD [157] and DDD violations in the context of coupling smells [154]. These smells and violations are especially problematic when used in distributed setting, as explained in Section 4.3. In this work, we investigate the next step in API design: the design of API operations. Based on the data sets created in our previous work, we have mined the API operation design patterns presented in this work.

Our main contributions contains two parts. The first part, based on the previously published patterns API DESCRIPTION and API CONTRACT, we explain how they can help in answering the design question *how to formally describe the API*. Next, we present a pattern on *how to derive APIs from a DOMAIN MODEL and its elements*. We identified the DOMAIN MODEL FACADE AS API pattern which derives an API as a FACADE [46] to an identified subset of DOMAIN MODEL elements well-suited to be exposed to the API[1] We describe it along with 4 pattern variants and two regularly occurring alternative practices. Next,

[1]Please note that we use the term “exposed to API” to indicate that one or more domain model elements are formally mapped to corresponding API elements, e.g. in a model or by implementation.

we present patterns on *how to derive API endpoints from domain model elements*. For this, we identified the AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, and DOMAIN PROCESSES AS API ENDPOINTS patterns. They all describe how to derive API endpoints from specific kinds of domain model elements. We present them together with 2 regularly occurring alternative practices (ENTITIES AS API ENDPOINTS and BOUNDED CONTEXTS AS API ENDPOINTS). Please note that in this work API endpoints cover all kinds of APIs realized with different kinds of technologies including RESTful HTTP, gRPC, SOAP, and messaging endpoints.

The second part, there are three patterns on API operation design derived from DDD designs, namely AGGREGATED DOMAIN OPERATION ON API ENDPOINT, EVENT-BASED API ENDPOINT OPERATION, and CRUD-BASED API OPERATION which are alternatives for API operation design. We describe the patterns in detail along with multiple pattern variants and known uses for each of the patterns. The target audience of this work are software/API developers and architects who are interested in the relations of DDD and APIs, as well as software engineering researchers studying those concepts. In particular, DDD is often applied in domains related to business problems and enterprise domains, but the patterns in the chapter are not limited to these domains.

This chapter is structured as follows: First, we explain our research method in Section 4.2. Next, we motivate the need for new patterns by discussing common API design smells in Section 4.3. After that, Section 4.4 provides pattern template used in this chapter. Section 4.5 then discusses the patterns on deriving APIs from DDD models. Section 4.6 discusses the patterns on API operation design derived from DDD designs. Then we discuss the related work in Section 4.7. Finally, in Section 4.8 we draw conclusions.

4.2 Research Method

The main knowledge sources used in this work are from our prior work [157]. In this work, we studied 32 practitioner sources from the grey literature (i.e., practitioner sources such as blog posts or system documentations [50]) in depth using the Grounded Theory (GT) research method [51, 24], a systematic research method for discovery of theory from data. We studied each knowledge source in depth, followed GT's coding process, as well as a constant comparison procedure to derive a model of architectural decisions on deriving APIs and API endpoints from domain model elements. Hentrich et al. provide details on how GT's coding process is mapped to pattern mining [62]. Riehle et al. [137] explain various such systematic pattern mining methods, and propose steps for discovering, codifying, evaluating, and validating the patterns during pattern mining. In GT-based pattern mining, those steps are embodied in the coding and constant comparison processes of GT. Our coding processes applied in this study are explained in detail in those two previous works [157, 154].

Using the same research methods, we also studied coupling smells based on 48 practitioner sources [154]. This study revealed many relations of coupling smells and principle violations to DDD models, and vice versa. As those tend to become specifically problem-

atic in distributed settings, many aspects in this study are core motivations of this work, as discussed in Section 4.3, and contribute to the key forces and consequences of our patterns. Besides those prior works, we also considered existing patterns and pattern languages to enhance and detail our patterns.

In addition to the detailed studies, we have modeled the DDD-to-API mappings of twelve system descriptions and open source systems by practitioners in UML. We have used these system models to confirm our patterns, and we also feature them below to present known uses of the patterns.

4.3 Motivation: Relations to Anemic Domain Models and Coupling Smells

This section explains why coupling smells related DDD violations are problematic in distributed settings and APIs as a motivation for this work. One of the key *anti-patterns* discussed in the realm of DDD is the ANEMIC DOMAIN MODEL[2]. This anti-pattern describes a DOMAIN MODEL that fails to combine data with logic processing it in its realization. Such domain objects look at first glance like good domain abstractions, as they are named with nouns from the domain's problem space and are connected with detailed relationships. Digging deeper and inspecting the object's behavior, however, shows that the objects actually carry little to no rich domain behavior (or business logic).

If an API is derived from such an ANEMIC DOMAIN MODEL often very basic elements of the DOMAIN MODEL, such as ENTITIES are exposed as e.g. RESTful services. This can lead to shallow API endpoints, where clients need to understand all the complexity in the backend. Transactional or data consistency boundaries between distributed services are missing, often leading to situations where the client needs to take part in ensuring data consistency in the backend services or where consistency management in the backend is hard to realize well. Such designs lead to chatty APIs with bad performance and scalability. APIs are becoming hard to understand, maintain, or evolve (see discussion and referenced gray literature in [157]).

A typical symptom of these problems visible at the API operation level, which we focus on in this chapter, are many simple and shallow ENTITIES or even VALUE OBJECTS, exposed as API endpoints, with CRUD (Create, Read, Update, Delete) operations only on them. This can lead to all negative consequences mentioned above. However, not every CRUD-based API endpoint is necessarily a bad design either: Some such endpoints expose rich domain abstractions well. Or the domain logic is inherently simple, meaning that the simple endpoint is the best possible design.

As it seems natural for a REST resource to refer to one or more nouns found in the DOMAIN MODEL, it is possible that, after these initial API designs are then fully specified and implemented, there is the danger that they could result in an ANEMIC DOMAIN MODEL.

[2]See e.g. `https://martinfowler.com/bliki/AnemicDomainModel.html`.

This happens if the search for finding rich[3] domain abstractions is replaced by simply using abstractions in the DOMAIN MODEL that are easy to realize as REST resources. If a substantial refactoring to API endpoints backed up by rich DOMAIN MODEL abstractions never happens, a shallow API on top of an ANEMIC DOMAIN MODEL is the consequence, with all kinds of negative consequences, such as the ones described above.

Our prior studies show that these issues can lead to poor quality *Domain Model* design [154] and in consequence bad API operation designs as well. In our prior study on coupling smells [154][4], a number of practitioner sources discuss relations of coupling smells to issues in DOMAIN MODEL design. Let us illustrate a few of those relations with their consequences on API operation design:

- The *Data Class* bad smell describes a class that only offers data. If exposed to an API, this can lead to shallow ENTITIES only with CRUD operations on them, if a naive object to resource mapping is used. Among other issues this can lead to data consistency issues or issues regarding transactional boundaries, chatty APIs, high API complexity, and so on.

- The *Feature Envy* bad smell describes a class or method that makes excessive use of a target class or its methods. If this happens for distributed API operations, this can lead to excessive distributed calls, which in turn leads to chatty APIs, performance and scalability issues, high API complexity, and interaction protocols that are hard to understand.

- The *Inappropriate Intimacy* bad smell describes a class using another class's implementation details. When this happens across different API operations of two API endpoints, this can lead to similar issues as for *Feature Envy*, just across multiple operations of the endpoints.

- The *Message Chain* bad smell describes designs containing a long sequence of method calls. If *Message Chains* are needed to work with API operations, then this leads to many distributed calls, which is a symptom of hard to understand, complex APIs with bad performance (so-called chatty APIs).

- The *Indecent Exposure* bad smell describes a class that exposes internal detail. If this happens in an API operation, clients get to know the service's internal DOMAIN MODEL or other backend details, which they do not need for the work. This means, the API is more complex than needed and hard to understand.

Our patterns introduced below help to relate DOMAIN MODEL method designs to API operations, thus helping to expose API operations that have a meaning in the DOMAIN

[3]Evans [33] uses the term "rich DOMAIN MODEL" to express a DOMAIN MODEL which contains a rich understanding of the processes and rules of a domain. The opposite is an ANEMIC DOMAIN MODEL which represents rather a shallow understanding of the domain. Our patterns, in part, aim to helpt in reflecting such rich domain abstractions in the API.

[4]See also `https://refactoring.guru/refactoring/smells/` for definitions of code smells.

MODEL. In this context, the patterns help avoiding API operation designs that lead to these and similar smells.

4.4 Pattern Template

We use the following pattern template, which is derived from the so-called CANONICAL FORM[5], for presenting our patterns.

- **Name.** The name of the pattern conveys characteristics of the solution. It also delivers the big picture of what the solution is.
- **Context.** The context describes the conditions in which a pattern exists or occurs. It is a part of a discourse that relates the problem and the solution.
- **Problem.** The problem statement describes the issues developers or architects might face in a design situation, and is phrased in form of a question.
- **Forces.** The forces are mainly identified and synthesized from our previous work (see discussion above). They represent the core decision driver to decide for or against the use of the solution, also in relation to the other patterns in this work, which might serve as alternative options.
- **Solution.** The solution provides an answer of the design issues posed in the problem statement.
- **Solution Details.** The solution details extend the answer provided by the solution by covering various aspects of it in more detail.
- **Example.** A source code based example is provided to illustrate the solution.
- **Pattern Variants.** Known pattern variants are discussed. They represent alternative ways for providing the given solution to the problem.
- **Consequences.** Consequences are addressing the resulting context. We designate them with (+) and (-) to indicate a positive or negative effect, respectively.
- **Related Patterns.** We discuss related patterns from the literature.
- **Known Uses.** Known uses provide evidence that the pattern is used in practice. We have modeled 14 systems as cases that contain known uses; in addition, we discuss known uses of the pattern from the literature.

Figure 4.1: Overview of the patterns for API descriptions and/or contracts

4.5 Patterns on Deriving APIs from DDD Models

We first report on two previously mined patterns, API DESCRIPTION and API CONTRACT. We discuss them rather briefly, as API DESCRIPTION [90] has been published as part of the Microservice API Patterns [212] before, and API CONTRACT is just a variant of the INTERFACE DESCRIPTION pattern in the Remoting Patterns [184]. The patterns' relationships are illustrated in Figure 4.1.

Next, we present the DOMAIN MODEL FACADE AS API pattern that describes how to derive an API from a DOMAIN MODEL and its elements. The pattern explains in its pattern text a number of variants on how to achieve deriving API elements from a DOMAIN MODEL (namely *Domain Model to API Model Mapping*, *Interface Bounded Context*, *Shared Kernel Based Interface*[6], and *Implicit Designation of API Model Subset*). We also describe two alternative practices[7] (namely *Expose the Whole Domain Model 1:1 as the API* and *Expose Each Bounded Context as its Own API*). An API design based on DOMAIN MODEL FACADE AS API can or cannot use the API CONTRACT and/or API DESCRIPTION for API documentation. Figure 4.2 illustrates those pattern relations. The DOMAIN MODEL FACADE AS API should use API names and abstractions from the UBIQUITOUS LANGUAGE [33] formally specified by the DOMAIN MODEL. This makes it easy to trace from API elements to the corresponding domain model elements, and enable domain experts to understand the API. An API is a PUBLISHED LANGUAGE [33] in DDD terminology. If an API CONTRACT or API DESCRIPTION is used to specify the API design, the API CONTRACT or API DESCRIPTION is a formal specification of the PUBLISHED LANGUAGE.

The following patterns AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, and DOMAIN PROCESSES AS API ENDPOINTS describe how to derive API endpoints from domain model elements. We describe AGGREGATE ROOTS AS API ENDPOINTS in detail. For the very similar, but less often used DOMAIN SERVICES AS API ENDPOINTS and DOMAIN PROCESSES AS API ENDPOINTS patterns, we describe the differences to AGGREGATE ROOTS AS API ENDPOINTS briefly. Finally, in the pattern text of AGGREGATE ROOTS AS API ENDPOINTS we describe the two alternative practices ENTITIES AS API ENDPOINTS and BOUNDED CONTEXTS AS API ENDPOINTS which only rarely deliver results on deriving an API from DOMAIN MODEL elements that balance the forces well. AGGREGATE ROOTS AS API ENDPOINTS (and all alternative patterns and practices) can use

[5] `https://wiki.c2.com/?CanonicalForm`

[6] Please note that a SHARED KERNEL can be defined as the shared interface that constitutes e.g. a solution internal APIs. For external or public APIs, usually rather interface BOUNDED CONTEXTS are used.

[7] We use the term "practice" if we found in our prior studies practices that are often applied by practitioners, but that should not be called a pattern or best practice. That is, they might work well in some design situations, but might in others be considered an anti-pattern.

API CONTRACT or API DESCRIPTION to specify the mapped API design as well as the API endpoints formally. Figure 4.3 illustrates those pattern relations.

Figure 4.3 also shows the relation of the API endpoints patterns to DOMAIN MODEL FACADE AS API: An API is composed of a number of endpoints. Thus, the API endpoint patterns determine the domain model subset to be exposed in the API. Please note that there is a certain overlap between the DOMAIN MODEL FACADE AS API patterns and the API endpoints patterns, as some APIs are based on BOUNDED CONTEXTS, especially if dedicated interface BOUNDED CONTEXTS are designed with the purpose of creating APIs from them. But smaller, more limited, or only partly exposed BOUNDED CONTEXTS are also sometimes used as basis for identifying API endpoints.

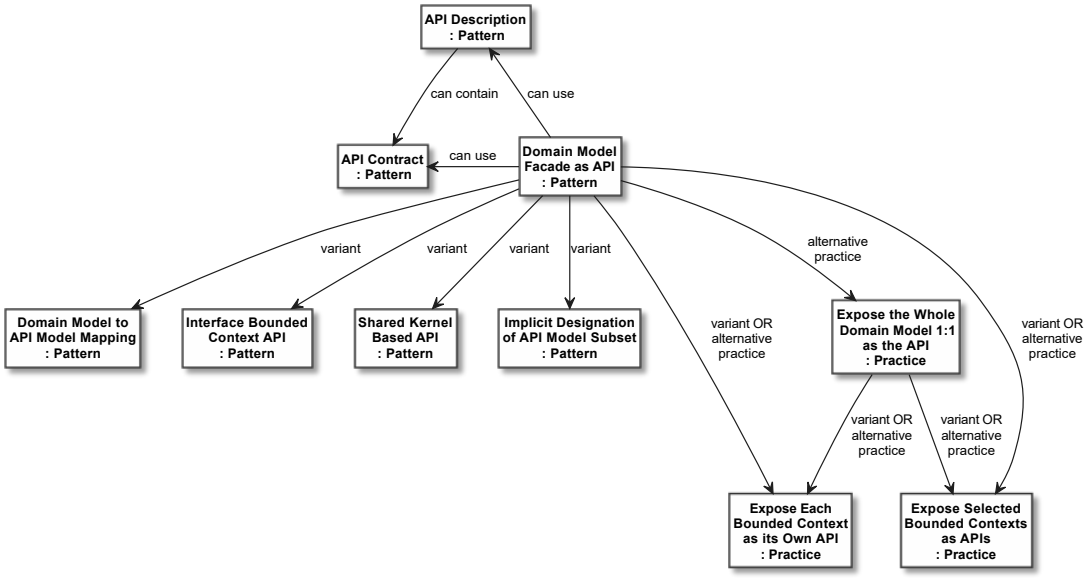

Figure 4.2: Overview of the patterns for how to derive an API from a DOMAIN MODEL

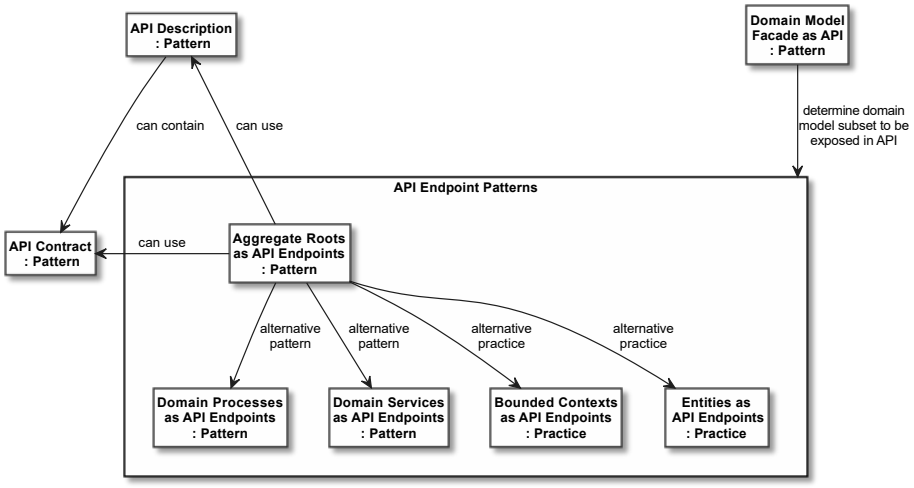

Figure 4.3: Overview of the patterns for how to derive API endpoints from domain model elements

Patterns: API Contract and API Description

An API DESCRIPTION [90, 212][8] contains the API CONTRACT that defines request and response message structures, error reporting, and other relevant parts of the technical knowledge to be shared between API provider and client. In addition to this syntactical interface description it contains quality management policies as well as semantic specifications and organizational information. The API CONTRACT part is essentially a special case or variant of the INTERFACE DESCRIPTION pattern [184] with the purpose of describing a remote API. For example, Swagger/Open API, WADL, or WSDL are INTERFACE DESCRIPTION languages used to describe API CONTRACTS. Please refer to the original pattern sources [90, 212, 184] for a detailed description of these patterns.

Pattern: Domain Model Facade as API

Alias

Remote Service Layer.

Context

In a software development project, you use DDD to design your DOMAIN MODEL and want to expose some parts of the software system you develop as an API.

Problem

How to derive an API from a DOMAIN MODEL?

Forces

- *Avoiding Brittle Interfaces*: A naive solution is to map every element of the DOMAIN MODEL to the API. This can lead to brittle interfaces as any change in the DOMAIN MODEL leads to a change in the API. DOMAIN MODEL and backend implementation should be kept flexible, while APIs should usually – as far as possible – stay stable. Of course, as a downside, the more the DOMAIN MODEL and the API drift apart, the higher the effort for API design and the more difficult it might become to understand and keep track of the mapping between them. A good balance has to be found.

- *API Complexity*: Low complexity of the API is essential to make it comprehensible both for consumers of the API and API developers (see [11] for a definition of complexity as it is used here). Exposing all elements of a DOMAIN MODEL would lead to unnecessary complexity of the API, if a smaller or more simplified abstraction is possible. However, the more the DOMAIN MODEL and the API drift apart, the more complex the mapping logic becomes, also possibly leading to complexity. A middle

[8] See `https://microservice-api-patterns.org/patterns/foundation/APIDescription`

ground where both API complexity and mapping complexity are kept in check have to be found. Following the POINT principles can help here[9]

- *API Usability and Understandability*: Aiming for low complexity and high stability means to make the API more usable to the consumer. As API consumers are often not experts for the backend implementation, any unnecessary details should be abstracted away from the API. Also, those properties make an API understandable both for API consumers and API developers. Ideally, API consumers should not have to know any details about the underlying domain model. However, any such measures require additional design effort for the API, and it must be decided if they really pay off. A good API design should not come at the cost of a bad DOMAIN MODEL or backend design.

- *Modifiability*: Most software systems have to change over time. As pointed out, the pace of changes in DOMAIN MODELS and APIs can differ significantly, as APIs should rather stay stable, whereas DOMAIN MODELS must change as the requirements change. This can lead to tensions between the two abstractions. Also, some DOMAIN MODEL changes lead to required API changes. Such changes should rather be localized both in the DOMAIN MODEL and in the API. Finally, the API can change independently of the DOMAIN MODEL. For example, if the DOMAIN MODELS is gradually exposed to the API. Or, if technology changes induce API changes, the API can change without a change in the DOMAIN MODEL.

- *Design and Implementation Effort*: As designer and developer time is costly, usually only a limited amount of design and implementation effort can be invested in the initial realization of an API and its evolution. As mentioned above, many desired API features require additional design and implementation effort.

- *API Maintainability*: High complexity, low usability and understandability, and low modifiability can all lead to maintainability problems. Again, as a downside, those might be lead to higher design and implementation efforts throughout the API evolution. [116]

- *Security and Privacy*: Security and privacy can be an important consideration when deriving an API from a domain model, too. Interesting considerations to be made during API design can be, for instance, to avoid exposing confidential domain elements, enabling auditability, and supporting API monitoring or observability.

Solution

Introduce a dedicated API view on selected parts of the DOMAIN MODEL to establish a PUBLISHED LANGUAGE that exposes parts of the UBIQUITOUS LANGUAGE of the domain

[9] See `https://medium.com/olzzio/apis-should-get-to-the-point-c79113efa31c`.

in a controlled, managed fashion. Mark the domain model elements exposed to the API clearly as API elements.

Solution Details

In DDD terminology an API is a PUBLISHED LANGUAGE offered by the API publisher's contexts to the API consumers' contexts (both of these kinds of contexts can be modeled as BOUNDED CONTEXTS). The parts of the DOMAIN MODEL selected to be exposed in the API are usually determined based on the coarse-grained parts required by API consumers. Here, especially a selection of domain model elements to be exposed as API endpoints is necessary. How to do the selection of individual API endpoints candidates well is explained in the patterns AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN PROCESSES AS API ENDPOINTS, and DOMAIN SERVICES AS API ENDPOINTS. Occasionally, the ENTITIES AS API ENDPOINTS and BOUNDED CONTEXTS AS API ENDPOINTS patterns may also deliver good results on deriving an API from domain model elements, as explained above. Based on the to-be exposed coarse-grained domain model elements, it is necessary to determine which of them should be exposed as API endpoint in which API. That is, not always all endpoints of an application are exposed in the same API. Just consider an application that needs an API for ordinary API consumers plus an administration API interface. Then, two APIs exposed by the application need to be derived from the set of API endpoints of one application.

The design process is usually iterative; this is emphasized strongly in the literature on object-oriented analysis and design [81] and in the Design Practice Repository [210] that collects general purpose activity descriptions and artifact templates. It can start by considering API endpoints candidates using patterns such as AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN PROCESSES AS API ENDPOINTS, and DOMAIN SERVICES AS API ENDPOINTS. As explained above, this is intertwined with the design of the *Application Services* in a SERVICE LAYER. The design process can also start by considerations about the whole API, e.g. the design of one or more DOMAIN MODEL FACADE AS APIS that are needed by API consumers. Usually, multiple iterations of the design are needed to find a good compromise.

In addition to coarse grained domain model elements, also more fine-grained elements need to be exposed. For example, messages and message contents of the API can be derived from links, data types, and operations.

Not every part of the API must have a representation in the DOMAIN MODEL. For example, DATA TRANSFER OBJECTS [42] might be introduced covering multiple domain model elements or only parts of some of them. But in order to make the API understandable to domain experts, it is essential that names and abstractions in the API follow the terms defined in the UBIQUITOUS LANGUAGE which is formally specified by the DOMAIN MODEL. Thus, any deviation from the DOMAIN MODEL should have a good reason.

Example

To illustrate the pattern let us again consider the excerpt of a DDD model of a publication management system from Section 4.5.

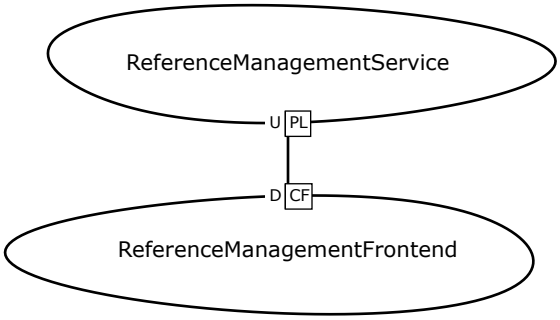

Figure 4.4: Publication Management Context Map (adapted from [204])

As shown in Figure 4.4, the *Reference Management Service* BOUNDED CONTEXT is a PUBLISHED LANGUAGE (indicated by *PL* in the CONTEXT MAP) for the *Reference Management Frontend* BOUNDED CONTEXT, which has just a CONFORMIST relation (indicated by *CF* in the CONTEXT MAP) to the Service BOUNDED CONTEXT. A CONFORMIST relation means that the downstream context (indicated by *D* in the CONTEXT MAP), which is dependent on the upstream context (indicated by *U* in the CONTEXT MAP), "eliminates the complexity of translation between BOUNDED CONTEXTS by slavishly adhering to the model of the upstream team" [33]. That is, here it would not make much sense to expose any domain model elements in the *Reference Management Frontend* BOUNDED CONTEXT as they would not contribute anything new to the API. In the context of the *Reference Management Service* BOUNDED CONTEXT it may also not make sense to see all elements of the DOMAIN MODEL as equally important for being exposed in the API. So instead of selecting the whole domain model as the scope making up the API, or creating APIs based on all the BOUNDED CONTEXTS, here a better choice is to select only the *Reference Management Service* BOUNDED CONTEXT as the scope for the API.

Next, we select the specific elements to be exposed to the API in this BOUNDED CONTEXT. In Figure 4.9 an API mapping is shown where the *Reference Management Service* BOUNDED CONTEXT is exposed as the *API* scope. This is denoted by the BOUNDED CONTEXT being exposed to the *Reference Management Service API*. This API offers an API endpoint which is exposed by the *Paper Archive Facade* AGGREGATE. The *Paper Item* and *Paper Item Key* domain model elements are both exposed as *API Data Types*. Via the *«exposed to API as»* relations, the subset of the API that is exposed to the API is clearly designated in this model.

Pattern Variants and Alternative Practices

There are a number of variants of DOMAIN MODEL FACADE AS API on how to mark elements clearly as API elements, as shown in Figure 4.5. The variants and alternative practices are used here to explain different ways how the pattern is applied in practice.

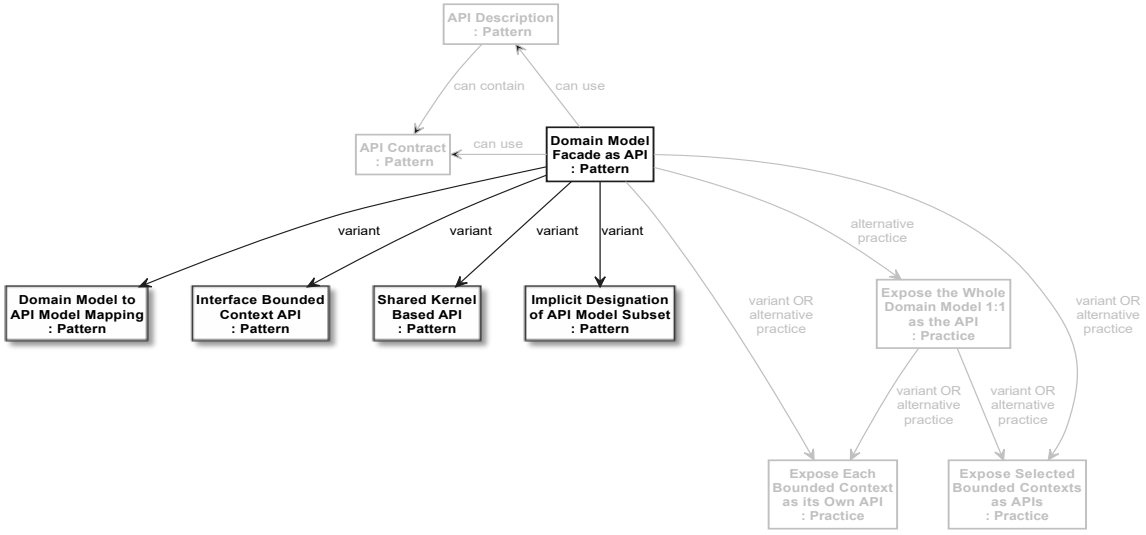

Figure 4.5: Domain Model Facade as API – Main Variants

- *Domain Model to API Model Mapping:* One option is to model the API explicitly alongside the domain model. Then the API model elements can be derived from domain model elements. Those can for instance be highlighted using dedicated stereotypes or relationship types.

- *Interface Bounded Context:* Another option is to design an explicit BOUNDED CONTEXT that contains the interface to be exposed in the API. This BOUNDED CONTEXT is designated as an Interface Bounded Context in the DOMAIN MODEL design.

- *Shared Kernel Based Interface:* A similar option is to design a SHARED KERNEL [33] for the interface between API consumer and publisher. The content of this SHARED KERNEL designates the domain model elements to be exposed in the API. This option assumes that the client scope is and can be modeled, too. For example, for public APIs this might be hard to do; but in more closed settings, e.g. where the teams developing the clients are known partners, this option is applicable. As a consequence, a SHARED KERNEL based interface would be used e.g. in a solution internal APIs. For external or public APIs, usually rather interface BOUNDED CONTEXTS are used.

- *Implicit Designation of API Model Subset to be Exposed in the Facade:* While it is advisable to mark domain model elements clearly as API elements, often less clear options than dedicated stereotypes or relationship types are chosen. For example, a variant of the pattern just uses the same or similar names for domain model element and API element. Such practices can lead confusions and inconsistencies in the mappings.

Two other *variants* of DOMAIN MODEL FACADE AS API, as shown in Figure 4.6, are to *Expose Each Bounded Context as its Own API* [157] or *Expose Selected Bounded Context as APIs* [157]. These two practices select the BOUNDED CONTEXT as API scope. They are a variant of DOMAIN MODEL FACADE AS API if the DOMAIN MODEL of each BOUNDED

CONTEXT is exposed to the API following the guidances provided here in the DOMAIN MODEL FACADE AS API pattern.

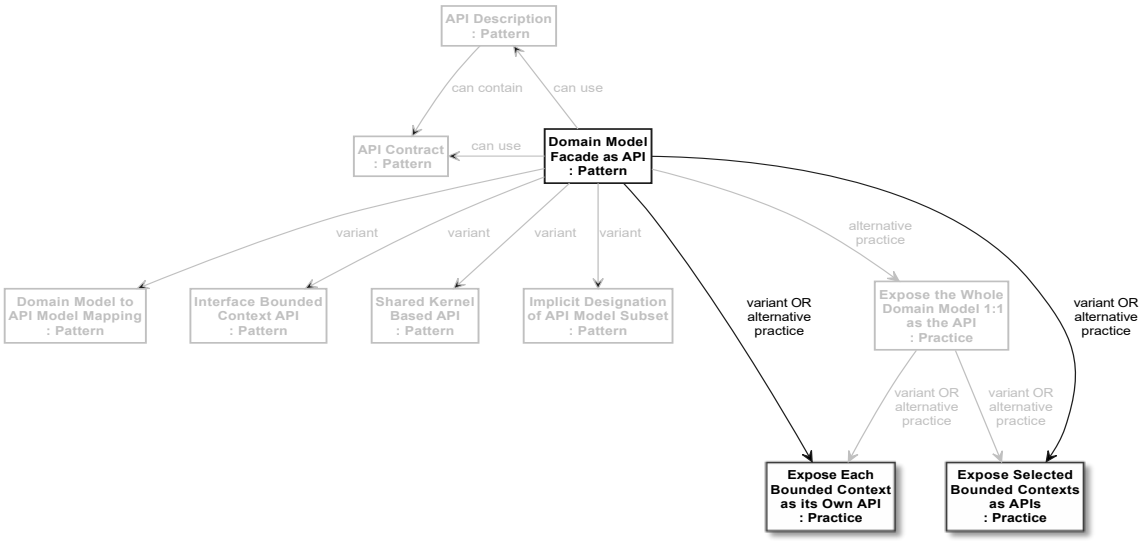

Figure 4.6: Domain Model Facade as API – Other Variants

There are two possible alternative solutions, as shown in Figure 4.7, sometimes discussed by practitioners, but which only rarely lead to acceptable results:

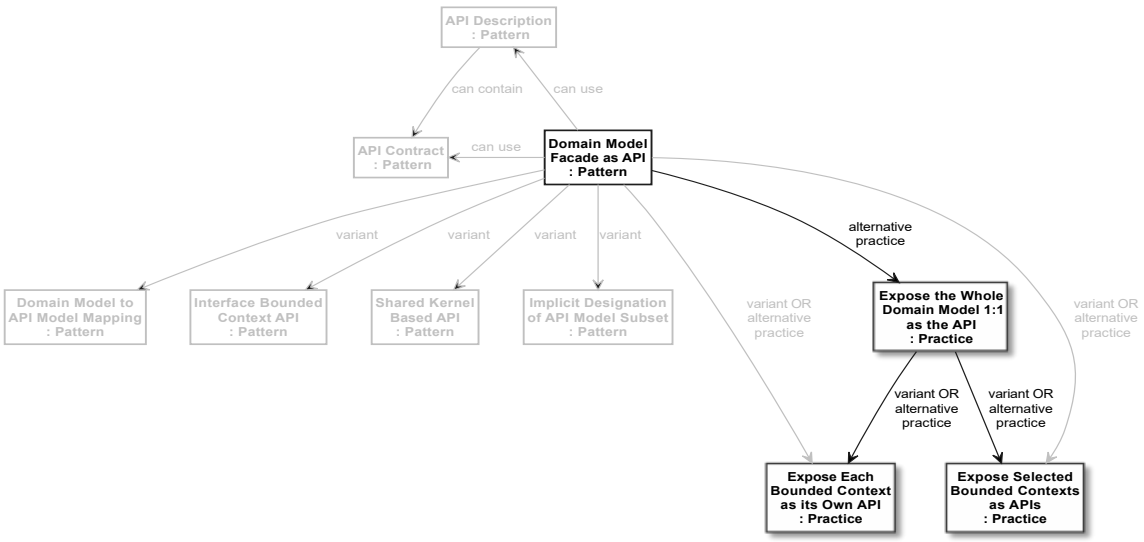

Figure 4.7: Domain Model Facade as API – Alternative Practices

- One alternative to applying this pattern is to *Expose the Whole Domain Model 1:1 as the API* [157]. At least in non-trivial domains, this solution is in many situations rather an anti-pattern. For example, it can lead to brittle interfaces, high API complexity, understandability issues, and API modifiability problems.

- The two practices *Expose Each Bounded Context as its Own API* or *Expose Selected Bounded Context as APIs* are merely alternative practices to the DOMAIN MODEL FACADE AS API pattern, if there API mapping is not following the DOMAIN MODEL

FACADE AS API pattern. Especially, if their mapping is akin to *Expose the Whole Domain Model 1:1 as the API*, i.e. exposing the whole domain model of each of those BOUNDED CONTEXTS to the API, it is again rather an anti-pattern. That is, while those alternatives are then more fine-grained than *Expose the Whole Domain Model 1:1 as the API*, still all elements of those BOUNDED CONTEXTS are exposed, including many elements that do not have to be exposed. Thus those solutions tends to lead to similar problems as exposing the whole DOMAIN MODEL 1:1 as the API. For these reasons, *Expose Each Bounded Context as its Own API* or *Expose Selected Bounded Context as APIs* are shown in Figure 4.2 as *variant OR alternative practice* both for DOMAIN MODEL FACADE AS API and *Expose the Whole Domain Model 1:1 as the API*, depending on how the mapping of the DOMAIN MODELS of the API BOUNDED CONTEXTS is done.

Consequences

+ *Stable Interfaces:* The pattern leads to more stable interfaces than solutions like exposing the whole DOMAIN MODEL or exposing all BOUNDED CONTEXTS as the API scope. The reason is that only selected interface elements are exposed, meaning that changes in other parts of the DOMAIN MODEL do not affect the API.

+ *API Complexity:* The pattern has a positive effect on API complexity as specific elements of the DOMAIN MODEL are selected and designed to be part of the API. Thus the API gets smaller and tends to contain abstractions and aggregated elements rather than every DOMAIN MODEL detail.

+ *API Usability and Understandability:* The pattern is beneficial to API usability and understandability because of the lowered complexity and increased stability it offers.

+ *API Modifiability:* Detailed selection of API-exposed domain elements is positive for API modifiability as it is easy to change and extend the selection.

+ *Avoiding API Maintainability Issues and Reducing Design and Implementation Effort:* As the pattern can lower complexity, improve usability and understandability, and improve modifiability it can help to avoid API Maintainability problems. This can lead to less design and implementation effort throughout the API evolution.

+ *Traceability:* If a systematic and formal mapping between domain model elements and API elements is used, the pattern enables traceability from API elements to contributing domain model elements.

+ *Security and Privacy:* This pattern creates a clear mapping between DOMAIN MODEL and API, making it easier to trace which parts of the domain model (at a coarser-grained level) are exposed via the API. This can help to fulfill auditability and/or monitoring requirements e.g. for possibly confidential parts of the domain model.

- *Design and Implementation Effort:* Compared to naively exposing the whole DOMAIN MODEL or exposing all BOUNDED CONTEXTS, the pattern requires initially a higher design and implementation effort.
- *API Usability, Understandability, Modifiability, and Maintainability:* Compared to the INTERFACE BOUNDED CONTEXT or SHARED KERNEL BASED INTERFACE variants, described above, the detailed selection of exposed elements has a negative effect on API usability and understandability, as well as API Maintainability, as the mapping might require both for the consumers and maintainers more effort for comprehending the API. This can be negative for API modifiability, too.
- *Clients Might Have to Manage Crossing Model Boundaries:* Another possible downside of this solution is that clients might have to manage crossing model boundaries, i.e., the boundaries between the BOUNDED CONTEXTS, but only if domain model elements from different contexts are exposed.
- *Traceability:* If no systematic and formal mapping between domain model elements and API elements is used, traceability can be lost, meaning that inconsistencies and confusion can arise. This can mean that many of the benefits of the pattern cannot be achieved.

Related Patterns

The DOMAIN MODEL FACADE AS API pattern describes how to establish one or more APIs based on the domain model elements to be exposed to an API through *Application Services* [182]. Together the *Application Services* form a SERVICE LAYER [42]. The SERVICE LAYER defines an application's boundary as a layer of (micro)services that implement the API. An API defines the client-visible interfaces of a subset of those services exposed to API consumers. For example, an application might define five *Application Services* in its SERVICE LAYER, each with its own endpoint. It might further define two APIs, one for ordinary API consumers and one an administration API; the first API exposes three of the *Application Service* endpoints, the second one the two other endpoints. The AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN PROCESSES AS API ENDPOINTS, and DOMAIN SERVICES AS API ENDPOINTS patterns, explained later on ,as well as their alternative practices, are ways how find candidates for the mapping of coarse-grained domain model elements to API endpoints.

A REMOTE FACADE "provides a coarse-grained facade on fine-grained objects to improve efficiency over a network." [42]. Typically a DOMAIN MODEL FACADE AS API is special form of REMOTE FACADE which does not directly expose objects, but rather *Application Services* in the SERVICE LAYER [42] which then access objects and other implementation structures in their own service implementations or in backends. The relations to these patterns, is the reason for the alias *Remote Service Layer* of the DOMAIN MODEL FACADE AS API pattern.

Please note that DOMAIN EVENTS [33] and event-driven architecture related patterns such as EVENT SOURCING [136] or CQRS [136] are often important in API derivation. That is, if the backends use events and event-driven architectures, some of those or abstractions of those events can be exposed in the API. However, this is usually in first place a consideration of more detailed API design than the patterns covered in this chapter (see [157] for more detailed architecture design decisions on this).

The API GATEWAY pattern [136] is often the place where the API is technically offered. Sometimes different APIs are offered via different gateway, e.g. as in the BACKENDS FOR FRONTENDS pattern [136]. In such cases, multiple APIs need to be derived from the same domain model. That is, the DOMAIN MODEL FACADE AS API is often useed to design the APIs exposed on API GATEWAY or BACKENDS FOR FRONTENDS.

Known Uses

- The publication management system [204] used as an example above uses the pattern to select specific API elements that are described via the API DESCRIPTION language MDSL. MDSL can be used to generate the API CONTRACT in respective technologies such as Swagger/Open API, and/or generate models and code.
- In an online shop system model [209] that is also specified with MDSL, an AGGREGATE is selected as the scope for an API, and three endpoints are defined as a DOMAIN MODEL FACADE AS API. From them, a DOMAIN MODEL with two SERVICES, five ENTITIES, and five VALUE OBJECTS is derived.
- Dugalic [30] discusses a purchase order management system with a Shipping and Order BOUNDED CONTEXTS. The two BOUNDED CONTEXTS are exposed to the API and use various AGGREGATES for their realization. That is a selection of domain model elements is made that are exposed as one API per BOUNDED CONTEXT, meaning that the *Expose Each Bounded Context as its Own API* variant of DOMAIN MODEL FACADE AS API is chosen. Each API offers two endpoints, one command and one query endpoint, to realize the COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS) pattern [136], two times: two endpoints for a RESTful and two for a gRPC-based API. Lastly, the APIs each offer one gRPC-based PUBLISH-SUBSCRIBE endpoint for exchanging events, e.g. between the services realizing the endpoints.

Pattern: Aggregate Roots as API Endpoints

Alias

Aggregate Root Wrappers, Aggregate-oriented API Endpoints.

Context

In a software development project, you use DDD to design your DOMAIN MODEL and want to expose some parts of the software as an API.

Problem

How to derive API endpoints from domain model elements?

Forces

- *Avoid Exposing Domain Model Details in API*: Exposing details of the domain models in the API that are not necessarily needed by at least some API consumers increases the API complexity and coupling between clients and server. This can lead to maintainability and modifiability problems. But avoiding domain model details in the API requires more intricate API design and consequently implementation, which leads to more design and implementation effort required for the initial API.

- *Data Consistency*: The API shall be designed in a way so that it is easy to maintain strict or eventual data consistency[10] as required in the given business domain context. If parts that need to be kept consistent are split across multiple API endpoints, distributed measures for ensuring consistency have to be used, which are more complex and error-prone, and offer worse performance than local consistency measures. However, a good design for data consistency is hard, especially in distributed setting. This can be avoid by co-locating data elements in the same service.

- *Chatty APIs*: If (almost) every step in the processing requires some distributed interaction, the application is much slower than a coarser design, where unnecessary distributed calls are avoided. Such APIs are too chatty. The same can happen also for the services in the backend (i.e. chatty microservices), but in this force many client-API interactions are meant that can be observed at the API surface. Also, API complexity rises because complex distributed interaction patterns need to be understood. This influences maintainability negatively, too. For example, often the client has to maintain state and orchestrate interactions between many chatty API calls.

- *Performance and Scalability*: APIs should perform as well as possible and be as scalable as required. Bad API endpoint choices, e.g. as discussed for chatty APIs and issues in data consistency, can lead to insufficient performance and scalability. Performance and scalability can be improved, if the domain object that are needed to sent back a particular response are kept together in one and the same API service where possible.

- *API Complexity*: A very fragmented or too detailed derivation of API elements from domain model elements, as well as chatty APIs, can lead to high complexity of the API, which should be avoided. For instance, fragmented or chatty APIs usually make many assumptions on complex interaction patterns or state to be kept between API

[10] Eventual consistency describes a weak consistency relation which requires that all replicas of an object (here: microservices) will only eventually reach the same correct value.

calls (on client side), making the APIs hard to understand and use. Conversation patterns describing such interactions have been mined [63].

- *Coupling of API Consumer and Supplier*: A very fragmented or too detailed derivation of API elements from domain model elements, can lead to a high coupling of API consumer and supplier. As a consequence, every change in the API interface can lead to (maybe complex) changes in many clients, making it progressively harder to evolve the API.

- *Security and Privacy*: As in the DOMAIN MODEL FACADE AS API pattern, security and privacy related aspects play a role, such as avoiding excessive data exposure, enabling auditability, and supporting monitoring/observability. For example, the OWASP API Security project[11] has identified a top 10 list of API security risks, especially excessive data exposure (the number 3 on the list) is relevant for this pattern. One of the recommendations is to "review all API responses and adapt them to match what the API consumers really need." This advice is in line with the position taken in this pattern and the related patterns and practices: a domain model element should never be fully exposed in a pass-through manner just because this is technically feasible; the API designers should rather provide views and facades on the required parts of the domain model in the API.

Most of the forces discussed above are an issue for an initial design of an API and of its API clients, but become more and more problematic as the API evolves and thus needs to be maintained [116]. Thus finding good solutions for the other forces is essential for supporting the *Maintainability of API and API Consumers*, too.

Solution

Expose selected AGGREGATE Roots as API endpoints. Mark those elements clearly as domain model elements being exposed as API endpoints.

Solution Details

As an AGGREGATE abstracts the implementation details of a number of related ENTITIES, VALUE OBJECTS, SERVICES, and other domain model elements, it naturally serves as an interface element that can be exposed to an API as an API endpoint.

An AGGREGATE is a cluster of domain model elements such as ENTITIES, VALUE OBJECTS, SERVICES, and so on. It usually contains a root which is one of those elements and which is designated in a model as the aggregate root.

Please note that the term "exposed to" can mean that the AGGREGATE interface is rather literally mapped to the API. It can, however, also mean that a representation of the

[11] See `https://owasp.org/www-project-api-security/`, `https://apisecurity.io/encyclopedia/content/owasp/api3-excessive-data-exposure`

AGGREGATE is provided in the API which contain some changes compared to what is in the DOMAIN MODEL (than rather a REMOTE FACADE [42], maybe as part of a SERVICE LAYER, to the AGGREGATE is offered). For example, different operations or data types can be used in the API representation, if needed. Consider an endpoint offers the possibility to provide a WISH LIST [165] for some API operations. WISH LIST means an API client can provide in the request an enumeration of all desired data elements of the requested endpoint. This can be used to optimize resource usage in the distributed setting for operations that send possibly large amounts of data. That is, in addition to the respective domain operations, additional operations and/or data types can be used to describe the WISH LIST requests which are only present in the API but not in the DOMAIN MODEL.

To be able to apply this pattern, a DOMAIN MODEL must contain suitable AGGREGATES in the first place. If this is not the case and it is perceived as an issue of the DOMAIN MODEL design, one option is to consider redesigning parts of the DOMAIN MODEL. If the DOMAIN MODEL design is of good quality, it might be better to look out for alternative DOMAIN MODEL elements which can be abstracted into API elements. That is, AGGREGATE Roots are a good starting point for API endpoints, but some other options exist: For instance, PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS, described as patterns below, are alternatives that often lead to very good results. In any case, usually deliberate, incremental design is required to find good API endpoints.

Example

To illustrate the pattern let us consider an excerpt of a DDD model of a publication management system [204]. Firstly, in Figure 4.4 a CONTEXT MAP is shown that describes the relations of two BOUNDED CONTEXTS, one for a *Reference Management Service* and one for a *Reference Management Frontend* part of the system. The *Reference Management Service* context's details are shown in Figure 4.8. It contains an AGGREGATE for paper archiving which is composed of a SERVICE for *Paper Archiving*, a *Paper Collection* ENTITY, a *Paper Item* ENTITY, and a *Paper Item Key* VALUE OBJECT.

In Figure 4.9, the *Paper Archive Facade* is exposed to the API as an API endpoint. This is clearly marked in the figure through the *«exposed to API as»* relation to an API endpoint. Please note that for each domain model element that is exposed as selection must be made what actually is exposed. For example, of course, not all attributes of the model element have to be exposed. This means that all members of the *Paper Archive Facade*, which here have an *«exposed to API as»* relation to an *API Element*, are offered as part of this API endpoint. That is, here the API elements *Paper Item DTO* and *Paper Item Key Data Type* are offered as elements of the API in this endpoint.

Pattern Variants and Alternative Practice

PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS, described as patterns below, are alternatives to AGGREGATE ROOTS AS API ENDPOINTS (see [157]). They

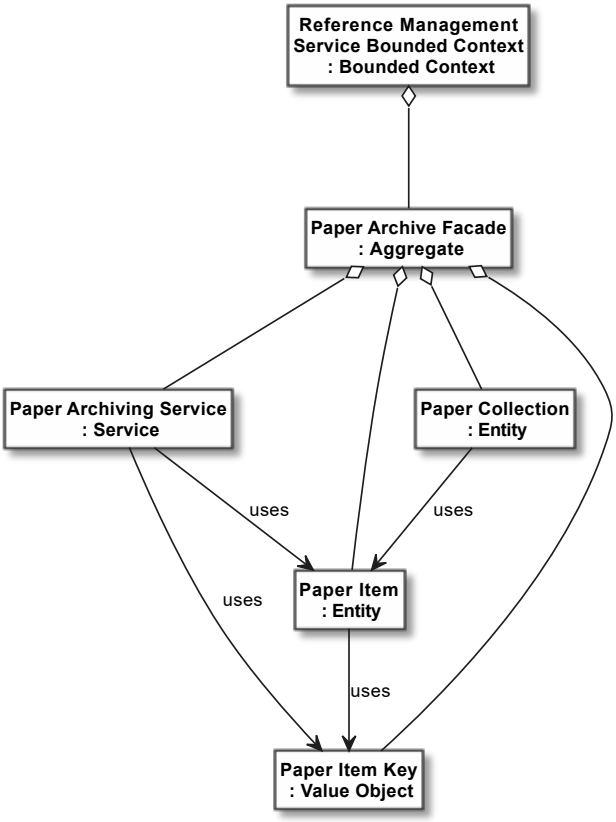

Figure 4.8: Publication Management Example (adapted from [204])

can also be seen as variants of this pattern.

