UNIVERSITY OF APPLIED SCIENCES OF EASTERN SWITZERLAND (HSR FHO)

MASTER THESIS

A Modeling Framework for Strategic Domain-driven Design and Service Decomposition

Author:
Stefan KAPFERER

Supervisor:
Prof. Dr. Olaf ZIMMERMANN

External Examiner:
Dr. Gerald REIF

A thesis submitted in fulfillment of the requirements for the degree of Master of Science FHO in Engineering focusing on Information and Communication Technologies

in the

Software and Systems
Master Research Unit

January 24, 2020

Declaration of Authorship

I, Stefan KAPFERER, declare that this thesis titled, "A Modeling Framework for Strategic Domain-driven Design and Service Decomposition" 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 work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Rapperswil, January 24, 2020

Stefan Kapferer

UNIVERSITY OF APPLIED SCIENCES OF
EASTERN SWITZERLAND (HSR FHO)

Abstract

Master of Science FHO in Engineering

A Modeling Framework for Strategic Domain-driven Design and Service Decomposition

by Stefan KAPFERER

The decomposition of a system into modules or services is a challenging practical problem and research question that has not been answered satisfactorily yet. With the current trend towards microservices, Strategic Domain-driven Design (DDD) has become a popular technique to decompose a domain into so-called Bounded Contexts. In our previous work we presented Context Mapper, an open source tool offering a Domain-specific Language (DSL) based on the DDD patterns. It supports the evolution of DDD pattern-based architecture models in a formal and expressive way. By applying Architectural Refactorings (ARs), systems can be decomposed in an iterative manner. However, our validation activities have shown that our tool-based approach requires additional capabilities to expand the target user group. For instance, support for reverse engineering has been requested since re-modeling existing systems is often too expensive in brownfield projects. Decomposition on the basis of a systematic approach and generating graphical Context Maps are other user requirements.

With this thesis we propose a modular and extensible component architecture for a modeling framework based on Strategic DDD. The already existing Context Mapper tool evolved into a framework offering components for reverse engineering, architecture modeling, refactoring, systematic decomposition, and generation of other representations from the Context Mapper DSL (CML) models. The DSL constitutes the core component of the framework. With our discovery library we propose a strategy-based approach to reverse engineer CML models. An extended set of ARs has been conceptualized allowing users to evolve the architecture models iteratively. With Service Cutter, we integrated a systematic service decomposition approach to derive new Context Maps that improve coupling and cohesion. A graphical Context Map generator enhances the transformation tools to convert CML code into visual diagrams.

The proposed framework supports architects and business analysts in creating DDD-based models and improve their productivity at the same time