Two alternative practices to this pattern and its related patterns are *Entities as API Endpoints* or *Bounded Context as API Endpoints* as shown in Figure 4.10. *Entities as API Endpoints* means that ENTITIES are offered as API endpoints. They, however, are often too fine-grained in the sense that then simple, shallow ENTITIES with CRUD (Create, Read, Update, Delete) operations are offered in the API. This can lead to the *Data Class* bad smell between distributed API endpoints. According to our practitioner sources, *Entities as API Endpoints* can lead to problems related to API complexity, data consistency, chatty APIs, performance and scalability isses, API understandability, API maintainability, and API evolvability. On the other hand, in some cases, *Entities as API Endpoints* can also lead to well-fitting designs where similar structures are used intentionally (e.g., the use of ENTITIES does not lead to the *Data Class* bad smell but the structurally identical DATA TRANSFER OBJECT pattern [42]). Again, careful and deliberate design is needed to come up with a well-fitting API design.

Instead of an AGGREGATE, *Bounded Context as API Endpoints* is a practice that offers BOUNDED CONTEXTS as composite units offered as API endpoints. But this often is much too coarse-grained leading to too large and complex API endpoints. Again, this is not true for every existing BOUNDED CONTEXT, and if careful and deliberate design uses *Bounded Context as API Endpoints*, it might produce well-designed API endpoints.

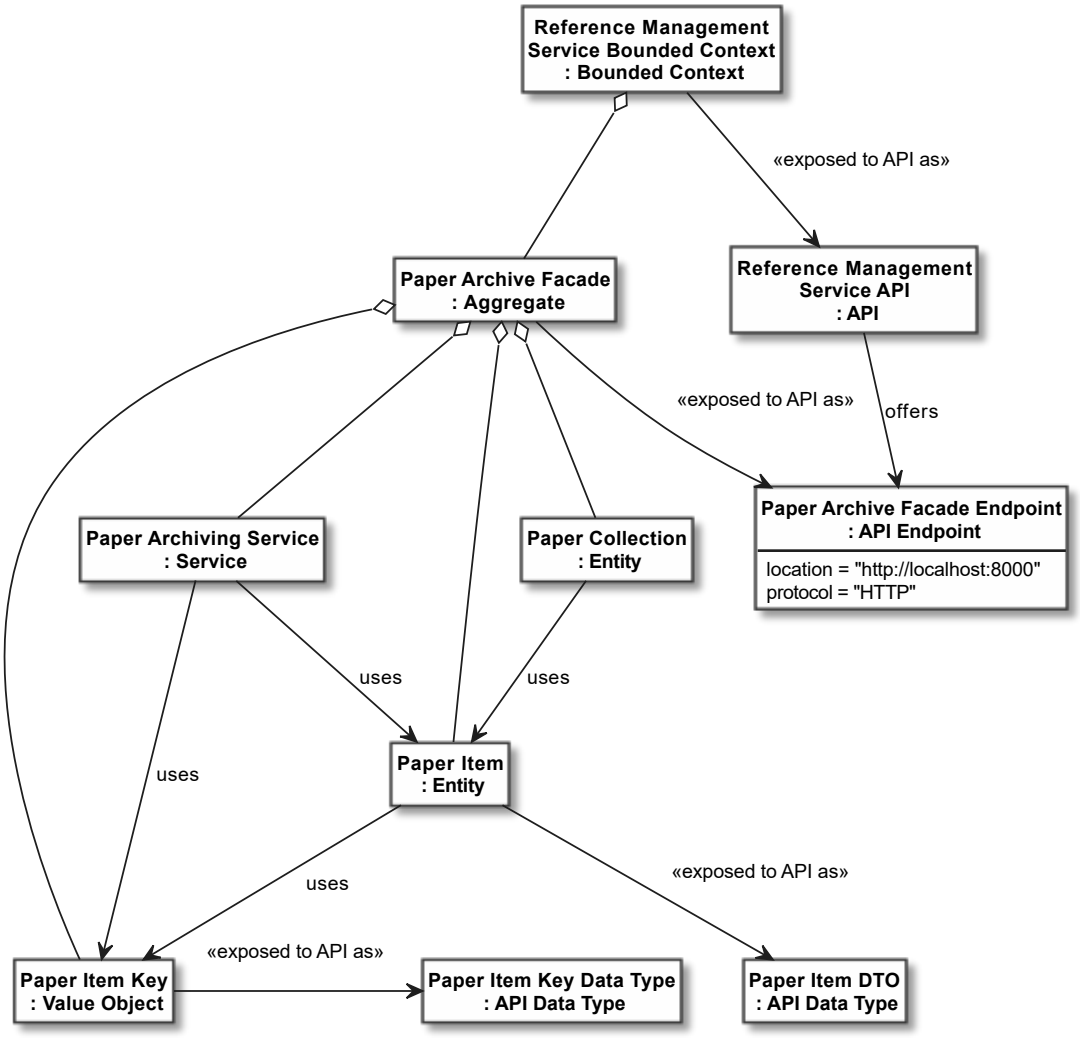

Figure 4.9: Publication Management Example Exposed to the API

Consequences

+ *Level of Domain Model Details Exposed:* The pattern usually provides good results regarding the amount and level of domain model details exposed in the API. AGGREGATES are coarse-grained structures that abstract the implementation details of a number of related ENTITIES, VALUE OBJECTS, SERVICES, and other domain model elements. Thus the AGGREGATES often works well as an endpoint (as also discussed in the small aggregates rule by Vernon [182]), whereas selected members can be mapped to or be represented by API elements, and other members are hidden from the API.

+ *Avoiding Data Consistency Issues:* AGGREGATES are often used as basic elements of data storage and thus it is advised that transactions should not cross AGGREGATE boundaries [12]. Thus using them in this way as API endpoints means that all "local" data consistency issues are handled by the local transactions performed within the AGGREGATE boundary. This avoids unnecessary distributed data consistency han-

[12] See e.g. `https://martinfowler.com/bliki/DDD_Aggregate.html`.

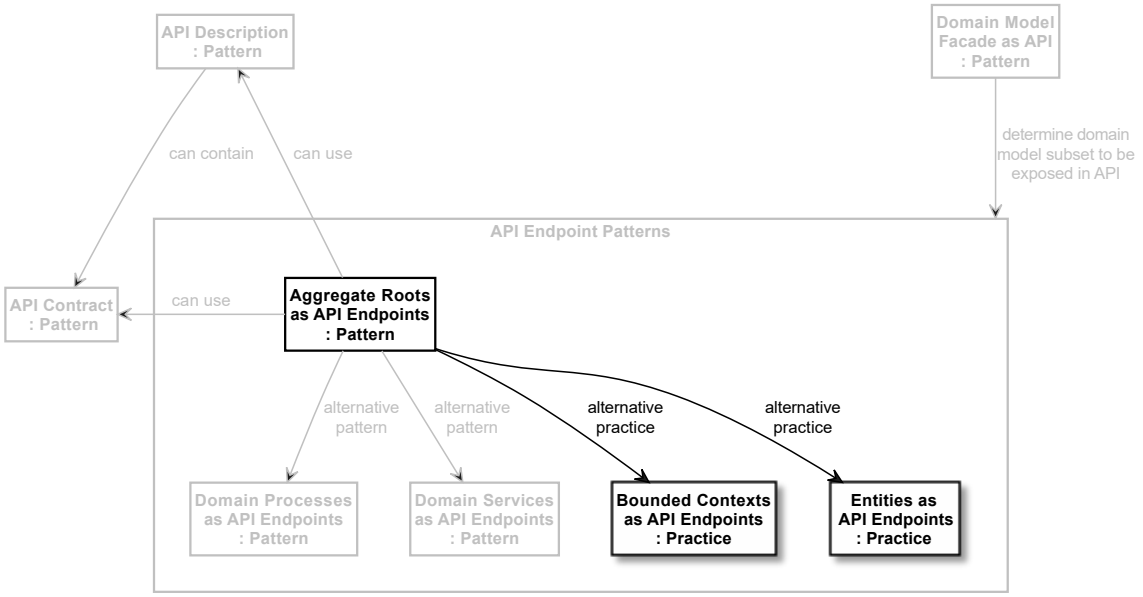

Figure 4.10: Aggregate Roots as API Endpoints – Alternative Practices

dling.

\+ *Performance and Scalability:* Coarser-grained structures enable less *chatty APIs* and thus better performance and scalability than more fragmented structures, where more distributed interactions are required. The Microservice API Patterns offer patterns that might optimize API designs in such situations. For example, the REQUEST BUNDLE [212][13] pattern defines a data container that assembles multiple individual requests in a single request message. The WISH LIST pattern discussed above is another option.

\+ *API Complexity and Coupling:* Coarser-grained structures that abstract details and interactions, such as AGGREGATES, reduce API complexity and coupling of API consumer and supplier.

\+ *Security and Privacy:* This pattern creates a clear mapping between domain model elements and API endpoints, making it easier to trace which domain model elements at a detailed design level are exposed via the API. This can help to avoid excessive exposure of data in domain model elements, as well as fulfill auditability and/or monitoring requirements e.g. for possibly confidential parts of the domain model.

– *Better Suited Domain Model Elements Might Exist:* In some cases the AGGREGATE is not the optimal structure to represent the endpoint. For example, in a AGGREGATE that contains multiple SERVICES or domain processes, sometimes the AGGREGATE is too coarse-grained, and the SERVICES or domain processes are better suited as endpoints. In rare cases, "lonely" ENTITIES exist in domain models that make little

[13] See `https://microservice-api-patterns.org/patterns/quality/dataTransferParsimony/RequestBundle.html`

sense to be included in any of the existing AGGREGATES. Then it might make sense to expose them directly as API endpoints.

– *Issues when Aggregate Boundaries Have to be Crossed:* Some benefits above work especially well as long as the API consumer only depends on one AGGREGATE. If transaction boundaries need to be crossed across AGGREGATES and/or chatty APIs with multiple involved AGGREGATES arise, the chosen AGGREGATE design might not yet be optimal for the purpose of exposing an API. Please note that in general crossing transaction boundaries across AGGREGATES is considered a bad practice in DDD-based design [182][14].

Related Patterns

PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS, described as patterns below, are alternatives to AGGREGATE ROOTS AS API ENDPOINTS that often lead to very good results, according to the practitioner sources in our prior study (see [157]). Less often the practices *Entities as API Endpoints* or *Bounded Context as API Endpoints*, described above may lead to acceptable results, according to the practitioner sources in our prior study (see [157]) as well.

AGGREGATE ROOTS AS API ENDPOINTS (and in the same manner all alternative patterns and practices including PROCESSES AS API ENDPOINTS and DOMAIN SERVICES AS API ENDPOINTS) implies to expose API endpoints as *Application Services* [182]. Vernon [182] (and also Evans [33]) distinguishes *Application Services* from *Domain Services*, i.e. services modeled as domain model elements in the DOMAIN MODEL. The set of those *Application Services* exposed together by an application form a SERVICE LAYER [42]. The SERVICE LAYER defines an application's boundary as an intermediate layer exposing a local, consumer-driven API. The API defines the client-visible interfaces of a subset of these services. For example, an application might define five *Application Services* in its SERVICE LAYER, each with its own endpoint. It might further define two APIs, one for ordinary API consumers and one an administration API; the first API exposes three of the *Application Services* as endpoints, the second one the remaining two. The DOMAIN MODEL FACADE AS API pattern describes how to establish one or more APIs based on the domain model elements to be exposed to an API through such *Application Services*.

AGGREGATE ROOTS AS API ENDPOINTS (and all alternative patterns and practices) can use API CONTRACT or API DESCRIPTION to specify the mapped API design as well as the API endpoints formally.

As discussed above, the Microservice API Patterns [212], such as WISH LIST or REQUEST BUNDLE, can help to optimize the API representation in AGGREGATE ROOTS AS API ENDPOINTS (and all alternative patterns and practices), for example, to improve performance and scalability, as well as avoiding chatty APIs.

[14] See also: `https://socadk.github.io/design-practice-repository/activities/DPR-TacticDDD.html`

Brown et al. [15] suggest to derive API endpoints from ENTITIES and AGGREGATES using the ENDPOINT API pattern [27], i.e., a plain mapping to endpoints following RESTful design principles. Domain processes, in contrast, are covered by the PROCESS API PATTERN, where processes are "nounified" to be mapped to RESTful resources. In all these cases, a mapping to an API endpoint is performed.

Known Uses

- The publication management system [204] used as an example above uses the pattern to select which element makes up the *Paper Archive Facade* endpoint, as shown in Figure 4.9.
- Dugalic [30] discusses a purchase order management system with a Shipping and Order BOUNDED CONTEXTS, which both contain various AGGREGATES. Both BOUNDED CONTEXTS contain "main" AGGREGATES also called Shipping and Order. Each of those is exposed to two endpoints, one command and one query endpoint, to realize the COMMAND QUERY RESPONSIBILITY SEGREGATION (CQRS) pattern [136], two times: two endpoints for a RESTful and two for a gRPC-based API. Lastly, the AGGREGATES each offer one gRPC-based PUBLISH-SUBSCRIBE endpoint for exchanging events, e.g. between the services realizing the endpoints.
- The Eventuate Tram Customers and Orders system[15] exposes some AGGREGATES to an event-driven API. Again, the CQRS pattern is used in the API design, too. Event handler and publisher abstractions are realized in a service that is a wrapper for the AGGREGATES.

Pattern: Domain Services as API Endpoints

We present this pattern here as an extension to AGGREGATE ROOTS AS API ENDPOINTS, explaining only the differences, as the patterns are very similar. In some cases, it might make more sense to consider DDD SERVICES instead of AGGREGATE roots as the foundation for the API endpoints. For example, if one AGGREGATE naturally is composed of multiple services, and no transaction that crosses services boundaries is found in the AGGREGATE, it might be an option to expose each of the services as an endpoint in the API. This makes for instance sense, if the AGGREGATE root based API endpoints are getting very large, and a split based on the SERVICES contained in them is beneficial for API complexity and comprehensibility. In some domain models, designers deliberately use SERVICES for larger decomposition units and not AGGREGATES. Of course, one option is to redesign the model with AGGREGATES, another one is to simply use those modeled SERVICES as basis for API endpoints.

Please note that usually in DDD AGGREGATES and SERVICES are not seen as two alternative concepts, they rather complement each other. For example, a SERVICE can be

[15] `https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders`

the aggregate root in an AGGREGATE. However, we document this pattern here explicitly, as our empirical data indicates that practitioners sometimes only model domain services to derive APIs from them, rather than using the AGGREGATES pattern as well.

If SERVICES are used in this way, as in the cases explained in the previous paragraph, are used as decomposition units very similar to AGGREGATES, the consequences of this pattern are more or less the same as those explained in AGGREGATE ROOTS AS API ENDPOINTS.

For instance, to model the DOMAIN SERVICES AS API ENDPOINTS in the example in Figure 4.9 the *Paper Archiving Service* could have the *«exposed to API as»* relation to the API endpoint instead of the *Paper Archive Facade*. As the two actually exposed domain model elements are used (only) by this SERVICE, likely the same or a very similar endpoint would be realized based on this changed model. However, please note that there is a difference: The *Paper Collection* entity that is used only in the backend is part of the AGGREGATE but not used by the SERVICE. That is, the mapping of this service might lead to problems with transaction boundaries in the future, if *Paper Collection* is part of vital transactions, but it is not visible from the model as it is part of the SERVICE based API endpoint. That is, for the reason of documenting elements in the transactional boundaries, in this particular model, the AGGREGATE root is probably the better choice as a endpoint abstraction.

Please note again that here we discuss *Domain Services* as domain model elements being exposed to an API, as opposed to the *Application Services* (that might form a SERVICE LAYER) which are the actual implementation services realizing the API. See the discussion in the Related Patterns subsection of Section 4.5.

Pattern: Domain Processes as API Endpoints

Alias

Long-running application and domain services as API endpoints.

In a long running business process-like context, for instance claims processing in an insurance form or order management in a shopping and fulfillment logistics scenario, an API operation may realize a single business activity in a business process or even wrap the complete execution of an entire process instance on the provider side. In DDD, the application layer manages process instances and delegates activity execution to the underlying domain layer.

We present this pattern here as an extension to AGGREGATE ROOTS AS API ENDPOINTS, explaining only the differences, as the patterns are very similar. In some cases, it might make more sense to consider domain (or business) processes instead of AGGREGATE roots as the foundation for the API endpoints. Like AGGREGATES, processes aggregate or compose a number of domain model elements, but offer a more step-wise or behavior-oriented view. Processes are especially useful, if the domain experts model and understand their domain in terms of process abstractions. As a domain process would be in DDD terms be modeled as a DDD SERVICE representing the process interface, maybe running on a process engine

in the implementation, this pattern is in its consequences with regard to the API mapping almost identical to DOMAIN SERVICES AS API ENDPOINTS. It is introduced here as a separate pattern, as this might not be obvious to stakeholders who are used to model with process abstractions, as DDD and business process modeling are two different modeling approaches with overlaps.

In the Microservice API Patterns Language, State Creation Operations and State Transition Operations in Processing Resources model these API capabilities and responsibilities (see [206]).

The Process-Driven SOA [60] patterns explain in detail how to map processes to services, both for services realizing the process tasks and application services used for accessing the processes. The DOMAIN PROCESSES AS API ENDPOINTS pattern can be seen as an augmentation of this pattern language making the links to DDD, on the one hand, and API design, on the other hand. Pautasso and Wilde [115] present an approach to map business processes onto RESTful push services so that business processes can be modeled and observed in a RESTful way. This is one possible way to design a RESTful API when applying the DOMAIN PROCESSES AS API ENDPOINTS pattern.

4.6 Patterns on API Operation Design Derived from DDD Designs

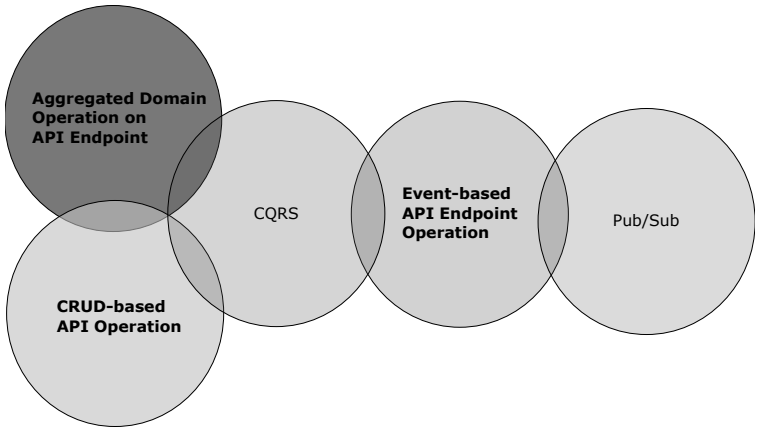

Figure 4.11: High-level overview of the patterns

The patterns presented in this section are summarized in a high-level overview in Figure 4.11. The patterns introduced in this chapter are high-lighted in bold-face font. They are related to two existing patterns CQRS and PUBLISH/SUBSCRIBE [17]. The overlaps of the patterns define the pattern variants described below. In Figure 4.12 the detailed relations of these patterns and patterns variants are shown.

The AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern describes the practice of exposing aggregated or abstracted domain operations on the API endpoints. An alternative is EVENT-BASED API ENDPOINT OPERATION which aims to expose DOMAIN EVENTS or abstracted DOMAIN EVENTS as state transition operations on the API endpoint. Alter-

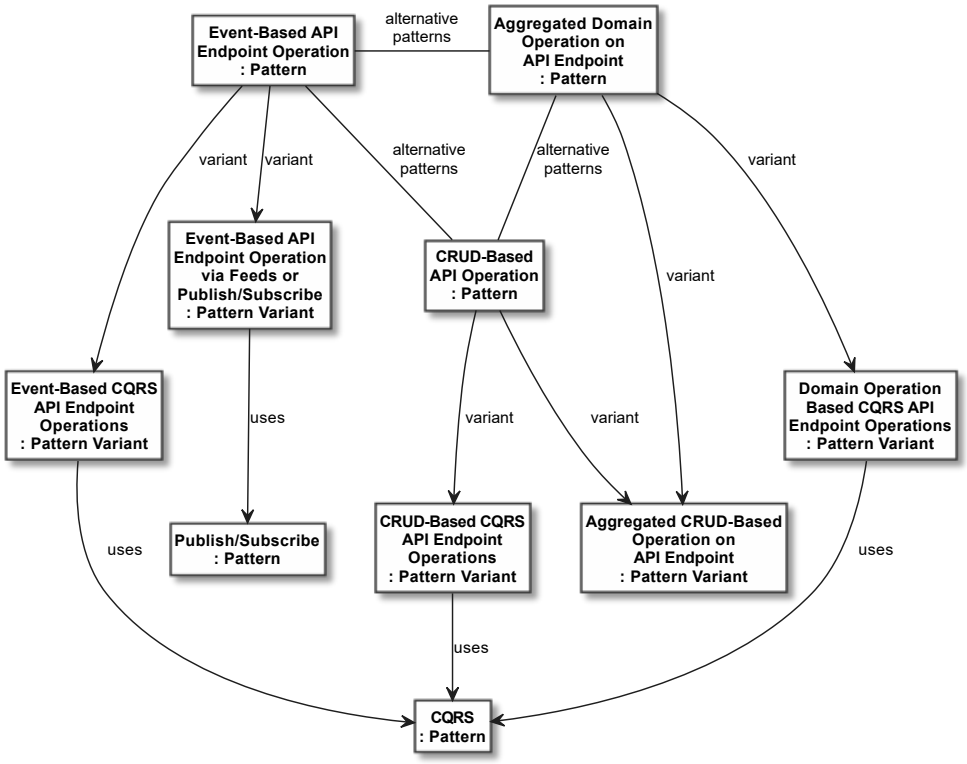

Figure 4.12: Overview of the patterns and their relations

natively, in its variant *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* DOMAIN EVENTS or abstracted DOMAIN EVENTS are offered via a PUBLISH/SUBSCRIBE [17] architecture or event feeds instead.

Finally, the CRUD-BASED API OPERATION pattern describes the practice to design operations on API endpoints based on the well-known *Create, Read, Update, and Delete (CRUD)* primitives. As this pattern can generate negative impact on many of its forces, when overly or wrongly applied, it shall be applied with care and only where CRUD operations are really needed by clients. Sometimes AGGREGATED DOMAIN OPERATION ON API ENDPOINT can be CRUD-based, too. Then, the variant of the two pattern *Aggregated CRUD-Based Operation on API Endpoint* can be applied.

All three patterns have a variant which combines the respective pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [136]. CQRS advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. The application of CQRS usually has consequences for the API operation designs, as in the backend the views are then only eventually consistent with actions offered by the commands. That is, transactional or data consistency boundaries do not exist anymore or require further measures, and clients need to be aware of those consequences. Also, additional or other API operations might be required, e.g. API operations for compensation actions should a "transaction" fail.

All patterns described here require deriving an API endpoint from the DOMAIN MODEL first. For this we have in our prior work mined a number of patterns: AGGREGATE ROOTS AS API ENDPOINTS [158] starts out with AGGREGATES and their roots to derive API end-

point. DOMAIN SERVICES AS API ENDPOINTS [158] and DOMAIN PROCESSES AS API ENDPOINTS [158] use domain SERVICES or processes as starting points instead. Those three options are advised most strongly by practitioners. If BOUNDED CONTEXTS or ENTITIES fit well as API boundaries, for example, they are not too large or small for the API, and cover one design concern completely, it makes sense to consider ENTITIES AS API ENDPOINTS [158] or BOUNDED CONTEXTS AS API ENDPOINTS [158], too.

Aggregated Domain Operation on API Endpoint

Context

In a software development project, you use DDD to design your DOMAIN MODEL and want to design the operations on the endpoints of an API for this project. Consider further that your DOMAIN MODEL is designed in detail, especially, it is modeling operations on the DOMAIN MODEL elements.

Problem

How to design the operations of an API endpoint in relation to the methods modeled on DOMAIN MODEL elements?

Forces

- *Avoid Exposing Domain Model Details in API*: In the design of API operations, we want to expose the domain model details that are needed by clients to work, but no more. Any other aspects exposed might reveal more domain model details (and thus details about the backend implementation) to clients than necessary. This could e.g. lead to unnecessary coupling.

- *Avoid Interface Design that Limits Domain Model Design*: In DDD, a DOMAIN MODEL is the core model and the basis for the UBIQUITOUS LANGUAGE of a software project. Thus, it shall not be limited in its design because of interface design considerations. For instance, consider a project starts with a simple prototype of an API operation design exposing shallow resources in CRUD style only, which sometimes happens e.g. when naively designing RESTful HTTP resource, as the HTTP verbs POST, GET, PUT, PATCH, and DELETE seems to imply this interaction style. If then the DOMAIN MODEL is designed e.g. based on those resources, it is likely that at least parts of the DOMAIN MODEL are anemic as a result.

- *API Understandability*: At the operation level, the number and complexity of operations and the abstractions they use, influence the API understandability. This is determined by how the DOMAIN MODEL operations are mapped to API operations. Also, it is important for easing the understanding of the API by client developers that the operations are meaningful in the scope clients require. API implementation

developers require a well-understandable mapping to the DOMAIN MODEL to understand the API in the context of the backend realization which is based on the DOMAIN MODEL.

- *Coupling of Clients to Server*: The coupling of clients and servers is at the API operation level determined by the number and kinds of dependencies from clients to the operations exposed by the API. Less and less tightly coupled links or connectors (i.e., if possible, asynchronous links and indirect links such as those in the PUBLISH/-SUBSCRIBE pattern [17]) should be preferred. This needs to decided based on the semantics of the links in the DOMAIN MODEL which are mapped to API links. For example, eventually consistent API links are usually less tightly coupled, but might make not much sense, if the DOMAIN MODEL abstractions demand a transactional relation.

- *Maintainability of API and API Consumers*: More exposed elements of the API than necessary, more dependencies than needed, and higher coupling all can lead to negative impact on various maintainability aspects of API and API consumer, such as modular APIs, modular API clients, as well as independent testability, modifiability, extensibility, and analyzability. At the operation level, this is determined by which DOMAIN MODEL operations, links, and data elements are exposed in the API.

- *Chatty API, Performance, and Scalability*: Too many or too fine-grained operations which need to be invoked in combination by API consumers to achieve their goals will lead to many distributed calls. If substantially more calls are exchanged than necessary, an API is called a chatty API. This should be avoided by aggregating fine-grained operations together so that the number of invocations and the length of client conversations is reduced. For example, if DOMAIN MODEL operations on AGGREGATES are exposed, instead of the operations of all members of the AGGREGATE, this usually helps to avoid chatty APIs.

  More generally speaking, at API operation level, performance and scalability (in terms of requests per client, number of concurrently served requests, or computational load caused by those requests) are important forces and can be positively influenced by designing operations in such a way that unnecessary distributed calls are avoided. As discussed above, careful selection of what is exposed from the DOMAIN MODEL can help here, too, but the general impacts of design options on performance and scalability might be less obvious than just observing chatty APIs, and thus require prototyping and measurement. Also, it should be carefully considered how to expose those operations; for example, exposing many unnecessary data elements in the API operations' payload, can reduce the API provider's response behavior.

  Further, the repeated calling of operations can be limited, e.g. by using patterns such as API RATE LIMITING [212, 165]. API RATE LIMITING suggests to introduce rate limits

per client, e.g. with the goal to avoid abusive clients being able to overload the system. All this requires careful investigation of the DOMAIN MODEL.

- *API Operations Reuse*: Optimizing API operation design for some of the forces properties above can mean to deviate from the DOMAIN MODEL operation design. For example, by offering multiple operations for different clients, each with different performance and scalability characteristics, means to offer multiple operations per corresponding DOMAIN MODEL operation. This might again make it hard to understand the API well in relation to the DOMAIN MODEL, but also it reduces the opportunity for reuse. There is a trade-off between reuse of operations shared by different clients and how many operations are provided overall.

- *Data Consistency*: Operations usually process data. It is important to consider data consistency measures such as transaction boundaries and eventual consistency in the backend when designing API operations. It can be hard to manage data consistency correctly if the API consumer is responsible for ensuring data consistency guarantees. In general, all data consistency requirements stem from the DOMAIN MODEL. Here, also abstractions such as AGGREGATES are used to indicate transactional or consistency boundaries. API operation design should, if possible, adhere to those boundaries.

Solution

Primarily expose operations that represent abstractions of the details in the DOMAIN MODEL, exposing only the DOMAIN MODEL elements required by clients and nothing more. To reach this, from the set of all domain operations modeled for the Domain Model elements to be exposed to the API, select the subset that can directly be traced back to client use cases and use this subset as a starting point for designing the API operations. Aim to offer more coarse-grained or aggregated operations on the API, rather than exposing every fine-grained operation, to avoid bloating the API with unnecessary operations which make it harder to understand and would increase coupling between API and backend implementation.

This might lead to designing API operations on coarser-grained API elements such as those representing AGGREGATES, BOUNDED CONTEXTS, or DOMAIN SERVICES, rather than those representing finer-grained API elements such as ENTITIES or VALUE OBJECTS. It might also lead to designing API operations that abstract or aggregate aspects covered in a number of detailed domain operations.

Solution Details

Our patterns on deriving API endpoints from DOMAIN MODEL elements advise us to use AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, or DOMAIN PROCESSES AS API ENDPOINTS, which are all coarse-grained DOMAIN MODEL elements abstracting from other DOMAIN MODEL elements such as ENTITIES or VALUE OBJECTS. If not

of those abstractions is modeled (e.g., the model contains no AGGREGATES) or their scope does not fit well to the scope of the API to be designed (e.g., to small or to large to be easy to understand as an API), next BOUNDED CONTEXTS AS API ENDPOINTS shall be considered, which are typically yet coarser-grained structures. It is also possible to expose ENTITIES AS API ENDPOINTS but it is advised to expose them with caution because an API design where every ENTITY (or VALUE OBJECT) is exposed to the API can suffer from negative consequences. For instance, not required domain model details might be exposed through the API, which makes the API hard to understand and change, raise its complexity, introduce unnecessary coupling, and so on. This would also lead to chatty APIs, with possibly low performance and bad scalability. Those choices should be reflected at the operation level by exposing primarily operations on those coarser-grained structures. Our operation level is not only focusing on DDD and how to implement them but it is considering other architectural elements. This is why the core idea of the AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern is to primarily expose operations that represent abstractions of the details in the DOMAIN MODEL, exposing only those DOMAIN MODEL elements required by clients and nothing more.

In many protocols there is a clear-cut way how to encode the domain operations. For instance, in gRPC usually the operation would be directly encoded using the means of the protocol, typically with the same operation name as in the DOMAIN MODEL. RESTful HTTP is different here, as it usually advises us to use HTTP operations (such as POST, GET, PUT, PATCH, and DELETE) instead. Thus a mapping between domain operations and HTTP operations is required. The downsides of introducing such mappings can be an additional level of indirection, the need for maintenance of the mapping during evolution, keeping backwards compatibility, and so on. Then it might make sense to consider other implementation options such as to *encode operations as commands in the payload*. For example, this is sometimes used to encode various possible actions flexibly in one HTTP verb, e.g. actions such as `rename` are not encoded in the protocol or URL, but in the payload instead. Please note that there are also many extant APIs in which use *actions encoded in the URL* (e.g. as in "*POST /rename*"). Please note that this practice is similar to encoding commands in the payload, but it is usually not seen as a recommended practice in RESTful HTTP. That is, RESTful HTTP guidelines often advice to use the only the HTTP verbs as actions and not encode actions in URLs.

While all such practices of circumventing RESTful conventions (encoding operations in the payload or the URL) are debatable, there are many other reasons, why encoding operations in the payload might make sense (and here encoding in the payload usually makes indeed more sense than encoding in the URL). For example, this practice can be used to encode a complex operation consisting possibly of many atomic operations. Consider realizing a REQUEST BUNDLE [212] in which multiple requests are grouped and sent together in one request. This could be realized by encoding the commands required for the individual requests in the payload of the request bundle operation.

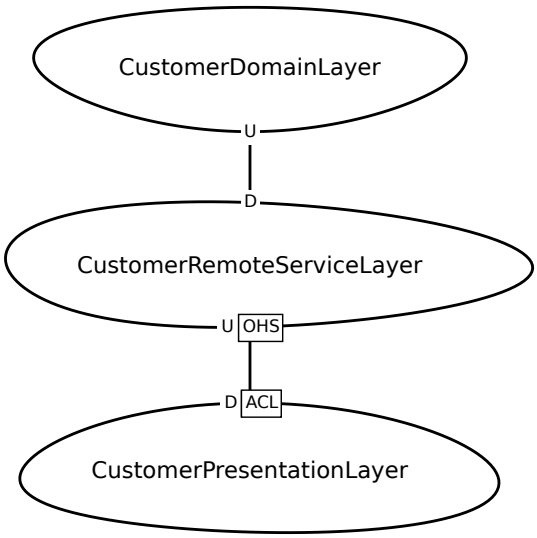

Figure 4.13: Context Map for the Customer Management Example

Example from Customer Self-Service API of the Lakeside Mutual Project

The following code shows a number of operations exposed via RESTful HTTP in the *Customer Self-Service* API of the Lakeside Mutual open source project[16]. *Customer Self-Service* is an AGGREGATE which is offered as an API to clients, and, as can be seen, it aggregates various elements of the DOMAIN MODEL, such as those necessary for handling customer details or insurance quote requests. Instead of exposing detailed CRUD-based operations on each of the respective ENTITIES, the AGGREGATE-based endpoint offers only those coarse-grained domain operations which are required by clients for self-service. Please note that some of the operations perform abstracted CRUD-BASED API OPERATION; that is, those operations realize the *Aggregated CRUD-Based Operation on API Endpoint* pattern variant of this pattern.

Please note that the exposed functions realize an abstraction in the scope of the DOMAIN MODEL which is meaningful to the client ("customer self-service") as opposed to offering the detailed abstractions realizing the backend (aka the full DOMAIN MODEL).

```
export function getUserDetails(token: string): Promise<User> {
  const url = urlForEndpoint("/user")
  return getAuthenticatedJson(url, token)
}

export function getCustomer(
token: string,
customerId: CustomerId
): Promise<Customer> {
  const url = urlForEndpoint(`/customers/${customerId}`)
  return getAuthenticatedJson(url, token)
}

export function changeAddress(
```

[16] https://github.com/Microservice-API-Patterns/LakesideMutual

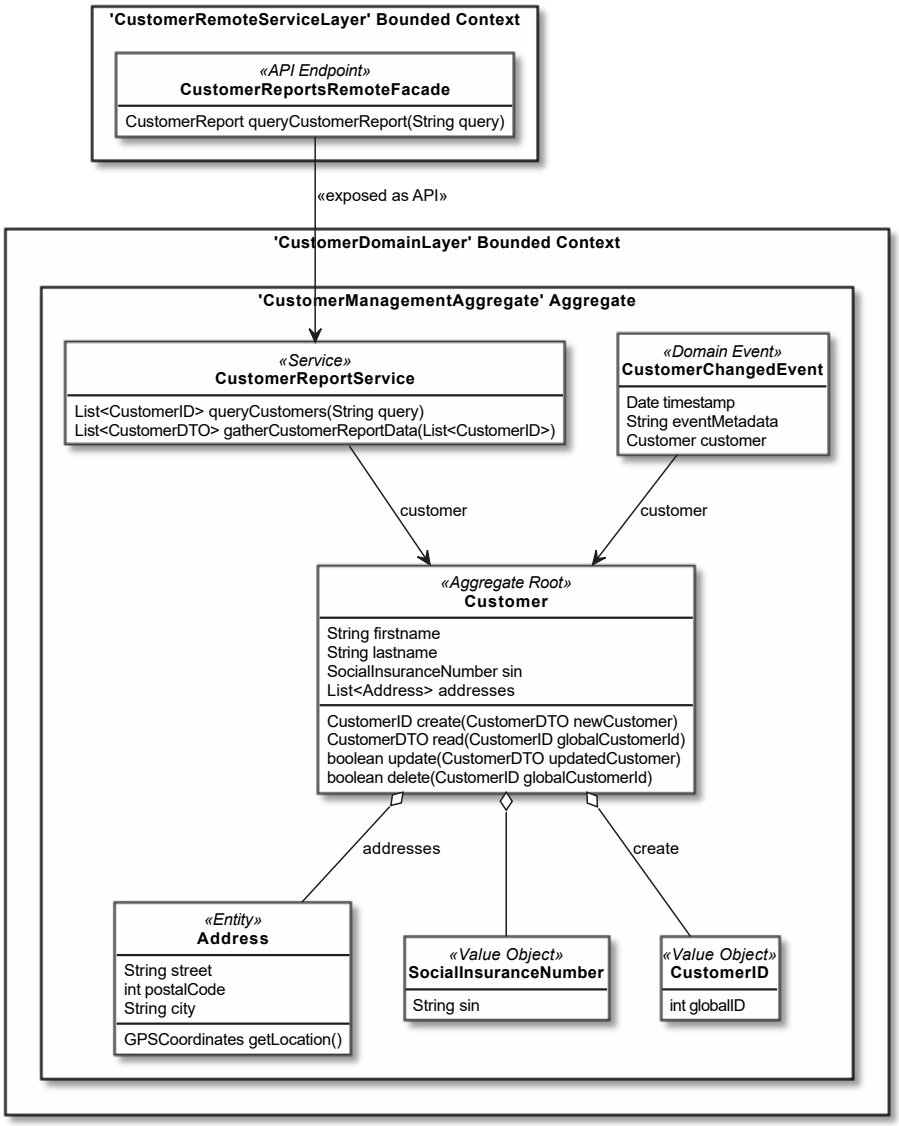

Figure 4.14: Aggregated Domain API Operation for Querying Customer Reports

```
token: string,
customer: Customer,
address: Address
): Promise<Address> {
  // Instead of assembling the URL to change the address ourselves,
  // we use the one provided in the customer response.
  const url = customer._links["address.change"].href
  return putAuthenticatedJson(url, token, address)
}

export function completeRegistration<T>(
token: string,
data: T
): Promise<Customer> {
  const url = urlForEndpoint("/customers")
  return postAuthenticatedJson(url, token, data)
```

```
}

export function createInsuranceQuoteRequest(
token: string,
data: InsuranceQuoteRequest
): Promise<InsuranceQuoteRequest> {
  const url = urlForEndpoint("/insurance-quote-requests")
  return postAuthenticatedJson(url, token, data)
}

export function getInsuranceQuoteRequests(
token: string,
customerId: CustomerId
): Promise<[InsuranceQuoteRequest]> {
  const url =
      urlForEndpoint(`/customers/${customerId}/insurance-quote-requests`)
  return getAuthenticatedJson(url, token)
}

export function getInsuranceQuoteRequest(
token: string,
id: string
): Promise<InsuranceQuoteRequest> {
  const url = urlForEndpoint(`/insurance-quote-requests/${id}`)
  return getAuthenticatedJson(url, token)
}
```

Please note that in this code, helper functions such as postAuthenticatedJson are called, which handle the actual construction of the responses in a Promise:

```
export async function postAuthenticatedJson<T, U>(
url: string,
token: string,
params: T
): Promise<U> {
  const response = await fetch(url, {
    method: "POST",
    headers: {
      "X-Auth-Token": token,
      "Content-Type": "application/json",
      Accept: "application/json",
    },
    body: JSON.stringify(params),
  })
  return checkStatus(response)
}
```

Pattern Variants

There are three variants of the pattern:

- *Aggregated CRUD-Based Operation on API Endpoint:* In the CRUD-BASED API OPERATIONS pattern, we mainly discuss fine-grained CRUD-BASED API OPERATIONS. AGGREGATED DOMAIN OPERATION ON API ENDPOINT in contrast focus on the notion to offer domain operations on aggregated or coarse-grained DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS. Having CRUD-BASED API OPERATIONS on those can alleviate a couple of possible risks and drawbacks mentioned in the consequences of CRUD-BASED API OPERATIONS, especially those that are related to the fine-grained nature of CRUD-BASED API OPERATIONS. The example for this pattern above contains some samples of this pattern variant. This variant is applicable when "aggregating" DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS contain CRUD-like operations in the DOMAIN MODEL.

- *Domain Operation Based CQRS API Endpoint Operations:* The CQRS (*Command Query Responsibility Segregation*) pattern [136] advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. Both APIs can use aggregated domain operations, as explained in this pattern. For instance, in the example above, we would simply need to segregate the GET operations from the PUT and POST operations to create the two views. This variant is applicable when this pattern needs to be combined with CQRS.

Consequences

+ *Avoid Exposing Domain Model Details in API* : Only selected domain operations that are actually needed for client interactions are offered via the API. Other details of the DOMAIN MODEL remain hidden.

+ *Chatty API, Performance, and Scalability*: The aggregated nature of AGGREGATED DOMAIN OPERATION ON API ENDPOINT means bundling of operations and reduction of the total number of client/service interactions, which can help to avoid chatty APIs. In the same sense, they can be designed in a way optimized for performance and scalability. For example, this could be achieved by avoiding unnecessary distributed operations or introducing measures that reduce the number of requests such as REQUEST BUNDLING [212] (explained briefly above). In addition to avoiding unnecessary distributed operation calls, it is possible to introduce further measures to reduce the maximum requests that need to be served at a time. For example, smaller rate limits in API RATE LIMITING [212, 165] (a pattern introducing rate limits per client, e.g. with the goal to avoid abusive clients being able to overload the system) can lead to a lower number of total request that need to be served at a particular point in time.

+ *Avoid Interface Design that Limits Domain Model Design*: As explained above, this pattern tends to enable API designers to provide an abstraction in the scope of the

DOMAIN MODEL which is meaningful to the client as opposed to offering the detailed abstractions realizing the backend (aka the full DOMAIN MODEL).

\+ *Maintainability of API and API Consumers*: Lowering the number of possible dependencies (as suggested by this pattern in comparison to e.g. only CRUD-BASED API OPERATIONS) and thus reducing the complexity on operation level, helps to improve maintainability properties, such as modularity of the API and its consumers, as well as more independent testability, modifiability, and analyzability.

\+ *Data Consistency*: Aggregated operations are often designed with transaction boundaries in mind. For example, if an operation on an AGGREGATE performs a transaction on a couple of ENTITIES in its scope, this is one data consistency issue that is solely handled in the backend and thus not a possible issue at the API level anymore.

\+ *API Understandability*: This pattern tends to reduce the number of API operations (compared to using only CRUD-BASED API OPERATIONS for instance). Those API operations can be designed closer to the scope required by the clients. Those measures can improve the understandability of the API.

+/- *Coupling of Clients to Server*: On the one hand, e.g. compared to CRUD-BASED API OPERATIONS, more coarse-grained or aggregated operations can reduce number of links that introduce coupling. So overall this pattern is beneficial for coupling in many cases. On the other hand, compared to EVENT-BASED API ENDPOINT OPERATION, especially if event abstractions are offered to the client, as in the *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* variant, there is still a substantial coupling at the operation-level possible. This can especially cause problems is operations are invoked synchronously, and thus can be avoid through asynchronous operation calls to a large extent.

Related Patterns

As explained, the two patterns CRUD-BASED API OPERATIONS and EVENT-BASED API ENDPOINT OPERATION are alternatives to this pattern. Whereas CRUD-BASED API OPERATIONS is in many cases not an advisable alternative, its variant *Aggregated CRUD-Based Operation on API Endpoint* which combines CRUD-BASED API OPERATIONS with this pattern, usually is a possible option.

As explained in the Solution Details section, ideally, this pattern is placed on a endpoint derived using one of the following patterns: AGGREGATE ROOTS AS API ENDPOINTS [158], DOMAIN SERVICES AS API ENDPOINTS [158], or DOMAIN PROCESSES AS API ENDPOINTS [158]. Sometimes it makes sense to combine it with the ENTITIES AS API ENDPOINTS [158] and BOUNDED CONTEXTS AS API ENDPOINTS [158] practices, too.

The *encode operations as commands in the payload* implementation option can be used to realize patterns such as REQUEST BUNDLE [212] which require complex specifications of

the request. Such patterns can then improve performance. Scalability can be improved using patterns such as API RATE LIMITING [212, 165].

The *Domain Operation Based CQRS API Endpoint Operations* variant combines this pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [136].

Known Uses

- The *Lakeside Mutual* open source system used in the example above uses a number of AGGREGATE-based endpoints which offer domain operations which are aggregating aspects of the DOMAIN MODEL elements in their AGGREGATE scopes.

- The publication management demo case for MDSL[17] uses a paper archive facade AGGREGATE which offers coarse-grained operations such as *Lookup Paper from Author*, *Create Paper Item*, and *Convert to Markdown for Website* on its aggregate root *Paper Archive Service*. Please note that one of those operations is an abstracted CRUD-BASED API OPERATION (*Create Paper Item*) which uses a concrete implementation on the *Paper Collection Backend* ENTITY.

- The Cinema Microservices[18] open source system uses aggregated domain operations on DOMAIN SERVICES AS API ENDPOINTS. For instance, its *Payment* service exposes domain operations such as *makePurchase* or *getPurchaseByID* as RESTful HTTP operations as shown in Figure 4.15.

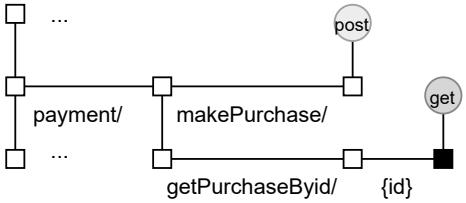

Figure 4.15: API Tree[19]of Payment Endpoints in the Cinema Microservices system

- The Online Shop example of the Design Practice Repository [20] exposes various AGGREGATE root SERVICE as DOMAIN SERVICES AS API ENDPOINTS. They then offer aggregate operations. For example, the *Browse and Buy* service is the AGGREGATE root on the *Business to Consumer* AGGREGATE. It offers aggregate API operations such as *Create Order Item*, *Create Order*, and *Read Product Information*. Please note that all of those are abstracted CRUD-BASED API OPERATIONS which require implementations on the members of the AGGREGATE.

[17] `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html`

[18] `https://github.com/Crizstian/cinema-microservice`

[19] The used API tree notation: ◯ = The HTTP methods, where each method has a specific color (GET method is green, POST method is yellow, PUT method is blue, PATCH method is gray, DELETE method is red), □ = Path segment with no parameter, ■ = Path segment with single parameter. For more details on the API tree diagrams see [150].

[20] `https://github.com/socadk/design-practice-repository/blob/master/tutorials/DPR-Tutorial1.md`

Pattern: Event-Based API Endpoint Operation

Context

Many microservice systems are event-based systems. Consider further that in a software development project, you use DDD to design your DOMAIN MODEL which can include DOMAIN EVENTS.

Problem

How to design the operations of an API endpoint in relation to DOMAIN EVENTS in the DOMAIN MODEL?

DOMAIN EVENTS are usually modeled as domain methods in DOMAIN MODEL, and consequently API operations in the API, that handle event emission and event reception. That is, the pattern is best applicable to design problems where the domain problem can be modeled in terms of event emissions and receptions.

Forces

This pattern has the same forces as discussed for its alternative pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT. The idea here is that the forces to be considered are essentially the same, even though they need to be considered in the context of DOMAIN EVENTS, but the consequences on them are different. For example, Loose Coupling or Avoiding Exposing Domain Model Details in APIs are forces in both cases, but the effects on the forces (aka the consequences) are fundamentally different.

Solution

Select a subset from the DOMAIN EVENTS and related technical events in the scope of an API endpoint that API clients have a need to know about, as incoming and/or outgoing events. Consider to design API-level events at a more coarse-grained or higher abstraction level than the system-internal events. For outgoing events, provide ways for clients to get notified or be able to poll for events in the API. For events incoming from the client, expose those events as STATE TRANSITION OPERATIONS[21] on the API endpoint, or provide ways with which the API can get notified about or poll for client-side events.

Solution Details

Overall, this pattern is in a number of ways similar to its alternative pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT. The main difference is that the operations exchanged as supposed to be part of an event-driven architecture. That can actually refer to different event-driven architecture patterns [45], but here we usually assume one or both of the following are realized by the API operations: Often event-driven architecture refers to a

[21] A STATE TRANSITION OPERATIONS [206, 212] is an operation on an API endpoint that combines client input and current state to trigger a provider-side state change f: (in,S) -> (out,S').

use of EVENT NOTIFICATIONS, i.e. "a system sends event messages to notify other systems of a change in its domain [45]." It can also refer to EVENT-CARRIED STATE TRANSFER, i.e. "clients of a system get updated in such a way that they do not need to contact the source system in order to do further work [45]." The EVENT-BASED API ENDPOINT OPERATION explains how to realize such event-driven architectures as distributed systems with the API operations realizing the incoming and outgoing event messages.

For the domain model elements in scope of your API endpoint, such as an AGGREGATE root and the scope it represents, select those DOMAIN EVENTS that should be exposed to clients. For example, in a shopping application, DOMAIN EVENTS such as "order item added to a shopping cart" or "order finished" exist and are required for clients to work properly. Thus, they should be exposed to clients as outgoing operations informing clients about system events or incoming operations with which clients can raise system events. In contrast, system-internal DOMAIN EVENTS not needed by clients shall not be exposed via the API. Consider for instance a client calls an event-based operation signaling that an item has been added to a shopping cart on client side. This operation is invoked on the Cart service, and the client gets a confirmation that the call was received via the used communication protocol. The Cart service, after having performed the required actions to update the cart, might raise a "shopping cart updated" event directed to other backend services. The client, who has initiated the update and thus knows of the update already (and also has a confirmation that it was received), does not need to get informed about this system-internal event.

Next, you should relate this set of exposed DOMAIN EVENTS in a state transition model and identify technical gaps in those state transitions. That is, there might be steps required for the API to work that are not yet exposed to the API. If this is the case, augment the set of exposed DOMAIN EVENTS with the additional, technical events required for the API to work. For example, before a shopping cart can be filled with order items, it might be necessary to create the cart or create a user session. Those events might not be modeled in the DOMAIN MODEL, but nonetheless they are needed to support all possible client interactions.

Example of Customer Changed Event Operations Exposed to the API

Figure 4.16 extends the minimal example from Figure 4.14. It uses the same DOMAIN MODEL, but highlights different operations exposed to the API: One is an event retrieval operation called *emitEvent*, and the other one is the corresponding event processor called *receiveEvent*. Both use and abstract from the *CustomerChangedEvent* domain event in the DOMAIN MODEL.

Example of Event-Based Operations in an EShop System

To illustrate the pattern consider an example from the *eShopOnContainers* open source system[22]. This system has a *UserCheckoutAccepted* DOMAIN EVENT which is triggered by a

[22] `https://github.com/dotnet-architecture/eShopOnContainers`

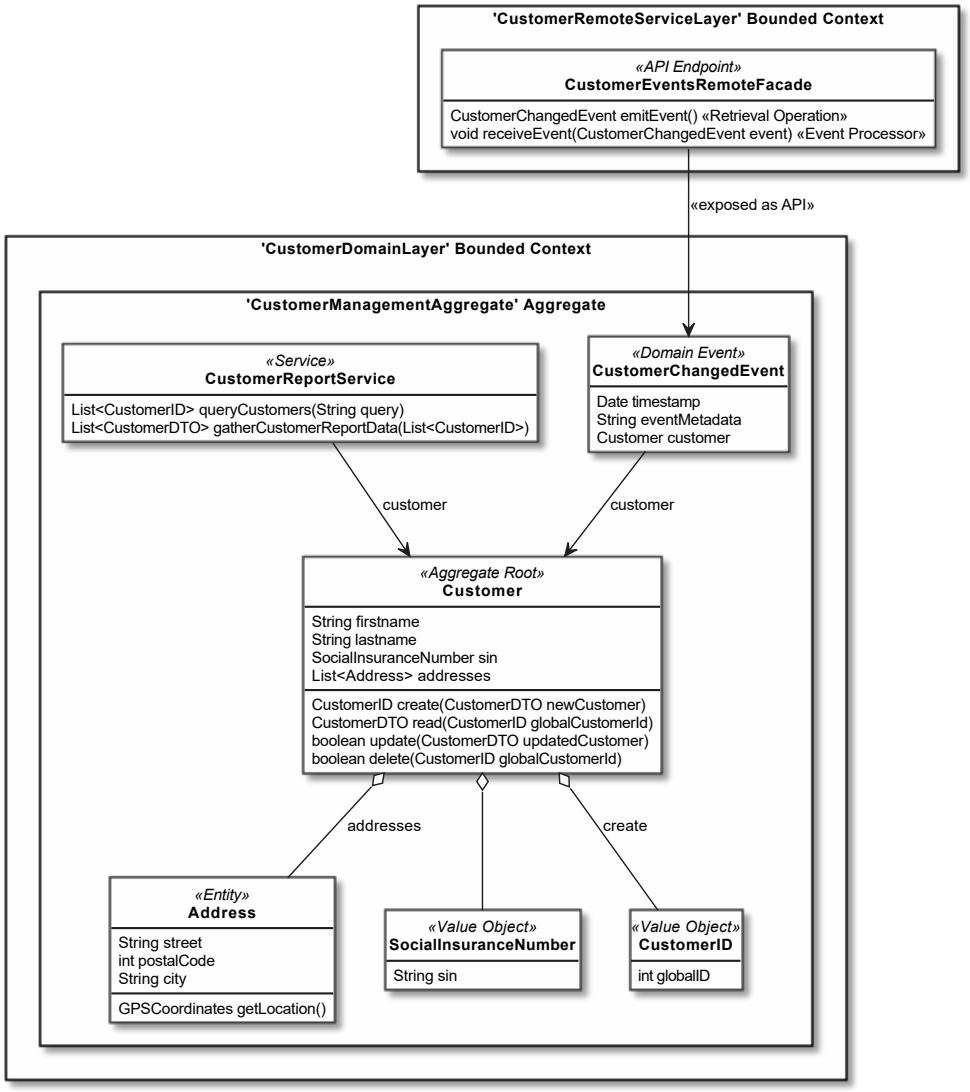

Figure 4.16: Event-Based API Endpoint Operations for a Customer Changed Event

corresponding operation CheckoutAsync() on the *Basket* API, provided via an HTTP POST on the route checkout. Internally, the UserCheckoutAcceptedIntegrationEvent is raised to signal the event created by calling the API operation to other backend microservices. Integration is performed using an Event Bus based PUBLISH/SUBSCRIBE architecture [17].

```
[Route("checkout")]
[HttpPost]
[ProducesResponseType((int)HttpStatusCode.Accepted)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
public async Task<ActionResult> CheckoutAsync([FromBody] BasketCheckout
    basketCheckout,
[FromHeader(Name = "x-requestid")] string requestId)
{
  var userId = _identityService.GetUserIdentity();

  basketCheckout.RequestId =
  (Guid.TryParse(requestId, out Guid guid) && guid != Guid.Empty) ?
```

```
    guid : basketCheckout.RequestId;

    var basket = await _repository.GetBasketAsync(userId);

    if (basket == null)
    {
      return BadRequest();
    }

    var userName = this.HttpContext.User.FindFirst(x => x.Type ==
        ClaimTypes.Name).Value;

    var eventMessage = new UserCheckoutAcceptedIntegrationEvent(userId,
        userName,
    basketCheckout.City, basketCheckout.Street, basketCheckout.State,
    basketCheckout.Country, basketCheckout.ZipCode,
        basketCheckout.CardNumber,
    basketCheckout.CardHolderName, basketCheckout.CardExpiration,
    basketCheckout.CardSecurityNumber, basketCheckout.CardTypeId,
    basketCheckout.Buyer, basketCheckout.RequestId, basket);

    // Once basket is checkout, sends an integration event to
    // ordering.api to convert basket to order and proceeds with
    // order creation process
    try
    {
      _eventBus.Publish(eventMessage);
    }
    catch (Exception ex)
    {
      _logger.LogError(ex,
      "ERROR Publishing integration event: {IntegrationEventId} from
          {AppName}",
      eventMessage.Id, Program.AppName);

      throw;
    }

    return Accepted();
  }
```

Pattern Variants

There are two variants of the pattern:

- *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe:* In the example above, we have seen how a PUBLISH/SUBSCRIBE architecture can be used to signal events internally, here to other microservices realizing the API. An option is to expose the PUBLISH/SUBSCRIBE architecture or event feeds directly to clients. That is,

clients publish and/or subscribe to events via the API using the internally used PUBLISH/SUBSCRIBE architecture or event feeds. If both clients and system are developed together and shall both follow an event-based architecture, this abstraction simplifies the client/system interactions as one common infrastructure is used. Very often this architecture is chosen for SOLUTION INTERNAL APIS [212]. In PUBLIC APIS [212] it is often not acceptable to expose the internally used infrastructure and assume clients are able or willing to use it. For example, if internally a PUBLISH/SUBSCRIBE architecture such as Apache Kafka is used, using this pattern variant would mean to expose Kafka to clients. To expose a RESTful API abstraction (maybe even on top of Kafka used in the backend) would be a solution that would not use the pattern variant, as from the viewpoint of the API, the PUBLISH/SUBSCRIBE architecture is not visible. This variant is applicable when this pattern shall be applied in a context where a PUBLISH/SUBSCRIBE architecture or event feeds are used as well.

- *Event-Based CQRS API Endpoint Operations:* The CQRS (*Command Query Responsibility Segregation*) pattern [136] advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. Both APIs can use event-based abstractions, e.g. as state transition operations as explained above. Also, both can use a PUBLISH/SUBSCRIBE architecture internally to realize the eventual consistency between the services and query views. This variant is applicable when this pattern shall be applied in a context where CQRS also shall to be applied.

Consequences

+ *Avoid Exposing Domain Model Details in API* : Only selected events that are actually needed for client interactions get published as state transition operations in the API. Other details of the DOMAIN MODEL remain hidden.

+ *Chatty API, Performance, and Scalability*: Events from a number of DOMAIN MODEL elements are bundled in one state transition model to optimize the client/system interaction. This way, it is possible to avoid introducing unnecessary interactions through distributed calls between client and system exchanging the events – that would yield a chatty API. For example, defining more aggregated or higher-level events at the API level could help to optimize client/system interactions in terms of requests that have to be exchanged. More generally speaking, such an optimized client/system interaction model can be designed for better performance and scalability. Again, additional measures than just those covering atomic operation design are possible. For example, just as discussed for AGGREGATED DOMAIN OPERATION ON API ENDPOINT, REQUEST BUNDLING [212] can be applied to exchange a couple of events in one joint (bundled) distributed call. Or introducing API RATE LIMITING [212, 165] can help to avoid abusive clients being able to overload the system.

\+ *Coupling of Clients to Server*: Event-based interactions usually lead to loosely coupled architectures. When exposing events as state transition operations, the clients are still coupled to those operations, but backend microservices can be loosely coupled to the receiving API microservices. In the *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* even the clients use loosely coupled dependencies.

\+ *Avoid Interface Design that Limits Domain Model Design*: In DDD, a DOMAIN MODEL is the core model and the basis for the UBIQUITOUS LANGUAGE of a software project. Thus, it shall not be limited in its design because of interface design considerations. An additional layer that abstracts the DOMAIN EVENTS, or exposing the event needed for client interactions, are both solutions to avoid a strong impact of interface design on DOMAIN MODEL design, if only selected or abstracted events are exposed.

+/- *Maintainability of API and API Consumers*: Loosely coupled relations with minimal required dependencies enable many positive maintainability properties, such as modularity of the API and its consumers, as well as more independent testability, modifiability, and analyzability. This is much harder to achieve e.g. when the fine-grained modularity implied by CRUD-BASED API OPERATIONS is needs to be maintained (i.e., many *Entities* are used as API modules), and the more strongly coupled dependencies make it harder to reach independent testability, modifiability, and analyzability. On the other hand, loose coupling often makes it more challenging to detect the impact of breaking changes: A tightly coupled system breaks immediately, a loosely coupled system may also break but will hide the root cause because of its indirection layers.

+/- *Data Consistency*: Event-based interactions often means to introduce eventual consistency [173], but so do other distributed system interactions, especially asynchronous ones [44]. That is, transactions for coordination between microservices are not used and consistency issues are dealt with by compensating operations. This is more complex and substantially harder to manage than transaction-based consistency [44]. However, if eventual consistency needs to be embraced, the event-driven approach is a well-established one with helpful supporting patterns and practices such as EVENT NOTIFICATION, EVENT-CARRIED STATE TRANSFER, EVENT SOURCING, and CQRS [45].

\- *API Understandability*: Events-based interactions can be harder to understand than a simple sequence of API operations, especially if they follow a simplistic scheme as in synchronous CRUD-BASED API OPERATIONS. That is, the more asynchronous and loosely coupled an architecture is, the harder it can get to detect the impact of one operation call or a particular change. A loosely coupled and asynchronous system may hide the root cause of a defect or breaking change due to its indirections layers. Because of them, it is also hard to understand the effects of a particular API call in the system.

Related Patterns

As explained, the two patterns AGGREGATED DOMAIN OPERATION ON API ENDPOINT and CRUD-BASED API OPERATIONS are alternatives to this pattern. Whereas CRUD-BASED API OPERATIONS is in many cases not an advisable alternative, its variant *Aggregated CRUD-Based Operation on API Endpoint* which combines CRUD-BASED API OPERATIONS with this AGGREGATED DOMAIN OPERATION ON API ENDPOINT, usually is a possible option.

As explained in the Solution Details section, ideally, this pattern is placed on a endpoint derived using one of the following patterns: AGGREGATE ROOTS AS API ENDPOINTS [158], DOMAIN SERVICES AS API ENDPOINTS [158], or DOMAIN PROCESSES AS API ENDPOINTS [158]. Sometimes it makes sense to combine it with the ENTITIES AS API ENDPOINTS [158] and BOUNDED CONTEXTS AS API ENDPOINTS [158] practices, too.

Patterns such as REQUEST BUNDLING [212] or API RATE LIMITING [212, 165] are options to improve performance and scalability.

The *Event-Based CQRS API Endpoint Operations* variant combines this pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [136].

The option *Event-Based API Endpoint Operation via Feeds or Publish/Subscribe* combines this pattern with a PUBLISH/SUBSCRIBE [17] architecture.

Known Uses

- The *eShopOnContainers* open source system used in the example above is a system in which API operations are derived as state transition operations from DOMAIN EVENTS, and corresponding integration events are used internally to realize microservice interaction in a PUBLISH/SUBSCRIBE architecture.

- Dugalic [30] presents a similar exposition of API operations as state transition operations derived from *Domain Events*. In addition, each BOUNDED CONTEXT of the application is divided following the CQRS pattern into commands and queries. That is, this known use realizes the *Event-Based CQRS API Endpoint Operations* variant of the pattern. For internal inter-service communication again a PUBLISH/SUBSCRIBE architecture is used. Various implementation of this, e.g. based on the AXON Web server and RESTful HTTP, are published, too[23].

- Another similar architecture is provided by the Eventuate Example application [24]. Here, DOMAIN EVENTS derived from AGGREGATES are exposed as state transition operations via CQRS command and query APIs. The Eventuate event store is used in the background as a PUBLISH/SUBSCRIBE architecture backbone. This example, additionally realized EVENT SOURCING [136].

[23] `https://github.com/idugalic/digital-restaurant`
[24] `https://github.com/cer/event-sourcing-examples`

- The Kanban board application[25] is an example where a microservices are used via a RESTful HTTP API and WebSockets APIs. Services are based on AGGREGATES and again CQRS command and query APIs are offered. Eventual consistency is managed via an event store.

CRUD-Based API Operation

Context

In a software development project, you use DDD to design your DOMAIN MODEL and want to design the operations on the endpoints of an API for this project. Consider further that your DOMAIN MODEL is designed in detail, especially, it is modeling operations on the DOMAIN MODEL elements.

Problem

How to design the operations of an API endpoint in relation to the operations modeled on DOMAIN MODEL elements?[26]

Forces

This pattern has the same forces as discussed for its alternative pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT.

Solution

Design operations on API endpoints based on the well-known *Create, Read, Update, and Delete (CRUD)* primitives, which are also the basis for many primitive datastore operations abstractions, as well as the main HTTP methods (i.e., POST, GET, PUT/PATCH, and DELETE). Use this endpoint design, if DOMAIN MODEL contain only or can easily be mapped to a subset of those primitives. This includes any CRUD subset, e.g. read-only endpoints, write-only endpoints, append-only endpoints, or endpoints with no option to delete.

Limit the use of such CRUD-BASED API OPERATIONS to cases where they are essentially the only possible option. For example, they should be used, if API consumers require such Create, Read, Update, or Delete operations to function and API designers find no better-fitting, more abstract API operation.

More design advice on this can be found in the Microservice API patterns [206, 208]. They contain several patterns that address the question of which architectural roles API endpoints play. *Information Holder Resources* are endpoints that primarily expose data management and storage operations, whereas *Processing Resources* are concerned with handling incoming action requests and commands. CRUD-BASED API OPERATIONS can typically

[25] `https://github.com/eventuate-examples/es-kanban-board`

[26] Please note that the problem of this pattern is very similar to the one of AGGREGATED DOMAIN OPERATION ON API ENDPOINT.

be found on *Information Holder Resources*. It should be avoided, to design a large number of *Information Holder Resources* which are offered as distributed services and expose high semantic and operational coupling. Nygard calls this the "Entity Service Anti-Pattern"[27]. That does not mean *Information Holder Resources* should be avoided completely. Instead any use should be a conscious decision motivated and justified by the design scenario at hand to avoid negative impacts such as the coupling impact Nygard describes [208].

To reach this, ideally, some abstraction happens in the API from DOMAIN MODEL details, for example by placing the CRUD-BASED API OPERATION on an aggregating DOMAIN MODEL element, such as an AGGREGATE, domain SERVICE, or BOUNDED CONTEXT, that is exposed on the API. In rare cases, more primitive DOMAIN MODEL elements such as ENTITIES or sometimes even VALUE OBJECTS are needed as-is by API consumers. If there is no meaningful way to aggregate or abstract those DOMAIN MODEL elements further in the API, it makes sense to expose those DOMAIN MODEL elements to the API and offer CRUD-BASED API OPERATIONS on them.

Solution Details

CRUD-BASED API OPERATION offers a simple solution that is easy to design. Especially in the context of RESTful APIs where the HTTP methods support CRUD-like abstraction or in the context of APIs heavily relying on database backends, many designers start out with this pattern. If a CRUD-like abstraction is the natural abstraction to represent the DOMAIN MODEL elements in the client scope, following this pattern makes sense.

But the pattern can also be deceiving and lead to less than optimal designs. Consider a system in which each and every data element on an ENTITY and VALUE OBJECT is simply offered as CRUD-BASED API OPERATIONS. This would lead to highly complex APIs with many internal DOMAIN MODEL elements being exposed to the API that are not useful or needed in the scope of the clients. Therefore, an API consisting solely of API elements representing fine-grained DOMAIN MODEL elements, such as ENTITIES and VALUE OBJECTS, and only offers CRUD-BASED API OPERATIONS is an anti-pattern.

Thus, before applying this pattern, it shall be considered if another abstraction might be better fitting. Here, the two prime alternatives are the patterns AGGREGATED DOMAIN OPERATION ON API ENDPOINT and EVENT-BASED API ENDPOINT OPERATION explained below. Please note in this context that there is also a variant of the CRUD-BASED API OPERATION pattern, explained below which offers CRUD-based operations on API elements representing aggregated or coarse-grained DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS. Such *Aggregated CRUD-Based Operation on API Endpoint* are usually preferable over many CRUD-BASED API OPERATIONS on ENTITIES, as they reduce the number of necessary operations to what is actually needed by clients, as abstractions in the form needed in the client scope.

[27] `http://www.michaelnygard.com/blog/2018/01/services-by-lifecycle/`

Finally, of course, some ENTITIES or other fine-grained DOMAIN MODEL elements are needed as is on client side. Then it makes no sense to introduce intermediate aggregated abstractions, but instead those ENTITIES should get exposed in the API, and if the client needs CRUD-operations on them, those should be offered as CRUD-BASED API OPERATIONS.

Our patterns on deriving API endpoints from DOMAIN MODEL elements advise us to use, if possible, AGGREGATE ROOTS AS API ENDPOINTS, DOMAIN SERVICES AS API ENDPOINTS, or DOMAIN PROCESSES AS API ENDPOINTS, which are all coarse-grained DOMAIN MODEL elements abstracting from other DOMAIN MODEL elements such as ENTITIES or VALUE OBJECTS. If this is not possible, next BOUNDED CONTEXTS AS API ENDPOINTS shall be considered, which are typically yet coarser-grained structures. It is also possible to expose ENTITIES AS API ENDPOINTS but it is advised to expose them with caution because an API design where every ENTITY or VALUE OBJECT (which is not an AGGREGATE root) is exposed to the API suffers typically from many negative consequences. For instance, DOMAIN MODEL details might be exposed through the API, which makes the API hard to understand and change, raise its complexity, introduce unnecessary coupling, and so on. This would also lead to chatty APIs, with possibly low performance and bad scalability. Those choices should be reflected at the operation level by exposing primarily operations on those coarser-grained structures. That is, the idea of the AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern is to primarily expose operations that represent abstractions of the details in the DOMAIN MODEL, exposing only those DOMAIN MODEL elements required by clients and nothing more.

Example of RESTful Customer API Following the CRUD-based API Operation Pattern

Figure 4.17 extends the minimal example from Figure 4.14. It uses the same DOMAIN MODEL, but highlights different operations exposed to the API: Here, four CRUD-based operations are exposed to the API, which are mapped to the CRUD-based operations on the *Customer* ENTITY in the DOMAIN MODEL.

Example of CRUD-based API Operations in the Lakeside Mutual Project

The following code shows an excerpt of the operations exposed via RESTful HTTP in the `Customer` information holder endpoint of the Lakeside Mutual open source project[28]. `Customer` is an ENTITY which is offered as an endpoint to clients. Clients such as the system's Web-based frontend pages use CRUD-BASED API OPERATIONS on it to GET, PUT, or POST customers and customer data. In this example, it makes perfect sense to expose the ENTITY with CRUD-BASED API OPERATIONS as clients such as the Web frontend page client or the customer self-service API need to perform read, update, and create operations on these data elements.

```
@RestController
```

[28] `https://github.com/Microservice-API-Patterns/LakesideMutual`

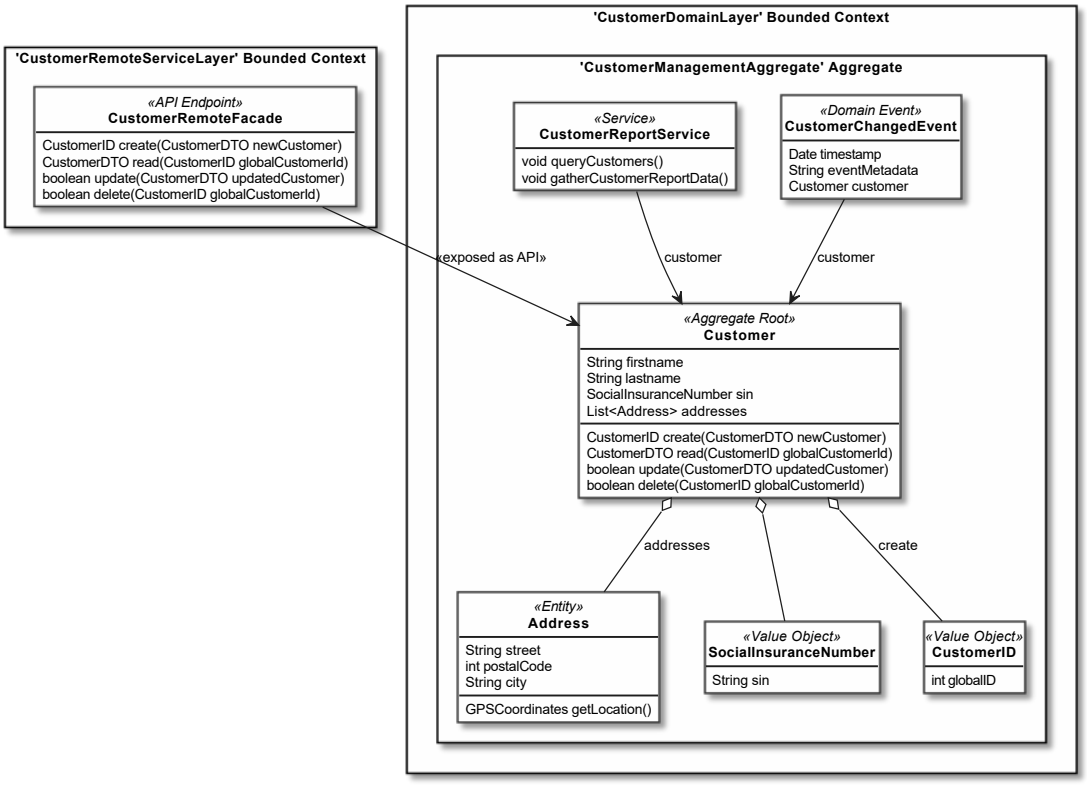

Figure 4.17: CRUD-based API Operation Exposing the Customer Entity

```
@RequestMapping("/customers")
public class CustomerInformationHolder {
  ...
  @ApiOperation(value = "Get a specific set of customers.")
  @GetMapping(value = "/{ids}")
  public ResponseEntity<CustomersResponseDto> getCustomer(
  @ApiParam(value = "a comma-separated list of customer ids", required =
      true)
  @PathVariable String ids,
  @ApiParam(value =
  "a comma-separated list of the fields that should be included in the
      response",
  required = false)
  @RequestParam(value = "fields", required = false, defaultValue = "")
      String fields) {
    List<CustomerAggregateRoot> customers =
       customerService.getCustomers(ids);
    List<CustomerResponseDto> customerResponseDtos = customers.stream()
    .map(customer -> createCustomerResponseDto(customer, fields))
    .collect(Collectors.toList());
    CustomersResponseDto customersResponseDto =
    new CustomersResponseDto(customerResponseDtos);
    Link selfLink = linkTo(methodOn(CustomerInformationHolder.class)
    .getCustomer(ids, fields)).withSelfRel();
    customersResponseDto.add(selfLink);
    return ResponseEntity.ok(customersResponseDto);
```

```
}

@ApiOperation(value = "Update the profile of the customer with the
    given customer id")
@PutMapping(value = "/{customerId}")
public ResponseEntity<CustomerResponseDto> updateCustomer(
@ApiParam(value = "the customer's unique id", required = true)
@PathVariable CustomerId customerId,
@ApiParam(value = "the customer's updated profile", required = true)
@Valid @RequestBody CustomerProfileUpdateRequestDto requestDto) {
  final CustomerProfileEntity updatedCustomerProfile = requestDto
  .toDomainObject();

  Optional<CustomerAggregateRoot> optCustomer =
  customerService.updateCustomerProfile(customerId,
      updatedCustomerProfile);
  if(!optCustomer.isPresent()) {
    final String errorMessage =
    "Failed to find a customer with id '" + customerId.toString() +
        "'.";
    logger.info(errorMessage);
    throw new CustomerNotFoundException(errorMessage);
  }

  CustomerAggregateRoot customer = optCustomer.get();
  CustomerResponseDto response = new
      CustomerResponseDto(Collections.emptySet(),
  customer);
  return ResponseEntity.ok(response);
}
...
```

Consequences

+/- *Avoid Exposing Domain Model Details in API* : CRUD-BASED API OPERATIONS only support selecting among the four operations CRUD offers. If those are exactly what is needed, this is optimal. In other cases, CRUD-BASED API OPERATIONS have the risk of exposing details of the DOMAIN MODEL not necessarily needed by clients.

+/- *Avoid Interface Design that Limits Domain Model Design*: This pattern leads to a good DOMAIN MODEL mapping, if the client needs the CRUD-like operations on a subset of the data elements on the DOMAIN MODEL element. Then the impact is positive; else, it can be highly negative. For instance, consider a DOMAIN MODEL element offers read-only operations of rather static information, e.g. it delivers the zip codes of a country. Then a CRUD-based DOMAIN MODEL design comes from the domain semantics and a 1:1 mapping makes sense. Consider further an early prototype of an API has been designed with only CRUD-like operations. If this design is then

used as input to change DOMAIN MODEL design to "better conform" to the API, in the worst case an anemic DOMAIN MODEL is the result.

+/- *Maintainability of API and API Consumers*: If the client needs the CRUD-like operations on a subset of the data elements on the DOMAIN MODEL element, this pattern offers a positive impact on maintainability as it offers the simplest mapping possible. If this is not the case, its impact on maintainability can be rather negative, e.g. if far too many CRUD-like operations need to be maintained or they lead to complex interactions. How many CRUD-like operations are actually needed shall be determined by the semantic of the DOMAIN MODEL.

+/- *Data Consistency*: If the data consistency boundary (e.g. transaction boundary) is the DOMAIN MODEL element on which the CRUD-BASED API OPERATIONS are offered, this pattern offers very good data consistency impact. If this is not the case, in the worst case the client needs to manage consistency. Or eventual consistency measures need to be introduced in the backend. These are rather negative impacts.

+/- *API Understandability*: If CRUD-like operations are needed on the client, this pattern offers the simplest possible mapping, which is thus easy to understand. If instead far too many CRUD-like operations are offered or they lead to complex interactions, this makes the API complex and thus hard to understand.

+/- *Coupling of Clients to Server*: CRUD-BASED API OPERATIONS usually lead to a substantial level of coupling, especially compared to EVENT-BASED API ENDPOINT OPERATION, if certain consistency boundaries need to be considered. However, CRUD-BASED API OPERATIONS can also help to decouple different API clients. For example, consider some shared data among different clients, and one client creates the data, and the other one reads it and then deletes it. The two clients never meet or even need to know about each other. Thus, impact on coupling highly depends on DOMAIN MODEL semantics and how well the mapping is designed.

- *Chatty API, Performance, and Scalability*: CRUD-BASED API OPERATIONS can be very chatty, as they are among the most fine-grained API operations possible. If that is exactly what is needed by clients, they can be applied nonetheless, but still lead to chattiness. Likewise, performance and scalability are highly dependent on the number of distributed calls exchanged. They are also influenced by message payloads. Performance and scalability can be degraded if CRUD-BASED API OPERATIONS are not exactly what is needed by clients. Patterns that help to aggregate responses such as PAGINATION [212] or WISH LIST [212], or help to aggregate requests such as REQUEST BUNDLE [212], can help to reduce these issues.

- *API Operations Reuse*: The rather fine-grained API operations that are a consequence of this pattern can quickly lead to many exposed operations and to optimizations per group of clients. As a consequence, it might be hard to reuse API operations well.

Pattern Variants

There are two variants of the pattern:

- *Aggregated CRUD-Based Operation on API Endpoint:* We mainly discussed fine-grained CRUD-BASED API OPERATIONS above. The previously discussed pattern AGGREGATED DOMAIN OPERATION ON API ENDPOINT discusses the notion to offer domain operations on aggregated or coarse-grained DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS. Having CRUD-BASED API OPERATIONS on those can alleviate a couple of possible risks and drawbacks mentioned in the consequences of CRUD-BASED API OPERATIONS above, especially those that are related to the fine-grained nature of CRUD-BASED API OPERATIONS. The example in the AGGREGATED DOMAIN OPERATION ON API ENDPOINT pattern contains some samples of this pattern variant. This variant is applicable when "aggregating" DOMAIN MODEL elements such as AGGREGATES, DOMAIN SERVICES, or BOUNDED CONTEXTS contain CRUD-like operations in the DOMAIN MODEL.

- *CRUD-Based CQRS API Endpoint Operations:* The CQRS (*Command Query Responsibility Segregation*) pattern [136] advises to use a different model to update data than the model that is used to read data. If CQRS is exposed to the API, the API is segregated into Command and Query APIs. Both APIs can use CRUD-BASED API OPERATIONS, as explained in this pattern. This then would require to segregate the Read operations from the Create, Update, and Delete operations in the Query and Command APIs. As then Read operations would be provided as only a view on the Command part of the API, data would be eventually consistent as a consequence. This can lead to different operations and interactions in the API than in a non-segregated view, e.g. additional compensation action commands might need to be added to cope with transaction issues due to eventual consistency. This variant is applicable when this pattern shall be applied in a context where CQRS also shall to be applied.

Related Patterns

As explained, the two patterns AGGREGATED DOMAIN OPERATION ON API ENDPOINT and EVENT-BASED API ENDPOINT OPERATION are the prime alternatives that should be considered before considering CRUD-BASED API OPERATIONS. *Aggregated CRUD-Based Operation on API Endpoint* is a combination of CRUD-BASED API OPERATIONS with AGGREGATED DOMAIN OPERATION ON API ENDPOINT.

Patterns that help to aggregate responses such as PAGINATION [212] or WISH LIST [212], or help to aggregate requests such as REQUEST BUNDLE [212], can help in reducing negative performance and scalability impacts, and thus avoid chatty APIs. See the Consequences section for details.

As explained in the Solution Details section, ideally, this pattern is placed on a endpoint derived using one of the following patterns: AGGREGATE ROOTS AS API ENDPOINTS [158],

DOMAIN SERVICES AS API ENDPOINTS [158], or DOMAIN PROCESSES AS API ENDPOINTS [158]. Sometimes it makes sense to combine it with the ENTITIES AS API ENDPOINTS [158] and BOUNDED CONTEXTS AS API ENDPOINTS [158] practices, too.

The *CRUD-Based CQRS API Endpoint Operations* variant combines this pattern with the CQRS (*Command Query Responsibility Segregation*) pattern [136].

Known Uses

- The *Lakeside Mutual* open source system discussed above contains a number of ENTITY and AGGREGATE-based endpoints which offer CRUD-like domain operations. Thus both the main pattern variant and *Aggregated CRUD-Based Operation on API Endpoint* are supported. The example of this pattern shows the first variant, the example of AGGREGATED DOMAIN OPERATION ON API ENDPOINT shows the second variant.

- The Pokemon API[29] offers many ENTITIES exposed to the API. On those it mainly offers a large collection of GET operations. Such read-only APIs in which each detail needs to be read by clients might offer some potential of aggregation and abstraction in performing multiple requests at once. Also, an API that uses a query language or interface on an AGGREGATE might be better suited. But as the fine-grained data is needed by the clients, CRUD-BASED API OPERATIONS is here not as negative as it would be on an API that would offer all four types of CRUD operations. But still it has some negative impacts on some of the forces discussed. For instance, the API is more complex than necessary and thus harder to understand and maintain. More fine-grained calls than necessary might lead to non-optimal performance, scalability, and chatty interactions, and so on.

- For similar reasons, the disease.sh open API for disease-related statistics[30] offers many GET operations on fine-grained API endpoints and is thus based on CRUD-BASED API OPERATIONS, too. As example from this API is shown in Figure 4.18. It has similar negative impacts on forces as discussed for the Pokemon API.

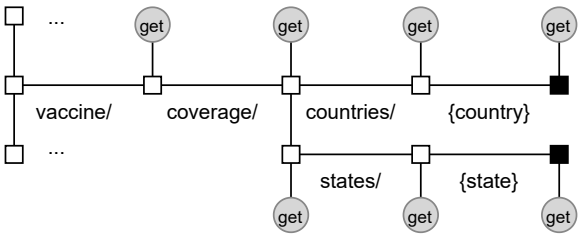

Figure 4.18: The API Tree of Vaccine Endpoints in the disease.sh

- The publication management demo case for MDSL[31] offers a CRUD-BASED API OPERATION *Create Paper Item* which is an instance of the *Aggregated CRUD-Based*

[29] `https://github.com/PokeAPI/pokeapi`
[30] `https://github.com/disease-sh/API`
[31] `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html`

Operation on API Endpoint variant. It requires a concrete implementation on the *Paper Collection Backend* ENTITY.

4.7 Related Work

This section outlines and compares to relevant related works. There are a substantial number of patterns and pattern languages in closely related areas. Firstly, core works on DDD such as those by Evans [33] and Vernon [182] describe DDD practices as patterns. Also various distributed systems patterns exist, too. The closest are our Microservice API patterns [212, 90] which describe best practices on the design of microservices APIs. In addition, patterns for various style of distribution have been discussed such as Messaging Patterns [64], Remoting Patterns [184], Enterprise Application Architecture patterns [42], Cloud Adoption Patterns [163, 15], and Service Design Patterns [27], to name just a few. Many of these works, hint at how to derive APIs and distributed systems in general from domain models, but so far this is not the core focus of any of these works.

This work is based on two of our prior studies. Firstly, we studied how practitioners understand coupling smells [154] and generated a Grounded Theory (GT) [51, 24] explaining those coupling smells and their relations. In this study we found a number of evidences which indicated a link between coupling smells and related software engineering principles to DDD. In essence, issues in DDD models can lead to coupling smells, and vice versa. This is interesting in the API context, as those smells and issues tend to become worse when they occur in a distributed setting, as discussed in Section 4.3. In distributed setting, it also needs to be considered that smells resemble established distribution patterns such as DATA TRANSFER OBJECT (DTO) [42], which can lead, if not carefully analyzed, to detrimental refactorings.

Secondly, we conducted a Grounded Theory study based on the grey literature mainly focusing on the interrelation of microservice API design and DDD [157]. We derived six architectural design decisions (ADDs) with 27 design options and 27 design drivers.

In another previous work [196] related to the Microservice API patterns [212, 90] we identified ADDs in the area of microservice API quality from the in-depth study of 31 widely used APIs and 24 specifications, standards, and technologies. We reported six ADDs with 40 decision options and 47 drivers.

Taibi and Lenarduzzi [171] define a number of microservice bad smells. As discussed in Section 4.3, some of those are relating bad smells and APIs. In this sense, this work also confirms our observation that coupling smells are relevant in the context of our work and may increase in their intensity in a distributed setting.

Stylos and Myers [167] categorized and organized API design decisions by conducting empirical research (in particular, a multi-vocal literature review). They investigated the literature in API usability, whereas we mainly focus on the interrelation between API and DDD.

Li and Chou [86] propose three design patterns for RESTful Web services based on a case study of computer-supported telecommunications application services. Their work concentrates on REST APIs, with basic abstraction such as session, event subscription and relationships using REST composition. Our focus is broader, as we concentrate on all kinds of API concepts and technologies.

Ayas et al. [6] conducted a Grounded Theory study to investigate the decision making in microservice migrations. Their data collection is from interviewing 19 participants, and evaluated by 52 professionals. They realized decision making on technical dimension that reflects the organizational and operational levels.

Brogi et al. [14] present a systematic literature review on design principles, architectural smells, and refactorings for microservices based on analysis of 54 sources. The chapter identified many design smells and their resolution. The smell resolution in the chapter primarily is on the infrastructure level (e.g., ESB rightsizing is suggested) and therefore complementary to our work.

There are number of API design patterns for specific domains, for example, northbound API of Software-Defined Networking (SDN) [87, 201], Internet of Things (IoT) [170], and biological data [188]. While API design in general has been studied, the specific relation of API design to design practices and models commonly used (such as those in DDD) is yet understudied. This is gap in the state-of-the-art led us to write our patterns on deriving APIs and API endpoints from DDD domain model elements.

Our work aims to present more general design patterns, for the specific problem of designing the combination of API and DDD. API endpoints definition in our context are broader than the key abstractions of REST resource specification, as e.g. in Fielding's work [39]. In REST, a resource being served usually refers to one or more nouns, whereas an endpoint is the location where a service can be accessed. In a non-REST context, API endpoints have various kinds of meaning, such as in URL (Uniform Resource Locator), where it is almost synonymous to an endpoint. The Microservice API patterns [212, 90, 165] use the following definitions: An API ENDPOINT is the provider-side end of a communication channel and a specification of where the API endpoints are located so that APIS can be accessed by API CLIENTS. Each endpoint thus needs to have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), as well as in HTTP-based SOAP or RESTful HTTP. Each API ENDPOINT belongs to an API; one API can have different endpoints. Our work has the goal to help in deriving API and API endpoints from domain model elements, which could potentially ease software engineering tasks and decision making, for instance in contexts such as migration to/of microservice architectures or API design.

Table 4.1 summarizes the main related works. We compare the core topics of the works to our work's core topics, i.e. if they address DDD, APIs, Microservices, Smells, and Design Decision, as well as the used methodologies.

There are a substantial number of patterns and pattern languages in closely related areas. Firstly, core works on DDD such as those by Evans [33] and Vernon [182] describe

Table 4.1: Comparison to Related Work for Patterns on Deriving APIs and their Endpoints from Domain Models

Work/Reference	Core Topics					Methodology
	DDD	API	Microservices	Smells	Decisions	
Singjai, Simhandl, and Zdun [154]	Yes	No	No	Yes	No	GT/Grey Literature
Singjai, Zdun, and Zimmermann [157]	Yes	Yes	Yes	Yes	Yes	GT/Grey Literature
Taibi and Lenarduzzi [171]	No	Yes	Yes	Yes	No	Interviews
Stylos and Myers [167]	No	Yes	No	No	Yes	Multi-vocal Literature Review
Li and Chou [86]	No	Yes	No	No	Yes	Case Study
Ayas, Leitner, and Hebig [6]	No	Yes	Yes	No	Yes	GT/Interviews
Brogi, Neri, Soldani, and Zimmermann [14]	No	Yes	Yes	Yes	No	Systematic Literature Review
Our work	Yes	Yes	Yes	Yes	No	Pattern Mining based on GT /Grey Literature Study

DDD practices as patterns. Many distributed systems patterns exist, too. The closest are our Microservice API patterns [212, 90] which describe best practices on the design of microservices APIs. In addition, patterns for various style of distribution have been discussed such as Messaging Patterns [64], Remoting Patterns [184], Enterprise Application Architecture patterns [42], Cloud Adoption Patterns [163, 15], and Service Design Patterns [27], to name just a few. Many of these works, hint at how to derive APIs and distributed systems in general from domain models, but so far this is not the core focus of any of these works.

Table 4.2 summarizes the main related works. We compare the core topics of the works to our work's core topics, i.e. if they address APIs, Microservices, Smells, Refactoring, and Design Decision, as well as the focus area of works and the research methodologies used in the works.

In our prior work [196] related to the Microservice API patterns [212, 90] we identified Architectural Design Decisions (ADDs) in the area of microservice API quality from the in-depth study of 31 widely used APIs and 24 specifications, standards, and technologies. We reported six ADDs with 40 decision options and 47 drivers. The Microservice API patterns [206, 208] language features several patterns that address the question of which architectural roles API endpoints play. Information Holder Resources are endpoints that primarily expose data management and storage operations, whereas Processing Resources are concerned with handling incoming action requests and commands. The general Information Holder Resource is further refined into sub-patterns depending on the life span and mutability of the data contained.

Context Mapper [72] provides a Domain-Specific Language (DSL) and supporting tools for model-driven Domain-Driven Design (DDD). It focuses on modeling based on strategic and tactical DDD patterns such as Bounded Context, Aggregate, Entity, and Service. The domain model to API mappings presented in this chapter are partially supported in Context Mapper. Platform-independent API descriptions in MDSL, another DSL, can be generated from the DDD models. These transformations map the operations in Aggregates and their Root Entities and Services to API endpoints and operations.[32] MDSL, in turn, provides

[32]See `https://contextmapper.org/docs/mdsl/`

Table 4.2: Comparison to Related Works with Focus Area for the Patterns on Designing API Endpoint Operations

Work/ Reference	**Core Topics**					**Focus Area**	**Methodology**
	API	**Micro services**	**Smells**	**Re factoring**	**Decisions**		
Zdun, Stocker, Zimmermann, Pautasso, and Lübke [196]	Yes	Yes	No	No	Yes	ADDs for Microservice APIs	Grounded Theory (GT)
Lübke, Zimmermann, Pautasso, Zdun, and Stocker [90]	Yes	Yes	No	No	No	Microservice API Patterns	Pattern Mining
Kapferer and Zimmermann [72]	Yes	Yes	No	Yes	No	Domain Driven Service Design	Empirical Validation
Zimmermann, Lübke, Zdun, Pautasso, and Stocker [206]	Yes	Yes	No	No	No	Microservice API Patterns	Pattern Mining
Zimmermann, Pautasso, Lübke, Zdun, and Stocker [208]	Yes	Yes	No	No	No	Microservice API Patterns	Pattern Mining
Taibi and Lenarduzzi [171]	Yes	Yes	Yes	No	No	Microservice Bad Smells	Interviews
Stylos and Myers [167]	Yes	No	No	No	Yes	API design decisions, API quality attributes	Multi-vocal Literature Review
Li and Chou [86]	Yes	No	No	No	Yes	RESTful Communication Web Services	Case Study
Ayas, Leitner, and Hebig [6]	Yes	Yes	No	No	Yes	Decision-Making in Microservices	GT/Interviews
Brogi, Neri, Soldani, and Zimmermann [14]	Yes	Yes	Yes	Yes	No	Microservice Architectural Smells	Systematic Literature Review
Our work	Yes	Yes	Yes	No	No	DDD-based API design	Pattern Mining based on GT/-Grey Literature Study

mappings to microservice technologies that can be generated. These mappings include OpenAPI/Swagger interface descriptions, gRPC Protocol Buffer specifications and Jolie services (that in turn can be transformed into port types in WSDL and XML Schema). An interface refactoring catalog and tool support for refactoring API designs according to patterns are emerging as well [164].

4.8 Conclusion

In this chapter, we described two set of patterns for deriving API operations from domain-driven design models. This work draws upon data sets we have created in our prior research. Firstly, we mined patterns and their relations on how to derive AGGREGATED DOMAIN OPERATIONS ON API ENDPOINTS, API ENDPOINTS BASED ON DOMAIN EVENTS, and CRUD-BASED API OPERATION endpoints. Secondly, we mined patterns and their relations on how to derive AGGREGATED DOMAIN OPERATIONS ON API ENDPOINTS, API ENDPOINTS BASED ON DOMAIN EVENTS, and CRUD-BASED API OPERATION endpoints. These patterns

originate from an in-depth empirical study of grey literature authored by practitioners. In our pattern mining, we have also considered twelve detailed open source system models (which we modeled from systems implemented or documented by practitioners), from which we have given examples and known uses in this chapter.

Part IV

Empirical Evaluation

Chapter 5

Conformance Assessment of Architectural Design Decisions

This chapter presents the empirical evaluation research stage. 14 case studies are observed to address the **RQ4**. The conformance assessment of architectural design decisions performs in the high-level (the mapping of domain model elements to APIs and API endpoint) and the low-level (API endpoint designs derived from domain models) designs. Moreover, there are five lesson learned which we gained from studying this chapter. The content of this chapter is based on a peer-reviewed article published in **ICSA'22** and a peer-reviewed journal published in **JSS'22**.

Microservice APIs are often identified and designed based on Domain-Driven Design (DDD). To help in the continuous analysis of mappings of domain model elements to APIs and API endpoints, we aim to automate the assessment of conformance to Architectural Design Decision (ADD) options. The ADDs, their decision options, and relevant decision drivers studied in this chapter originate from an empirical study on the mapping of domain model elements to APIs and API endpoints in practice. We propose automated detectors to detect the decision options of the ADDs taken in a given microservice API model, and an assessment scoring scheme based on the empirical knowledge codified in the ADDs. We evaluate our work, by first manually creating a ground truth in a multi-case study, and then comparing the results of our automated detectors to the ground truth. In the cases we were able to identify more than 80% of the decision points correctly, and a statistical analysis of our data shows only a negligible effect size for differences to the ground truth.

5.1 Introduction

Microservice architectures consist of independently deployable, scalable, and changeable services [203, 211, 104]. In microservices and related architectures, message-based APIs, such as RESTful HTTP, queue-based messaging, or event-based message exchanges, dominate over remote procedure calls [212, 211]. A critical aspect in designing a microservice architecture is API design which, in this chapter, is seen as all activities of planning and

making design decisions when building an API [197, 212]. It includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to offer links, and how to represent API messages [211, 212].

Domain-Driven Design (DDD) [33, 182] is a set of concepts and patterns that help in designing software systems based on the underlying model of the (business) domain, the Domain Model [42]. It is often used as a foundation to identify and design microservice architectures [36, 100, 157].

Architectural Design Decisions (ADDs) focus on the architecturally significant design decisions of a system, in the sense that a single decision change could significantly affect its entire architecture [82]. In the field of microservice APIs and their relations to Domain Models, many central API design problems revolve around API endpoint design. An *API endpoint* is the provider-side end of a communication channel and a specification of where the API endpoints are located so that APIs can be accessed by API clients [212, 211]. Changes in the API endpoints can significantly affect the architecture of the API clients and of the microservices in the backend serving the API [212].

In our study, we selected three empirically grounded ADDs from our prior work [157]. In particular, we studied ADDs on detailed API endpoint design explaining (1) how to derive links offered in API endpoints from the links in the domain model; (2) how to design the operations of an API endpoint; and (3) if and how to segregate the resource represented by the API endpoint.

Please note that these are all architectural design issues relevant only for remote, message-based APIs. While the overlap between a DDD model and an API can exist in a local context, too, none of the issues addressed in the studied ADDs appears for local APIs or API libraries

Many existing practitioner works use DDD to identify and model the microservices which are exposed via APIs [157, 212, 36, 100]. If this is the case, APIs should have a traceable mapping to the Domain Models of the microservices. In each evolution step of either the API or the microservice models, the respective other parts might have to be changed accordingly. To safeguard such evolution steps in practice, it is required to assess whether a given microservice API endpoint design correctly realizes the mapping to its microservice domain model and vice versa. This is especially problematic if such an assessment is needed continuously, e.g. in the context of continuous integration/delivery (CI/CD). Just consider today's large-scale microservice deployments that are deployed with high frequency (e.g. at least daily), such as those of Uber [148], Google [52], or Netflix [105]. To manually assess for each and every service, whether the mapping to the Domain Models and other such constraints imposed by backend or client architectures are still in place and correct, would be an extremely laborious and error-prone manual task. This kind of assessment can be classified as architecture conformance assessment. Conformance is generally defined as the consistency relation between models [127]. In our API endpoint design context, conformance concerns consistency in the mapping of API models to DDD models according to the relations from our empirically studied ADDs.

To address this problem, we aim to automate the assessment of conformance to ADD options in API endpoint designs. Some might argue that APIs are intended to change rarely, and thus such automation is not required. But actually there are many more reasons for API evolution [91], meaning that frequent system changes usually lead to frequent API changes. Also, our approach is on the links of APIs to DDD models. So, if the API stays stable but the DDD model changes, still conformance must be checked.

Here, we phrase our study design following the Goals/Questions/Metrics (GQM) approach [8]: It is the **research goal** of this chapter to *study how to provide support for assessing how well a microservice API is conforming to a Domain Model it was derived from.* In particular, we aim to study this goal in the context of three common API design decisions: (1) API endpoint operations in relation to Domain Model operations, (2) API links in relation to Domain Model links, and (3) resource segregation of API operations in relation to Domain Model operations. To address our research goal, we need to answer the following **research questions**:

- **RQ4.1** For which decision options in microservice API design decisions can we provide automated support for assessing conformance to the Microservice API's Domain Model?
- **RQ4.2** To which extent can we assess conformance of API design decisions to the Microservice API's Domain Model?
- **RQ4.3** How can we automatically assess conformance to ADD options used in the mapping of domain model elements to APIs and API endpoints?
- **RQ4.4** How well do such automated measures for assessing ADD conformance in the mapping of domain model elements to APIs and API endpoints perform?

The goal fulfillment will be measured using a couple of **metrics**: We use simple count-based metrics to measure for how many decision options our approach provides automatic conformance assessment. We also use count-based metrics for measuring the extent to which automated conformance assessment is supported; in addition, we use statistical methods to measure the effect size.

In particular, we performed a detailed study of the decision driver impacts in the decision options to derive an empirically grounded scoring scheme for the assessment of API endpoint designs, based on the empirical data from our prior study [157] (only summarized in this chapter). We collected and modeled cases in a multi-case study and manually assessed each case. This way we created a ground truth for evaluating our automated approach later on. Next, to support the automated assessment we developed detectors to identify each of the decision points. Finally, we mapped the detector results to our ground truth in order to automatically assess the cases in our case study evaluation.

This chapter is organized as follows: Next, in Section 5.2 we describe the three API endpoint ADDs from our prior study as background. Then we describe in Section 5.3 our

research methods. We give an overview of the case in our multi-case study and the scoring scheme in Section 5.4. Our automated detectors are described in Section 5.5. We discussed the results fo the case studies related to the mapping of domain model elements to APIs and API endpoints in and the Section 5.6 API endpoing Designs derived from domain models in Section 5.7 followed by a discussion of our multi-case study results, a statistical analysis, and lessons learned in . 5.8 Finally, we compare our finding to related work in Section 5.9, discuss threats to validity in Section 5.10, and draw the conclusion in Section 5.11.

5.2 Background

In this section, we discuss three ADDs on API endpoint design from our prior study [157] as background of this work. In our prior study, we have conducted a Grounded Theory study [51] based on the 32 grey literature sources (i.e., practitioner sources such as blog posts or system documentations [50]). Then, we present the ADDs along with their relations (to decision options, other decisions, and patterns/practices) and their decision drivers. All decisions drivers used in this section are defined in Table 5.1.

Table 5.1: Glossary of Decision Drivers Used in the ADDs

Decision Driver	**Definition**
Data Consistency	"[Data] consistency [...] can be seen as the guarantee that transactions started in the future see the effects of transactions committed in the past" [89].
API Evolvability	"The ability of [an API (was: a system)] to accommodate changes in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity" [141].
API Modifiability	"Modifiability describes that the code [and design of the API realization] facilitates the incorporation of changes, once the nature of the desired change has been determined" [12].
Message Sizes	The smaller the message sizes in an API, the less bandwidth is used. Larger messages can help avoid sending multiple distributed messages.
Protocol Complexity	Many links in API messages lead to a complex interaction protocol.
Performance	"Performance prescribes conditions on functional requirements such as speed, efficiency, availability, accuracy, throughput, response time, recovery time, and resource usage" [12].
Minimize API Calls	Each API call costs the performance of a distributed invocation, which is usually much higher than for many local operations.
Scalability	"Scalability is the ability to handle increased workload (without adding resources to a system)" [187]
Exposing Domain Model Details in API	The API should be an abstraction of the Domain Model only revealing those aspects needed for the API functions and nothing more.
Coupling of Clients to Server	API clients and servers should be as loosely coupled as possible, so that independence maintenance, testing, and motification is possible.
Interface Designs that Limit Domain Model Designs	An API design should not limit the design of the Domain Model, but provide an abstraction of it.
Maintainability	"The ability to undergo changes, including error correction and system adaptation" [12].
API Understandability	"[API] understandability describes that the purpose of the [API design and] code is clear to the inspector." [12].
Eventual Consistency Support	Eventual consistency describes a weak consistency relation which requires that all replicas of an object (here: API/DDD elements) will only eventually reach the same correct value.

This chapter assumes familiarity with basic DDD concepts, in particular: Evans [33] classifies domain objects into types such as *Entities*, *Value Objects*, and *Services*, which are then used to identify larger structures such as *Aggregates*. Such elements contain domain links in the domain model, as well as Domain Operations and/or Domain Events. At the next higher abstraction level, DDD introduces the notion of Strategic Design [33, 182]

which explains how to structure large domains into a number of *Bounded Contexts* and their relations. The ADDs described here explain how to map domain model elements to APIs and API endpoints (MMD, API-ED, API-DD) and how to derive API endpoint designs from the information in such DDD domain models (LMD, ODD, RSD).

Model Mapping Decision (MMD)

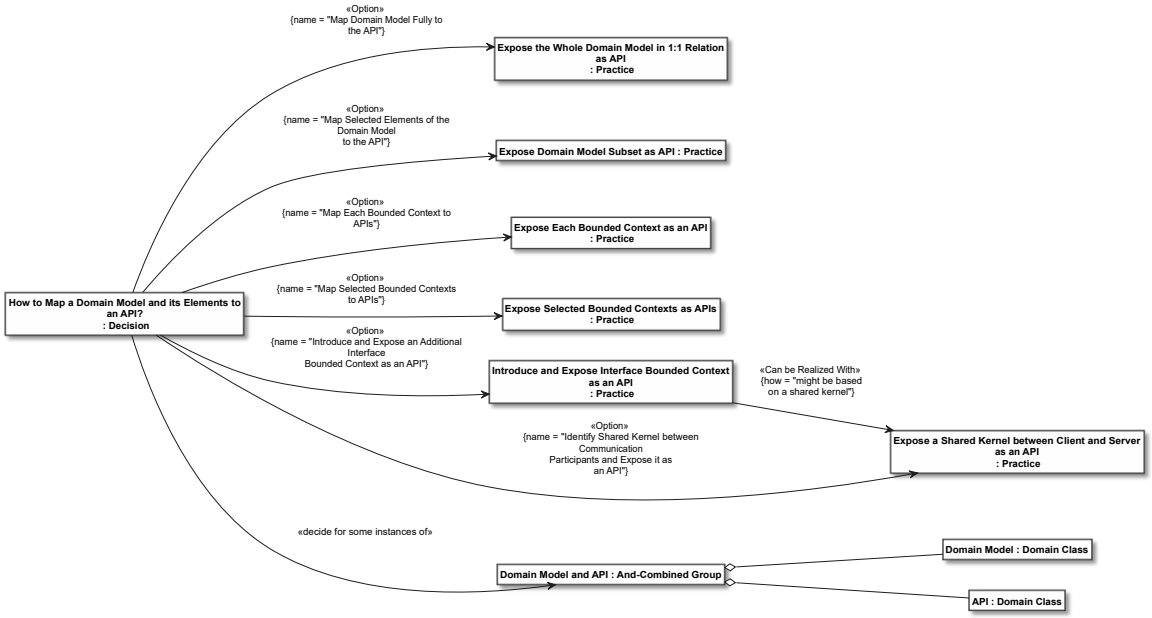

Figure 5.1: Model Mapping Decision

This ADD focuses on how to map the domain model and its elements to an API. There are five alternative design options practitioners commonly apply in this ADD. Firstly, one can *Expose the Whole Domain Model in 1:1 Relation as API*, but this is seen rather negatively as it increases coupling and adversely impacts many API maintainability aspects. An often better working solution is *Expose Domain Model Subset as API* as it can reduce some of those negative impacts. Many practitioners use DDD's *Bounded Context* pattern [33] for defining boundaries of the APIs with the options *Expose Selected Bounded Contexts as APIs* being seen more positive than the option *Expose Each Bounded Context as an API*. Still, even in selected Bounded Contexts, domain model elements that do not belong to the API might get exposed, thus the options *Introduce and Expose Interface Bounded Context as an API* and *Expose a Shared Kernel between Client and Server as an API* are seen as the most favourable options, as in them a special *Bounded Context* or a *Shared Kernel* [33] is designed with the goal to specify an API interface.

API Endpoints Decision (API-ED)

This ADD addresses the problem which domain model elements should be offered as endpoints in an API. Please note that it includes endpoints in various kinds of technologies such as RESTful HTTP, messaging, SOAP, gRPC, and so on. The options for this ADD

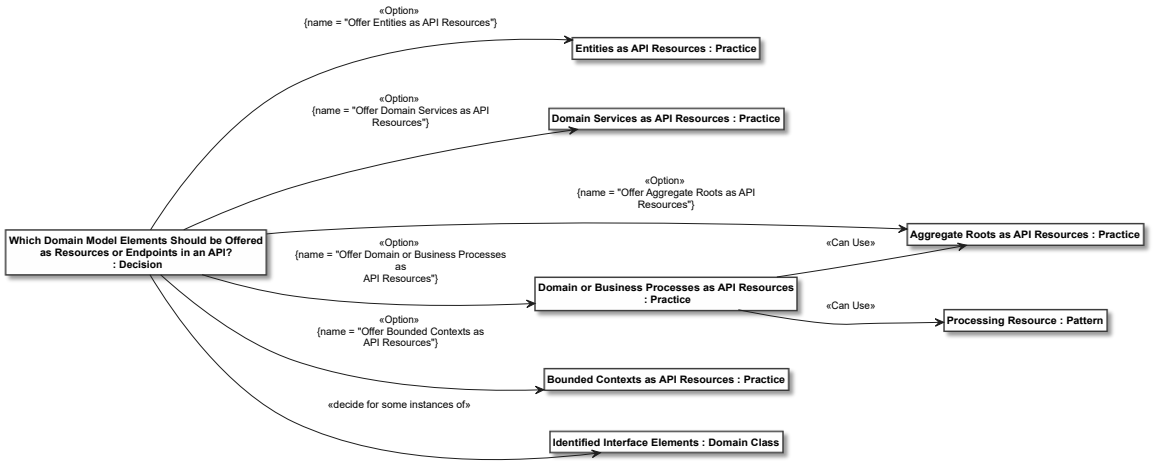

Figure 5.2: API Endpoint Decision

are related to the basic patterns in tactical DDD [33]. The most positive options of this decision are those that expose *Aggregate* roots or *Service* as API endpoints. The option to expose *Entities* as API endpoints is often discussed, but also has many negative impacts e.g. on exposing domain model details in the API, data consistency, and chatty APIs. Broader concepts such as *Bounded Contexts* and *Processes* are also options, but issues regarding API complexity, data consistency, and coupling are reported for these options.

API Documentation Decision (API-DD)

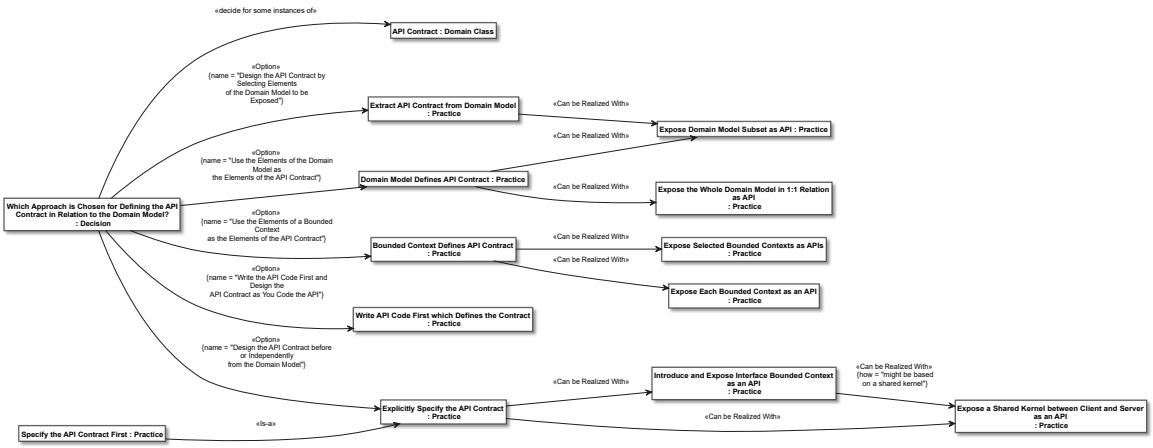

Figure 5.3: API Document Decision

This ADD is about how to document the API. There are two main options that can be combined in different ways: *API Contract* and *API Description* [90, 212]. The *API Contract* specifies structural information about the API in a formal language, e.g., based on Open API or RAML for RESTful APIs, WSDL for SOAP APIs, and so on. The *API Description* is a detailed specification of the API and contains more information than the contract, such as invocation sequences, pre-conditions, post-conditions, quality management policies, and so on. API Descriptions can be classified into *informal* or *structured* descriptions. Combinations of API contracts and structured descriptions available for all APIs are seen

as the most positive option. More incomplete or less formal combinations are gradually seen as more negative.

Link Mapping Decision (LMD)

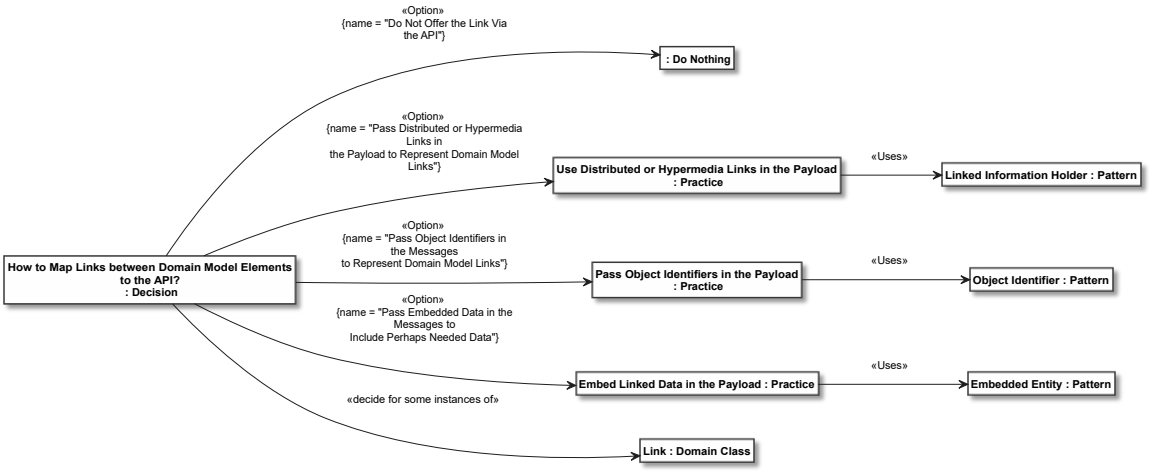

Figure 5.4: Link Mapping Decision

In APIs, the links between API endpoints play a central role. In DDD the links between model elements have a similar role. Obviously not all links in the DDD model are candidates to be exposed in an API, but only those between the model elements offered as API endpoints. The Link Mapping Decision with its decision options and links is shown in Figure 5.4. One decision option is *Use Distributed or Hypermedia Links in the Payload*. It means to use standard distributed/hypermedia links, such as URIs in RESTful HTTP or URIs and HAL/JSON-LDs in JSON. An alternative is *Embed Linked Data in the Payload* where the content is added to the message payload instead of linking to it. Finally, there is the option to *Pass Object Identifiers in the Payload* which sends an *Object Identifier* that is meaningful (only) in the server context, but contains no remote location information such as a distributed link. Compared to the embedding option, use of distributed links is beneficial for *Data Consistency* as the link is always up-to-date, and thus *API Evolvability* and *API Modifiability* are positively influenced. It leads also to smaller *Message Sizes*. Links however lead to higher *Protocol Complexity*, making it harder to *Minimize API Calls*, and can have a worse *Performance* and *Scalability* because of many resulting distributed calls. The *Object Identifier* based option is very similar in its effects to the distributed links based option. In direct comparison, it has the disadvantages of possibly *Exposing Domain Model Details in API* as well as higher *Coupling of Clients to Server*.

Operation Design Decision (ODD)

The ODD decision, shown in Figure 5.5, is on how to design the operations of an endpoint. A simplistic option, especially in a RESTful context, are *CRUD-style Operations on Endpoints*, which designs operations like primitive data store operations. This practice, while commonly used, is seen negatively for many decision drivers. In particular, in contrast to the options

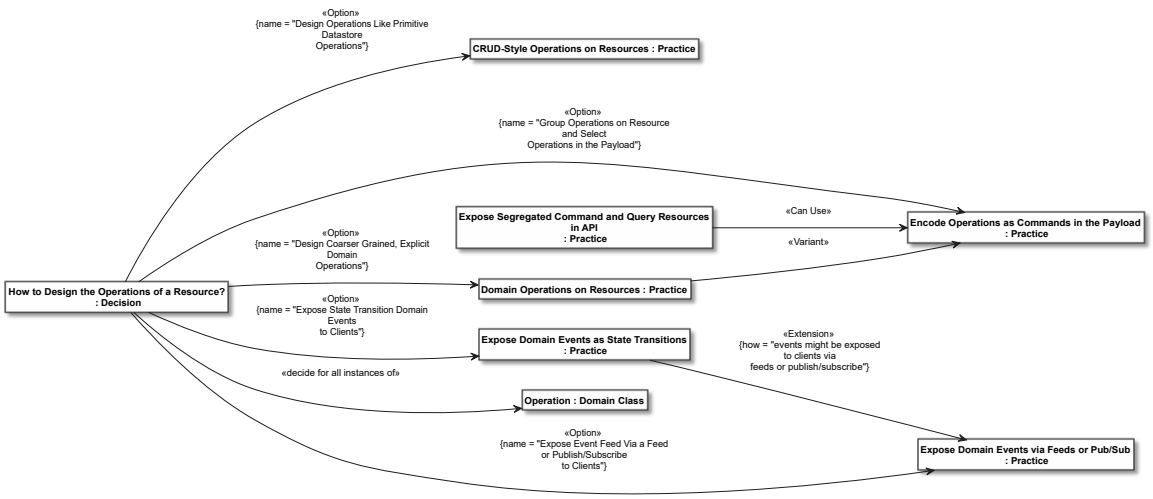

Figure 5.5: Operation Design Decision

below it is negative for *Avoiding Exposing Domain Model Details in API* and can lead to Chatty APIs, i.e. *Protocol Complexity* because of many small messages (*Minimize API Calls* not followed) with bad *Performance* and *Scalability*. It is argued that it can lead to *Interface Designs that Limit Domain Model Designs*, various *Maintainability* issues including *Coupling of Clients to Server* issues, and *Data Consistency* problems. On the positive side, *CRUD-style Operations on Endpoints* are simple and good for *API Understandability*. The option to expose *Domain Operations on Endpoints* leads to more coarse-grained designs. This option is usually more positive on the mentioned decision drivers, but needs more design work when mapped to a RESTful API. A variant of it is *Encode Operations as Commands in the Payload* which can be harder to understand than domain operations exposed via other means such as protocol features. It must further be noted that the distinction into good domain operations and bad CRUD-like operations is not valid either. In many cases, a CRUD-like operation is the best suited option. The actual decision criterion is: Is a further abstraction of the operation possible and does it make sense? For that reason, it is often beneficial that operations are exposed from an aggregated domain model element, such as an Aggregate, its Aggregate root, a Bounded Context, or a Service.

Many microservice systems are event-based systems. In such systems, another option is to *Expose Domain Events as State Transitions* (or its variant *Expose Domain Events via Feeds or Pub/Sub*). These options are also positive for most of the mentioned forces. They can even lead to a better solution for *Exposing the Domain Model in the API*, if events are used to model the domain, and/or improving *Scalability*. Events can be harder to understand and thus these options can have some issues with regard to *API Understandability*.

Resource Segregation Decision (RSD)

For API Endpoints, there sometimes is the option to decide whether or not to *Segregate Resources for Reading and Updating Information in an API*. This is closely related to the *Command Query Responsibility Segregation (CQRS)* Pattern [136]. The idea of CQRS is

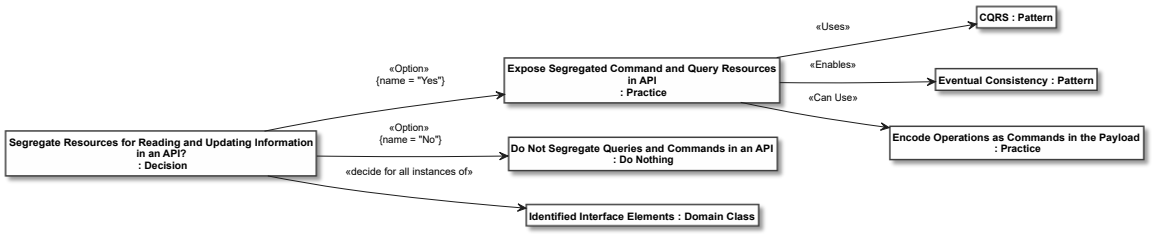

Figure 5.6: Resource Segregation Decision

to use a different model to update data than the model that is used to read data. The Resource Segregation Decision is shown in Figure 5.6.

The CQRS option has the benefit of possibly improving *Scalability* and enabling *Eventual Consistency Support* where it is needed, e.g. in long running transactions. Downsides are lower *API Understandability* due to API complexity and less *Data Consistency*. Thus, typically, it shall not be used, if simple domain model elements (e.g. just a few Entities or Value Objects) are offered on an API endpoint. Usually, it makes sense to apply this option together with event-based API operations. For the configuration, we have chosen the case studies that be able to define the ADD options.

Illustrative Example

Figure 5.7 shows an excerpt of a domain model, designed using DDD (adapted from the Case Study PM [204]). Such models are frequently used as a starting point to identify microservices and derive microservice APIs [36, 100, 157]. In the model, an Aggregate Paper Archive Facade is designed for managing Paper Collections, which is itself an Entity using Paper Item Entities and having Paper Item Keys as Value Objects. For exposing the Paper Archive, further a Service is designed, which contains three Domain Operations: Lookup Paper from Authors, Create Paper Item, and Convert to Markdown.

Many designs are possible to map this simple DDD model excerpt to an API. The one chosen in the PM Case Study [204] is provided in Figure 5.8. One API Endpoint is designed for the Paper Archive Facade, its Service, and the Collection Entity. Thus, it was chosen to not provide the domain model links from the Aggregate to the Service and Entity as explicit elements of the API model (i.e., the Do Nothing option of LMD is chosen here). The three domain operations of the Service are exposed via same-named API Operations. As Create Paper Item is a simple creation function, the CRUD-based API Operation option of the ODD decision was chosen here. The two other API operations were mapped as Domain-Based API Operations, as these operations provide coarser-grained operations, closer to the domain design. None of them uses the Encode Operations as Commands in the Payload variant from the ODD decision.

The Paper Collection Entity and the Paper Archiving Service have links to Paper Item and Paper Item Key. Here, the Embedded Data option of LMD is chosen for mapping all those links. Consequently, the Paper Item Entity and Paper Item Key Value Object are mapped to API Data Types. Further, decisions about the precise variant of the Embedded

Data option of LMD are made: whether the data is required or provided, and if the data elements are usually needed by clients or not (default value: False).

Finally, for technically realizing the API, another API Data Type for the parameters of the Paper creation (not modeled in the domain model) and a String type are needed. The String type is used once to provide an Object-Based Link as another option of LMD, for the other ones again the Embedded Data option is chosen.

Resource Segregation is not used in this API. Thus for RSD the option Do Nothing was chosen for all interface elements.

While this illustrative example is rather simple, many other possible designs could have been chosen even in this small-scale example that would have led to very different API designs. For instance, designers could have decided to expose all domain model links as hyperlinks, or resource segregation could have been offered as well. Assessing such designs, especially if they grow substantially larger and/or if the assessment is needed continuously, is tedious and error-prone. Our approach aims to automate the model-based assessment of the conformance between API and domain model designs.

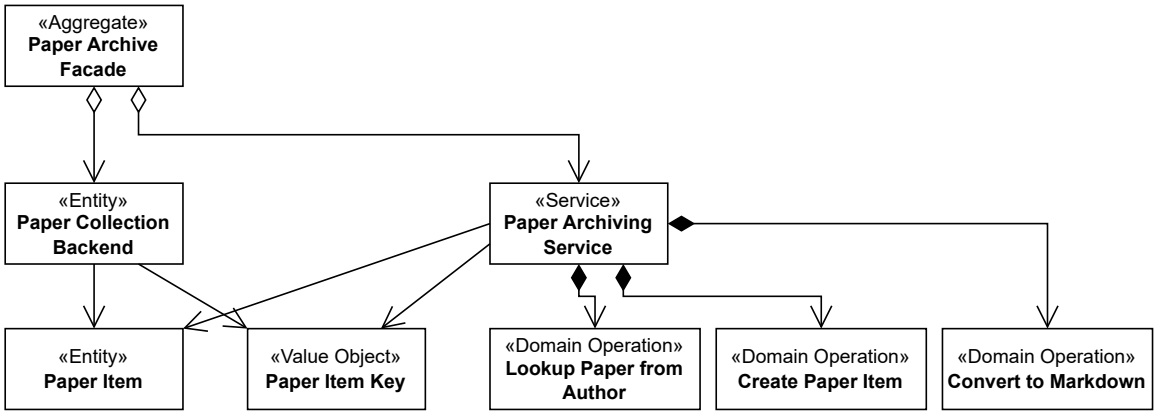

Figure 5.7: Example of a DDD Model (Excerpt from Case Study PM (adapted from [204])

5.3 Research Method

Figure 5.9 illustrates the research methods applied in our study. Please note that we have applied a very similar combination of research methods in earlier works [107, 108] before. To provide an audit trail of the research and enable repeatability of the study, we provide open access to our the data set and the source code[1].

Gray Literature Study of ADDs on API Endpoint Designs Derived from Domain Models In our previous work [157], we have empirically identified ADDs on deriving APIs from DDD [33] models. In particular, we conducted a Grounded Theory study [51] based on 32 gray literature [50] sources (i.e., practitioner sources such as blog posts or

[1]We provide derived coded models in Python, and generated models (in UML, Markdown, and Latex) as a replication package for download in the Zenodo long-term open access archive [155].

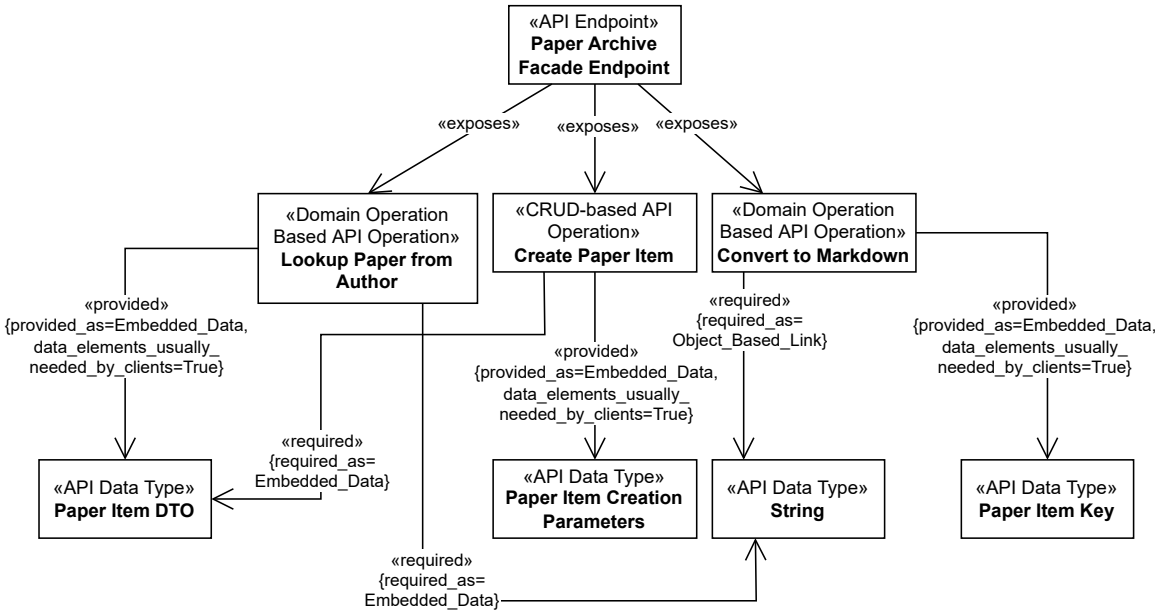

Figure 5.8: Example of an API Model (Excerpt from Case Study PM (adapted from [204])

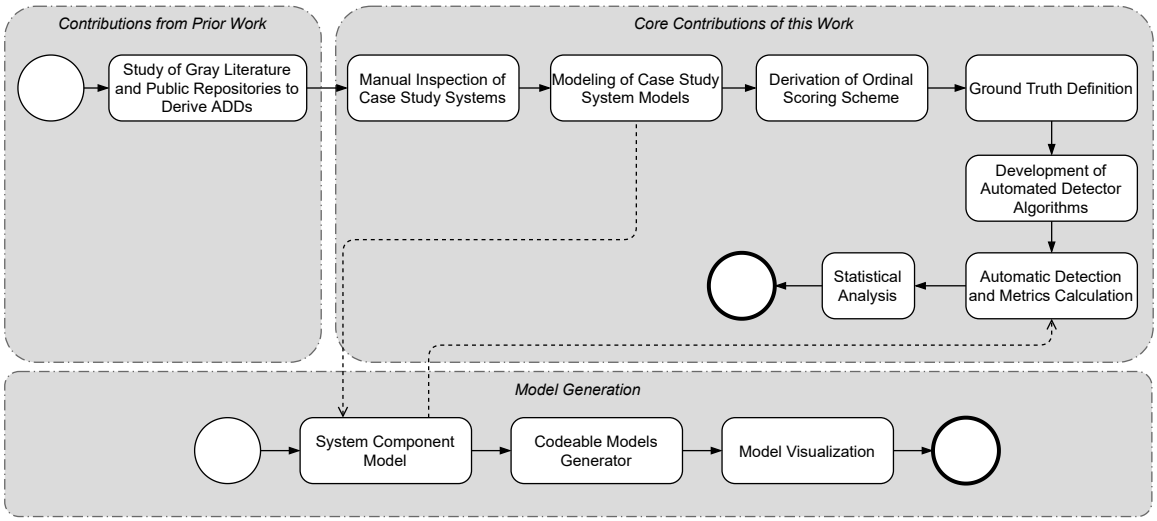

Figure 5.9: Research Methods Overview

system documentations) concentrating on the interrelation of microservice API design and DDD. We have identified ADDs frequently employed by practitioners, along with decisions options, decision relations, and decision drivers. For this study, we selected the three ADDs summarized in Section 5.2, which are all focusing on API endpoint design related issues. We have selected those ADDs, as many central API design problems revolve around API endpoint design [212, 211].

Multi-Case Study Preparation and System Modeling In the next steps, we manually inspected case studies of systems realized by practitioners, either from the published source code of these systems in public repositories, their system documentations, or both. In this, we followed the guidelines for conducting and reporting case study research in software engineering by Runeson et al. [142]. We used major search engines (e.g., Google, Bing,

DuckDuckGo) and topic portals (e.g., InfoQ, DZone) to find relevant cases. One major concern about search engines in such research is their search algorithm because the results are dependent on the user [123]. To avoid personal bias in the research, we used as search keywords that provided decision categories (i.e. the most general categorization) in our detailed prior study of the gray literature [157], in particular: "Application Programming Interface" or "API", "Domain Driven Design" or "DDD", and "Microservices" (all in singular and plural form). We had to check that the found cases contained information on their domain modeling, which substantially limited the possible cases we could consider. Further, we checked for each found case whether authors realized it with a substantial background in industrial practice. Finally, we selected systems from many different domains covering a broad range of our ADDs' decision options and their combinations (for details see Table 5.2 explained below). We assume that our evaluation systems are, or reflect, real-world practical examples of microservice architectures. As many of them are open source systems with the purpose of demonstrating practices or technologies, they are at most of medium size and modest complexity, though.

We then defined models and meta-models to precisely model the case study systems as UML models. We used our existing CodeableModels tool[2], a Python implementation for precisely specifying meta-models, models, and model instances in code. Based on CodeableModels, we realized automated code generators to generate graphical visualizations of all meta-models and models in PlantUML[3].

Please note that this modeling step was done manually in our work for most of the case study systems. However, in a prior work we have shown that it can be automated, too. Our existing static code analysis approach for architecture reconstruction of polyglot microservice systems [106] can be applied here. Due to the polyglot nature of microservice systems, this requires modest initial specification effort, though. So far, only two of the APIs of the systems in Table 5.2 have been automatically reconstructed, namely, the ES and LS API models (see [106]). To illustrate the manual effort, in [106] we compared the lines of code: For the 35.4 KLoC of code in LS, we required 804 LoC of specification code; for the 81.4 KLoC of code in ES, we required 716 LoC of specification code. As those are among the largest models in our model set, it is highly likely that the other system models can be reconstructed with significantly less effort. Please note that in [106] we only reconstructed the API part, not the DDD model elements and links yet.

Ground Truth Definition We then performed a systematic assessment on support or violation of the collected ADDs' decision options. The result is an ordinal scoring scheme which is directly based on the decision driver impacts empirically identified in our gray literature study. The ordinary scale helped us turn qualitative judgments in the practitioner texts into numerical assessments. We used the scoring scheme to manually assess the cases

[2] `https://github.com/uzdun/CodeableModels`
[3] `https://plantuml.com/en/`

in our multi-case study to create a ground truth, which we could use later on to validate our approach.

Automated Detector Algorithms and their Application To automate the assessment provided by the scoring scheme, we developed automated detector algorithms which aim to detect each relevant criterion deciding on scores in the scheme. Furthermore, we implemented each automated detector in Python (based on our coded UML meta-models) and wrote code generators to automatically enable running them on any system model conforming to our meta-models, such as the case study models.

We applied the automated detectors on our case study data set and calculated simple count-based metrics to measure for how many decision options our approach provides automatic conformance assessment and the extent to which automated conformance assessment is supported.

Statistical Analysis Finally, we performed an empirical assessment of our approach using a multi-case study using the cases inspected. We used the manual assessments in the cases of the multi-case study as a "ground truth" and compared them to our automatically derived scores from the detectors. Besides analyzing and discussing the results on a case-by-case basis for each decision option, we also perform a statistical analysis of the results. In it, we use R's *shapiro.test()* function to perform Shapiro-Wilk normality tests [151]. As the data sets are non-normally distributed, we used the Wilcoxon signed rank test with Pratt method [125], provided by the *wilcoxsign_test()* function from R's *coin* package, to test for a significant difference between the ground truth and the automated detector results. Finally, Cliff's δ [21] is used for effect size estimation via *cliff.delta()* function from R's *effsize* package.

5.4 Multi-Case Study Preparation and Inspection

This section explains how we prepared and inspected the cases for our multi-case study. Table 5.2 summarizes the cases which we have manually modeled based on available source code, documentation, descriptions in articles or blog posts, and so on. The table summarizes for each model the size of the cases in terms of modeled domain models and API model elements. We also modeled in each of those models the relations of the elements among each other, and the relations between DDD and API model elements in detail. The third column of the table contains a brief description of the case, the options selected in the case for each of the ADDs from Section 5.2, and the URL of the original case source.

We have modeled DDD domain models, microservice API models, and the mapping of DDD domain models to API models for each case. The information in the table includes the number of domain elements, number of API elements, a brief description, the solutions for each of the ADDs, and a URL for the original sources of the cases.

Table 5.2: Overview of 14 Cases in the Multi-Case Study:API Endpoint Designs Derived from Domain Models

ID	Elements	Description and Summary of ADD Inspection
AX	domain: 5 api: 14	Purchase Order System; applies DDD concepts and CQRS to decompose components; no details on link and operation design provided. ADD Options used: **LMD** (N/A), **ODD** (N/A), **RSD** (CQRS on aggregating endpoints, no details on operations). `https://dzone.com/articles/bounded-contexts-with-axon`
PM	domain: 10 api: 11	Publication Management System; applies DDD concepts to API design, detailed API specifications and code generation. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Explicit Domain Operations & Design Operations Like Primitive Datastore Operations, exposed to API by an aggregate root), **RSD** (N/A). `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html`
BA	domain: 8 api: 39	Bank Account System; focuses on event-based architecture using CQRS and event sourcing patterns. ADD Options used: **LMD** (Pass Object Identifiers in the Messages to Represent Domain Model Links & Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Event State Transition Operations, based on Domain Operations & CRUD-based operations, some exposed to API by aggregating services and aggregates), **RSD** (CQRS on aggregating endpoints, event-based operations). `https://github.com/cer/event-sourcing-examples`
OS	domain: 19 api: 18	Online Shop System; uses DDD and pattern-based modeling of APIs. ADD Options used: **LMD** (Pass Object Identifiers in the Messages to Represent Domain Model Links & Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Design Operations Like Primitive Datastore Operations, all exposed to API by Entities), **RSD** (N/A). `https://github.com/socadk/design-practice-repository/blob/master/tutorials`
CM	domain: 12 api: 52	Cinema Microservice System; models multiple frontends and REST-based APIs. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data, except for one operation), **ODD** (Explicit Domain Operations & Design Operations Like Primitive Datastore Operations, all exposed to API by aggregating services), **RSD** (CQRS on aggregating endpoints). `https://github.com/Crizstian/cinema-microservice`
ES	domain: 4 api: 142	E-Shop on Containers System; follows the CQRS pattern; uses event transitions and pub/sub for event-based interaction. ADD Options used: **LMD** (Pass Distributed or Hypermedia Links in the Payload to Represent Domain Model Links & Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Event State Transition Operations, based on Domain Operations & CRUD-based operations, & Expose Publish/Subscribe to Clients, some exposed to API by aggregating bounded contexts, some not), **RSD** (CQRS on aggregating endpoints, event-based operations). `https://github.com/dotnet-architecture/eShopOnContainers`
ET	domain: 8 api: 15	Customers and Orders System; implements transaction with the SAGA pattern and implements queries using CQRS. ADD Options used: **LMD** (N/A), **ODD** (Event State Transition Operations, based on CRUD-based operations, all exposed to API by aggregates), **RSD** (CQRS on aggregating endpoints, event-based operations). `https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders`
KB	domain: 11 api: 34	Kanban Board System; a multi-user collaborative application using event-sourcing and pub/sub. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Event State Transition Operations, based on CRUD-based operations, all exposed to API by bounded context), **RSD** (CQRS on aggregating endpoints, no details on operations). `https://github.com/eventuate-examples/es-kanban-board`
NC	domain: 8 api: 72	Disease Statistics App; a public API providing a wide range of virus information. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data, but not needed on some operations), **ODD** (Design Operations Like Primitive Datastore Operations, exposed to API by Entities), **RSD** (N/A). `https://github.com/disease-sh/API`
PC	domain: 63 api: 215	Pokemon App; a RESTful API for infos on Pokemon game series. ADD Options used: **LMD** (Pass Distributed or Hypermedia Links to Represent Domain Model Links, but in many cases it seems clients usually need the data immediately), **ODD** (Design Operations Like Primitive Datastore Operations, exposed to API by Entities), **RSD** (N/A). `https://github.com/PokeAPI/pokeapi`
RW	domain: 6 api: 67	Realworld Example App; provides API specs that support technology stack diversity. ADD Options used: **LMD** (Pass Embedded Data in the Payload, clients usually need the data), **ODD** (Design Operations Like Primitive Datastore Operations & Group Operations on Resource and Select Operations in the Payload, all exposed to API by aggregating bounded contexts), **RSD** (N/A). `https://github.com/gothinkster/realworld`
LS	domain:170 api: 92	Lakeside Mutual System; demonstrates DDD in microservices of an insurance product. ADD Options used: **LMD** (Use Pass Distributed or Hypermedia Links on frontend to connect to backends which also Pass Embedded Data in the Payload, where clients usually need the data), **ODD** (Explicit Domain Operations & Design Operations Like Primitive Datastore Operations, all exposed to API by aggregating bounded contexts), **RSD** (N/A). `https://github.com/Microservice-API-Patterns/LakesideMutual`

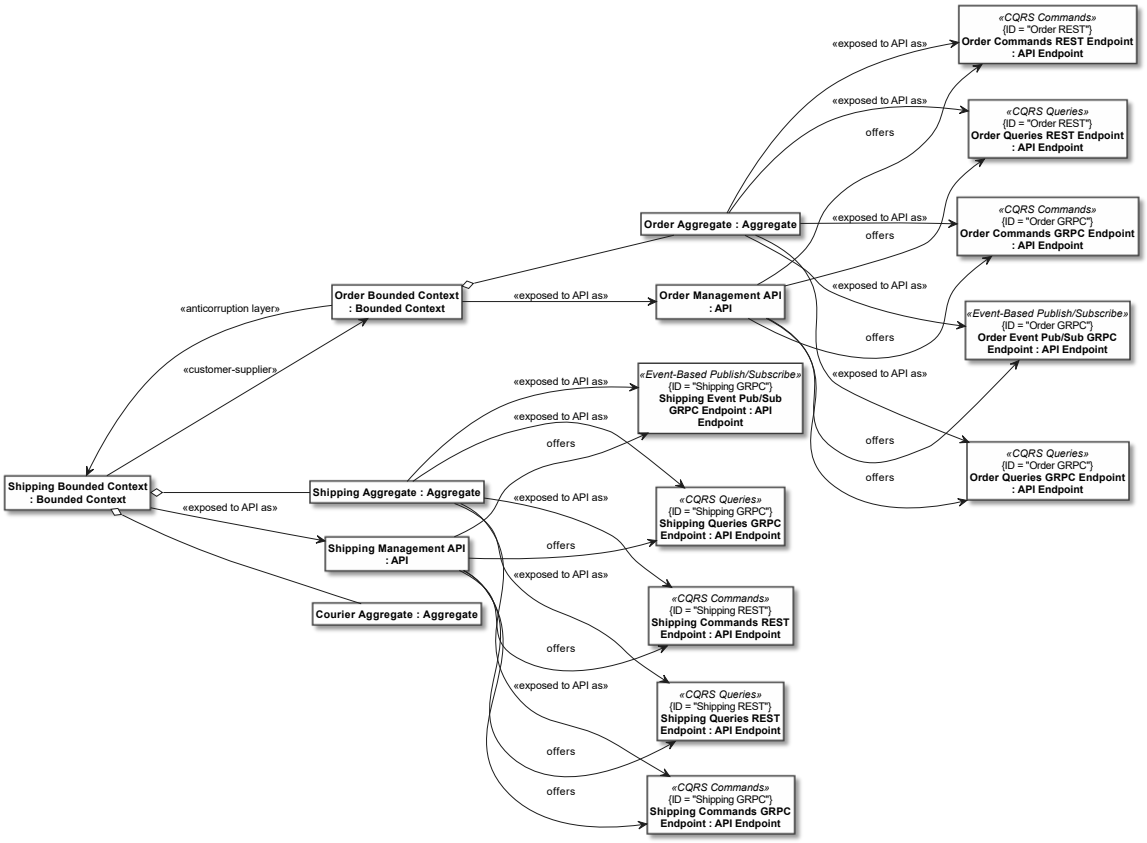

Figure 5.10: Illustrative Example: Overview of DDD/API Mapping in Model MD1

As an illustrative example of the modeling of the mapping of DDD domain models to API models, Figure 5.10 shows the Model MD1. In it, the detailed association between domain model elements and API elements of MD1 can be seen. The model contains five domain elements and 14 API elements. The domain elements are the two Bounded Contexts (*Order Bounded Context*, *Shipping Bounded Context*) and three aggregates (*Order Aggregate*, *Shipping Aggregate*, *Courier Aggregate*). *Order Aggregate* belongs to *Order Bounded Context*, whereas *Shipping Aggregate* and *Courier Aggregate* belong to *Shipping Bounded Context*. Moreover, the relation between these two bounded contexts follows the *Customer-Supplier* pattern [33], and in this relation *Order Bounded Context* offers additionally an *Anti-Corruption Layer* [33]. This domain model is mapped to two APIs called *Shipping Managment API* and *Order Management API*. The *Shipping Managment API* has a *Shipping Managment API Contract*, whereas *Order Management API* has an *Order Management API Contract*; both contracts are expressed as service API contracts published as a schema. Both APIs offer the combination of the RESTful and gRPC invocation options. Each API has five API endpoints that follow the *CQRS* [135] and *Publish-Subscribe* [17] patterns. The *exposed to API as* relations in Figure 5.10 represent the mappings between DDD and API model elements. From this relation, we can deduce the options selected for the decisions. As can be seen, the two Bounded Contexts are exposed to APIs; that is, the option *Expose Each Bounded Context as an API* of the *MMD Decision* is chosen in this model. Please also note in the context of this decision that not all DDD model elements

are 1:1 mapped into the APIs, but rather selected elements are exposed. For the *API-ED decision*, we can observe that the *exposed to API as* association is used between aggregates and API endpoints. Therefore, for this decision, the option *Aggregate Roots as API Endpoints* is chosen in this model. Lastly, for the *API-DD decision*, we can see that both APIs have an API Schema, which indicates that the chosen option for this decision is *Each API has an API Contract*.

As an illustrative example, we explain the case study preparation and ADD inspection of the *ES* model. Figure 5.11 shows the Basket RESTful Endpoint's elements and its associations, as well as the Bounded Context domain model element from which it is derived. Please note that this is just a small excerpt of the *ES* model, and we do not show other domain model elements for sake of brevity here. Most of models contain complex domain models with domain links, domain operations, and many other domain elements, which are related to API elements such as the ones shown in Figure 5.11.

Listing 5.1: GetBasketByIdAsync Operation from Basket API

```
[HttpGet("{id}")]
[ProducesResponseType(typeof(CustomerBasket), (int)HttpStatusCode.OK)]
public async Task<ActionResult<CustomerBasket>> GetBasketByIdAsync(string id)
{
  var basket = await _repository.GetBasketAsync(id);
  return Ok(basket ?? new CustomerBasket(id));
}
```

The endpoint offers four *API Operations*, namely *UpdateBasketAsync*, *GetBasketByIdAsync*, *DeleteBasketByIdAsync*, and *CheckoutAsync*. They use four *API Data Types*: (*Customer Basket Data Type*, *Customer Basket ID Data Type*, *Request ID Data Type*, and *Basket Checkout Data Type*). Firstly, for the LMD decision, we investigated the relationship between *API Operations* and *API Data Types*. For instance, the *GetBasketByIdAsync* operation shown in Listing 5.1, receives a Basket ID as a string which is modeled as an *ObjectIDBasedLink* and provides *CustomerBasket* as *EmbeddedData*. As for LMD the ≪*provided*≫ data elements are mainly needed to determine which decision option is chosen, we can conclude here that *GetBasketByIdAsync* represents passing embedded data in the messages. When inspecting the code closer, we can further find out that clients likely need the data provided immediately. Secondly, for the ODD decision, we studied the *API Operation* and how its endpoint is exposed. For instance, for the *CheckoutAsync* operation shown in Listing 5.2, it is a domain operation which is used to perform an event-based state transition. Thus, the two respective stereotypes ≪*Event State Transition Operation*≫ and ≪*Domain Operation Based API Operation*≫ are used on that operation in the model. Moreover, the operations are offered on an endpoint that is exposed by an aggregating domain model element (a Bounded Context), which are recommended practices for ODD (see Section 5.2). Lastly, for the RSD decision, the system documentation reveals that this endpoint is intended as a ≪*CQRS*≫ type endpoint. Further, we can see that this endpoint exposes event-based operations and already determined that it is exposed by a Bounded Context (i.e., is aggregating), which are recommended practices for RSD (see Section 5.2).

Table 5.3: Overview of 12 Cases in the Multi-Case Study:The Mapping of Domain Model Elements to APIs and API Endpoints

ID	Elements	Description and Summary of ADD Inspection
MD1	domain: 5 api: 14	Purchase Order System; applies DDD concepts and CQRS to decompose components. ADD Options used: **MMD** (Expose Each Bounded Context as an API), **API-ED** (Aggregate Roots as API Resources), **API-DD** (Each API has an API Contract). `https://dzone.com/articles/bounded-contexts-with-axon`
MD2	domain: 10 api: 11	Publication Management System; applying DDD concepts to API design, detailed API specifications and code generation. ADD Options used: **MMD** (Expose Selected Bounded Context as an API), **API-ED** (Aggregate Roots as API Resources), **API-DD** (Each API has an API Contract and a structured API Description). `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html`
MD3	domain: 8 api: 3	Bank Account System; focuses on event-based architecture, CQRS and event sourcing patterns. ADD Options used: **MMD** (Expose Domain Model Subset as API), **API-ED** (id=eve]some API resources it is unclear on which domain model elements they are based, and some are Domain Services as API Resources), **API-DD** (Each API has an API Contract). `https://github.com/cer/event-sourcing-examples`
MD4	domain: 19 api: 18	Online Shop System; uses DDD and pattern-based modelling of APIs. ADD Options used: **MMD** (Expose Selected Bounded Contexts as API), **API-ED** (Some Aggregate Roots and Entities as API Resources), **API-DD** (Each API has API Contract and structured API Desc.). `https://github.com/socadk/design-practice-repository/blob/master/tutorials`
MD5	domain: 12 api: 52	Cinema Microservice System; models multiple frontends and REST-based APIs. ADD Options used: **MMD** (Expose Each Bounded Context as an API), **API-ED** (Domain Services as API Resources), **API-DD** (Most APIs have an API Contract). `https://github.com/Crizstian/cinema-microservice`
MD6	domain: 4 api: 142	E-Shop on Containers System; follows both simple CRUD and DDD/CQRS patterns; uses pub/sub for event-based interaction. ADD Options used: **MMD** (Expose Each Bounded Context as an API), **API-ED** (For some API resources it is unclear on which domain model elements they are based), **API-DD** (Each API has an API Contract). `https://github.com/dotnet-architecture/eShopOnContainers`
MD7	domain: 8 api: 15	Customers and Orders System; implements transaction with the SAGA pattern and implements queries using CQRS. ADD Options used: **MMD** (Domain Model Subset Exposed to API), **API-ED** (Some Aggregate Roots as API Resources), **API-DD** (Each API has an API Contract). `https://github.com/eventuate-tram/eventuate-tram-examples-customers-and-orders`
MD8	domain: 3 api: 3	E-commerce Application; has a Web UI directly accessing microservices and an API gateway for service-based API. ADD Options used: **MMD** (Expose Each Bounded Context as API), **API-ED** (N/A), **API-DD** (No API description or contract). `https://microservices.io/patterns/microservices.html`
MD9	domain: 11 api: 34	Kanban Board System; a multi-user collaborative application using event-sourcing and pub/sub. ADD Options used: **MMD** (Expose Each Bounded Context as an API), **API-ED** (id=eve]Bounded Contexts and Domain Services as API Resources), **API-DD** (Some APIs have an API Contract). `https://github.com/eventuate-examples/es-kanban-board`
MD10	domain: 8 api: 72	Disease Statistics App; a public API providing a wide range of virus information. ADD Options used: **MMD** (Map Domain Model Fully to the API), **API-ED** (Entities as API Resources), **API-DD** (Each API has an API Contract and a structured API Description). `https://github.com/disease-sh/API`
MD11	domain: 63 api: 215	Pokemon App; a RESTful API for infos on Pokemon game series. ADD Options used: **MMD** (Domain Model Exposed 1:1 (except Bounded Contexts)), **API-ED** (Entities as API Resources), **API-DD** (Each API has an API Contract and a structured API Description). `https://github.com/PokeAPI/pokeapi`
MD12	domain: 6 api: 56	Realworld Example App; provides API specs that support technology stack diversity. ADD Options used: **MMD** (Expose Selected Bounded Context as API), **API-ED** (Bounded Context as API Resources), **API-DD** (Each API has API Contract and structured API Description). `https://github.com/gothinkster/realworld`
MD13	domain: 12 api: 6	Taxi hailing Application; uses REST APIs, multiple frontends, and databases per services. ADD Options used: **MMD** (Expose Each Bounded Context as an API), **API-ED** (N/A), **API-DD** (no formal or informal description or API contract). `https://www.nginx.com/blog/introduction-to-microservices`
MD14	domain: 170 api: 92	Lakeside Mutual System; demonstrates DDD in microservices of an insurance product. ADD Options used: **MMD** (Expose Selected Bounded Context as an API), **API-ED** (Bounded Contexts as API Resources), **API-DD** (Each API has an API Contract). `https://github.com/Microservice-API-Patterns/LakesideMutual`

Listing 5.2: CheckoutAsync Operation from Basket API

```
[Route("checkout")]
[HttpPost]
[ProducesResponseType((int)HttpStatusCode.Accepted)]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
public async Task<ActionResult> CheckoutAsync(
[FromBody] BasketCheckout basketCheckout,
[FromHeader(Name = "x-requestid")] string requestId)
{
  var userId = _identityService.GetUserIdentity();

  basketCheckout.RequestId =
  (Guid.TryParse(requestId, out Guid guid) && guid != Guid.Empty) ?
  guid : basketCheckout.RequestId;

  var basket = await _repository.GetBasketAsync(userId);

  if (basket == null)
  {
    return BadRequest();
  }

  var userName =
  this.HttpContext.User.FindFirst(x => x.Type == ClaimTypes.Name).Value;

  var eventMessage = new UserCheckoutAcceptedIntegrationEvent(
  userId, userName, basketCheckout.City, basketCheckout.Street,
  basketCheckout.State, basketCheckout.Country, basketCheckout.ZipCode,
  basketCheckout.CardNumber, basketCheckout.CardHolderName,
  basketCheckout.CardExpiration, basketCheckout.CardSecurityNumber,
  basketCheckout.CardTypeId, basketCheckout.Buyer,
  basketCheckout.RequestId, basket);

  // Once basket is checkout, sends an integration event to
  // ordering.api to convert basket to order and proceeds with
  // order creation process
  try
  {
    _eventBus.Publish(eventMessage);
  }
  catch (Exception ex)
  {
    _logger.LogError(ex,
    "ERROR Publishing integration event: {IntegrationEventId} from {AppName}",
    eventMessage.Id, Program.AppName);
    throw;
  }

  return Accepted();
}
```

Derivation of Scoring Scheme of the Mapping of Domain Model Elements to APIs and API Endpoints

In this section, we explain the scoring scheme to assess the ADDs in Section 5.2. We derived the assessment, which as objectively as possible follows the results of our empirical study by considering the various impacts on decision drivers reported by practitioners. Based on this empirical evidence, we decided to provide an assessment on an ordinal scale [++: very positive, + : positive, o : neutral, -: negative, - -: very negative] as it would have been hard to derive a more precise, numerical assessment from the qualitative judgments in the practitioner texts. For each of the decisions introduced in Section 5.2, we present precise

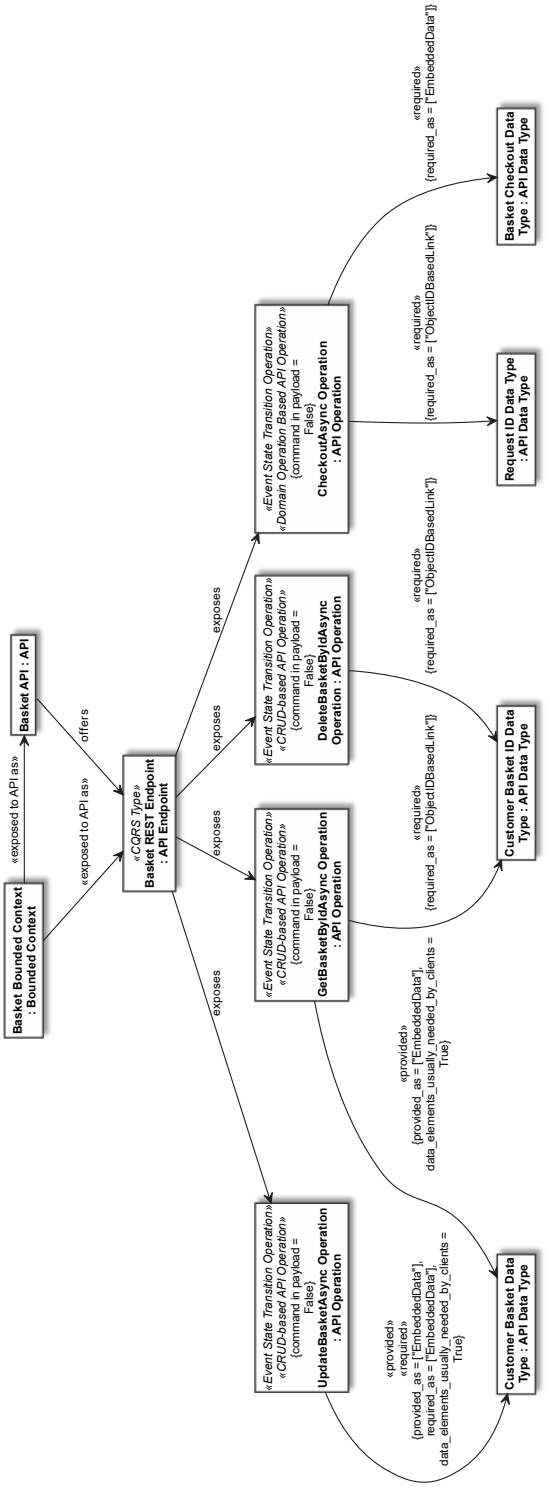

Figure 5.11: ES Model Excerpt: Basket RESTful API and its Mapping to a Bounded Context

decision points in conditional statements and boolean logic operators based on the possible decision options of the ADDs.

For example, consider the MMD decision, as discussed in [157]. According to the empirical evidence in the practitioner sources, *Expose the Whole Domain Model in 1:1*

Relation as API is seen as working only for small examples, as it leads to negative impacts on coupling and maintainability decision drivers such as *Brittle Interfaces*, *API Complexity*, and *Avoiding Exposing Domain Model Details in API*. A usually better working solution is *Expose Domain Model Subset as API* which partly improves on the negative decision driver impacts. Both solutions have benefits like little required *Design and Implementation Effort*. For complex domains, it can be advisable to consider the *Bounded Contexts* as well. Then there are the options to *Expose Each Bounded Context as an API* or *Expose Selected Bounded Contexts as APIs*. In most large domains, the latter solution is seen as being better suited to avoid *Brittle Interfaces*, *Exposing of Domain Model Details in the API*, and *API Complexity*, and improve *API Usability* and *API Modifiability*. Both solutions offer positive impact on *Design and Implementation Effort*, but require more effort than the first two solutions. The downside of those solutions is that *Clients Need to Manage Crossing Model Boundaries*, i.e., the boundaries between the *Bounded Contexts*. One suggested solution to this problem is to *Introduce and Expose Interface Bounded Context as an API* (or alternatively realized as a *Shared Kernel* [33]). That is, a new special *Bounded Context* or *Shared Kernel* that represents the API interface is exposed. These solutions are neutral or positive on all so far mentioned decision drivers, except the *Design and Implementation Effort* where they lead to additional effort compared to all other so far mentioned solutions. All those solutions are better than any *combinations of API and domain model where no fully traceable mapping between them can be discerned*.

These considerations have led to a literal derivation of the scoring scheme presented in the next section of the Decision MMD. For the other two decisions, their scoring scheme based on the empirical study results from [157].

MMD: How to Map a Domain Model and its Elements to an API?

- IF for some APIs no traceable mapping to domain model elements can be discerned, THEN assessment = (- -).
- ELSE IF the options *Introduce and Expose Interface Bounded Context as an API* and/or *Expose a Shared Kernel between Client and Server as an API* are used AND no other kinds of Bounded Contexts are exposed to the API, THEN assessment = (++).
- ELSE IF *Expose the Whole Domain Model in 1:1 Relation as API* is used, THEN assessment = (-).
- ELSE IF for all APIs *Expose Domain Model Subset as API* is used (where some of those might be realized using *Introduce and Expose Interface Bounded Context as an API* OR *Expose a Shared Kernel between Client and Server as an API*), THEN assessment = (+)
- ELSE IF for all APIs *Expose Selected Bounded Context as an API* (implies: NOT *Expose Each Bounded Context as an API*), THEN assessment = (+)

- ELSE assessment = (o) (e.g. (*Expose Each Bounded Context as an API*).

API-ED: Which Domain Model Elements Should be Offered as Endpoints in an API?

- IF for some API endpoints it cannot be discerned on which domain model elements they are based THEN assessment = (- -).
- ELSE IF for all API endpoints *Entities as API Endpoints* OR other domain model elements than [Services, Aggregates, Domain Processes, Bounded Contexts] as API Endpoints are used THEN assessment = (-).
- ELSE IF for all API endpoints either *Domain Services as API Endpoints* OR *Aggregate Roots as API Endpoints* OR *Domain OR Business Processes as API Endpoints* are used, THEN assessment = (++).
- ELSE IF for some API endpoints, *Entities as API Endpoints* OR other domain model elements than (Services, Aggregates, Domain Processes, Bounded Contexts) as API Endpoints are used, THEN assessment = (o).
- ELSE: assessment = (+) (e.g., if Bounded Contexts are used as Endpoints).

API-DD: How to formally describe the API?

- IF no API has a formal OR informal *API Description* OR *API contract*, THEN assessment = (- -).
- ELSE IF each API has an *API Contract* AND a structured *API Description*, THEN assessment = (++).
- ELSE IF each API has an *API Contract* AND an informal *API Description* THEN assessment = (+).
- ELSE IF each API has an *API Contract* OR a structured *API Description* THEN assessment = (o).
- ELSE: assessment = (-) (e.g. only an informal *API Description* or some APIs are not documented).

Derivation of Scoring Scheme of API Endpoint Designs Derived from Domain Models

We have established the scoring scheme to assess the ADDs in 5.2. This scoring scheme is based on our prior empirical study [157] by considering the various impacts on decision drivers reported by practitioners. Grounded on this empirical evidence, we decided to provide an assessment on an ordinal scale [++: very positive, + : positive, o : neutral, -:

negative, - -: very negative]. The ordinary scale could turn qualitative judgments in the practitioner texts to numerical assessment.

For example, consider the LMD decision, as discussed in [157]. According to the empirical evidence in the practitioner sources, if domain links are identified to be needed by clients, the option *Do not offer the Link via the API* shall not be used. It shall only be used for those links in the domain model, which clients do not require. If this derivation from the domain model is performed correctly, the most positive impacts on decision drivers are achieved, if *Embed Linked Data in the Payload* is used for the required linked data elements, and *Use Distributed or Hypermedia Links in the Payload* in all other cases. Embedding means to minimize the number of messages that need to be exchanged, leading to positive impacts on *Minimize API Calls*, *Performance*, *Scalability*, and *Protocol Complexity* decision drivers. However, *Message Sizes* are lower with distributed links, and as they are always up-to-date, also *API Evolvability* and *API Modifiability* are positively influenced. A still good, but less optimal variant is to use *Pass Object Identifiers in the Payload* in those cases where distributed links are applicable. In direct comparison, it has the disadvantages of possibly *Exposing Domain Model Details in API* as well as higher *Coupling of Clients to Server*. If embedded data is not used, where clients usually require most of the linked data elements immediately, consequently a negative impact on some of the forces can be expected. Likewise, if embedded data is used, but clients do not require most of the linked data elements immediately, some of the forces are influenced negatively, too.

These considerations have led to a literal derivation of the scoring scheme of the Decision LMD (see next section). For the other decisions we have derived the scoring schemes in the same way based on the empirical results from [157].

Scoring Scheme for Link Mapping Decision (LMD)

- IF FOR SOME domain links which are needed by the clients: *Do not offer the Link via the API* THEN assessment = (- -).
- IF (FOR ALL domain links *dl* which are needed by the clients: (IF for *dl* clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* is used)) THEN assessment = (++).
- IF ALL domain links *dl* which are needed by the clients: (IF For *dl* clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* OR *Pass Object Identifiers in the Payload* is used)) THEN assessment = (+).
- IF (FOR SOME domain links *dl* which are needed by the clients: If for *dl* clients usually require most of the linked data elements immediately: *Embed Linked Data in the Payload* is NOT used for *dl*) THEN assessment = (-)

- IF any other combination is used (e.g. for some domain links *dl* which are needed by the clients, and for *dl* clients usually do NOT require most of the linked data elements immediately AND *Embed Linked Data in the Payload* is used for *dl*) THEN assessment = (o).

Scoring Scheme for Operation Design Decision (ODD)

Please note that a model can contain more than one API. In some of the following decision points different choices are possible for the operation of each of the APIs in one model. We use the "*Has Aggregating Endpoint*" to denote that an API endpoint is exposed by a domain element of aggregating nature, such as an Aggregate (or its Aggregate root), a Bounded Context, or a Domain Service. For example, Entities and Value Objects are domain model elements that are usually not of aggregating nature.

- IF FOR EACH API: (FOR ALL API Operations: (*Expose Domain Events as State Transitions* OR *Expose Domain Events via Feeds or Pub/Sub*) AND *Has Aggregating Endpoint* is applied) OR ((FOR ALL API Operations: *Domain Operations on Resources* OR *Encode Operations as Commands in the Payload*) AND *Has Aggregating Endpoint* is applied) THEN assessment = (++).

- IF FOR EACH API: (FOR ALL API Operations: *Expose Domain Events as State Transitions* OR *Expose Domain Events via Feeds or Pub/Sub* AND *Has Aggregating Endpoint* is applied) OR ((FOR ALL API Operations: *Domain Operations on Resources* OR *Encode Operations as Commands in the Payload* OR *CRUD-Style Operations on Resources*) AND *Has Aggregating Endpoint* is applied) THEN assessment = (+).

- IF FOR ALL API Operations OF ALL APIs: (*Domain Operations on Resources* OR *Encode Operations as Commands in the Payload* OR *Expose Domain Events as State Transitions* OR *Expose Domain Events via Feeds or Pub/Sub* OR *CRUD-Style Operations on Resources*) AND *Has Aggregating Endpoint* is applied. THEN assessment = (o).

- IF FOR ALL API Operations OF ALL APIs: *CRUD-Style Operations on Resources* is applied and not *Has Aggregating Endpoint*. THEN assessment = (- -).

- IF any other combination is used, e.g. some API endpoints are not aggregating endpoints THEN assessment = (-).

Scoring Scheme for Resource Segregation Decision (RSD)

Please note that the decision points use *Is Aggregating Endpoint* in the same way as explained for the ODD scoring scheme, just not operating on operations, but on endpoints.

- IF FOR ALL API Endpoints *e* that are of *CQRS type*: At least 1 API Operation is exposed by *e* AND all API Operations exposed by *e* are *Event-based Pub/Sub Operations* or *Event Transition Operations* AND *e Is Aggregating Endpoint* THEN assessment = (++).

- IF FOR ALL Endpoints *e* that are of *CQRS type*: At least 1 API Operation is exposed by *e* AND all API Operations exposed by *e* are *Event-based Pub/Sub Operations* or *Event Transition Operations* AND *e Is or Is Not Aggregating Endpoint* THEN assessment = (+).

- IF FOR ALL Endpoints *e* that are of *CQRS type*: All API Operations exposed by *e* are or are not *Event-based Pub/Sub Operations* or *Event Transition Operations* AND *e Is Aggregating Endpoint* THEN assessment = (o).

- IF FOR ALL Endpoints *e* that are a *CQRS type*: At least 1 API Operation is exposed by *e* AND All API Operations exposed by *e* are not *Event-based Pub/Sub Operations* or *Event Transition Operations* AND *e Is Not Aggregating Endpoint* THEN assessment = (- -).

- IF any other combination is used for API Endpoints *e* that are of *CQRS type*, e.g. only some API operations are event-based THEN assessment = (-).

Ground Truth Assessment

To generate a ground truth for later evaluation, we have assessed each of the cases in our multi-case study manually based on our scoring scheme from Section 5.4. The results are presented in Table 5.5. Please note that this is not the result of the automated detectors presented below but the manually derived ground truth which we used to compare later to our detectors' results. Some assessments are not applicable ("n/a") as this aspect is not implemented or explained in the case's source, meaning that we cannot judge the ADD based on this case's model.

Table 5.4: Ground Truth: the Mapping of Domain Model Elements to APIs and API Endpoints

ID	MMD	API-ED	API-DD	**ID**	MMD	API-ED	API-DD
MD1	o	++	o	MD8	o	n/a	- -
MD2	+	++	++	MD9	o	+	-
MD3	+	- -	o	MD10	-	-	++
MD4	+	o	++	MD11	-	-	++
MD5	o	++	-	MD12	+	+	++
MD6	o	- -	o	MD13	o	n/a	- -
MD7	+	++	o	MD14	+	+	o

Table 5.4 contains the assessment of the 14 cases using our ordinal scoring scale (summaries of the ADD inspections can be found in Table 5.3). As can be seen, the case models cover for the decisions a wide range of possible combinations of the decision options.

Table 5.5: Ground Truth: API Endpoint Designs Derived from Domain Models

ID	LMD	ODD	RSD	**ID**	LMD	ODD	RSD
AX	n/a	n/a	++	ET	n/a	++	++
PM	++	+	n/a	KB	++	++	++
BA	+	-	++	NC	o	- -	n/a
OS	+	+	n/a	PC	-	- -	n/a
CM	o	+	o	RW	++	+	n/a
ES	o	-	++	LS	++	+	n/a

Table 5.5 contains the assessment of the 12 cases using our ordinal scoring scale (summaries of the ADD inspections can be found in Table 5.2). As can be seen, the case models cover for the decisions a wide range of possible combinations of the decision options.

5.5 Detectors for ADD Conformance Assessment

Detectors for the Mapping of Domain Model Elements to APIs and API Endpoint

In this section, we describe details about our detectors approach for automatically assessing conformance to decision options in the mapping of domain model elements to APIs and API endpoints. We propose a modular detector approach, in which one or more detectors are responsible for detecting each of the decision points in the scoring scheme from Section 5.4. Table 5.6 provides an overview of all detectors we have defined for the three decisions. The *assessment* column indicates how each detector helps in making a decision on the scoring scheme: s means that the detector must be *successful* for leading to the respective assessment, f means that the detector must fail to contribute to the assessment; u means that the detectors is *unused* in the respective assessment.

CodeableModels[4], a Python tool for the precise specification of meta-models, models, and model instances in code. Based on the meta-models and decision models, we manually created model instances for every case study system, and realized automated PlantUML code generators to generate graphical visualizations of all model instances.

To illustrate the detector approach further, let us explain one detector and how it contributes to the automatic assessment in detail: the Detector d2 from the MMD Decision. As indicated in the *assessment* column of Table 5.6, if this detector is *successful* and no other detectors have indicated other options, it leads to the resulting score "o". In addition, this detector's *failing* is considered in the decision on the score "++": if it fails and not all violations detected by Detector d2 are also detected as either Interface Bounded Contexts or Bounded Contexts linked to a Shared Kernel, then the score "++' cannot be achieved anymore, because "++" requires that *only* Interface Bounded Contexts or Bounded Contexts linked to a Shared Kernel are exposed. Similarly, this detector must fail for achieving "+", as in it only selected Bounded Contexts shall be exposed.

[4] `https://github.com/uzdun/CodeableModels`

Table 5.6: Overview for Detectors for ADD Conformance Assessment

ADD	**ID**	**Detectors**	**Description**	**Assessment**				
				++	+	o	-	- -
MMD	d1	`detect_is_each_domain_model_element_exposed_to_api` (*input: Domain_Model*)	To detect if each domain model element is exposed to the API.	u	u	u	s	u
	d2	`detect_is_each_bounded_context_exposed_to_api` (*input: Domain_Model*)	To detect if each Bounded Context is exposed to the API.	f	f	s	u	u
	d3	`detect_are_selected_bounded_ontext_exposed_to_api` (*input: DomainModel*)	To detect selected Bounded Contexts are exposed to the API.	f	s	u	u	u
	d4	`detect_all_apis_are_exposed_by_subset_of_domain_model_elements` (*input: Domain_Model*)	To detect all APIs are exposed by a subset of domain model elements (Services or Processes).	u	s	u	u	u
	d5	`detect_is_each_api_element_exposed_by_domain_model_element` (*input: API_Elements*)	To detect all APIs have a traceable mapping to domain model elements.	u	u	u	u	f
	d6	`detect_interface_bounded_context` (*input: Domain_Model*)	To detect if a dedicated Interface Bounded Context is used.	s	s	u	u	u
	d7	`detect_shared_kernel` (*input: Domain_Model*)	To detect if a Shared Kernel is used as interface between Bounded Contexts.	s	s	u	u	u
API-ED	d8	`detect_is_each_api_endpoint_exposed_to_domain_model_element` (*input: API_Elements*)	To detect if all API endpoints have a traceable mapping to domain model elements.	u	u	u	u	f
	d9	`detect_all_api_endpoints_exposed_by_an_entity` (*input: API_Elements*)	To detect all API endpoints are exposed by Entities which are not Aggregate roots.	u	u	u	s	u
	d10	`detect_api_endpoints_exposed_by_a_bounded_context` (*input: API_Elements*)	To detect API endpoints are exposed by Bounded Contexts.	u	s	u	f	u
	d11	`detect_all_endpoints_exposed_by_aggregates_services_process` (*input: API_Elements*)	To detect all API endpoints are exposed by Aggregates roots, Services, or Processes.	s	u	u	f	u
	d12	`detect_some_endpoints_exposed_by_entities` (*input: API_Elements*)	To detect some API endpoints are exposed by Entities or domain elements other than (Services, Aggregates, Processes, Bounded Contexts).	u	u	s	u	u
API-DD	d13	`detect_each_api_has_structured_description_and_contract` (*input: API_Elements*))	To detect if each API has a structured API description and an API contract.	s	u	u	u	u
	d14	`detect_each_api_has_informal_description_and_contract` (*input: API_Elements*)	To detect if each API has an informal API description and an API contract.	u	s	u	u	u
	d15	`detect_each_api_has_structured_description_or_contract` (*input: API_Elements*)	To detect each API has either a structured API description or an API contract.	u	u	s	u	u
	d16	`detect_api_has_description_or_contract` (*input: API_Elements*)	To detect an API has either an (informal or structured) API description or an API contract.	u	u	u	u	f

Algorithm 1: Detector d2:
detect_is_each_bounded_context_exposed_to_api

```
input: Domain_Model DM
output: Tuple<Boolean, Set, String>
begin
BCS ← bounded_contexts(DM)
if BCS = ∅:
return (False, ∅, 'no bounded context defined')
API_BCS ← api_exposed_bounded_contexts(BCS)
violations ← BCS \ API_BCS
if violations ≠ ∅:
return (False, violations,
'bounded contexts not being exposed to API')
return (True, ∅, 'all bounded contexts exposed to API')
end
```

The pseudo code for Detector d2 is shown in Algorithm 1. The detector receives a domain model as input. It delivers a tuple indicating (1) detector success or failure as a Boolean; (2) a set of all detected violations; (3) an explanation of the result. Please note in this context that our detectors not only provide binary assessments of the scores, they also provide a list of violations that led e.g. to failing an assessment. In the example in the previous paragraph we have illustrated how this is used in the assessment of MMD to combine the results of multiple detectors. It is also helpful for the human architect to inspect and fix violations, if they occur.

In its implementation, the Detector d2 uses a function call *bounded_contexts()* to traverse the model to find the set of all Bounded Contexts BC defined in it. If there is none, this detector fails without detected violations. Next, we use another model traversal function *api_exposed_bounded_contexts()* to find the set of those Bounded Contexts that are exposed to the API, called API_BCS. Then *violations* are calculated by removing the API_BCS set from the BCS set. If it is not an empty set, a detector failure with violations indicating the Bounded Contexts not being exposed to an API is returned. Otherwise, the detector is successful, i.e. all Bounded Contexts are exposed to an API.

Detectors for the API Endpoint Designs Derived from Domain Models

In this section, we describe details of our detector approach for automatically assessing conformance to the ADD options from Section 5.2. We propose a modular detector approach where one or more detectors are responsible for detecting each of the decision points in the scoring scheme from Section 5.4. Tables 5.7-5.9 gives an overview of all the detectors we defined for the three ADDs.

It also explains in the *Use in Assessment* column how each scoring scheme result of each of the three ADDs is derived during the assessment by the detectors. Please note that in many cases combinations of detectors are needed, and that they are applied in the given contexts of API elements that are passed in as lists (e.g. in ODD all API elements of each API are usually used, whereas RSD applies the detectors to the elements of CQRS type

endpoints).

Table 5.7: Detectors Overview: LMD

ID	Detectors	Description	Use in Assessment
d1	detect_each_API_endpoint_has _links_to_API_operations (*input: List⟨APIElements⟩*)	To detect whether each API endpoint has links to API operations. Helps to find those exposed to clients.	The detector is required (with results SUCCESS, PARTIAL SUCCESS) in the assessment.
d2	detect_api_operations_with _embedded_data_elements_usually _needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide embedded data as data elements usually needed by clients.	For '++' assessment, this detector or d5 only one of them can be successful (i.e. SUCCESS or PARTIAL_SUCCESS). If the d5 is successful, this detector must fail in '++' assessment. For '-' assessment, this detector must be FAILED when d3 or d4 is partialy successful or successful.
d3	detect_api_operations_with _distributed_links_to_data_elements _usually_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide distributed links to data elements usually needed by clients.	For '++' assessment, this detector must be failed but it can be partial successful in '+' assessment.
d4	detect_api_operations_with_object _id_based_links_to_data_elements _usually_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide object-ID based links to data elements usually needed by clients.	For '++' assessment, this detector must be failed but it can be partial successful in '+' assessment.
d5	detect_api_operations_with _distributed_links_to_data_elements _usually_not_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide distributed links to data elements usually not needed by clients.	For '++' assessment, this detector or d2 only one of them can be successful (i.e. SUCCESS or PARTIAL_SUCCESS). If the d2 is successful, this detector must fail in '++' assessment
d6	detect_api_operations_with_object _id_based_links_to_data_elements _usually_not_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide object-ID based links to data elements usually not needed by clients.	For '++' assessment, this detector must be failed but it can be successful or partial successful in '+' assessment.
d7	detect_api_operations_with _embedded_data_elements_usually _not_needed_by_clients (*input: List⟨APIElements⟩*)	To detect API operations that provide embedded data as data elements usually not needed by clients.	For '++' assessment, this detector must be failed . For '+' assessment, this detector should not be successful nor partial sucessful, unless the assessment result will be 'o' instead.
Note: The detectors-based assessment for each decision will result in "n/a" if at least one of decision's detectors has the result NOT_APPLICABLE.			

All detectors are implemented in Python and work mainly by traversing models. All models are implemented with CodeableModels[5], a Python tool for precise specification of meta-models, models, and model instances in code. Based on the meta-models and decision models, we manually created model instances for each case system and implemented automated PlantUML code generators to produce graphical visualizations of all model instances.

To illustrate the approach further let us explain one exemplary detector in detail as pseudo code. Algorithm 5.4 shows the detector to detect each API operation in a list of API elements *AE* that is of event state transition type. In this algorithm we first get the API operations from the *AE* list. If no operations are defined, the detector is *NOT_APPLICABLE*. If this is not the case, we next detect the possible violations by traversing all API operations and checking whether they conform to the respective stereotype *≪Event State Transition Operation≫*. If there are only members ofthis stereotype, we report a *SUCCESS*; if there

[5] `https://github.com/uzdun/CodeableModels`

Table 5.8: Detectors Overview: ODD

ID	Detectors	Description	Use in Assessment
d8	detect_each_crud_based_api _operation (*input: List⟨APIElements⟩*)	To detect each CRUD-based API operation on API endpoints.	It can be successful or partially successful for APIs for the '++' ,'+', 'o', or '-' assessments with at least one other detectors of d9-d12 used (partially) successfully with it. If this is the only detector that is successful, the assessment will turn to '--'.
d9	detect_each_domain_operation_based _api_operation (*input: List⟨APIElements⟩*)	To detect each domain operation-based API operation on API endpoints.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d10	detect_each_api_operation_encoded _as_commands_in_the_payload (*input: List⟨APIElements⟩*)	To detect each API operation on API endpoints that has the operation encoded as commands in the payload.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d11	detect_each_api_operation_with _event_based_state_transition (*input: List⟨APIElements⟩*)	To detect each API operation on API endpoints that represents an event-based state transition.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d12	detect_each_api_operation_with _event_based_pub_sub (*input: List⟨APIElements⟩*)	To detect each API operation on API endpoints that offers events via feeds or pub/subs.	It can be successful or partially successful for APIs for the '++', '+', 'o', or '-' assessments.
d13	detect_aggregating_endpoint_and _crud_based_operation (*input: List⟨APIElements⟩*)	To ensure that API endpoints with CRUD-based API operations are aggregating endpoints.	This detector is used alongside d8, verifying the investigated endpoints are aggregating ones with CRUD-based API operations. When d8 is used together with other detectors (d9-d12), this detector should be a SUCCESS to reach '++', and it should not be FAILED to reach '+'. If it has FAILED and only d8 succeeded, the assessment will be '--'. If it and d8 are a PARTIAL_SUCCESS, the assessment will be '-'.
d14	detect_aggregating_endpoint_and _domain_based_operation (*input: List⟨APIElements⟩*)	To ensure the investigated API endpoints with domain-based API operations are aggregating endpoints.	This detector is used alongside d9 and d10, verifying that the investigated endpoints are aggregating ones with domain-based API operations. When d9 or d10 are used together with it, it should be a SUCCESS to reach '++'. It should not be FAILED to reach '+'. If it is PARTIAL_SUCCESS, the assessment will be '-'.
d15	detect_aggregating_endpoint_and _event_based_operation (*input: List⟨APIElements⟩*)	To ensure the investigated API endpoints with event-based API operations are aggregating endpoints.	This detector is used alongside d11 and d12, verifying that the investigated endpoints are aggregating ones with event-based API operations. When d11 or d12 are used together with it, it should be a SUCCESS to reach '++'. Itt should not be FAILED to reach '+' . If it is PARTIAL_SUCCESS, the assessment will be '-'.
Note: The detectors-based assessment for each decision will result in "n/a" if at least one of decision's detectors has the result NOT_APPLICABLE.			

are violations and members of this stereotype, we report a *PARTIAL_SUCCESS*; else a *FAILURE* is reported. Please note that we also return the list of *violations*, which helps humans to later inspect the violation, and maybe correct either the model or the detection. The other detectors in Tables 5.7-5.9 are constructed in a similar fashion, inspecting and

Table 5.9: Detectors Overview: RSD

ID	Detectors	Description	Use in Assessment
d16	detect_api_endpoints_exposed _by_aggregating_domain_model _element (*input: List⟨APIElements⟩*)	To detect API endpoints exposed by aggregating domain elements (such as Aggregate, Aggregate root, Service, Bounded Context).	For '++' and 'o', it must succeed (SUCCESS, PARTIAL_SUCCESS) for a given endpoint, and for '--' it must fail for the endpoints under investigation.
d17	detect_each_cqrs_endpoint_with _event_based_operations (*input: List⟨APIElements⟩*)	To detect API endpoints offer CQRS Queries or Commands, as well as event-based API operations.	It must be SUCCESS, PARTIAL_SUCCESS, or FAILED for the decision to be applicable.
Note: The detectors-based assessment for each decision will result in "n/a" if at least one of decision's detectors has the result NOT_APPLICABLE.			

detecting other aspects in the model, as explained in the *Description* column of Tables 5.7-5.9.

Algorithm 2: Detector d12 — detect_each_api_operation_with_event_based_state_transition

```
input: List ⟨Api_Element⟩ AE
output: Tuple⟨Detector_Results_Enum, Set, String⟩
begin
API_OPS ← api_operations(AE)
if (API_OPS = ∅):
return (NOT_APPLICABLE, ∅,
'No API operations defined')
violations ← ∅
members ← ∅
for OP in API_OPS:
if (OP.is_of_stereotype(
≪Event State Transition Operation≫):
members ← members ∪ OP
else:
violations ← violations ∪ OP
if (violations ≠ ∅):
if (members ≠ ∅):
return (PARTIAL_SUCCESS, violations,
'Some operations are Domain Events ' +
'as State Transitions')
else:
return (FAILURE, violations,
'No operation is Domain Events ' +
'as State Transitions')
return (SUCCESS, ∅, 'All operations are Domain ' +
'Events as State Transitions')
end
```

5.6 Multi-Case Study Results of the Mapping of Domain Model Elements to APIs and API Endpoint

In this section, we discuss the multi-case study results of the mapping of domain model elements to APIs and API endpoint. First, we analyze the results on a case-by-case basis for each decision option. In doing so, we simply count the correctly identified results and discuss interesting findings. Next, we statistically analyze our dataset and show that there

Table 5.10: Assessment Result of Each Model from Detector

ID	MMD	Reason	API-ED	Reason	API-DD	Reason
MD1	o	*Successful Detectors*: d2, d3, d5 *Failed Detectors*: d1, d4, d6, d7	++	*Successful Detectors*: d8, d11, d12 *Failed Detectors*: d9, d10, d13	o	*Successful Detectors*: d16, d17 *Failed Detectors*: d14, d15
MD2	+	*Successful Detectors*: d3, d5 *Failed Detectors*: d1, d2, d4, d6, d7	++	*Successful Detectors*: d8, d11, d12 *Failed Detectors*: d9, d10, d13	++	*Successful Detectors*: d14, d16, d17 *Failed Detectors*: d15
MD3	- -	*Successful Detectors*: d4 *Failed Detectors*: d1, d2, d3, d5, d6, d7	- -	*Successful Detectors*: d12 *Failed Detectors*: d8, d9, d10, d11, d13	o	*Successful Detectors*: d16, d17 *Failed Detectors*: d14, d15
MD4	+	*Successful Detectors*: d3, d5 *Failed Detectors*: d1, d2, d4, d6, d7	o	*Successful Detectors*: d8, d12, d13 *Failed Detectors*: d9, d10, d11	++	*Successful Detectors*: d14, d16, d17 *Failed Detectors*: d15
MD5	o	*Successful Detectors*: d2, d3, d5 *Failed Detectors*: d1, d4, d6, d7	++	*Successful Detectors*: d8, d11, d12 *Failed Detectors*: d9, d10, d13	-	*Successful Detectors*: d17 *Failed Detectors*: d14, d15, d16
MD6	- -	*Successful Detectors*: d2, d3, d4 *Failed Detectors*: d1, d5, d6, d7	- -	*Successful Detectors*: d10 *Failed Detectors*: d8, d9, d11, d12, d13	o	*Successful Detectors*: d16, d17 *Failed Detectors*: d14, d15
MD7	+	*Successful Detectors*: d2, d3, d4, d5 *Failed Detectors*: d1, d6, d7	++	*Successful Detectors*: d8, d11, d12 *Failed Detectors*: d9, d10, d13	o	*Successful Detectors*: d16, d17 *Failed Detectors*: d14, d15
MD8	o	*Successful Detectors*: d2, d3, d5 *Failed Detectors*: d1, d4, d6, d7	n/a	*Successful Detectors*: d11 *Failed Detectors*: d8, d9, d10, d12, d13	- -	*Successful Detectors*: none *Failed Detectors*: d14, d15, d16, d17
MD9	o	*Successful Detectors*: d2, d3, d5 *Failed Detectors*: d1, d4, d6, d7	+	*Successful Detectors*: d8, d10, d12 *Failed Detectors*: d9, d11, d13	-	*Successful Detectors*: d17 *Failed Detectors*: d14, d15, d16
MD10	- -	*Successful Detectors*: d4 *Failed Detectors*: d1, d2, d3, d5, d6, d7	-	*Successful Detectors*: d8, d9, d13 *Failed Detectors*: d10, d11, d12	++	*Successful Detectors*: d14, d16, d17 *Failed Detectors*: d15
MD11	- -	*Successful Detectors*: d4 *Failed Detectors*: d1, d2, d3, d5, d6, d7	-	*Successful Detectors*: d8, d9, d13 *Failed Detectors*: d10, d11, d12	++	*Successful Detectors*: d14, d16, d17 *Failed Detectors*: d15
MD12	- -	*Successful Detectors*: d2, d3, d4 *Failed Detectors*: d1, d5, d6, d7	+	*Successful Detectors*: d8, d10 *Failed Detectors*: d9, d11, d12, d13	++	*Successful Detectors*: d14, d16, d17 *Failed Detectors*: d15
MD13	o	*Successful Detectors*: d2, d3, d5 *Failed Detectors*: d1, d4, d6, d7	n/a	*Successful Detectors*: d11 *Failed Detectors*: d8, d9, d10, d12, d13	- -	*Successful Detectors*: none *Failed Detectors*: d14, d15, d16, d17
MD14	- -	*Successful Detectors*: d2, d3, d4 *Failed Detectors*: d1, d5, d6, d7	+	*Successful Detectors*: d8, d10 *Failed Detectors*: d9, d11, d12, d13	o	*Successful Detectors*: d16, d17 *Failed Detectors*: d14, d15

is no significant difference between the ground truth data and the detector results and the effect size between the two variables is negligible.

Detailed Discussion of the Results

Table 5.10 shows the results of the automated assessments based on our detectors approach for the 14 models in our multi-case study. The column *Reason* shows which specific detectors from Table 5.6 have failed or been a success, and thus led to the automated assessment. Comparing the overall results in Table 5.10 to our ground truth in Table 5.4, we can summarize the matching score ratios of MMD as 8/14 (57%), API-ED as 14/14 (100%), and API-DD as 14/14 (100%), respectively. In total, 86% of the decision points in our multi-case study have been correctly identified by the automated detectors. The places where ground truth and detector results diverge illustrate well those cases where human judgment is needed. This indicates, especially in the ADD MMD where we observe divergence, the detectors should not be applied "blindly" as automated tests but rather as indicators of possible conformance violations. When we apply the detectors, we also get the violation set as part of the detector result. These violations can help to spot the aspects humans need to inspect more closely.

Firstly, it can be observed that all decision points of API-ED and API-DD are matching. For API-ED to be 100% correctly assessed, we needed to extend our scoring scheme slightly, to also cover the case that no API endpoints are modeled (yielding "n/a" instead of the default case). This is because this decision is about the concrete mapping of DDD model elements to API endpoints, which is rather easy to specify precisely in automated detectors. In contrast to the other ADDs, the API-DD decision is pretty straightforward to model and realize with detectors, and thus no ambiguous cases have been observed.

Let us discuss a few notable cases where MMD diverges in detail. One case where the ground truth and the detector assessment can diverge is the option *Expose Domain Model Subset as API* of the MMD ADD. For instance, consider MD3. Here, a domain model subset is exposed as API elements, but this is only modeled at the level of API endpoints, not for the API. If the model would be corrected, to expose also the API, the detector would yield the correct result. So, here the detector has spotted an omission in the model, which could be fixed. A correctly identified case is the MMD ADD for MD7 where again a domain model subset is exposed as an API (here services). This could easily be wrongly modeled as in MD3, e.g. by exposing the services to API endpoints. But here this is not the case because aggregates are used as well, and they are exposed to API endpoint. In summary, for the option *Expose Domain Model Subset as API* of MMD there might be modeling issues in those cases where MMD and API-ED overlap. Here, modelers might feel the relations to APIs and API endpoints are redundant and thus omit them. The detectors deliver precise results on where the problem occurred and can thus help to fix the problem.

Another critical case is the detection of *no traceable mapping to domain model elements can be discerned* for MMD. This option can in a number of cases take precedence

in detectors over other options. Interestingly, again this concerns cases where MMD and API-ED overlap: For example, in MD11, the assessment result should be "-" because the domain model is exposed 1:1 to the API. As some Bounded Contexts in the model are pure aggregation elements not exposed to the API (but all their members are exposed), formally the model is not fully exposed, leading to the difference in human and automatic judgment. A similar issue occurs in MD12 where selected Bounded Contexts are exposed. But as those are mapped to API endpoints for the API-ED decision, they are not mapped to the API again, leading to the difference in the assessment results. Finally, in MD14 the same issue as in MD12 has happened. Also selected bounded contexts are exposed, but this is modeled only at for the API-ED decision, not for the API. All of these cases can be fixed by exposing the respective elements to the API as well, and the detectors pinpoint the problem location.

Both problems have their root cause in the fact that there is a slight overlap in the MMD and API-ED decisions, and redundant modeling of API and API endpoint mapping is needed to avoid issues. Instead of requiring the redundant modeling, we could modify the named detectors to fall back to the exposed API endpoints, if the API is not exposed.

Another issue occurred in MD6 and MD10 for decision MMD: Here "--" is reported because in each case one of many APIs (a "helper" API) is not linked to any domain model elements. For a human it is obvious how the overall mapping logic is constructed, the machine identifies no traceable mapping for those. Here, completing the model would be need for fixing the model, and the detectors clearly indicate where to fix this.

In summary, our case studies have shown that all ADD options could automatically be detected based on our scoring scheme (**RQ4.1**). Regarding **RQ4.2**, 86% of the decision points have been correctly identified, and the remaining cases are those where a modeling omission has occurred due to the conceptual overlaps of the MMD and API-ED decisions. Would we use the detector modification explained in this section, we could even reach 12/14 (86%) for MMD, and thus 95% in total. Please note that this modification would be only based on the experiences of this study, not on the empirical data from [157]. In cases, where case study models were missing certain redundant expose-relations, our detectors identified the locations of these modeling issues and indicated how to fix those issue.

Statistical Analysis

To confirm the results for **RQ4.2** and get a more precise estimate for the effect size, we statistically analyzed the results in R. In our data set, we had to deal with ordinal variables, a rather small sample size, and data that is not normally distributed. We first confirmed the non-normal distribution with a Shapiro-Wilk test [151] using R's *shapiro.test()* function. The data in Table 5.11 shows that, as the p-values for both ground truth (0.0002982) and detector results (0.0004204) data are significant, we must reject the null hypothesis that the data is normally distributed for both variables. Thus, the t-test, which assumes the normal distribution, is not applicable. For our problem the Wilcoxon signed-rank test would be applicable, but as many data points are identical, we get many zero values, which are in

Wilcoxon's method removed from the test, making the results non-exact for our data set. The Wilcoxon signed rank test with Pratt method can handle those zero values [125], which means that it is more appropriate for our data set. For Wilcoxon-Pratt Signed-Rank Test calculation, we used the *wilcoxsign_test()* function from R's *coin* package. The test's result (0.3173) is not significant (see Table 5.11), meaning that the null hypothesis cannot be rejected. Thus, we must assume the true μ is close to 0.

To confirm this result and assess the relevance of the result further, we computed the effect size. Here, Kitchenham et al. [77] suggest Cliff's δ [21] as a robust method for empirical software engineering. For this calculation, we used the *cliff.delta()* function from R's *effsize* package. As shown in Table 5.11, the delta estimate is 0.009375, which is to be interpreted as: the effect size is negligible [140].

Table 5.11: Statistical Analysis Results

Shapiro-Wilk Test for Ground Truth Data	
p-value	0.0002982
Shapiro-Wilk Test for Detector Results Data	
p-value	0.0004204
Wilcoxon-Pratt Signed-Rank Test	
p-value	0.3173
Cliff's Delta	
delta estimate	0.009375 (negligible)

5.7 Multi-Case Study Results of API Endpoint Designs Derived from Domain Model

In this section, we discuss the multi-case study results of API endpoint designs derived from domain model. We first analyze the results case by case for each decision option. Here, we simply count the correctly identified results and discuss interesting insights. Next, we analyze our data set statistically and show (1) that there is no significant difference between ground truth and detector result data and (2) that the effect size between the two variables is negligible.

Detailed Discussion of the Results

Table 5.12 summarizes the results of the automated assessments based on our detectors approach for the 12 models in our multi-case study. Comparing the overall results in Table 5.12 to our ground truth in Table 5.5, we can calculate matching score ratios for each of the decisions: LMD – 8/12 (67%), ODD – 12/12 (100%), RSD – 10/12 (83%). In total, 83% of the decision points in our multi-case study have been correctly identified by the automated detectors. The places where ground truth and detector results are different are all cases where additional human judgment is needed. When we apply the detectors, we do not only get the results, but in case of violations, we also get the violation set as part of the detector result. These violations can help to spot the aspects humans need to inspect more closely.

Table 5.12: Automated Assessment Results for Each Model from the Detectors

ID	**LMD**	**Reasons: Detector Results**	**ODD**	**Reasons: Detector Results**	**RSD**	**Reasons: Detector Results**
AX	n/a	*Not Applicable*: d1, d2, d3, d4, d5, d6, d7	n/a	*Failed Detectors*: d13, d14, d15 *Not Applicable*: d8, d9, d10, d11, d12	o	*Successful Detectors*: d16 *Failed Detectors*: d17
PM	++	*Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Successful Detectors*: d13, d14 *Partially Successful Detectors*: d8, d9 *Failed Detectors*: d10, d11, d12, d15	n/a	*Successful Detectors*: d16 *Not Applicable*: d17
BA	+	*Successful Detectors*: d1 *Partially Successful Detectors*: d2 *Failed Detectors*: d3, d4, d5, d6, d7	-	*Successful Detectors*: d11 *Partially Successful Detectors*: d8, d9, d10, d13, d14, d15 *Failed Detectors*: d12	++	*Successful Detectors*: d17 *Partially Successful Detectors*: d16
OS	+	*Successful Detectors*: d1 *Partially Successful Detectors*: d2, d6 *Failed Detectors*: d3, d4, d5, d7	+	*Successful Detectors*: d8 *Partially Successful Detectors*: d13 *Failed Detectors*: d9, d10, d11, d12, d14, d15	n/a	*Partially Successful Detectors*: d16 *Not Applicable*: d17
CM	++	*Partially Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Partially Successful Detectors*: d8, d9, d13, d14 *Failed Detectors*: d10, d11, d12, d15	o	*Successful Detectors*: d16 *Failed Detectors*: d17
ES	++	*Partially Successful Detectors*: d1, d2, d5 *Failed Detectors*: d3, d4, d6, d7	-	*Partially Successful Detectors*: d8, d9, d11, d12, d13, d14, d15 *Failed Detectors*: d10	++	*Successful Detectors*: d17 *Partially Successful Detectors*: d16
ET	n/a	*Successful Detectors*: d1 *Not Applicable*: d2, d3, d4, d5, d6, d7	++	*Successful Detectors*: d8, d11, d13, d15 *Failed Detectors*: d9, d10, d12, d14	++	*Successful Detectors*: d16, d17
KB	++	*Successful Detectors*: d2 *Partially Successful Detectors*: d1 *Failed Detectors*: d3, d4, d5, d6, d7	++	*Successful Detectors*: d8, d11, d13, d15 *Failed Detectors*: d9, d10, d12, d14	o	*Partially Successful Detectors*: d16 *Failed Detectors*: d17
NC	++	*Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	- -	*Successful Detectors*: d8 *Failed Detectors*: d9, d10, d11, d12, d13, d14, d15	n/a	*Failed Detectors*: d16 *Not Applicable*: d17
PC	++	*Successful Detectors*: d1, d5 *Failed Detectors*: d2, d3, d4, d6, d7	- -	*Successful Detectors*: d8 *Failed Detectors*: d9, d10, d11, d12, d13, d14, d15	n/a	*Failed Detectors*: d16 *Not Applicable*: d17
RW	++	*Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Successful Detectors*: d8, d13 *Failed Detectors*: d9, d10, d11, d12, d14, d15	n/a	*Successful Detectors*: d16 *Not Applicable*: d17
LS	++	*Partially Successful Detectors*: d1, d2 *Failed Detectors*: d3, d4, d5, d6, d7	+	*Partially Successful Detectors*: d8, d9, d13, d14 *Failed Detectors*: d10, d11, d12, d15	n/a	*Partially Successful Detectors*: d16 *Not Applicable*: d17

Let us discuss a few notable cases where detector results diverge or could have diverged in detail. The *RSD* detector delivers "o" for the *AX* and *KB* models. This deviates from our manual assessment, where we judged those models as "++". The reason for this is that the model is detailed enough to see that all CQRS endpoints are aggregating endpoints, but is missing detailed modeling of operations. Thus it cannot automatically be judged

whether event-based operations are used. However, as in both cases of *AX* and *KB* models, model and system documentation contain information on the event-based endpoints (the naming of endpoints), and there is a description that event-based communication is used in the documentation, too, we can make a manual conclusion here. As the detailed operation modeling is missing, the detectors cannot decide for any of the other options and thus uses the fallback "o". This could be mitigated by more detailed modeling of operations or introducing a flag that signals the event-driven nature of the system to the detectors.

The *LMD* detector for the *CM* model delivers "++" which deviates from the "o" in our manual assessment. We inspected the operations carefully and found one that uses embedded data, but seems to convey possibly a lot of information that might not always be needed by clients. If this interpretation is correct, "o" is the correct assessment, but this can only be found out by source code analysis, not on the model. This issue could be mitigated by changing the *data_elements_usually_needed_by_clients* flag manually to *False*. The same case happens for the *ES* model, also with just one operation. It also occurs for the *NC* model, where this violation is even worse, as a couple of operations are candidate for using embedded data, but it seems the operations convey a lot of information that is not always needed by clients.

The *LMD* detector for the *PC* model reports "++" which deviates from the "-" in our manual assessment. We inspected the operations carefully and that many of the operations providing URL-based links actually contain data immediately needed by clients. If this interpretation is correct, "-" is the correct assessment, but this can only be found out by source code analysis, not on the model. This issue could be mitigated by changing the *data_elements_usually_needed_by_clients* flag manually to *True*.

The *LS* model is another interesting case. The system uses frontend endpoints that deliver URLs to access the backend services, which then deliver the actual data in embedded form. If we would model just based on the frontends, *LMD* would be falsely detected as in the *PC* model, explained in the previous paragraph. This technical feature could also have led to the misinterpretation in ODD (and RSD) that the frontends do not belong to the aggregating bounded contexts; which both would have led to yet another deviation in manual and automatic assessment. As we, however, modeled the *LS* model considering the relation to backend services, manual and automatic assessment match.

The *ODD* cases of the *BA* and *ES* models are assessed as "-", consistently in ground truth and automatic assessment. But with a few changes those could turn into "++", as just a few endpoints are not covered in domain model. If corresponding aggregating domain model elements were modeled, the asssessments could easily be improved to the best score. Here, the failed detectors clearly pinpoint the few endpoints that require attention.

The "--" assessment of LMD did not appear in our cases. We did not implement a detector for it yet, as this detector would be pretty trivial. Here, human judgment if a domain link is needed by clients but not in the API is needed. This cannot be found out automatically. Thus a simple flag to indicate this fact on domain links in our models, which can be set by a human designer, together with a simple detector whether there is an API

link corresponding to the domain link would be enough to support this assessment, too.

The *ODD* cases of the *BA* and *ES* models are assessed as "-", consistently in ground truth and automatic assessment. But with a few changes those could turn into "++", as just a few endpoints are not covered in domain model. If corresponding aggregating domain model elements were modeled, the asssessments could easily be improved to the best score. Here, the failed detectors clearly pinpoint the few endpoints that require attention.

In summary, our multi-case study have shown that all ADD options could possibly automatically be detected based on our scoring scheme (**RQ4.3**). Regarding **RQ4.4**, 83% of the decision points have been correctly identified, and the remaining cases are those where more detailed modeling of aspects beyond architecture-level models, e.g. with a simple additional flag, could have solved the issue. That is, a single manual inspection and correction could lead to correct detection in future runs, e.g. in a continuous delivery pipeline context.

Statistical Analysis of the Results

To confirm the results for **RQ4.4** and get a more precise estimate for the effect size, we statistically analyzed the results in R. In our data set, we had to deal with ordinal variables, a rather small sample size, and data that is not normally distributed. We first confirmed the non-normal distribution with a Shapiro-Wilk test [151] using R's *shapiro.test()* function. The data in Table 5.13 shows that, as the p-values for both ground truth and detector results data are significant, we must reject the null hypothesis that the data is normally distributed for both variables. Thus, the t-test, which assumes the normal distribution, is not applicable. In general, for our problem the Wilcoxon signed-rank test would be applicable, but as many data points are identical, we get many zero values, which are in Wilcoxon's method removed from the test, making the results non-exact for our data set. The Wilcoxon signed rank test with Pratt method can handle those zero values [125], which means that it is more appropriate for our data set. For Wilcoxon-Pratt Signed-Rank Test calculation, we used the *wilcoxsign_test()* function from R's *coin* package. The results of the test in Table 5.13 are not significant, meaning that the null hypothesis cannot be rejected. Thus, we must assume the true μ is close to 0.

Table 5.13: Statistical Analysis Results

Shapiro-Wilk Test for Ground Truth Data	
p-value	0.000448
Shapiro-Wilk Test for Detector Results Data	
p-value	2.728e-05
Wilcoxon-Pratt Signed-Rank Test	
p-value	0.369
Cliff's Delta	
delta estimate	-0.1042524 (negligible)

To confirm this result and assess the relevance of the result further, we computed the effect size. Here, Kitchenham et al. [77] suggest Cliff's δ [21] as a robust method for empirical software engineering. For this, we used the *cliff.delta()* function from R's *effsize*

package. As shown in Table 5.13, the delta estimate is -0.1042524, which is interpreted as: the effect size is negligible [140].

5.8 Lessons Learned

In this section, we summarize the major lessons learned of our study, especially from the multi-case study that illustrate the relevance of our results.

Lesson 1. *A rough estimation of manual architectural conformance checking strongly indicates that automation is required, if a rapid release schedule is used.*

As discussed below in Section 5.10, each of the two authors has performed a complete manual conformance assessment for our case study model set at least three times during the writing of this chapter, and before that numerous times for all single models during the development of our approach. Thus, we can provide a rough estimate of a manual conformance checking effort. An initial manual checking takes in our experience substantially more time than subsequent re-assessments. While some links require initially a few minutes to check, with experience and practice one can reach about 10-30 seconds per link between domain model and API element on average. That is, our smaller models initially took about 10-20 minutes to check, going down to a few minutes to check in later iterations. The larger models required about 40-60 minutes initially, and went down to 8-15 minutes in later iterations. Assuming that a very small-size industrial system is about the size of our largest models, a small medium-sized industrial system is about the size of our complete model set, and large-scale systems (such as those in [148, 52, 105]) are way beyond our models' scale, it is immediately clear that even for small to medium real-life systems it is almost impossible to check them manually in each run of a CI/CD pipeline (e.g. for daily or hourly release). For large-scale systems, even a single manual check might be infeasible. Thus, a significant lesson learned is that manual conformance checking might be possible for release schedules measured in years, but for contemporary rapid release schedules automation is often the only feasible option.

Please note that this kind of automation for frequent releases might require automatic reconstruction of the model. In our approach, all aspects except the case study modeling were automated. We discuss in Section 5.3 how this step can be automated using our existing static code analysis approach [106]. In fact, the evaluation [106] of our prior work demonstrates the automatic API reconstruction of our case study systems ES and LS used in this chapter, too (see Table 5.2).

Lesson 2. *Accurate automatic conformance checking is possible, especially if domain-specific system knowledge is utilized.*

In our approach we have shown that for a given modeling of 12 systems according to existing industry best practices (empirically studied in [157]) and using a given interpretation of these industry best practices (i.e., our scoring scheme), following our approach, it is straightforward to provide detectors that accurately detect conformance to the ADDs

describing the industry best practices. Further our detailed assessment in Section 5.7 shows that with domain-specific knowledge about the systems (which an e.g. industry team working on a system has), the detection can be improved up to a 100% accuracy.

Lesson 3. *The approach can be adapted or calibrated easily, if the detector results provide detailed traceability.*

The results in our evaluation use a concrete modeling or interpretation of the industry best practices (i.e., our scoring scheme), but do not depend on them. During our many iterations and refactorings, we have many times improved the models, scoring scheme, and detectors, and still about the same level accuracy of the detectors was possible to achieve. A consequence of this observation is that it is easy and straightforward to adapt models, change the detectors, or calibrate the scoring scheme. All steps in our approach are fully automated and can be re-run after such a change. The major reason why this works smoothly in our experience is the provided traceability in detector results—i.e. status (Success, Failed, or Partial Success), the affected modeling elements (violations), and a reason for the outcome. That is, if our detectors fail to produce the expected results, the traceability information provided by the detectors helps to pinpoint the model elements or detector implementations which failed.

Lesson 4. *Once the detector infrastructure is realized, writing or changing detectors is straightforward and comparatively low effort.*

The effort for providing the automation of a single detector, once the general detector framework, modeling framework, model visualization, model traversal methods, traceability support, reporting framework, and other generic components are in place, is relatively low. Most single detectors have been written in 20-40 minutes time, with a couple of refactorings, requiring on average again as much time. That is, it is feasible to adapt detectors to new circumstances, if needed. Furthermore, as we provide our results as open access artifacts, even our implementation can be reused, thus reducing the initial required effort substantially.

Lesson 5. *For full automation of our approach, the API model must either be automatically extracted or API code must be generated from it.*

Even though a detector is often not difficult to implement and maintain, the initial creation and maintenance of the API models need to be considered, too (This does usually not apply to the DDD models as they are usually intended to be provided as manually modeled artifacts.) This often requires some level of architecture reconstruction, as not only the API calls need to be understood, but also calls to the backend microservices they interact with.

We have outlined two feasible approaches in Section 5.3 to deal with this challenge: Firstly, our extraction from the code approach described in [106] can be applied, which enables automatic reconstruction of APIs but requires an initial specification effort. This is applied to the API models of our case studies ES and LS (see [106]). Secondly, a model-driven

approach to generate API code from a model can be chosen, as e.g. offered in MDSL [204] and used in our Case Study PM.

5.9 Related Works

In this section, we compare to the related work. We first discuss related works on API and microservice design, and show that we close the gap in the literature. Most of the existing works mainly provide informal guidance that is not yet supported in an automated fashion. Works that provide automation for conformance assessment are already being discussed, but they either only cover very general conformance assessment not suitable for API/DDD problems or address specific microservice patterns. None exist for the problem of supporting the mapping of APIs and DDD yet. On the other hand, in the existing works on the relation of APIs and DDD, mostly broader microservice architecture aspects are covered, and API topics are rather a side-note. An automated mapping or conformance assessment approach does not exist in this area of research either.

Related Works on API and Microservice Design

Various works address the design of API endpoints, often in the form of patterns, which we consider in our decision options, or as decision models. The Microservice API patterns [212] contain patterns on operation and link/embedded data design [207, 211]. The Enterprise Integration patterns [64] contain various patterns on message construction, such as Command Message or Event Message, that are relevant in message-based endpoint design. The Service design patterns [27] contain high-level patterns on API styles, as well as client-service interactions, which all are relevant for service-based endpoint design.

Fowler [42] links patterns such as Domain Model to distributed system patterns in an enterprise context. His work is one of the first patterns works linking domain models with distributed systems designs using pattern-based design advice.

Richardson [136] presents microservice patterns which are linked in examples to DDD models. Some solutions for those patterns relevant to microservice API endpoints are in our model offered as ADD options. None of these approaches is supported by automated detectors as in our approach yet.

Our work considers solutions akin to such patterns in the decision options, and the patterns contain forces akin to the decision drivers which have led to our ground truth assessment. However, none of the pattern works contains an automated approach, as proposed in our work.

Related Works on Conformance Assessment

Conformance assessment has been applied in various areas of software engineering such as service composition [127] and traceability to guidelines [134]. In general, the conformance relation is defined as the consistency between models [127]. In software architecture, con-

formance assessment addresses relation between a software system's architecture and its intended architecture [28]. With those works our approach shares the general notion of architectural conformance assessment.

We study conformance of an ADD model and a microservice API model for API endpoints. Very close to our work is architectural conformance assessment [193, 107, 108] in which first metrics are derived from a knowledge source and then those metrics are compared to expert judgments based on patterns. Our work has some commonalities with those earlier works, such as deriving a scoring scheme based on human judgement (in the earlier works from patterns; in our work from an empirical study of ADDs) and evaluating the investigated systems using the scoring scheme. In contrast to these works, our work uses detectors instead of special-purpose metrics to automatically detect the core decision points from an empirically grounded ADD model. The reason for the differences is that we investigate ADDs on the mapping of API and DDD models, whereas the prior works study conformance to architectural microservice patterns. To establish traceability and clear judgments on each decision point in the more complex mappings we investigated, we required more detailed assessments (Success, Failure, and Partial Success) with links to the identified violations for traceability, and reasons for detected failures (see Section 5.5) for details).

Related Works on Automated API Approach

Concerning API design from a DevOps perspective, automated tasks are beneficial in terms of time management. In addition, engineering designers are intimate with automated tools because of their speed and accuracy. There are many tasks in the design phase which hard to process automatically. However, the following related works also promote the automated approach over the manual one.

Martin et al. [139] surveyed automated API property inference techniques. Regarding their systematic review, they conclude 60 different techniques into five categories. Their work has the empirical result as an outcome, whereas ours uses empirical results as an input.

Scheller and Kühn [146] proposed the API concepts framework for automated measurement of API usability. Their potential factors influencing API usability are from a literature review. Besides the evaluation metrics, our approach and their approach take context into account. Our context is ADDs, whereas, they shape the context by analyzing those factors closely.

Sohan et al. are interested in the automated API documentation. They introduced automatic RESTful API documentation using an HTTP proxy server [161] and demonstrated their technique by using a computerized software as a service tool called spyREST [160]. The HTTP proxy server-based solution was presented after they analyzed the automated tool's requirements. Meanwhile, their work tends to resolve the problems related to API documentation which aims to bridge the understanding between API provider and API client. Our work concerns more about the conformity of the API, which seeks to promote

conformance assessment of ADDs on API endpoint designs derived from the domain model.

Related Works on Checking Local APIs

Prior works have proposed various approaches to perform checks of certain kinds of conformance for local APIs or API libraries as opposed to the remote, message-based microservice APIs in focus of our work. For example, Zhong et al. [200] propose a method for deriving an API specification from a natural language API documentation. They show that APIs specifications can be inferred with high accuracy scores and the derived specifications can help to find defects in the projects. Tan et al. [172] propose an approach to test the conformance of Javadoc comments and source code, specifically for testing method properties about null values and related exceptions. The approach is tested on six open-source projects and 24 inconsistencies between Javadoc comments and method bodies have been found.

In the PaRu approach [199] API parameter rules describe constraints on parameters of API methods. These can be described as document rules in Javadoc or code rules where the rule is inferred from the source code. PaRu enables the automatic extraction of such local API parameter rules. It can also find overlaps between these two types of rules, which very seldomly occur in the studied open-source systems. In contrast, the API and DDD models addressed in our approach should always have substantial overlaps, as one is derived from the other, and it is the goal of our approach to infer whether the two kinds of models are correctly mapped according to the ADDs informally expressed in Section 5.2.

Such local API approaches are substantially different from the conformance checking provided in our approach, as firstly they do not address properties that require architectural abstraction from the code, such as our ADDs. Thus, the approaches can check directly against source code fragments, and the code's documented defects or existing inconsistencies can be used for building a ground truth. In contrast, as the realization of ADDs cannot easily be spotted from the code, we needed to design our detector approach and construct the ground truth based on our scoring scheme.

Secondly, local APIs do not expose the level of complexity of interaction that remote, message-based APIs offer, e.g. none of the ADDs described in Section 5.2 needs to be considered at all in a local context. Finally, our focus is not on the link of documentation/comments to source code fragments, but on the consistency between API and DDD models. Nonetheless, the named approaches use natural language-based methods, and it would be interesting to study as possible future work if such approaches would work well in our context, too, e.g. applied on structured API documentation.

Related Works on Microservices/APIs and DDD

Various approaches to modeling microservices with DDD, or integrating the two concepts have been proposed [36, 100, 128, 129, 117]. Evans [36] a microservice partitioning approach based on DDD's Bounded Contexts. Merson and Yoder suggest five strategies for microservice design based on DDD aggregates, bounded contexts (BC), domain events and other

strategies. Rademacher et al. [128] propose a UML profile for Domain-driven microservice modeling. In another work, Rademacher et al. [129] discuss challenges of using DDD modeling for microservice architectures in the context of model-driven software development. Pautasso et al. [117] discuss the use of DDD in the context of microservice design in practice. Fan and Ma [37] provide an experience report on migrating a mobile application to microservices, in which they use DDD concept to establish the mapping. Our approach includes modeling and integrating such concepts, with more detailed mappings to API concepts than prior approaches, and in contrast to the other approach also offers conformance assessment. While most of the named approaches consider API aspects, most cover broader microservice architecture aspects as well, which are not part of our approach.

The Context Mapper tool [74] goes one step further than the other microservice/DDD integration approaches and generates various kinds of API specifications from high-level DDD domain models. It is a tool that implements parts of the design options covered in this chapter, especially those options related to the API contracts, including links and operations. In this it is complementary to our work, as our Context Mapper could help in the manual creation of the models required for our approach, and our detectors could check models created with Context Mapper.

Petrillo et al. [122] conducted an empirical study to answer the question how well REST APIs for cloud computing are designed. The conformance of these REST APIs is evaluated by investigating best practices. While their work concentrates on REST APIs only, our work focus on wider range of microservices APIs and includes the DDD and modeling perspective.

CHARMY [118] aims to provide a tool for the model-based design and validation of software architectures. In contrast to our work, it is focused on the early stages of software development.

A number of other empirically grounded studies on API design exist, such as Murphy et al.'s study [102] on usability from the API designers' perspective or Robillard's study [138] on learnability of APIs. However, no other empirically grounded studies on the relation of APIs and DDD exist yet.

5.10 Threats to Validity

Our ground truth assessment depends on the interpretation of the ADDs, and different practitioners might come to slightly different assessments. We mitigated this subjectivity by comparing the ground truth assessment in each iteration of our research study to multiple data points from our prior study [157]. In this empirical work, we considered a relatively high number of practitioner sources (32). Nonetheless, some misinterpretation or bias could have been introduced.

Regarding internal validity, we avoided researcher bias by faithful modeling from the evidence-based information. However, the modeling process can be another source of an internal validity threat. We mitigated this threat by independently cross-checking our models

numerous times in the author team. Each author studied the sources line-by-line, and then refined them in at least five alternating iterations by both authors, until consensus of correct and consistent modeling was reached. We also confirmed that all modeled practices conform to industry best practices reported in [157]. That is, our modeling procedure has produced acceptable models with a high degree of confidence. Even if minor misinterpretations made their way into our models, as all models follow existing industry best practices, the systems still could have been implemented in this way. This means, such misinterpretations might invalidate the empirical results for a specific model (which are a by-product of our study, not the study goal), but not the evaluation results of our automation approach.

Regarding validity of the detector results, there is a threat that our detector implementation contains defects. To mitigate this threat, each detector implementation and changed detector result, throughout our study, was double-checked independently by both authors. Due to the full traceability provided by our approach, our implementation alerts us of changes. It offers a full trace to respective detector causing a change, its status (Success, Failed, or Partial Success), the affected modeling elements (violations), and a reason for the outcome (see Section 5.5). Upon a change in an assessment, both authors inspected the change in-depth to determine whether the result conforms to the expected result.

In addition to these measures, during writing of this chapter both authors independently checked the complete data and all models of this study 3 times each, to avoid mistakes in earlier steps of the research. We also provide all artifacts (code, data set) as open access artifacts to enable reproducibility, so that our modeling and assessments can be independently reproduced.

To avoid system composition and structure bias, we studied many cases in a multi-case study from various third-party authors. Also, the generalizability (external validity) is increased due to the broad range of third-party systems. Nevertheless the threat to validity remains that most of our systems and the case authors have a business/enterprise system background (where DDD is usually applied). Thus our results might not be easily transferable to other contexts such as embedded systems. Further, some systems are built for demonstration purpose. Thus, it is possible that some aspects important in full-scale commercial systems are missing. To mitigate these threats, we aimed in our selection of the systems, summarized in Table 5.2, to cover many different kinds of domains (all within the broader enterprise system context), including purchase ordering, publication management, banking, shopping, process tools, game-related knowledge-bases, disease statistics, and insurance. In addition, they cover all of the possible decision options in our ADDs in many different combinations.

The search process for the case systems might have led to the unconscious exclusion of certain sources. We mitigated this by collecting a relatively high number of cases (12), and checking for each background, for example including that all case authors are practitioners or have a practitioner background.

The construction of our scoring scheme is based on an interpretation and aggregation of practitioner texts in a qualitative, empirical study [157]. While precise decision drivers

and impacts have been identified in the empirical study and followed by us in our scheme, an exact mapping e.g. to crisp numerical assessments would have introduced a significant threat of misinterpretations. In contrast, the used ordinal scale enabled us to turn the qualitative practitioner judgments into numerical information, significantly reducing this threat.

While ordinal scales are commonly used to reflect qualitative judgments [80, 119, 18], the threat to validity remains that some interpretations might not reflect the practitioner judgments accurately. Please note that this threat is mitigated by the fact that our study in first place provides a method for automation, not an empirical assessment of 12 system. If others have a different interpretation of some practitioner judgments in [157], it is easily possible to adapt the respective failing detector(s) accordingly. As we provide all artifacts (code, data set) as open access artifacts to enable reproducibility, such a calibration of our approach can be easily performed. Our approach even provides traceability helping to locate the failing detectors. The concrete system assessment results in Table 5.12 would then change, but the automation approach would not require any alterations.

5.11 Conclusions

In this chapter we studied the conformance assessment of architectural design decision on 1.) the mapping of domain model elements to APIs and API endpoints and 2.) API endpoint designs derived from Domain Models.

Firstly, to answer **RQ4.1**, we introduced an automated assessment approach for conformance to ADDs on the mapping of domain model elements to APIs and API endpoints. Based on the data from an empirical study on this subject, we derived an empirically grounded scoring scheme. Then, we developed automated detectors to identify each of the decision points in our scoring scheme. We evaluated this approach in a multi-case study in which we compared a manually created ground truth to the detector results. To answer **RQ4.2**, in the case studies we were able to identify 86% of the decision points from our scoring scheme correctly. The remaining 14% were mainly cases where redundant modeling at the overlap of two of the ADDs was necessary, but was not correctly performed in the case study models. In two cases, for one of many APIs (a helper API) the mapping to the domain model was omitted. As shown in Section 5.6 with a simple extension of our detectors to cover the redundant cases, we could even reach 95% correct identification.

Secondly, we introduced an automated assessment approach for conformance to ADDs on API endpoint design based on DDD domain models, specifically focusing on operations, links, and resource segregation (to answer **RQ4.3**). Using the data set from an empirical study on those ADDs, we created an empirically grounded scoring scheme. Based on this, we developed novel automated detectors to identify each of the decision points in our scoring scheme. We evaluated this approach in a multi-case study in which we compared the manually created ground truth to the detector results. In the cases of the multi-case study we were able to identify 83% of the decision points from our scoring scheme correctly (to

answer **RQ4.4**). The remaining approximately 17% are those cases where more detailed modeling of aspects beyond architecture-level models, e.g. with a simple additional flag, could have solved the issue. That is, a single manual inspection and correction could lead to correct detection in future runs, e.g. in a continuous delivery pipeline context.

To confirm the results, we performed a statistical evaluation which showed no statistically significant difference between the two variables (ground truth and automated detector results), as well as a negligible effect size between the two variables. In general, manually establishing ADD conformance requires high and continuous effort. Our detectors can spot the violations automatically, support the human-in-the-loop, and support the automated assessment. Our detectors delivered for each of the diverging cases regarding precise results where the problem was located and how to fix it. Thus, in summary, our detectors always yielded clear results for human architects to either assess the models correctly, or inspect and potentially fix modeling issues. Due to the proposed modeling framework and the traceability supported by our detector results, it is relatively easy to adapt or calibrate the approach to a given system setting and set of best practices, and create or improve detectors in a straightforward manner with low effort. Due to the high level of automation and accuracy achieved, our approach is applicable in frequent release and/or large-scale system setting, in which a frequent manual inspection would be clearly infeasible. Our results indicate that such an automated approach is needed for architectural conformance checking, especially when a frequent release schedule is used, as often repeated manual checking is rather infeasible. A modeling framework and traceability support, as provided in our approach, can help to adapt or calibrate the approach to a given system setting and set of best practices, and create or improve detectors in a straightforward manner with low effort. In our approach all aspects, except the case study modeling were automated; we discuss in Section 5.3 how this step can be automated, too, using our existing static code analysis approach [106].

Chapter 6

API Description-Based Conformance Assessment of Architectural Design Decision

This chapter explains a novel approach for automated conformance assessment of API designs in relation to API design decisions. Consequently, it is the first fully automated conformance assessment approach based solely on the code of API Descriptions such as OpenAPI. It responds **RQ4** and is published in **SOSE'22**.

Microservice APIs are often designed based on Domain-Driven Design. It can be challenging to judge the quality of the relation between such a design and the implemented API, especially when facing frequent releases and changes in an API and its design models. Conformance assessment aims to improve software quality by bridging the gap between design and implementation. However, manual conformance assessment of this relation is time-intensive and error prone. This chapter proposes a novel approach for automated conformance assessment of API designs in relation to API design decisions. Our approach aims to provide the first fully automated conformance assessment approach based solely on the code of API Descriptions such as OpenAPI. We applied and exemplified our approach for checking conformance to the patterns and practices in an API design decision for mapping links in a system's domain model to API representations. The total accuracy score (F1-measure) for decision option identification in our multi-case studies for this decision is more than 90%.

6.1 Introduction

Microservice architectures consist of independently deployable, scalable, and changeable services [203, 211, 104]. API design is an important aspect of microservice architectures and includes aspects such as which microservice operations should be offered in the API, how to exchange data between client and API, how to offer links, and how to represent API messages [211, 212]. Many microservices and API abstractions are identified using

Domain-Driven Design (DDD) [33]. DDD is a design approach that places the (business) domain at the center of software designing and architecting. In this chapter, we focus on a specific aspect of mapping domain models to APIs: We apply and exemplify our approach using an Architectural Design Decision (ADD) on link mapping.

Supporting rapid release cycles is one of the key goals of many microservices architectures. Development teams want to deliver their APIs through a Continuous Integration/Continuous Delivery (CI/CD) pipeline. Continuous updates of APIs can quickly lead to violations of conformance of ADDs being taken in API design. Manual checking for such violations is error-prone and time-consuming. In general, such conformance evaluation has been used in a variety of domains of software engineering, including service composition [127] and conformance to standards [134]. The conformance relation is often described as the consistency of models [127]. In our context, architectural conformance assessment is used to verify that ADDs are implemented appropriately in a particular API endpoint. API Descriptions, such as OpenAPI[1], Swagger[2], or RAML[3], are widely used today. Our approach aims to provide the first fully automated conformance assessment based solely on the code of API Descriptions (we focus on OpenAPI). We set out to answer the following research questions:

- **RQ4.5** How to provide a fully automated conformance assessment of API ADDs?
- **RQ4.6 a)** To which extent can OpenAPI-based parsing provide sufficient information for design option identification in API ADDs? **b)** Which level of accuracy can be expected in such fully automated conformance assessment of API ADDs?

To address the research questions, we obtained empirically validated scoring schemes for the exemplary ADD LMD from our previous research. We then derived assessment algorithms and realized novel OpenAPI parser and assessor tools for conformance assessment. In parallel, we selected and analyzed case studies. Then we conducted a manual identification for ground truth assessment in our multi-case study. After that, we executed the automatic parsing and analysis of the case studies. Finally, we performed an empirical validation. For this, firstly, we computed accuracy measures (Precision, Recall, and F1-measure) for the identification of decision options compared to our ground truth. Secondly, we calculated the simple count comparison metric between the correct identification of conformance results and the number of API endpoints.

To address RQ4.5, we built the OpenAPI parser and assessor tools to automate the following tasks: parsing the OpenAPI description, identifying decision options, and assessing conformance. Regarding RQ4.6, we reached an overall average accuracy score of 97.22 (F1-measure) in our multi-case study.

[1] `https://www.openapis.org/`
[2] `https://swagger.io/`
[3] `https://raml.org/`

These numbers seem to indicate that the OpenAPI descriptions (in the cases and for the selected ADD) provide enough information for an acceptable automatic conformance identification.

This chapter is organized as follows. Section 6.2 we describe our motivation and background. Section 6.3 we discuss our research methods. Then, we explain how we prepared the case studies in Section 6.4. In Section 6.5, we present the automated approach for assessing the conformance, and present the empirical validation results in Section 6.6. After that, we discuss our results in Section 6.7. We compare to related works in Section 6.8. Finally, we discuss threats to validity in Section 6.9 and draw conclusions in Section 6.10.

6.2 Background

The primary motivation for this work can be linked to the results from our earlier empirical study of practitioners' perspectives on the interrelations between APIs and DDD [157]. In this work, we empirically identified multiple ADDs in API design that contains API design best practices as decision options. We selected one ADD, the Link Mapping Decision (LMD), to exemplify and validate our work in this chapter. LMD was selected because it proved to be the hardest to assess automatically [156].

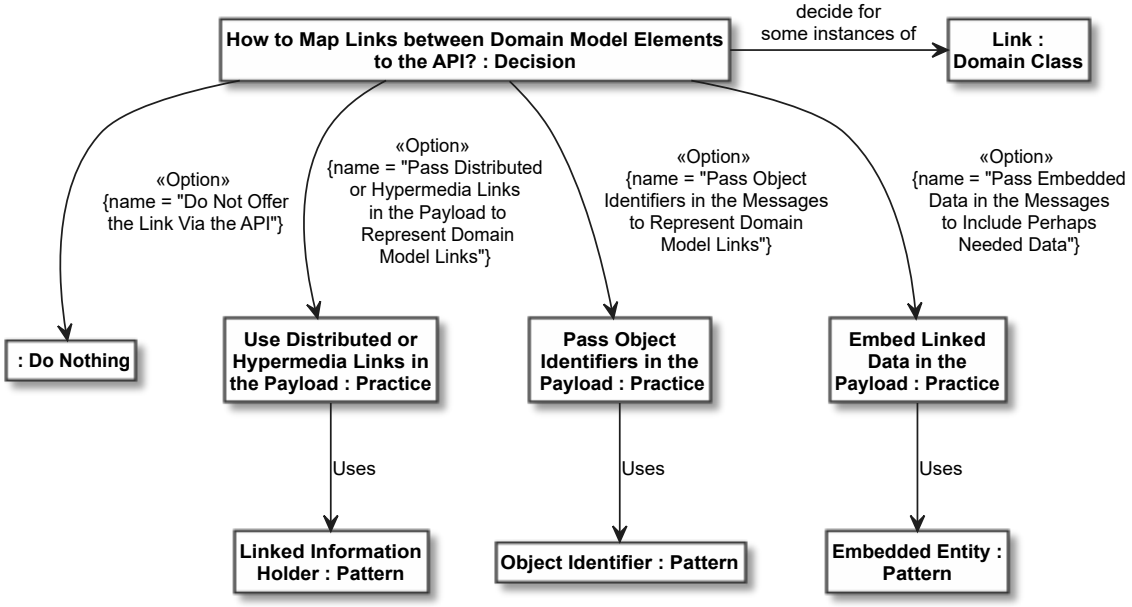

Figure 6.1: Link Mapping Decision

The Link Mapping Decision (LMD) describes possible options for the connections between API endpoints. Links between model elements in a domain model describing the API have a similar function in DDD. Obviously, not all links in a DDD model are eligible for API exposure, but only those between model elements that are designated to be exposed in API endpoints. The Link Mapping Decision is shown in Figure 6.1 along with its alternative decision options. A link in a domain model to be exposed to an API endpoint can be mapped in the following possible ways: Firstly, the option *Use Distributed or Hypermedia Links in*

the Payload maps the domain model link to a standard distributed or hypermedia link, such as URIs in RESTful HTTP or URIs and HAL/JSON-LDs in JSON. An alternative is *Embed Linked Data in the Payload* where the content is added to the message payload rather than connecting to it via a distributed link. Next, there is the option to *Pass Object Identifiers in the Payload* which sends an *Object Identifier* that is meaningful (only) in the server context, but contains no remote location information. Note that each of these options is linked to an API Design Pattern described in the literature, namely LINKED INFORMATION HOLDER [212], OBJECT IDENTIFIER [184, 212], and EMBEDDED ENTITY [212]. Finally, there is the option to choose not to map the domain model link to the API.

The choice of an option has positive or negative influences on desired API qualities. For example, compared to the EMBEDDED ENTITY option, use of distributed links is beneficial for *Data Consistency* as the link is always up-to-date, and thus *API Evolvability* and *API Modifiability* are positively influenced. It leads also to smaller *Message Sizes*. Links however lead to higher *Protocol Complexity*, making it harder to *Minimize API Calls*, and can have a worse *Performance* and *Scalability* because of many resulting distributed calls. The *Object Identifier* based option is very similar in its effects to the distributed links based option. In direct comparison, it has the disadvantages of possibly *Exposing Domain Model Details in API* as well as higher *Coupling of Clients to Server*.

In our previous work [156], we derived an *ordinal scoring scheme* for manual assessment of an API—Domain Model mapping. It is our goal in this work, to automate the human judgement in this scoring scheme solely based on the OpenAPI description of an API. The scoring scheme reflects the empirical insights how practitioners judge different combinations of the above explained decision options:

- IF FOR SOME domain links which are needed by the clients: *Do not offer the Link via the API* THEN assessment = (- -).
- IF (FOR ALL domain links *dl* which are needed by the clients: (IF for *dl* clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* is used)) THEN assessment = (++).
- IF ALL domain links *dl* which are needed by the clients: (IF For *dl* clients usually require most of the linked data elements immediately: THEN *Embed Linked Data in the Payload* is used for the required linked data elements, ELSE *Use Distributed or Hypermedia Links in the Payload* OR *Pass Object Identifiers in the Payload* is used)) THEN assessment = (+).
- IF (FOR SOME domain links *dl* which are needed by the clients: If for *dl* clients usually require most of the linked data elements immediately: *Embed Linked Data in the Payload* is NOT used for *dl*) THEN assessment = (-)
- IF any other combination is used (e.g. for some domain links *dl* which are needed by the clients, and for *dl* clients usually do NOT require most of the linked data

elements immediately AND *Embed Linked Data in the Payload* is used for *dl*) THEN assessment = (o).

In general, our conformance assessment approach is based on the notion of a structured API description or API contract [159] as input. API DESCRIPTION [90] is a pattern that was previously released as part of the Microservice API Patterns [212], and API CONTRACT is a variant of the general INTERFACE DESCRIPTION pattern [184].

6.3 Research Methodology

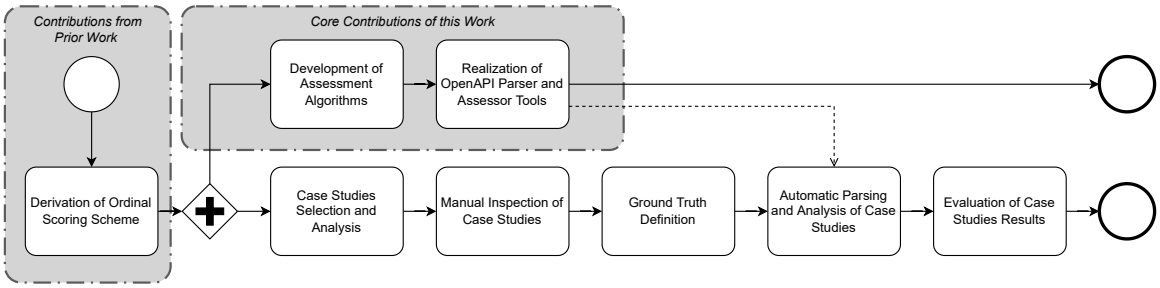

Figure 6.2: Research Methodology Overview

Figure 6.2 illustrates the research methods applied in our study. The *derivation of ordinal scoring scheme* from the previous section has been performed in previous work [156]. Github[4] served as the primary source for *case studies selection and analysis*. We followed the recommendations established by Petersen et al. [121] (originally for systematic mapping studies) for establishing search phrases according to the PICO principles [75]. We applied two principles from PICO, which are *population* and *intervention*. Then, we applied inclusion/exclusion rules. The case studies were chosen independently and in parallel to our approach's development. We performed a *manual inspection of case studies* and created the *ground truth definition* for each API endpoint in the case studies. This ground truth is based on the scoring scheme from the previous work. We have followed the following definitions during our inspections [212, 90, 165]: An *API endpoint* is the provider-side end of a communication channel and a specification of where the *API* endpoints are located so that *APIs* can be accessed by *API clients*. Each endpoint thus needs to have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web (WWW), as well as in HTTP-based SOAP or RESTful HTTP. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints.

For *Automatic Parsing and Analysis of Case studies*, we applied the Panda and Numpy libraries. We compared the results of the automatic assessment with the ground truth assessments in our *Evaluation*. We quantified our findings by counting the correct assessment identifications and calculating accuracy scores.

In the following subsections, we explain the methods for each of these core contributions.

[4] `https://github.com/`

Realization of OpenAPI Parser and Assessor Tools

We constructed the parsing and assessment algorithms using our prior work's scoring scheme. While the conditions remained constant, the scope of identification shifted. Rather than analyzing the whole system, we opted to assess individual API endpoints. Then, we created an algorithm that receives the endpoint name, HTTP methods, and the binary value of each decision option. It returns the assessment result and the endpoint name. We used the OpenAPI specification as a starting point to identify three kinds of relations:

1. single decision option per data elements,
2. multiple decision options per operation, and
3. the existence of decision options per API endpoint.

We analyze these three relations in turn. To identify the operation, we used the *operationId* element of OpenAPI-based specification in the first place. When the *operationId* is omitted; we inspect the HTTP method instead. To obtain this relation, we use a parser-based detection technique. We adapted the swagger-reader [120] for this purpose. Our OpenAPI parser converts the information into a decision option identification. Moreover, our output identifies the interrelation between the set of decision options and the API operation.

Decision Options Identification and Verification For the decision option identification, we started by applying the parser-based detection approach. The result is the automated generation of the option per API data element (Relation 1 in the previous section). For validation, we generated the raw accuracy data (True Positives: TP, True Negatives: TN, False Positives: FP, and False Negatives: FN) by comparing the automated detection with the manual detection. Then, we calculated the accuracy scores (Precision, Recall, and F1-Measure) using the following equations:

$$Precision = \frac{\sum_{n=1}^{\infty} TP}{\sum_{n=1}^{\infty}(TP + FP)} \tag{6.1}$$

$$Recall = \frac{\sum_{n=1}^{\infty} TP}{\sum_{n=1}^{\infty}(TP + FN)} \tag{6.2}$$

$$F1 - Measure = 2 * \left[\frac{Precision * Recall}{Precision + Recall}\right] \tag{6.3}$$

6.4 Case Studies Selection and Analysis

This section elaborates on how we selected the case studies and what the ground truth assessments of each case study are that resulted from our manual assessment of the cases. Figure 6.3 summarizes the number of case studies that were considered during the study selection procedure.

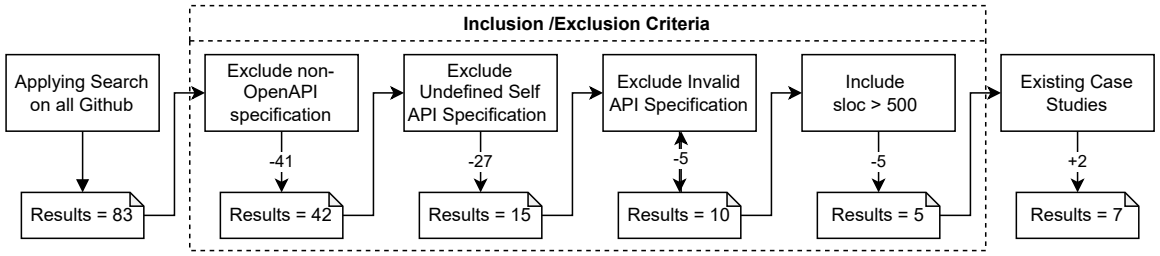

Figure 6.3: Number of Included Case Studies During the Study Selection Process

Firstly, we determined the search string keywords. Our population was restricted to the Github repositories. The interventions are the API DESCRIPTION pattern and OpenAPI. As a result, we utilized the Github Search API to do the queries based on a number of criteria:

- The search string ("OpenAPI", "API", "specification") in project name, description, and/or README file.
- The repository's size in the rank 10-100 MB.
- The author has more than 9 followers.
- The search string("OpenAPI", "API specification") without any conditions.

As a result, we obtained 83 repositories in total from the Github repository. Then, we subjected all sources to an in-depth investigation. The following inclusion/exclusion criteria were used to include sources:

- Sources with OpenAPI specifications: We reviewed each source individually and rejected sources that did not use OpenAPI.
- Sources with API specifications relevant to the project itself: we examined each API specification and filtered out test files, package files, sample files, example files, configuration files, and generated API specification files.
- Sources with valid API specifications: We used the OpenAPI Tree tool[5] to test whether or not an API specification was legitimate. A graphical representation, such as the one provided in Section 6.5, is generated when the API specification is valid. We also included sources where the API specification came with API contracts and the ability to assess the LMD was given, even if the OpenAPI Tree tool failed.
- Sources with more than 500 SLOCs.

Moreover, we included two sources from the prior case studies that met these criteria. They came with API contracts and the ability to assess the LMD as well. The Table 6.1 shows a brief summary of each of the inspected cases studies, ID, number of slocs, number of endpoints, number of pairings between API endpoint/API operation, and the number of pairings between API endpoint/API data element.

[5] `http://api-ace.inf.usi.ch/openapi-to-tree/`

Table 6.1: Overview of the Inspected Case Studies

ID	**Description**	**sloc**	**endpoint (E)**	**E - operation**	**E - data**
RW	Realworld Example App: The API specification supports technology stack diversity. The documentation of a full-stack app called Conduit is available in the YAML format (version 3.0.1). Ref:`https://github.com/gothinkster/realworld`	916	12	19	61
DX	Graph DevX API: It provides a backend RESTful API that has resources used by the Microsoft Graph documentation, Graph Explorer, Powershell SDK, and Graph samples workload teams, all of which are handled by the Graph PM team. Ref: `https://github.com/microsoftgraph/microsoft-graph-devx-api`	875	10	11	55
NC	Disease Statistics App: It delivers a variety of virus-related information. Its public API is an open-source project that is spearheaded by four primary contributors from four different countries. The specification for is available in JSON format (version 3.0.0). Ref: `https://github.com/disease-sh/API`)	2773	39	39	107
EA	Admin Example API Description: It provides the activity of the admin. The current version of the OpenAPI specification is 3.0.1. Ref: `https://github.com/mohsenTalal/OpenAPI-Swagger`	865	7	10	95
PT	Pinterest's REST API OpenAPI Description: This description is utilized internally by Pinterest's API architecture. It contains OpenAPI specifications in version 3.0.3 Ref: `https://github.com/pinterest/api-description`	4981	23	30	190
RG	Registry API: It enables teams to maintain API descriptions. The Registry API is a database of business APIs that is machine-readable and serves as the basis for web directories, portals, and workflow managers. Ref:`https://github.com/apigee/registry`	1846	20	35	203
ST	STITCH API Description: STITCH is a database that projected chemical-protein interactions. Correlations are formed both directly (physically) and indirectly (functionally) through computer prediction, information transfer between species, and aggregated interactions from other (primary) databases. Ref: `https://github.com/micheldumontier/API-descriptions`	716	12	12	65

To provide a baseline for further assessment, we manually analyzed each API endpoint in our multi-case studies based on the scoring scheme from Section 6.2. Regarding the ground truth agreement, one author used the scoring scheme to derive the ordinal scale of each API endpoint. Then, the other author verified these judgements. Table 6.2 summarizes the ground truth assessments. Please note that this is not the result of the automated assessment approach presented below but the manually derived ground truth obtained by manually assessing every single endpoint of the 7 cases. We use it later for scientific evaluation of the automatically computed results. As can be seen, the cases cover a wide range of possible combinations of the decision options for the decisions.

Table 6.2: Ground Truth Assessment: Scores for API Endpoints

Assessment Results	RW	DX	NC	EA	PT	RG	ST
+ +	5	6	28	0	5	0	0
+	0	0	0	0	3	6	0
o	5	3	0	7	13	11	12
-	2	1	11	0	1	3	0
- -	0	0	0	0	0	0	0
Total	12	10	39	7	23	20	12

6.5 Proposed Approach

This section elaborates on the proposed approach, i.e. how API developers would use our approach. Our approach starts by parsing an API description and then performs an identification of the decision options. In particular, we first identify the decision option chosen in the API description per API data element, then per API operation, and then per API endpoint. Three levels of step-wise decision option identification are needed to reflect the complexity of the decision's relation, as explained in Section 6.2.

Figure 6.4 shows an overview of the three levels of step-wise decision option identification. To illustrate these levels, we selected three API meta-model elements—*API Endpoint*, *API Operation*, and *API Data Element*—that are shown on the left-hand side of the figure. We show these meta-model elements in Figure 6.4 in relations using three kinds of associations: *Group*, *Generic Association*, and *Specific Association*. One API endpoint, two API operations, and five API data elements are depicted in Figure 6.4 (a.)-(c.). They depict the relationships between API endpoints/API data elements, API endpoints/API operations, and API endpoints with six, two, and one pairs between the API endpoint and its data elements, respectively. Our three identification levels are used to identify each of these possible structures in OpenAPI.

Based on the detected information, the approach then assesses the conformance using the scoring scheme. Finally, it generates the assessment results to be displayed to the API developer. Please note that these steps are to be conducted entirely based on OpenAPI-based API descriptions, i.e. requiring no human effort for modeling creation.

Figure 6.5 shows the tree model of the *Real World Example App* case's OpenAPI specification, which we use as a running example. It contains 12 API endpoints for which

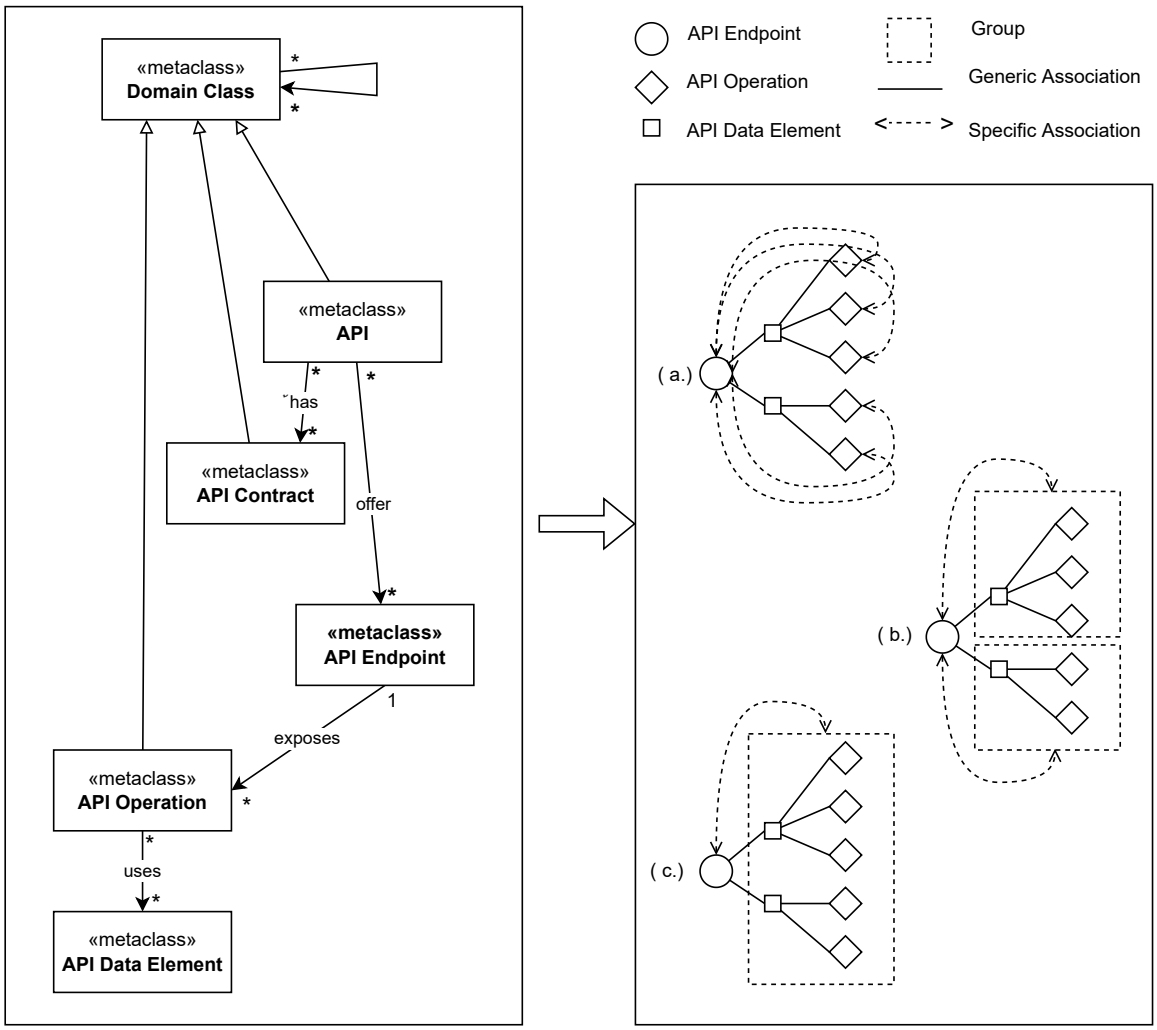

Figure 6.4: Three Levels of Step-wise Decision Option Identification

we evaluated LMD conformance. The decision option chosen for each endpoint depends on the relationship between the API endpoint and the API data elements used in its operations. Since we studied both the incoming and outgoing data elements in the API specification, we have 61 pairings between the API endpoint and its data elements. Further, there are 19 relations between the API endpoints and their operations. To illustrate this further, we use the endpoint "`/profiles/{username}/follow`" highlighted in red in Figure 6.5 as a running example.

Parsing API Description

For the LMD decision, we parse the following information using our OpenAPI Parser: 1.) *Endpoint Data* 2.) *Operation ID* 3.) *Incoming Distributed Link* 4.) *Incoming Embedded Data* 5.) *Incoming Object ID Based Link* 6.) *Response Code* 7.) *Outgoing Distributed Link* 8.) *Outgoing Embedded Data* 9.) *Outgoing Object ID Based Link*.

The 1.) *Endpoint Data* here contains the API endpoint's name and the HTTP method. Since one API endpoint can bind with more than one HTTP method, we also included the

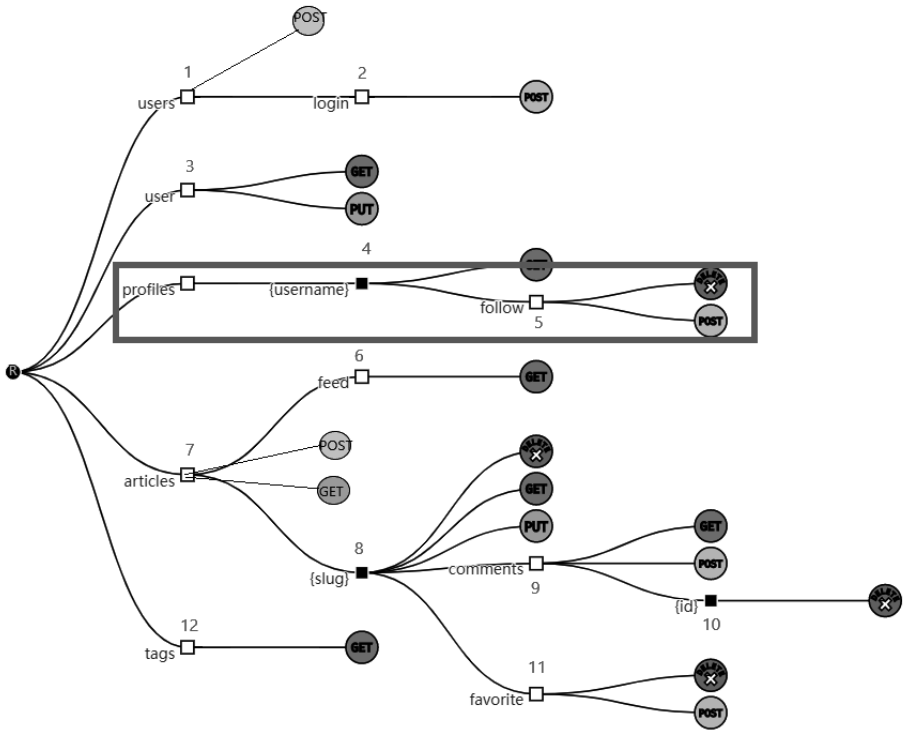

Figure 6.5: OpenAPI Tree of the *Real World Example App* case

HTTP method to make this endpoint data be usable as an identifier. The HTTP method identification, performed here, is helpful for API data element identification later on. The 2.) *Operation ID* helps to check whether the operation is already implemented in the backend, while the 6.) *Response Code* helps to check the availability of the outgoing data.

The three incoming data patterns, 3.) *Incoming Distributed Link*, 4.) *Incoming Embedded Data*, 5.) *Incoming Object ID Based Link*, as well as the three outgoing data patterns, 7.) *Outgoing Distributed Link*, 8.) *Outgoing Embedded Data*, 9.) *Outgoing Object ID Based Link*, are the specific information needed for identifying the LMD decision options (using the approach explained above).

Table 6.3 shows the sample output that we retrieve from the OpenAPI Parser for the *Real World Example App* study case. This table is also the sample input of the assessorin Section 6.5.

Identification of Decision Options

There are 916 SLOCs in RW's OpenAPI specification. The code snippet of the endpoint "`/profiles/{username}/follow`" using the POST HTTP method is shown in Listing 6.1. Listing 6.1 contains one API endpoint (`/profiles/{username}/follow`), one relation between API endpoint and API operation (`/profiles/{username}/follow` - POST or FollowUserByUsername), and three relations between API endpoint and API data element that we used to identify the decision option as follows:

- `/profiles/{username}/follow` - username

Table 6.3: Sample Output for *Real World Example App* from the OpenAPI Parser

ID	1	2	3	4	5	6	7	8	9
RW-A1	`/profiles/{username}/follow-DELETE`	UnfollowUserByUsername	null	username:true	null	200|401|422	`application/json:#/components/schemas/ProfileResponse|application/json:#/components/schemas/GenericErrorModel`	null	null
RW-A2	`/profiles/{username}/follow-POST`	FollowUserByUsername	null	username:true	null	200|401|422	`application/json:#/components/schemas/ProfileResponse|application/json:#/components/schemas/GenericErrorModel`	null	null

- `/profiles/{username}/follow` - ProfileResponse
- `/profiles/{username}/follow` - GenericErrorModel

Listing 6.1: The API Description Example

```
"/profiles/{username}/follow": {
  "post": {
    "tags": ["Profile"],
    "summary": "Follow a user",
    "description": "Follow a user by username",
    "operationId": "FollowUserByUsername",
    "parameters": [{
      "name": "username",
      "in": "path",
      "description":
      "Username of the profile you want to follow",
      "required": true,
      "schema": {"type": "string"}
    }],
    "responses": {
      "200": {
        "description": "OK",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/ProfileResponse"
            }
          }
        }
      },
      "422": {
        "description": "Unexpected error",
        "content": {
          "application/json": {
            "schema": {
              "$ref": "#/components/schemas/GenericErrorModel"
            }
          }
        }
      },
```

Listing 6.1 contains snippets that correspond to the detected options RW-V5 and RW-V7 in Table 6.4.

Table 6.4: Decision Options Detection

ID	IDENTIFIER	DECISION-OPTION
RW-V1	`/profiles/{username}/follow;DELETE;UnfollowUserByUsername; application/json:#/components/schemas/GenericErrorModel`	RES-DistributedLink
RW-V2	`/profiles/{username}/follow;DELETE;UnfollowUserByUsername; application/json:#/components/schemas/ProfileResponse`	RES-DistributedLink
RW-V3	`/profiles/{username}/follow;DELETE;UnfollowUserByUsername; username:true`	REQ-EmbededData
RW-V4	`/profiles/{username}/follow;POST;FollowUserByUsername;application/ json:#/components/schemas/GenericErrorModel`	RES-DistributedLink
RW-V5	`/profiles/{username}/follow;POST;FollowUserByUsername;application/ json:#/components/schemas/ProfileResponse`	RES-DistributedLink
RW-V6	`/profiles/{username}/follow;POST;FollowUserByUsername;username: true`	REQ-EmbededData

Table 6.4 presents the decision option identifications. The Columns *identifier* and *decision-option* are generated automatically. Moreover, we included an ID column for the reporting purpose. The six entries (RW-V1 to RW-V6) are all addressing the same API endpoint (/profiles/username/follow). The two operations are *UnfollowUserByUsername* and *FollowUserByUsername*. Even though, we can retrieve these operations' name from the "operationTd" element, the HTTP methods are interchangeable because the relation between the HTTP method and "operationId" elements is 1:1. There are three data elements involved in the example with these two operations: *GenericErrorModel*, *ProfileResponse*, and *username*. In conclusion, the API endpoint contains two relations between an API endpoint and API operations, and six relations between the API endpoint and API data elements.

Due to the fact that we have three solution patterns and we investigated all of them in both incoming and outgoing data, we have six different decision options to detect as shown in Table 6.5. The table summarizes the concepts applied to retrieve the decision option from API description.

Table 6.5: The rule of Decision Options Identification

Decision Option	**Request**		**Response**
element	*parameters*	*requestBody*	*responses*
DISTRIBUTED -LINK	$ref	$ref	$ref + checkResponse()
EMBEDDED -DATA	name, description + !checkKeywords()	x	item.name + !checkKeywords() + checkResponse()
OBJECT-ID -LINK	name, description + checkKeywords()	x	item.name + checkKeywords() + checkResponse()

The input is the *OpenAPI* description. To collect information from OpenAPI, we adopted the swagger model and swagger parser libraries. The output is a hashmap of the *identifier* and the *decision option* as shown in Table A 6.4. We aimed to identify the decision options of LMD between an API endpoint and a single data element of this

API endpoint. We investigated the *parameters* and *requestBody* elements for request data (incoming data), while the *responses* element was investigated for response data (outgoing data). The *parameters* element holds the information related to the GET, DELETE, and HEAD HTTP methods. The *requestBody* element contains the information related to the POST, PUT, and PATCH HTTP methods. Moreover, we added *checkKeywords()* to allow us identifying id-related keywords, such as “id" or “identifier". For DISTRIBUTED LINKS, the investigated sub-element is *§ref*. For EMBEDDED DATA and OBJECT ID LINKS, the syntax is almost identical. We checked the *name* and *description* sub-elements in the *parameters* element and the *item.name* sub-element in the *responses* element. The association between API endpoint and the API data element turns to the OBJECT ID LINKS decision option when *checkKeywords()* is positive, but the EMBEDDED DATA decision option is the opposite.

Conformance Assessment

To provide our assessment algorithm, we further needed to resolve the difficulty of identifying decision options through the parser-based approach. We were able to associate the multiple decision options per operation using the parser-based detection technique. This association is rendered in terms of the long string, so we extracted the API endpoint and operation. This extraction allowed us to combine all the decision options of each API endpoint. To assess the conformance assessment per API endpoint, we focused on the existence of decision options. That is why we used the set of booleans as an input. Concerning the scoring scheme in Section 6.2, we interpreted the information for two conditions: C1.) domain links are needed by the clients or not? C2.) clients usually require most of the linked data elements immediately or not? The answer for C1 should be “need" or “no need", whereas the answer of C2 should be “yes" or “no". Based on this, we can then fully assess the conformance to one of the decision options (“Distributed Link", “Embeded Data", “Object ID Link", and “None Link"). Due to the scoring scheme’s complexity, we included the HTTP method to represent the operation since certain API endpoints in the API specification lacked an *operationID*.

Table 6.6: The Output of *Real World Example App* from the Assessor Tool

Index	API-Endpoint(1–)	HTTP-Method (–1)	3	4	5	7	8	9	Actual-Assessment	Expected-Assessment
...										
7	`/profiles/{username}/follow`	POST; DELETE	F	T	F	T	F	F	Neutral	Neutral
...										

The handling of a single decision option per data elements was explained in Section 6.5. The judgment of LMD was based on the relation between the single data element and the API endpoint. The handling of multiple decision options per operation were identified by grouping the decision options based on their operation. We modified the prior parser-based detection rule to enable it to extract the association between the multiple decision options and the operation. The existence of decision options per API endpoint was explained in the following algorithm.

Algorithm 3: Assessment Pseudocode

```
input: String api_description
output: DataFrame assessment_results
begin
assessment_results = parseAPIDescription(api_description)
assessment_results.groupByAPIEndpoint()
assessment_results.transformOptionsToBoolean()
addAssessmentColumn(assessment_results)
return assessment_results
end
```

Algorithm 3 presents the assessment pseudocode. The input is the String from API description. The output is a DataFrame with assessment result. *addAssessmentColumn()* handles the multiple decision options per operation that have been identified by grouping the decision options based on their operation. We modified the prior parser-based detection rule to enable it to extract the association between the multiple decision options and the operation. Since one API Endpoint possibly contains multiple operations, we applied *groupByAPIEndpoint()* to reduce the investigated entries. These investigated entries are the information per one API endpoint. After that, we turn the decision options which is the String value in to the Boolean value using *transformOptionsToBoolean()*. Thus the existence of each decision option is denoted as TRUE or FALSE. For *addAssessmentColumn()*, the input—*assessment_results*—refers to the endpoint name, the HTTP method, and a collection of Boolean values. The set of Boolean values represent the existence of that particular decision options of each API enpdoint. The results are returned as a tuple.

As previously stated, the /profiles/username/follow API endpoint has two associations between the API endpoint and the operation. Table 6.3 shows these two associations, whereas Table 6.6 shows the final outcome, the conformance assessment result of a single API endpoint.

6.6 Empirical Validation Results

To empirically validate our approach, we conducted two empirical validations: the *verification of decision options* and the *comparison to the ground truth*. While the verification of decision options are measured by the accuracy scores, the comparisons to the ground truth are using a simple count metric.

Table 6.7: Information of Accuracy Scores

ID	ED	TP	FP	FN	TN	P	R	F1
RW	61	61	3	0	306	0.9531	1	0.976
DX	55	55	4	0	271	0.9322	1	0.9649
NC	107	107	20	0	515	0.8425	1	0.9145
EA	95	95	0	0	475	1	1	1
PT	190	172	18	0	950	0.9503	1	0.9503
RG	203	203	0	0	1015	1	1	1
ST	65	65	0	0	325	1	1	1
Total	776	**Average Accuracy Scores**				0.9479	1	0.9722

Verification of Decision Options Table 6.7 summarizes the results for the seven API descriptions that we selected from the third-party sources in our multi-case study. Accuracy score information includes the number of inspected relations (ED), and the prediction outcome in terms of True Positives (TP), False Positives (FP), False Negatives (FN), and True Negatives (TN). The number of inspected relations for the cases varied from 55 to 203. These relations refer to the links between the API endpoint and the single data elements. There are a total of 776 inspected relations (ED).

Consequently, the average Precision, Recall, and F1-Measure values are 0.9479, 1, and 0.9722, respectively. To provide an audit trail of the research and enable repeatability of the study, we provide open access to our the data set and the source code[6].

Comparison to the Ground Truth on the Scoring Scheme This comparison is for the conformance assessment correctness of the LMD decision based on the extracted data are shown in Table 6.8.

Table 6.8: Information of Accuracy Scores

Assessment Results		(+ +)	(+)	(o)	(-)	(- -)	Correct Identification API Endpoints' number
RW	AR	4	0	6	2	0	
	ER	5	0	5	2	0	11/12
DX	AR	7	0	2	1	0	
	ER	6	0	3	1	0	9/10
NC	AR	28	0	0	11	0	
	ER	28	0	0	11	0	39/39
EA	AR	0	0	7	0	0	
	ER	0	0	7	0	0	7/7
PT	AR	3	1	15	4	0	
	ER	5	3	13	2	0	19/23
RG	AR	0	6	11	3	0	
	ER	0	6	11	3	0	20/20
ST	AR	0	0	12	0	0	
	ER	0	0	12	0	0	12/12
Total							117/123 = 95.12%

The endpoints of *Real World Example App* (RW), *The Graph DevX API* (DX), *Disease Statistics App* (NC), *Admin Example API Description* (EA), *Pinterest's REST API OpenAPI Specification* (PT), *The Register API* (RG), *STITCH API Description* (ST) are 12, 10, 39, 7, 23, 20, and 12, respectively. AR stands for an actual assessment, which denotes an actual assessment produced automatically by our assessor tool. On the other hand, ER stands for an expected result, which conforms to the ground truth. Table 6.8 summarizes the evaluation results for the seven case studies using simple count metrics, with an overall match of 95.12 %. This score was determined by dividing the number of correct identifications by the number of endpoints. *Disease Statistics App*, *Admin Example API Description*, *The Register API*, and *STITCH API Description* are four case studies with perfect matching.

[6]We provide generated data, Python source code, and two excutable java files with dependencies as a replication package for download in the Zenodo long-term open access archive (`10.5281/zenodo.6564304`)

6.7 Discussion

Our automated conformance assessment is intended to substitute the time-consuming manual assessment. Concerning the API design from a DevOps or CI/CD perspective, automated quality control tasks are beneficial in terms of ensuring quality and avoiding error-prone and time-consuming manual inspections. We provided the novel concepts for the OpenAPI parser and the assessor tools to address RQ4.5. The OpenAPI parser is used to generate the input for the assessor tool that reports the ordinary scale conformance assessment per API endpoint. These tools aid in the following manual tasks: parsing OpenAPI description, identifying the decision options, and assessing conformance. Additionally, we demonstrated an approach using the LMD ADD and *Real World Example App* case study as examples.

To answer RQ4.6a, we conducted an empirical study to validate the decision option as discussed in Section 6.6 above. Overall, with 97.22% the accuracy score (F1 measure) is quite high (substantially better than 0.8). Occasionally, Embedded Data is unrecognized by the parser-based detector. Because in OpenAPI the source of the root of the OBJECT ID and EMBEDDED DATA is the same, we may presume that they are not different. Please note that this problem may be resolved by assigning an array string datatype rather than a single string datatype. While this would require changing the OpenAPI specifications that are inspected, with a simple developer guideline and an automated test of its application, we could potentially achieve 100% accuracy in our cases.

To answer RQ4.6b, we conducted an empirical study to compare the automated results to the application of the scoring scheme in the ground truth. We also illustrate these comparison results of all seven API specifications with the stack bars in Figure 6.6. The stack bars show the assessment information for both the expected results (ER) and the actual results (AR). Our case studies include a wide range of endpoints, ranging from seven in the *Admin Example API Description* case study to 39 in the *Disease Statistics App* case study.

In the *Real World Example App* and *The Graph DevX API* case studies, one result does not match due to two API operations' involvement. We could achieve 100% matching in these scenarios by implementing further processing to analyze the relationship before combining the API operation element.

For the *Pinterest's REST API OpenAPI Specification* case study, four results are not matching. There are two reasons: 1.) from *Negative* to *Positive*: sometime, the OBJECT ID and EMBEDDED DATA can refer to the same item in certain cases. 2.) from *Neutral* to *Very Positive*: more caution is required when two operations are involved. Additionally, such mismatches could be caused in cases where just the ID-object is specified or where double linking is used to facilitate distributed links.

Four case studies got a fully correct match to the ground truth, namely *Disease Statistics App*, *Admin Example API Description*, *Registry API*, and *STITCH API Description*. Due to the fact that *Disease Statistics App* is composed entirely of GET HTTP methods,

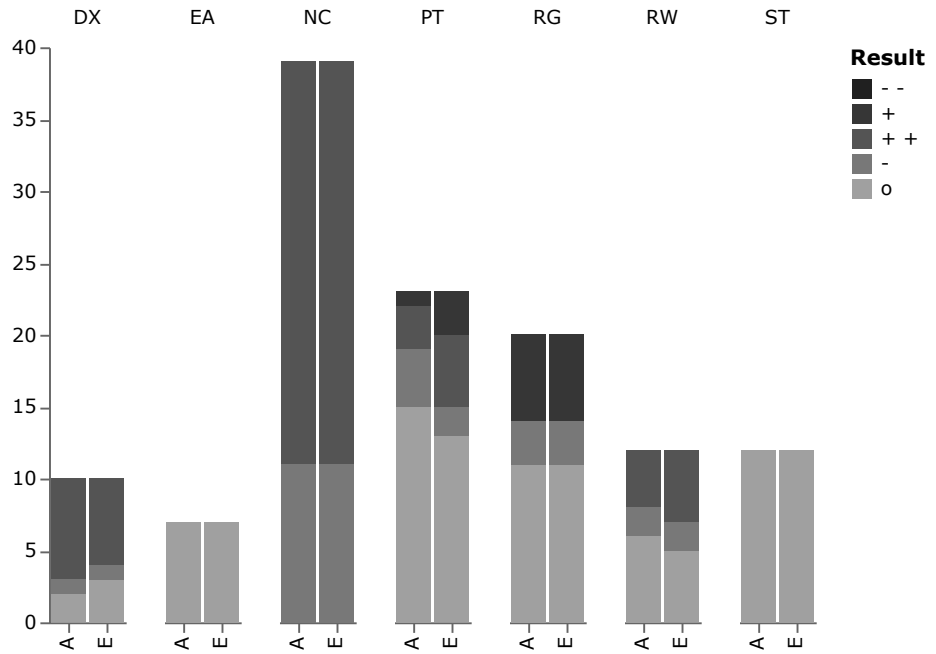

Figure 6.6: The Comparison of Case Studies' Assessment Results

it is relatively simple to retrieve a correct conformance assessment here. Also, this project's evaluation result has just two values: very positive (++) and negative (-). When clients require the majority of the linked data elements instantly, very positive means, "Embed Linked Data in the Payload" is utilized for the necessary linked data, unless "Distributed or Hypermedia Links in the Payload" is used. The negative value indicates that "Embed Linked Data in the Payload" is not utilized for certain domain connections required by clients. Both the *Admin Example API Description* and *STITCH API Description* case studies, even though their assessment results show only neutral values (o), allow clients to input the request in various forms ("application/json-patch+json", "application/json", "text/json", and "application/*+json"). Nevertheless, we treated it as one association between an API endpoint and a single API data element. For the RG, we added the "id." keyword to checkKeywords() since the authors of this API description often used "id." to indicate their OBJECT ID LINK. As a result, we were able to achieve a 100% of the matching score. This shows that sometimes minimal development effort is needed, like adding an identifier name, to achieve fully correct results.

In conclusion, 117 endpoints out of 123 were assessed correctly. The overall matching score is 95.12 percent.

6.8 Related Works

Three strategies exist for ensuring static architectural conformance: dependency-structure matrices, source code query languages, and reflexion models [114]. Architecture conformance or compliance checking has been employed to ensure architecture consistency [176]. Both terms are often used synonymously. Architectural documentation that preserves the

architectural knowledge becomes irrelevant when the conformance between documentation and implementation is non-existent [58]. This kind of conformance is constructed through model-driven engineering or reverse engineering. Some studies have been conducted on the conformance assessment of ADDs [193, 107, 108]. Zdun et al. [193] studied ensuring and assessing architecture conformance to microservice decomposition patterns. Their work consists of two main contributions: microservice design constraints and metrics to evaluate the architecture conformance to microservice patterns. Ntentos et al. [107] proposed a model-based technique for evaluating architectural conformance to microservice architecture patterns and practices. They also offered the metrics for assessing architecture conformance if an architecture complies with a typical architectural design option in the microservice domain [108]. In contrast to our approach, these approaches work from models; our approach only requires the OpenAPI code.

Some researchers studied conformance related to API or DDD. Athanasopoulos et al. [5] interpreted the framework for assessing interface uniformity in REST by discussing a conceptual framework and a criteria's collection. Domain-specific strategies might be used to map these criteria in terms of directing and/or inspecting the level of uniformity of a REST-based API. Kapferer and Zimmermann [73] proposed domain-driven architecture modeling and rapid prototyping with a tool called Context Mapper. They considered the conformance of the Domain-Specific Language (DSL) with the original DDD patterns. Our approach is the first to study conformance in the mapping between DDD models and APIs, ADDs in this context, and works by checking the API description code directly without human specification effort.

Concerning API design from a DevOps perspective, automated tasks are beneficial. Numerous tasks during the design phase are difficult to automate. The following related works, however, advocate for an automated API method over a manual one, similar to what we propose. Robillard et al. [139] surveyed automated API property inference techniques. They classify 60 various strategies into five groups based on their systematic review. Scheller and Kühn [146] proposed the API concepts framework for automated measurement of API usability. Sohan et al. are interested in automated API documentation. They introduced automatic RESTful API documentation using an HTTP proxy server [161].

6.9 Threats to Validity

This section discusses threats to the validity of our study. Our ground truth assessment is dependent on how the ADDs are interpreted, and different practitioners might get slightly different conclusions. We mitigated this subjectivity by comparing the ground truth assessment in each iteration of our research study to the prior study [157]. In this empirical work, we considered a relatively high number of practitioner sources (32). Nonetheless, some misinterpretation or bias could have been introduced. The construction of our scoring scheme is based on an interpretation and aggregation of practitioner texts in a qualitative, empirical study [157] While precise decision drivers and impacts have been identified in the

empirical study and followed by us in our scheme, an exact mapping e.g. to crisp numerical assessments would have introduced a significant threat of misinterpretations. In contrast, the ordinal scale allowed us to convert qualitative practitioner assessments to numerical data, considerably lowering this threat.

Ordinal scales are often used to reflect qualitative judgments. The threat to validity remains that some interpretations may not reflect the practitioner's judgments. This threat is mitigated by the fact that our study presents a strategy for automation rather than an empirical judgment. If others perceive any practitioner's decisions differently, it is simple to update the source code. As we provide all artifacts (code, data set) as open access artifacts to enable reproducibility, such calibration of our approach can be easily performed. The specific findings of the system evaluation would vary, but the automation technique would remain the same.

Regarding the validity of the outputs of the parser-based detectors, there is a threat that they might contain defects. To counteract this issue, we have included an empirical validation effort to ensure that the decision options are legitimate.

For the internal validity, we minimized the researcher bias by doing several independent cross-checks of our findings within the author team. Both authors independently reviewed the whole data in this study three times throughout the writing process to prevent errors in the early stages of the research. We also offered all artifacts (code, data set) as open access artifacts to facilitate reproducibility, allowing for independent replication of our tools and evaluations.

6.10 Conclusion

The Link Mapping Decision (LMD) is one of the architectural design decisions in the context of the interrelation between API and domain models. It is strongly associated with API endpoints. An API description contains mainly information on API endpoints. For this reason, in this chapter, we investigated how far it is possible to automatically assess conformance to LMD solely based on OpenAPI code, which has not been pursued before. To address RQ4.5, we developed the OpenAPI parser and assessor tools to support the following manual tasks: parsing OpenAPI descriptions, identifying the decision options, and assessing conformance. To address RQ4.6, we conducted empirical validations. For RQ4.6a, our accuracy scores towards the identification of decision options are as follows: Precision of 94.79 %, recall of 100 %, and F1-measure of 97.22 %. For RQ4.6b, we compared our automated approach's outcomes to the ground truth. We discovered that our results were on matching about 95.12 % correctly. This indicates that full automation of conformance assessment is indeed possible.

Part V

Conclusion

Chapter 7

Conclusion

This chapter concludes the thesis by revisiting the research questions introduced in Chapter 1 summarizing the contributions, and describing the limitations of presented approaches. We will discuss to what extent the research questions have been covered, our provided replication packages for replicability of the research, and state opportunities for the potential future work based on the achieved results.

7.1 Research Questions Revisited

This section discusses our findings in the relation to research questions and then we explain how the study aims are fulfilled.

Research Question 1 (RQ1)

How do practitioners understand DDD-based microservice API design?

There was a lack of studies on the interrelation of microservice APIs and DDD related to coupling smells. We have addressed this challenge by conducting an empirical research to make sure how we should understand this context. We have investigated 32 grey literature sources to figure out the ADDs related to the DDD-based Microservice design. Based on Grounded Theory, we identified six ADDs with 27 decision options, numerous relations between them, and 27 decision drivers. Our results can help scientists to get a better understanding of practitioner concerns, and practitioners to get an overview of the current view of other practitioners on the interrelation of microservice APIs and DDD. Moreover, the design guidance by our ADD model also can help to reduce design efforts and risks.

RQ1.1 What are the possible ADDs and corresponding decision options in DDD-based Microservice API design?

Based on Grounded Theory, we have identified six ADDs concerning the mapping of domain models to APIs (MMD), defining API contracts in relation to domain models (API-DD), designing API resources based on domain model elements (API-ED), segregation of API resources (RSD), mapping domain model links to the API (LMD), and designing the operations of an API resource (RSD). These ADDs answer **RQ1.1**.

RQ1.2 What are forces (decision drivers) relevant in those design decisions?

To answer **RQ1.2**, we identified 27 decision drivers (forces) considered by practitioners in those decisions, which can be broadly categorized into forces on *loosely coupled relation of API/its clients and the Domain Model*, *maintainability aspects of the API such as complexity and understandability*, *consistency of data*, *runtime properties*, the *relation of API client and server (API provider)*, and *costs and effort*.

RQ1.3 Which relations do the decisions and decision options have?

To answer **RQ1.3**, we identified numerous decision-option relations and other relations. Those usually require a deliberate, incremental human design effort (see Section 2.5).

Research Question 2 (RQ2)

How do practitioners understand coupling smells?

When we realized the different aspects among the practitioners on coupling smells, we have decided to clarify this knowledge onward. We have investigated 48 grey literature sources and elicited the practitioners' perspective on the coupling smell. Based on this knowledge, we identified gaps in the understanding of coupling smells between science and practice. Finally, we derived opportunities and challenges for future scientific work.

We discovered defining factors of coupling smells and relevant impacts on and trade-offs to be made with regard to code and design quality in a Grounded Theory study of grey literature sources containing practitioner views on coupling smells. As coupling smells commonly discussed by practitioners, we identified *Feature Envy*, *Inappropriate Intimacy*, *Data Class*, *Indecent Exposure*, *Message Chain*, and *Middle Man*. We attempted to look into the relationships between those coupling smells, other related smells, software code and design concepts, and options for fixing coupling smells. These coupling smells answer **RQ2.1**.

RQ2.1 How are coupling smells understood by practitioners?

- **RQ2.1.1** What are the defining factors of coupling smells as perceived by practitioners?
- **RQ2.1.2** What are the impacts on and trade-offs to be made with regard to code and design quality when considering coupling smells as perceived by practitioners?
- **RQ2.1.3** What are the relations and interactions among coupling smells, to other related smells, and to other software code and design concepts as perceived by practitioners?
- **RQ2.1.4** What are the options for fixing coupling smells as perceived by practitioners?

RQ2.2 What are gaps in the understanding of coupling smells between the practitioner's view and the scientific literature?

To answer **RQ2.2**, we found a number of evidences which indicated a link between coupling smells and related software engineering principles to DDD. In essence, issues in DDD models can lead to coupling smells, and vice versa. This is interesting in the API context, as those smells and issues tend to become worse when they occur in a distributed setting. In distributed setting, it also needs to be considered that smells resemble established distribution patterns such as DATA TRANSFER OBJECT (DTO) [42], which can lead, if not carefully analyzed, to detrimental refactorings.

Currently, there appears to be a low adoption of scientific findings regarding the coupling smells by practitioners. It is necessary to adapt future research to practitioner needs. To allow practitioners to interact with scientific findings, scientific research should employ accuracy measures and oracles that account for the semantically dense definitions and understandings of coupling smells found in practitioner literature.

RQ2.3 What are opportunities and challenges for future scientific work to address the practitioner concerns on coupling smells well?

In particular to answer **RQ2.3**, we have derived eight lessons that represent opportunities and challenges for future research. From those we can conclude that the definitions of coupling smells used in approaches and tools need to be broadened to the various practitioner concerns identified in Chapter 3. Coupling smells detection tools and approaches could aim to better include human design expertise in the detection and fixing decisions on coupling smells. Coupling smell fixing often could be understood as a complex multi-

criteria optimization or decision problem, not simply applying one or a few refactorings. The interaction of coupling smells and domain modeling could be studied more intensely. In general, the information related to the context, the domain, the size, and the design of the analyzed system is usually not considered. These insights should be incorporated into scientific precision measures and oracles so that practitioners can more easily relate to scientific findings.

Research Question 3 (RQ3)

What are recurring patterns in the ADDs and relations to coupling smells?

To address **RQ3**, we have described patterns from data sets we have created in our prior research, and linked to three existing patterns (API CONTRACT, API DESCRIPTION, and FACADE). These recurring patterns are divided into two groups according to their proximity. 1) patterns on deriving APIs and endpoints from domain models (endpoint-level) and 2) patterns on designing API endpoint operations (operation endpoint-level). In particular, we have mined patterns and their relations on how to formally describe APIs, how to derive APIs from DOMAIN MODELS, how to derive API endpoints from domain model elements, how to derive AGGREGATED DOMAIN OPERATIONS ON API ENDPOINTS, CRUD-BASED API OPERATION endpoints, and API ENDPOINTS BASED ON DOMAIN EVENTS.

At first, we presented the aggregate roots as api endpoints, domain services as api endpoints, and domain processes as api endpoints patterns. For deriving APIs and their endpoints from domain models, we have these three patterns that have been mentioned before and we also described it along with 4 pattern variants and two regularly occurring alternative practices.

Then, for designing API endpoint operations, we present three complementary patterns to derive API operations from the operations of Domain Services and Entities as well as Domain Events, namely Aggregated Domain Operation on API Endpoint, Event-Based API Endpoint Operation, and CRUD-Based API Operation. We also discussed variants of these patterns, such as their combination with the patterns Command Query Responsibility Segregation (CQRS) and Publish/Subscribe.

The models served as cases used for confirming the patterns in the context of systems described, implemented, or modeled by practitioners. The study on recurring patterns in the ADDs and relations to coupling smells is challenging. We have added the examples and usages of the particular topic which comprehend the current studies. Moreover, the multi-case study approach is used in evaluation process.

Research Question 4 (RQ4)

How can we automatically assess conformance to ADDs?

For ADDs' model elaboration and evaluation, we have introduced an automated assessment approach for conformance to ADDs based on the data from an empirical study on this subject. Using the data set from an empirical study on those ADDs, we created an empirically grounded scoring scheme. Based on this, we developed novel automated tools (detectors, parsers, assessors) to identify each of the decision points in our scoring scheme. In a multiple-case evaluation, we compared the manually generated ground truth to the outcomes of the tools.

Firstly, our detectors delivered for each of the diverging cases regarding precise results where the problem was located and how to fix it. Thus, in summary, our detectors always yielded clear results for human architects to either assess the models correctly, or inspect and potentially fix modeling issues. That is, a single manual inspection and correction could lead to correct detection in future runs, e.g. in a continuous delivery pipeline context. To confirm the results, we performed a statistical evaluation which showed statistically insignificant difference between the two variables (ground truth and automated detector results), as well as a negligible effect size between the two variables.

Secondly, the parsers and assessors, we selected one of the architectural design decisions in the context of the interrelation between API and domain models to demonstrate our approach. It is strongly associated with API endpoints. An API description contains mainly information on API endpoints. For this reason, we investigated how far it is possible to automatically assess conformance to LMD solely based on OpenAPI code, which has not been pursued before.

In general, to establish ADD conformance manually requires high and continuous effort. Our tools can spot the violations automatically, support the human-in-the-loop, and support the automated assessment. Our results indicate that such an automated approach is needed for architectural conformance checking, especially when a frequent release schedule is used, as often repeated manual checking is rather infeasible. A modeling framework and traceability support, as provided in our approach, can help to adapt or calibrate the approach to a given system setting and set of best practices, and create or improve tools in a straightforward manner with low effort.

RQ4.1 For which decision options in microservice API design decisions can we provide automated support for assessing conformance to the Microservice API's Domain Model?
RQ4.3 How can we automatically assess conformance to ADD options used in ADDs on deriving API endpoint operations, links, and resource segregation from Domain Models?

To answer **RQ4.1** and **RQ4.3**, we introduced an automated assessment approach for conformance to ADDs. Meanwhile RQ4.1 focuses on the mapping of domain model elements to APIs and API endpoints, RQ4.3 concerns on API endpoint design based on DDD domain models, specifically focusing on operations, links, and resource segregation. Using the data set from an empirical study on those ADDs, we created an empirically grounded scoring scheme. Based on this, we developed novel automated detectors to identify each of the decision points in our scoring scheme. We evaluated this approach in a multi-case study in which we compared the manually created ground truth to the detector results.

RQ4.2 To which extent can we assess conformance of API design decisions to the Microservice API's Domain Model?

To respond to the **RQ4.2**, it is necessary to consider the high-level design (the mapping of domain model elements to APIs and API endpoint). In the case studies we were able to identify 86% of the decision points from our scoring scheme correctly. The remaining 14% were mainly cases where redundant modeling at the overlap of two of the ADDs was necessary, but was not correctly performed in the case study models. In two cases, for one of many APIs (a helper API) the mapping to the domain model was omitted. As shown in Section 5.6 with a simple extension of our detectors to cover the redundant cases, we could even reach 95% correct identification.

RQ4.4 How well do such automated measures for assessing ADD conformance perform?

This measurement belongs to the low level design (API endpoint designs derived from domain models). In the cases of the multi-case study we were able to identify 83% of the decision points from our scoring scheme correctly (to answer **RQ4.4**). The remaining approximately 17% are those cases where more detailed modeling of aspects beyond architecture-level models, e.g. with a simple additional flag, could have solved the issue. That is, a single manual inspection and correction could lead to correct detection in future runs, e.g. in a continuous delivery pipeline context.

RQ4.5 How to provide fully automated conformance assessment of API ADDs?

To address **RQ4.5**, we developed the OpenAPI parser and assessor tools to support the following manual tasks: parsing OpenAPI descriptions, identifying the decision options, and assessing conformance.

RQ4.6

a) To which extent can OpenAPI-based parsing of information on APIs provide sufficient information for design option identification in API ADDs? **b)** Which level of accuracy can be expected in fully automated conformance assessment of API ADDs?

To address **RQ4.6**, we conducted empirical validations. For RQ4.6a, our accuracy scores towards the identification of decision options are as follows: Precision of 94.79 %, recall of 100 %, and F1-measure of 97.22 %. For RQ4.6b, we compared our automated approach's outcomes to the ground truth. We discovered that our results were on matching about 95.12 % correctly. This indicates that full automation of conformance assessment is indeed possible.

7.2 Empirical Contributions

To provide an audit trail of the research and enable replicability of the study, we provide open access to our data set and source code. We provide derived coded models in Python, and generated models (in UML, Markdown, and Latex) as a replication package for download in the Zenodo long-term open access archive as follows.

- **Coupling Smells**
  Singjai, Apitchaka, Simhandl, Georg, & Zdun, Uwe. (2021). On the Practitioners' Understanding of Coupling Smells – A Grey Literature Based Grounded-Theory Study: Dataset and Code [Data set]. Zenodo. https://doi.org/10.5281/zenodo.4476077
- **Interrelation between API and DDD**
  Singjai, Apitchaka, Zdun, Uwe, & Zimmermann, Olaf. (2021). Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design – A Grey Literature Study Based on Grounded Theory: Dataset and Code [Data set]. 18TH IEEE INTERNATIONAL CONFERENCE ON SOFTWARE ARCHITECTURE (ICSA 2021), STUTTGART, GERMANY. Zenodo. https://doi.org/10.5281/zenodo.4569578
- **Conformance Assessment**
  - Singjai, Apitchaka, & Zdun, Uwe. (2021). Conformance Assessment of Architectural Design Decisions on the Mapping of Domain Model Elements to APIs and API Endpoints: Dataset and Code [Data set]. Zenodo. https://doi.org/10.5281/zenodo.5176174
  - Singjai, Apitchaka, & Zdun, Uwe. (2021). Conformance Assessment of Architectural Design Decisions on API Endpoint Designs Derived from Domain Models: Dataset and Code [Data set]. Zenodo. https://doi.org/10.5281/zenodo.5031272
  - Singjai, Apitchaka, & Zdun, Uwe. (2022). API Description-Based Conformance Assessment of Architectural Design Decision: Dataset and Code [Data set]. Zenodo. https://doi.org/10.5281/zenodo.6564304

7.3 Future Works

This section introduces possible future works based on the revisited research questions and the existing limitations in our studies.

Practitioners' Perspective: Based on the conclusion of the practitioners' point of view on coupling smells and the interrelation between API and DDD, practitioners should consider other practitioners' perspectives. Likewise, the researchers should study design patterns, principles, and best practices more deeply. Therefore, future works could be conducting the interview and gathering the experts' opinions in these fields. To better understand the implication of these results, future studies could address how the experts think about the practitioners who share their knowledge online.

Based on our findings, as future work, we plan to examine more closely some of the lessons learned in our study, especially more specialized and domain-driven approaches. We also plan to confirm some of the lessons learned in other empirical studies. It would be interesting to perform further quantitative studies specifically focused on the relations of smells to practices, liabilities, fixes, and so on. We also plan to use our findings to provide automated design advice to API designers and improve tools that generate API code. Further research on anti-patterns at the overlap of DDD, APIs and microservices might help in providing automated design advice. We plan to perform research on further ADDs model elaboration and validation. Further research on related practices such as event storming and event sourcing might be interesting as well. Many important API concepts are not covered in this work, as they are not directly related to DDD, such as MIME type selection, remoting errors, reporting, security aspects, and so on. Further studies on more general API-related design is needed to include those.

Patterns: While the found recurring patterns have been evaluated with the case studies, further research is needed to determine the effects of applying those patterns in a new project from scratch. Moreover, the relationship between the case studies themselves is also interesting to investigate. As future work, we plan to mine additional patterns in this context and to study metrics for detecting our patterns in existing models.

Modeling-based Conformance Assessment: In general, manually establishing ADD conformance requires high and continuous effort. Our detectors can spot the violations automatically, support the human-in-the-loop, and support the automated assessment. Therefore, further work on automated assessments may help development teams to apply our approach in the context of continuous delivery. Even though the modeling allows us to access the different artifact types, an automatic assessment is not an easy task for real-world systems yet, and more work might be needed to enable this. In our future work, we plan to work on other ADDs for API design, as well as related design patterns. We aim to apply our approach as part of CI/CD pipelines which perform a series of related architectural conformance assessments.

API Description-based Conformance Assessment: In our future work, we plan to support other ADDs related to the interrelation between API and DDD. Our approach is potentially applicable to a variety of other model-based techniques, too. In addition, it is possible to implement our approach to other fields of interest, such as applying it to the machine-readable document format in the machine learning field or in the security and privacy field.

Appendix A

Instruction for the Repository of the interrelation of DDD and APIs

TITLE: A Reproduction Guideline for the Paper "Practitioner Views on the Interrelation of Microservice APIs and Domain-Driven Design: A Grey Literature Study Based on Grounded Theory"

WHO: Apitchaka Singjai* (apitchaka.singjai@univie.ac.at),
Uwe Zdun (uwe.zdun@univie.ac.at), Olaf Zimmermann (olaf.zimmermann@ost.ch)
Paper on Zenodo: https://doi.org/10.5281/zenodo.4493865
Repository on Zenodo: https://doi.org/10.5281/zenodo.4569578

WHAT: For coding processes in the Grounded Theory, we applied text-based coding only initially and then applied UML-based modeling instead (encoded in Python) to develop a precise and consistent theory.

WHY: The UML-based modeling could precisely encode findings in the Grounded Theory. Concerning the repeatability aspect, it benefits the iterative and incremental constant comparison. It also supports the automated generation of the results. This automation could enhance the traceability in terms of references and assure the variety of results' format.

HOW: After initial text-based open coding, we used precised UML-based modelling for axial and selective coding, instead of the often-used text-based coding process, in order to develop a precisely defined and consistent theory. We used the CodeableModel tool for this. Formal modelling also eased establishing an audit trail of the research, and thus enable repeatability of the study.

WHERE: The grey literature study is applied as the data collection technique for this Grounded Theory. When the new sources are added, the coding-loop is triggered. It also

means the coding process is reproduced. Because Grounded Theory does not completely collect the data before it starts the data analysis, but the data collection and analysis are performed in each iterative research step.

DISCUSSION: As Grounded Theory is mainly concerned with phenomena that have specifically been observed to exist, the coding process with the assist of UML-based modelling could comprehend the phenomenon. Other researchers might have coded or modelled differently. As our goal was only to find one model that is able to specify all observed phenomena, and this was achieved, we consider this threat not to be a major issue for our study. At any rate, our results are only valid in our set scope.

IMPORTANT: Please note that the code in the repository only regenerates the PlantUML models and the Latex tables used in this work. Mainly, this artifact contains the data on the open and axial coding, as well as the formal coding in Python. This repository / data set contains the following elements:

- The open coding data for each of the sources can be found a separate .md file in the folder:‘field_notes_open_coding_axial_coding‘
- In each file we also documented the axial coding steps performed in the coded models, to ensure traceability between open coding and formal coding in Python
- All classes and instances are in the ‘api_ddd_models‘ folder
  - The file ‘api_ddd_models.py‘ contains all models of ADDs
  - The file ‘sources_and_codes_models.py‘ contains all models of evidences
- The ‘generators‘ folder contains generators for Plant UML models, markdown files, and Latex tables. These are generated into ‘_generated’. This markdown file renders almost the complete model
- The file ‘GT_coding.py‘ in the ‘metamodels‘ folder shows the meta-class of grounded theory coding
- The ‘map_models‘ folder is a dependency model

A.1 Getting Started

In order to generate the files, the following artifacts must be downloaded and installed.

- **Codeable Models** `https://github.com/uzdun/CodeableModels/`
- **PlantUML** `http://plantuml.com/download`
- **Graphviz** `https://graphviz.org/download/`

Take a look at: generators/generate_all.py which generates all figures for the models and .md/latex files. Run it in the ‘generators‘ directory using:

```
python .\generate_all.py
```

A.2 Prerequisites

PYTHONPATH must be correctly set:

- The directory containing ‘codeableModels‘ and ‘plantUMLRenderer‘ must be on the PYTHONPATH.
- The directory containing this project must be on the PYTHONPATH.

See plant_uml_renderer directory in Codeable Models for instructions how to set up the plant UML jar. Without further configuration it is assumed to be in the default directory:

```
self.directory = "../_generated"
self.plant_uml_jar_path = "../../libs/plantuml.jar"
```

A.3 Remarks

To double-check we tested on a couple of other machines. In the following configurations our instructions worked without problems for us.

1. Installing prerequisite
    - o https://plantuml.com/starting
    - o https://plantuml.com/graphviz-dot
2. Setting PYTHONPATH
    - o ../DDDAPIModelsArtifact/CodeableModels
    - o ../DDDAPIModelsArtifact/python/ddd_api_codeablemodels
3. Running the script
    - o ../DDDAPIModelsArtifact/python/ddd_api_codeablemodels generators
4. Getting the results
    - o ../DDDAPIModelsArtifact/python/ddd_api_codeablemodels_generated

If everything is alright, you will see four subdirectories in the ‘_generated‘ directory.

- ddd_api
- domain_model
- latex_files
- Textual_model_rendering

Please note that, this artifact have been test with Windows10 + python3.6 or more, Ubuntu18.04.3 + python3.6, macOS Big sur + python3.6

A.4 Reproduction

Below you find a clear step-by-step description how we have achieved the following data collection and coding steps.

1. How to analyze the source in field notes, here using the example of: s1.md

    a) Fill in the attributes for the source (mandatory) and the reference (if it exists), e.g.:

```
# s
1
## url
https://nhpatt.com/bounded-context-in-apis/
## tiny url
https://tinyurl.com/api-ddd-s1
## archive url
https://bit.ly/2BsvnjK
## title
Bounded Context in APIs (1/2)
## source code
no
## example
no
## source type
Practitioner Audience Article
## author type
Practitioner
**AXIAL CODING TRACE:**
``` python
s1 = CClass(source, "s1", values={
  "title": "Bounded Context in APIs (1/2)",
  "url": "https://nhpatt.com/bounded-context-in-apis/",
  "archive url": "https://bit.ly/2BsvnjK",
  "author type": "Practitioner",
  "type": "Practitioner Audience Article",
  "example": False,
  "source code": False})
```
```

```
    ddd_vernon_book_2013 = CClass(reference,
        "vernon2013implementing", values={
      "bibliographic reference": "Vernon, Vaughn: Implementing
          domain-driven design}" + "Addison-Wesley Professional,
          2013",
      "author type": "Practitioner",
      "type": "Practitioner Book"})
  add_links({s_example: ddd_vernon_book_2013},
      role_name="referenced")
'''
```

2. An example of coding from the example file: s1.md. Of course, other structures for recording the coding process can be chosen in a replication.

Lines from knowledge sources:
This is an open question: Does it makes sense to expose the bounded contexts (from DDD) in your REST APIs?
Open Coding:
- Option: Expose Bounded Context in API - this is not REST specific, would be the same question in grpc API, so removed REST here. - This is about how to map the domain model to the API; the context of the decision is the domain model.
Axial Coding:

`expose_bounded_context_in_API = CClass(practice, "Expose Bounded Context in API")`
`domain_model_mapping_decision = CClass(decision, "How to map a domain model and its elements to an API?")`
`add_links({domain_model_mapping_decision: domain_model_and_api}, role_name="context",`
`stereotype_instances=decide_for_some_instances_of)`

Selective Coding:

`expose_each_bounded_context_as_an_API = CClass(practice,`
`"Expose Each Bounded Context as an API",`
`superclasses=expose_bounded_contexts_as_APIs)`
`expose_selected_bounded_context_as_an_API = CClass(practice,`
`"Expose Selected Bounded Contexts as APIs",`
`superclasses=expose_bounded_contexts_as_APIs)`

3. How to add the evidences into the model coded in Python. Of course, for replication any other modelling tool or informal coding only can be used, too.

```
add_links({s30: [ api_as_contract_decision, api_contract_specified,
  api_contract_specified_first, api_code_first, api_stability,
     api_modifiability,separation_of_api_contract_and_domain_concerns,
  initial_effort_required, design_and_implementation_effort,
  maintainability_of_api_and_consumers, api_understandability
  ]}, role_name="contained_code")
```

4. How to generate the figures of ADDs (To reproduce the generation of figures simply run the whole file as a Python script).

```
python .\generate_all.py
```

```
from plant_uml_renderer import PlantUMLGenerator
from api_ddd_models.api_ddd_model import ddd_api_views

# UMLgenerator
generator = PlantUMLGenerator(delete_gen_dir_during_init=True)
generator.object_model_renderer.name_break_length = 45
generator.object_model_renderer.left_to_right = True

object_models = {'ddd_api': ddd_api_views}
for key, value in object_models.items():
generator.generate_object_models(key, value)
```

5. How to generate the tables (To reproduce the generation of figures simply run the whole file as a Python script).

```
python .\generate_all.py
```

a) knowledge source table : generate_sources_table()

b) study result table : generate_overview_table()

```
from generators.generate_latex_files import LatexTableRenderer
from metamodels.guidance_metamodel import decision

# Latex
source_table_renderer = LatexTableRenderer()
source_table_renderer.generate(decisions)
```

Appendix B

Google Scholar Results from 2003-2021

These results aim to provide the overview of three important terms which are APPLICATION PROGRAMMING INTERFACE (API), DOMAIN DRIVEN DESIGN (DDD), and MICROSERVICES OR MICROSERVICE (MS) between year 2003 to the 2021. In 2003, Eric Evans introduced his book called "Domain-Driven Design: Tacking Complexity In the Heart of Software" [33], which we applied this year as a starting point of our literature review investigation. The figure B.1 illustrates the venn diagram to show the original idea where the searching algorithm based on. The Table B.1 presents the 7 possibilities of the relations among those three terms. Regarding the terms, we avoided using abbreviation because they can mislead to the unrelated terms.

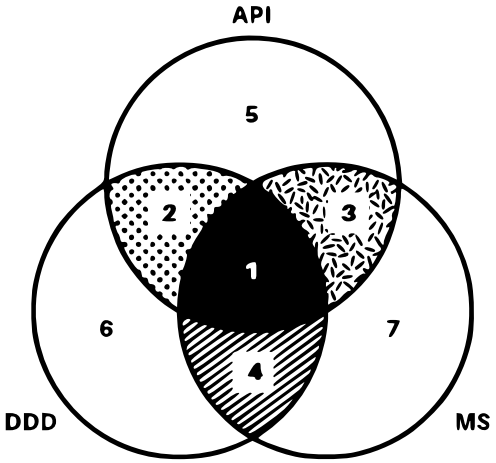

Figure B.1: The unweighted Venn diagram

For more detail, the Table B.2 shows the results of each possibilities in each year and the total from 2003 to 2021. The overall trend is up every year. The results clearly show that the interrelation between API, DDD, and Microservice started in 2015. In addition, the Figure B.2 illustrates the weighted venn diagram perspective of those terms.

Table B.1: The keyword algorithm to explore Google Scholar

No.	Equation	Searching Query Keywords
1	API ∩ DDD ∩ MS	"application programming interface" AND "domain driven design" AND (microservices OR microservice)
2	(API ∩ DDD) - MS	("application programming interface" AND "domain driven design") AND -microservices -microservice
3	(API ∩ MS) - DDD	("application programming interface" AND (microservices OR microservice)) AND –“domain driven design”
4	(DDD ∩ MS) - API	("domain driven design" AND (microservices OR microservice)) AND –“application programming interface”
5	API – DDD – MS	"application programming interface" AND -"domain driven design" -microservices -microservice
6	DDD – API – MS	"domain driven design" AND -"application programming interface" -microservices -microservice
7	MS – API – DDD	(microservices OR microservice) AND -"application programming interface" -"domain driven design"

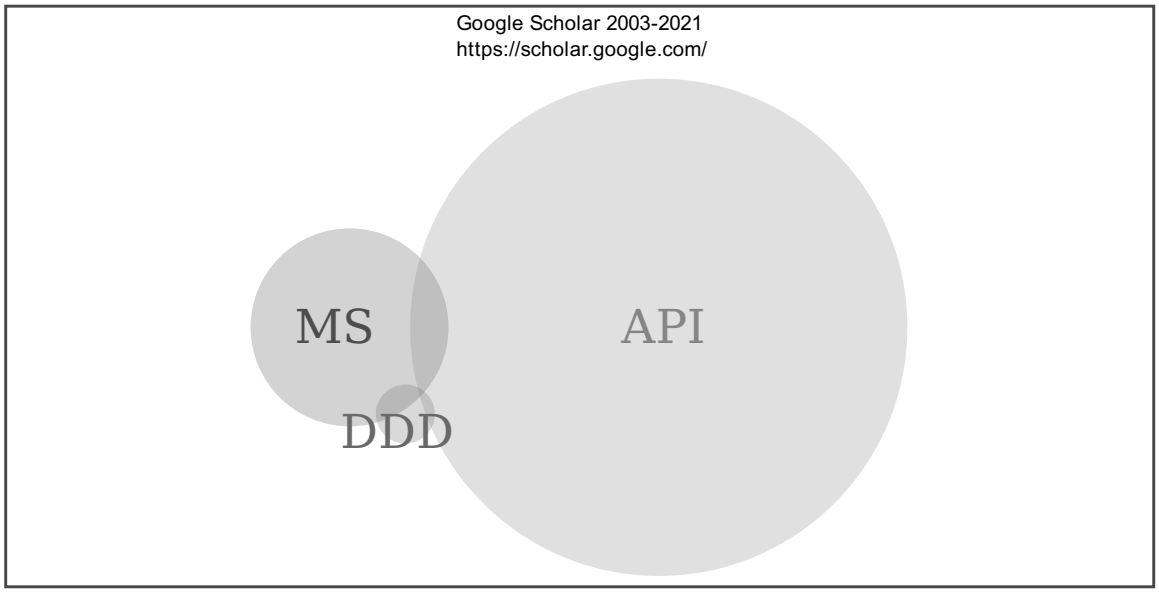

Figure B.2: Graphical Representation on exploring Google Scholar

Table B.2: The result from Google Scholar between 2003 - 2021

year	**1**	**2**	**3**	**4**	**5**	**6**	**7**
2003	0	0	0	0	2920	11	24
2004	0	0	7	0	3340	19	33
2005	0	2	0	0	3570	47	55
2006	0	3	3	0	3930	63	54
2007	0	5	1	0	4670	79	76
2008	0	10	3	0	5180	86	80
2009	0	11	7	1	6130	128	98
2010	1	13	9	0	7230	158	117
2011	0	19	6	0	7930	126	109
2012	0	5	4	0	8980	138	122
2013	0	11	10	0	9930	137	128
2014	0	18	11	3	10800	152	249
2015	2	13	42	18	11800	200	649
2016	10	28	124	57	12400	206	1460
2017	11	21	267	110	13500	175	2440
2018	28	17	459	145	15000	196	3840
2019	40	21	671	177	16400	183	5220
2020	41	24	784	196	16700	178	6230
2021	46	30	920	228	17200	219	7150
Total	**180**	**253**	**3331**	**939**	**177615**	**2507**	**28141**

List of Figures

List of Tables

Bibliography

[1] *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[2] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16(4):487–513, 2011.

[3] F. Arcelli Fontana, V. Ferme, and M. Zanoni. Filtering code smells detection results. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 803–804, 2015.

[4] Francesca Arcelli Fontana, Mika V. Mäntylä, Marco Zanoni, and Alessandro Marino. Comparing and experimenting machine learning techniques for code smell detection. *Empirical Softw. Engg.*, 21(3):1143–1191, June 2016.

[5] Michael Athanasopoulos, Kostas Kontogiannis, and Chris Brealey. Towards an interpretation framework for assessing interface uniformity in rest. In *Proceedings of the Second International Workshop on RESTful Design*, pages 47–50, 2011.

[6] Hamdy Michael Ayas, Philipp Leitner, and Regina Hebig. Facing the giant: a grounded theory study of decision-making in microservices migrations. *arXiv preprint arXiv:2104.00390*, 2021.

[7] Soon K Bang, Sam Chung, Young Choh, and Marc Dupuis. A grounded theory analysis of modern web applications: knowledge, skills, and abilities for devops. In *Proceedings of the 2nd annual conference on Research in information technology*, pages 61–62, 2013.

[8] Victor R. Basili and David M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, SE-10(6):728–738, 1984.

[9] Lujo Bauer, Cristian Bravo-Lillo, Lorrie Cranor, and Elli Fragkaki. Warning design guidelines. Technical report, Carnegie Mellon University, Pittsburgh, PA, 2013.

[10] Joshua Bloch. How to design a good api and why it matters. In *Proc. 21st ACM SIGPLAN Conference (OOPSLA)*, pages 506–507, Portland, Oregon, 2006. ACM New York, NY, USA.

[11] Eric Bouwers, Joost Visser, Carola Lilienthal, and Arie van Deursen. A cognitive model for software architecture complexity. In *2010 IEEE 18th International Conference on Program Comprehension*, pages 152–155, 2010.

[12] Hongyu Pei Breivold and Ivica Crnkovic. Analysis of software evolvability in quality models. In *2009 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 279–282. IEEE, 2009.

[13] Gleison Brito, Andre Hora, Marco Tulio Valente, and Romain Robbes. On the use of replacement messages in api deprecation: An empirical study. *Journal of Systems and Software*, 137:306–321, 2018.

[14] Antonio Brogi, Davide Neri, Jacopo Soldani, and Olaf Zimmermann. Design principles, architectural smells and refactorings for microservices: A multivocal review. *CoRR*, abs/1906.01553, 2019.

[15] Kyle Brown, Cees De Groot, and Chris Hay. Cloud adoption patterns: A set of patterns for developers and architects building for the cloud. `https://kgb1001001.github.io/cloudadoptionpatterns/Cloud-Native-Architecture/`, 2019.

[16] Reinhard Budde, Karlheinz Kautz, Karin Kuhlenkamp, and Heinz Züllighoven. What is prototyping? *Information Technology & People*, 1992.

[17] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture - Volume 1: A System of Patterns.* Wiley Publishing, 1996.

[18] Gerardo Canfora, Luigi Cerulo, and Luigi Troiano. Transforming quantities into qualities in assessment of software systems. In *Proceedings 27th Annual International Computer Software and Applications Conference. COMPAC 2003*, pages 312–319. IEEE, 2003.

[19] Kathy Charmaz. *Constructing grounded theory.* Sage, 2014.

[20] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of bad smells in object-oriented code. In *2010 Seventh International Conference on the Quality of Information and Communications Technology*, pages 106–115, 2010.

[21] Norman Cliff. *Ordinal methods for behavioral data analysis.* Psychology Press, 2014.

[22] Alistair Cockburn. *Agile software development: the cooperative game.* Pearson Education, 2006.

[23] J. Coplien. *Software Patterns: Management Briefings*. SIGS, New York, 1996.

[24] Juliet Corbin and Anselm L. Strauss. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13:3–20, 1990.

[25] John W Creswell, VL Plano Clark, Michelle L Gutmann, and William E Hanson. An expanded typology for classifying mixed methods research into designs. *A. Tashakkori y C. Teddlie, Handbook of mixed methods in social and behavioral research*, pages 209–240, 2003.

[26] G. d. F. Carneiro, M. Silva, L. Mara, E. Figueiredo, C. Sant'Anna, A. Garcia, and M. Mendonça. Identifying code smells with multiple concern views. In *2010 Brazilian Symposium on Software Engineering*, pages 128–137, 2010.

[27] Robert Daigneau. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Addison-Wesley, 2011.

[28] F. Deissenboeck, L. Heinemann, B. Hummel, and E. Juergens. Flexible architecture conformance assessment with conqat. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, volume 2, pages 247–250, 2010.

[29] Atish Kumar Dipongkor, Iftekhar Ahmed, and Nadia Nahar. Move method recommendation using call frequency of methods and attributes. In *2018 Joint 7th International Conference on Informatics, Electronics & Vision (ICIEV) and 2018 2nd International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 76–81. IEEE, 2018.

[30] Ivan Dugalic. A pattern language for microservices. `https://dzone.com/articles/bounded-contexts-with-axon`, 2019.

[31] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008.

[32] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in api design: A usability evaluation. In *29th International Conference on Software Engineering (ICSE'07)*, pages 302–312, Washington, DC, USA, 2007. IEEE.

[33] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley, Reading, MA., 2003.

[34] Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[35] Eric Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. Dog Ear Publishing, 2014.

[36] Eric Evans. Ddd and microservices: At last, some boundaries! `https://www.youtube.com/watch?v=sFCgXH7DwxM`, 2016.

[37] Chen-Yuan Fan and Shang-Pin Ma. Migrating monolithic mobile application to microservice architecture: An experiment report. In *2017 IEEE International Conference on AI & Mobile Services (AIMS)*, pages 109–112. IEEE, 2017.

[38] A. M. Fard and A. Mesbah. Jsnose: Detecting javascript code smells. In *2013 IEEE 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125, 2013.

[39] Roy T Fielding. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Irvine, 2000.

[40] Marios Fokaefs, Nikolaos Tsantalis, and Alexander Chatzigeorgiou. Jdeodorant: Identification and removal of feature envy bad smells. In *ICSM*, pages 519–520. IEEE Computer Society, 2007.

[41] F. A. Fontana, V. Ferme, and M. Zanoni. Towards assessing software architecture quality by exploiting code smell relations. In *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*, pages 1–7, 2015.

[42] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, USA, 2002.

[43] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., USA, 2002.

[44] Martin Fowler. Microservice trade-offs. `https://martinfowler.com/articles/microservice-trade-offs.html`, 2015.

[45] Martin Fowler. What do you mean by "event-driven"? `https://martinfowler.com/articles/201701-event-driven.html`, 2017.

[46] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[47] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *Proc. of the 13th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 255–258, 2009.

[48] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In *Proceedings of the 20th international conference on evaluation and assessment in software engineering*, pages 1–6, 2016.

[49] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Inf. Softw. Technol.*, 106:101–121, 2019.

[50] Vahid Garousi, Michael Felderer, Mika V. Mäntylä, and Austen Rainer. Benefitting from the grey literature in software engineering research. *CoRR*, 2019.

[51] Barney G. Glaser and Anselm L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. de Gruyter, New York, NY, 1967.

[52] Google. Devops tech: Architecture. `https://cloud.google.com/architecture/devops/devops-tech-architecture`, 2021.

[53] Peter Leo Gorski, Yasemin Acar, Luigi Lo Iacono, and Sascha Fahl. Listen to developers! a participatory design study on security warnings for cryptographic apis. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, pages 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[54] Tomasz Górski and Ewa Wojtach. Use case api-design pattern for shared data. In *2018 26th International Conference on Systems Engineering (ICSEng)*, pages 1–8, Washington, DC, USA, 2018. IEEE.

[55] Yuepu Guo, Carolyn Seaman, Nico Zazworka, and Forrest Shull. Domain-specific tailoring of code smells: An empirical study. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, page 167–170, New York, NY, USA, 2010. Association for Computing Machinery.

[56] Johanna Gustafsson. Single case studies vs. multiple case studies: A comparative study, 2017.

[57] C Hagstrom, S Kendall, and H Cunningham. Googling for grey: Using google and duckduckgo to find grey literature. In *23rd Cochrane Colloquium. Cochrane database systematic reviews supplements*, pages 1–327, 2015.

[58] Wilhelm Hasselbring. Software architecture: Past, present, future. In *The Essence of Software Engineering*, pages 169–184. Springer, Cham, 2018.

[59] Michi Henning. Api design matters. *Queue*, 5(4):24–36, 2007.

[60] Carsten Hentrich and Uwe Zdun. *Process-Driven SOA - Patterns for Aligning Business and IT*. Infosys Press. CRC Press, 2012.

[61] Carsten Hentrich, Uwe Zdun, Vlatka Hlupic, and Fefie Dotsika. An approach for pattern mining through grounded theory techniques and its applications to process-driven soa patterns. In *Proceedings of the 18th European Conference on Pattern Languages of Program*, pages 9:1–9:16, 2015.

[62] Carsten Hentrich, Uwe Zdun, Vlatka Hlupic, and Fefie Dotsika. An approach for pattern mining through grounded theory techniques and its applications to process-driven soa patterns. In *Proceedings of the 18th European Conference on Pattern Languages of Program*, EuroPLoP '13, New York, NY, USA, 2015. Association for Computing Machinery.

[63] Gregor Hohpe. Workshop report: Conversation patterns. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil M. P. van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, volume 06291 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[64] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.

[65] André Hora, Romain Robbes, Marco Tulio Valente, Nicolas Anquetil, Anne Etien, and Stéphane Ducasse. How do developers react to api evolution? a large-scale empirical study. *Software Quality Journal*, 26(1):161–191, 2018.

[66] Zahid Hussain, Wolfgang Slany, and Andreas Holzinger. Investigating agile user-centered design in practice: A grounded theory perspective. In *Symposium of the Austrian HCI and Usability Engineering Group*, pages 279–289. Springer, 2009.

[67] Hilary Hutchinson, Wendy Mackay, Bo Westerlund, Benjamin B Bederson, Allison Druin, Catherine Plaisant, Michel Beaudouin-Lafon, Stéphane Conversy, Helen Evans, Heiko Hansen, et al. Technology probes: inspiring design for and with families. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 17–24, 2003.

[68] Bala Iyer and George Wyner. Evaluating apis: a call for design science research. In *Design Science Research in Information Systems. Advances in Theory and Practice*, pages 28–35, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[69] R Burke Johnson and Larry Christensen. *Educational research: Quantitative, qualitative, and mixed approaches*. SAGE Publications, Incorporated, 2019.

[70] R Burke Johnson, Anthony J Onwuegbuzie, and Lisa A Turner. Toward a definition of mixed methods research. *Journal of mixed methods research*, 1(2):112–133, 2007.

[71] Toni Stokes Jones and Rita C Richey. Rapid prototyping methodology in action: A developmental study. *Educational Technology Research and Development*, 48(2):63–80, 2000.

[72] Stefan Kapferer and Olaf Zimmermann. Domain-driven service design - context modeling, model refactoring and contract generation. In *Proc. of the 14th Advanced Summer School on Service-Oriented Computing (SummerSOC'20) (to appear)*, 2020.

[73] Stefan Kapferer and Olaf Zimmermann. Domain-driven service design-context modeling, model refactoring and contract generation. *Proc. of the 14th Advanced Summer School on Service-Oriented Computing (SummerSOC'20)(to appear). Springer CCIS*, 2020.

[74] Stefan Kapferer and Olaf Zimmermann. Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, pages 299–306. Scitepress, 2020.

[75] Staffs Keele et al. Guidelines for performing systematic literature reviews in software engineering. Technical report, Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007.

[76] Meabh Kenny and Robert Fourie. Contrasting classic, straussian, and constructivist grounded theory: Methodological and philosophical conflicts. *Qualitative Report*, 20:1270–1289, 08 2015.

[77] Barbara Kitchenham, Lech Madeyski, David Budgen, Jacky Keung, Pearl Brereton, Stuart Charters, Shirley Gibbs, and Amnart Pohthong. Robust statistical methods for empirical software engineering. *Empirical Software Engineering*, 22(2):579–630, 2017.

[78] Kenneth E Lantz. *The prototyping methodology*. Prentice-Hall, Inc., 1986.

[79] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice - Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[80] OI Larichev and HM Moskovich. Unstructured problems and development of prescriptive decision making methods. In *Advances in multicriteria analysis*, pages 47–80. Springer, 1995.

[81] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall PTR, USA, 2004.

[82] Larix Lee and Philippe Kruchten. A tool to visualize architectural design decisions. In *International Conference on the Quality of Software Architectures*, pages 43–54. Springer, 2008.

[83] Jun Li, Yingfei Xiong, Xuanzhe Liu, and Lu Zhang. How does web service api evolution affect clients? In *2013 IEEE 20th International Conference on Web Services*, pages 300–307, Washington, DC, USA, 2013. IEEE.

[84] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–422, Washington, DC, USA, 2016. IEEE.

[85] Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. Cid: Automating the detection of api-related compatibility issues in android apps. In *27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 153–163. ACM, 2018.

[86] Li Li and Wu Chou. Design patterns for restful communication web services. In *2010 IEEE International Conference on Web Services*, pages 512–519, Washington, DC, USA, 2010. IEEE, IEEE.

[87] Li Li, Wu Chou, Wei Zhou, and Min Luo. Design patterns and extensibility of rest api for networking applications. *IEEE Transactions on Network and Service Management*, 13(1):154–167, 2016.

[88] Hui Liu, Zhifeng Xu, and Yanzhen Zou. Deep learning based feature envy detection. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, page 385–396, New York, NY, USA, 2018. Association for Computing Machinery.

[89] João Ricardo Lourenço, Bruno Cabral, Paulo Carreiro, Marco Vieira, and Jorge Bernardino. Choosing the right nosql database for the job: a quality attribute evaluation. *Journal of Big Data*, 2(1):1–26, 2015.

[90] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, EuroPLop '19. ACM, 2019.

[91] Daniel Lübke, Olaf Zimmermann, Cesare Pautasso, Uwe Zdun, and Mirko Stocker. Interface evolution patterns: balancing compatibility and extensibility across service life cycles. In *Proceedings of the 24th European Conference on Pattern Languages of Programs*, pages 1–24, 2019.

[92] Jinyu Ma, Zheng Li, and Yan Liu. Including pervasive web content in evidence-based software engineering: A case study. In *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pages 55–62. IEEE, 2017.

[93] Amine El Malki and Uwe Zdun. Guiding architectural decision making on service mesh based microservice architectures. In Tomás Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture - 13th European Conference, ECSA 2019, Paris, France, September 9-13, 2019, Proceedings*, volume 11681 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2019.

[94] M. Mantyla, J. Vanhanen, and C. Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, pages 381–384, 2003.

[95] M. V. Mantyla, J. Vanhanen, and C. Lassenius. Bad smells - humans as code critics. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 399–408, 2004.

[96] Mika V Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[97] C. Marinescu. Identification of design roles for the assessment of design quality in enterprise applications. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 169–180, 2006.

[98] Raúl Marticorena, Carlos López, and Yania Crespo. Extending a taxonomy of bad code smells with metrics. In *Proceedings of 7th International Workshop on Object-Oriented Reengineering (WOOR)*, page 6. Citeseer, 2006.

[99] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of api stability and adoption in the android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79, Washington, DC, USA, 2013. IEEE.

[100] Paulo Merson and Joseph Yoder. Modeling microservices with ddd. In *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pages 7–8. IEEE, 2020.

[101] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[102] Lauren Murphy, Mary Beth Kery, Oluwatosin Alliyu, Andrew Macvean, and Brad A Myers. Api designers in the field: Design practices and challenges for creating usable apis. In *2018 ieee symposium on visual languages and human-centric computing (vl/hcc)*, pages 249–258. IEEE, 2018.

[103] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[104] Sam Newman. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly, 2015.

[105] Cao Duc Nguyen. A design analysis of cloud-based microservices architecture at netflix. `https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45fe` 2020.

[106] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Patric Genfer, Sebastian Geiger, Sebastian Meixner, and Wilhelm Hasselbring. Detector-based component model abstraction for microservice-based systems. *Computing*, 103:2521–2551, 2021.

[107] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Assessing architecture conformance to coupling-related patterns and practices in microservices. In *European Conference on Software Architecture*, pages 3–20. Springer, 2020.

[108] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Sebastian Meixner, and Sebastian Geiger. Metrics for assessing architecture conformance to microservice architecture patterns and practices. In *International Conference on Service-Oriented Computing*, pages 580–596. Springer, 2020.

[109] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Daniel Schall, Fei Li, and Sebastian Meixner. Supporting architectural decision making on data management in microservice architectures. In Tomás Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture - 13th European Conference, ECSA 2019, Paris, France, September 9-13, 2019, Proceedings*, volume 11681 of *Lecture Notes in Computer Science*, pages 20–36. Springer, 2019.

[110] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, and A. D. Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 101–110, 2014.

[111] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Mining version histories for detecting code smells. *IEEE Transactions on Software Engineering*, 41(5):462–489, 2015.

[112] F. Palomba, D. Di Nucci, M. Tufano, G. Bavota, R. Oliveto, D. Poshyvanyk, and A. De Lucia. Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 482–485, 2015.

[113] Chris Parnin, Christoph Treude, Lars Grammel, and Margaret-Anne Storey. Crowd documentation: Exploring the coverage and the dynamics of api discussions on stack overflow. Technical report, Georgia Institute of Technology, USA, 2012.

[114] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor Mendonça. Static architecture-conformance checking: An illustrative overview. *IEEE Software*, 27(5):82–89, 2010.

[115] Cesare Pautasso and Erik Wilde. Push-enabling restful business processes. In Gerti Kappel, Zakaria Maamar, and Hamid R. Motahari-Nezhad, editors, *Service-Oriented Computing*, pages 32–46, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[116] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M. Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(2):97–104, 2017.

[117] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai M Josuttis. Microservices in practice, part 2: Service integration and sustainability. *IEEE Software*, 34(2):97–104, 2017.

[118] Patrizio Pelliccione, Paola Inverardi, and Henry Muccini. Charmy: A framework for designing and verifying architectural specifications. *IEEE Transactions on Software Engineering*, 35(3):325–346, 2008.

[119] William D Perreault Jr and Laurence E Leigh. Reliability of nominal data based on qualitative judgments. *Journal of marketing research*, 26(2):135–148, 1989.

[120] Daniel Persson. Swagger reader. `https://github.com/kalaspuffar/swagger-reader`, 2020.

[121] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and software technology*, 64:1–18, 2015.

[122] Fabio Petrillo, Philippe Merle, Naouel Moha, and Yann-Gaël Guéhéneuc. Are rest apis for cloud computing well-designed? an exploratory study. In *International Conference on Service-Oriented Computing*, pages 157–170. Springer, 2016.

[123] Jan Piasecki, Marcin Waligora, and Vilius Dranseika. Google search as an additional source in systematic reviews. *Science and engineering ethics*, 24(2):809–810, 2018.

[124] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. An empirical study of api usability. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 5–14, Washington, DC, USA, 2013. IEEE.

[125] John W Pratt. Remarks on zeros and ties in the wilcoxon signed rank procedures. *Journal of the American Statistical Association*, 54(287):655–667, 1959.

[126] D. Quartel and M. van Sinderen. On interoperability and conformance assessment in service composition. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 229–229, 2007.

[127] Dick Quartel and Marten van Sinderen. On interoperability and conformance assessment in service composition. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 229–229. IEEE, 2007.

[128] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. Towards a uml profile for domain-driven design of microservice architectures. In *International Conference on Software Engineering and Formal Methods*, pages 230–245. Springer, 2017.

[129] Florian Rademacher, Jonas Sorgalla, and Sabine Sachweh. Challenges of domain-driven microservice design: a model-driven perspective. *IEEE Software*, 35(3):36–43, 2018.

[130] Md Masudur Rahman, Rashed Rubby Riyadh, and Md Rayhanur Rahman. Recommendation of move method refactorings using coupling, cohesion and contextual similarity. In *2017 IEEE International Conference on Imaging, Vision & Pattern Recognition (icIVPR)*, pages 1–6. IEEE, 2017.

[131] Austen Rainer and Ashley Williams. Using blog articles in software engineering research: benefits, challenges and case–survey method. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, pages 201–209. IEEE, 2018.

[132] Austen Rainer and Ashley Williams. Using blog-like documents to investigate software practice: benefits, challenges and research directions. *Journal of Software: Evolution and Process*, 8 2019.

[133] Daniel Ratiu, Stéphane Ducasse, Tudor Gîrba, and Radu Marinescu. Using history information to improve design flaws detection. In *Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering (CSMR'04)*, CSMR '04, page 223, USA, 2004. IEEE Computer Society.

[134] Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Jane Cleland-Huang. Mind the gap: assessing the conformance of software traceability to relevant guidelines. In *Proceedings of the 36th International Conference on Software Engineering*, pages 943–954, 2014.

[135] Chris Richardson. Pattern: Command query responsibility segregation (cqrs). `https://microservices.io/patterns/data/cqrs.html`, 2017.

[136] Chris Richardson. A pattern language for microservices. `https://microservices.io/patterns/index.html`, 2018.

[137] Dirk Riehle, Nikolay Harutyunyan, and Ann Barcomb. Pattern discovery and validation using scientific research methods. `https://dirkriehle.com/2020/03/05/pattern-discovery-and-validation-using-scientific-research-methods-technical-report/` 2020.

[138] Martin P Robillard. What makes apis hard to learn? answers from developers. *IEEE software*, 26(6):27–34, 2009.

[139] Martin P Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2012.

[140] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, volume 13, pages –, 2006.

[141] David Rowe, John Leaney, and David Lowe. Defining systems evolvability-a taxonomy of change. *Change*, 94:541–545, 1994.

[142] Per Runeson, Martin Host, Austen Rainer, and Bjorn Regnell. *Case study research in software engineering: Guidelines and examples*. John Wiley & Sons, 2012.

[143] Fatima Sabir, Francis Palma, Ghulam Rasool, Yann-Gaël Guéhéneuc, and Naouel Moha. A systematic literature review on the detection of smells and their evolution in object-oriented and service-oriented systems. *Software: Practice and Experience*, 49(1):3–39, 2019.

[144] Vitor Sales, Ricardo Terra, Luis Fernando Miranda, and Marco Tulio Valente. Recommending move method refactorings using dependency sets. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 232–241. IEEE, 2013.

[145] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. Data-driven solutions to detect api compatibility issues in android: an empirical study. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 288–298, Washington, DC, USA, 2019. IEEE.

[146] Thomas Scheller and Eva Kühn. Automated measurement of api usability: The api concepts framework. *Information and Software Technology*, 61:145–162, 2015.

[147] Judith Schoonenboom and R Burke Johnson. How to construct a mixed methods research design. *KZfSS Kölner Zeitschrift für Soziologie und Sozialpsychologie*, 69(2):107–131, 2017.

[148] Mathias Schwarz. Uber engineering's micro deploy: Deploying daily with confidence. `https://eng.uber.com/micro-deploy-code/`, 2016.

[149] Michael Sedlmair, Miriah Meyer, and Tamara Munzner. Design study methodology: Reflections from the trenches and the stacks. *IEEE transactions on visualization and computer graphics*, 18(12):2431–2440, 2012.

[150] Souhaila Serbout, Cesare Pautasso, Uwe Zdun, and Olaf Zimmermann. From openapi fragments to api pattern primitives and design smells. In *European Conference on Pattern Languages of Programs (EuroPLoP'21)*, Virtual Kloster Irsee, Germany, July 2021. ACM, ACM.

[151] Samuel Sanford Shapiro and Martin B Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[152] Helen Sharp, Yvonne Dittrich, and Cleidson R. B. de Souza. The role of ethnographic studies in empirical software engineering. *IEEE Transactions on Software Engineering*, 42(8):786–804, 2016.

[153] Lin Shi, Hao Zhong, Tao Xie, and Mingshu Li. An empirical study on evolution of api documentation. In *International Conference on Fundamental Approaches To Software Engineering*, pages 416–431, Berlin, Heidelberg, 2011. Springer.

[154] Apitchaka Singjai, Georg Simhandl, and Uwe Zdun. On the practitioners' understanding of coupling smells – a grey literature based grounded-theory study. *Accepted for publication in Information and Software Technology*, 134:106539, 2021.

[155] Apitchaka Singjai, Georg Simhandl, and Uwe Zdun. On the Practitioners' Understanding of Coupling Smells – A Grey Literature Based Grounded-Theory Study: Dataset and Code. Zenodo, January 2021.

[156] Apitchaka Singjai and Uwe Zdun. Conformance assessment of architectural design decisions on api endpoint designs derived from domain models. *Submitted for Publication*, 2022.

[157] Apitchaka Singjai, Uwe Zdun, and Olaf Zimmermann. Practitioner views on the interrelation of microservice apis and domain-driven design: A grey literature study based on grounded theory. In *18th IEEE International Conference on Software Architecture (ICSA 2021)*, pages –, Washington, DC, USA, March 2021. IEEE.

[158] Apitchaka Singjai, Uwe Zdun, Olaf Zimmermann, and Cesare Pautasso. Patterns on deriving apis and api endpoints from domain model elements. In *Proceedings of the European Conference on Pattern Languages of Programs 2021*, EuroPLoP '21, New York, NY, USA, 2021. Association for Computing Machinery.

[159] Apitchaka Singjai, Uwe Zdun, Olaf Zimmermann, and Cesare Pautasso. Patterns on deriving apis and api endpoints from domain model elements. In *European Conference on Pattern Languages of Programs (EuroPLoP'21)*, July 2021.

[160] S M Sohan, Craig Anslow, and Frank Maurer. Spyrest in action: An automated restful api documentation tool. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 813–818, 2015.

[161] S.M. Sohan, Craig Anslow, and Frank Maurer. A case study of web api evolution. In *2015 IEEE World Congress on Services*, pages 245–252, 2015.

[162] Leonardo Sousa, Anderson Oliveira, Willian Oizumi, Simone Barbosa, Alessandro Garcia, Jaejoon Lee, Marcos Kalinowski, Rafael de Mello, Baldoino Fonseca, Roberto Oliveira, et al. Identifying design problems in the source code: A grounded theory. In *Proceedings of the 40th International Conference on Software Engineering*, pages 921–931, 2018.

[163] Tiago Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, 2021.

[164] Mirko Stocker and Olaf Zimmermann. From code refactoring to api refactoring: Agile service design and evolution. In Johanna Barzen, editor, *Service-Oriented Computing*, pages 174–193, Cham, 2021. Springer International Publishing.

[165] Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. Interface quality patterns: Communicating and improving the quality of microservices apis. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[166] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. Grounded theory in software engineering research: a critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering*, pages 120–131, 2016.

[167] Jeffrey Stylos and Brad Myers. Mapping the space of api design decisions. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*, pages 50–60, Washington, DC, USA, 2007. IEEE, IEEE.

[168] Vijay Surwase. Rest api modeling languages–a developer's perspective. *Int. J. Sci. Technol. Eng*, 2(10):634–637, 2016.

[169] Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

[170] Rasmus Svensson, Adell Tatrous, and Francis Palma. Defining design patterns for iot apis. In *European Conference on Software Architecture*, pages 443–458, Cham, 2020. Springer, Springer International Publishing.

[171] Davide Taibi and Valentina Lenarduzzi. On the definition of microservice bad smells. *IEEE software*, 35(3):56–62, 2018.

[172] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012.

[173] Sarah Taraporewalla. Event driven architecture terminology. `https://sarahtaraporewalla.com/architecture/Event-Driven-Architecture-Terminology`, 2020.

[174] Ricardo Terra, Marco Tulio Valente, Sergio Miranda, and Vitor Sales. Jmove: A novel heuristic and tool to detect move method refactoring opportunities. *Journal of Systems and Software*, 138:19 – 36, 2018.

[175] Steven R Terrell. Mixed-methods research methodologies. *Qualitative report*, 17(1):254–280, 2012.

[176] Fangchao Tian, Peng Liang, and Muhammad Ali Babar. Relationships between software architecture and source code in practice: An exploratory survey and interview. *Information and Software Technology*, 141:106705, 2022.

[177] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. Ten years of jdeodorant: Lessons learned from the hunt for smells. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 4–14, 2018.

[178] Nikolaos Tsantalis and Alexander Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Trans. Softw. Eng.*, 35(3):347–367, May 2009.

[179] Jaroslav Tulach. *Practical api design*. Apress, NY, USA, 2014.

[180] Jan Salvador van der Ven, Anton GJ Jansen, Jos AG Nijhuis, and Jan Bosch. Design decisions: The bridge between rationale and architecture. In *Rationale management in software engineering*, pages 329–348. Springer, 2006.

[181] Oleh Velychko, Valentyn Gaman, Tetyana Gordiyenko, and Oleh Hrabovskyi. Testing of measurement instrument software with the purpose of conformity assessment. *Eastern-European Journal of Enterprise Technologies*, 1(9):19–26, 2019.

[182] Vaughn Vernon. *Implementing Domain-Driven Design*. Addison-Wesley Professional, Boston, USA, 2013.

[183] S. Vidal, E. Guimaraes, W. Oizumi, A. Garcia, A. D. Pace, and C. Marcos. Identifying architectural problems through prioritization of code smells. In *2016 X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 41–50, 2016.

[184] Markus Voelter, Michael Kircher, and Uwe Zdun. *Remoting Patterns - Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*. J. Wiley & Sons, Hoboken, NJ, USA, 2004.

[185] Michael Waterman, James Noble, and George Allan. How much up-front? a grounded theory of agile architecture. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 347–357. IEEE, 2015.

[186] Jane Webster and Richard T Watson. Analyzing the past to prepare for the future: Writing a literature review. *MIS quarterly*, pages xiii–xxiii, 2002.

[187] Charles B Weinstock and John B Goodenough. On system scalability. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2006.

[188] Mark Wilkinson, Benjamin Vandervalk, and Luke McCarthy. The semantic automated discovery and integration (sadi) web service design-pattern, api and reference implementation. *Nature Precedings*, 2:8–8, 2011.

[189] Joost F Wolfswinkel, Elfi Furtmueller, and Celeste PM Wilderom. Using grounded theory as a method for rigorously reviewing literature. *European journal of information systems*, 22(1):45–55, 2013.

[190] Wei Wu, Adrien Serveaux, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. The impact of imperfect change rules on framework api evolution identification: an empirical study. *Empirical Software Engineering*, 20(4):1126–1158, 2015.

[191] Aiko Yamashita and Leon Moonen. To what extent can maintenance problems be predicted by code smell detection?–an empirical study. *Information and Software Technology*, 55(12):2223–2242, 2013.

[192] Robert K Yin et al. Case study research and applications: Design and methods. 2018.

[193] Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and assessing architecture conformance to microservice decomposition patterns. In *International Conference on Service-Oriented Computing*, pages 411–429. Springer, 2017.

[194] Uwe Zdun, Evangelos Ntentos, Konstantinos Plakidas, Amine El Malki, Daniel Schall, and Fei Li. On the design and architecture of deployment pipelines in cloud- and service-based computing - A model-based qualitative study. In Elisa Bertino, Carl K. Chang, Peter Chen, Ernesto Damiani, Michael Goul, and Katsunori Oyama, editors, *2019 IEEE International Conference on Services Computing, SCC 2019, Milan, Italy, July 8-13, 2019*, pages 141–145. IEEE, 2019.

[195] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing - 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*, volume 11236 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 2018.

[196] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice apis. In Claus Pahl, Maja Vukovic, Jianwei Yin, and Qi Yu, editors, *Service-Oriented Computing*, pages 73–89, Cham, 2018. Springer International Publishing.

[197] Uwe Zdun, Mirko Stocker, Olaf Zimmermann, Cesare Pautasso, and Daniel Lübke. Guiding architectural decision making on quality aspects in microservice APIs. In *16th International Conference on Service-Oriented Computing ICSOC 2018*, pages 78–89, November 2018.

[198] Hao Zhong and Hong Mei. An empirical study on api usages. *IEEE Transactions on Software Engineering*, 45(4):319–334, 2017.

[199] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. An empirical study on api parameter rules. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 899–911, 2020.

[200] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language api documentation. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 307–318, 2009.

[201] Wei Zhou, Li Li, Min Luo, and Wu Chou. Rest api design patterns for sdn northbound api. In *2014 28th international conference on advanced information networking and applications workshops*, pages 358–365, Washington, DC, USA, 2014. IEEE, IEEE.

[202] Franz Zieris and Lutz Prechelt. On knowledge transfer skill in pair programming. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM'14, New York, NY, USA, 2014. Association for Computing Machinery.

[203] Olaf Zimmermann. Microservices tenets. *Computer Science-Research and Development*, 32(3-4):301–310, July 2017.

[204] Olaf Zimmermann. Domain-driven service design with context mapper and mdsl. `https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html`, 2020.

[205] Olaf Zimmermann, Jana Koehler, Frank Leymann, Ronny Polley, and Nelly Schuster. Managing architectural decision models with dependency relations, integrity constraints, and production rules. *J. Syst. Softw.*, 82(8), August 2009.

[206] Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. Interface responsibility patterns: Processing resources and operation responsibilities. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[207] Olaf Zimmermann, Daniel Lübke, Uwe Zdun, Cesare Pautasso, and Mirko Stocker. Interface responsibility patterns: Processing resources and operation responsibilities. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20, pages –, New York, NY, USA, 2020. ACM.

[208] Olaf Zimmermann, Cesare Pautasso, Daniel Lübke, Uwe Zdun, and Mirko Stocker. Data-oriented interface responsibility patterns: Types of information holder resources. In *Proceedings of the European Conference on Pattern Languages of Programs 2020*, EuroPLoP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[209] Olaf Zimmermann and Mirko Stocker. Dpr tutorial 1: Api design in an online shop. `https://github.com/socadk/design-practice-repository/blob/master/tutorials/DPR-Tutorial1.md`, 2020.

[210] Olaf Zimmermann and Mirko Stocker. Design practice reference guides and templates to craft quality software in style. `https://leanpub.com/dpr`, 2021.

[211] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. Introduction to microservice api patterns (map). *Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019)*, 78(4):1–17, 2020.

[212] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, Cesare Pautasso, and Uwe Zdun. Microservice api patterns. `https://microservice-api-patterns.org/`, 2021.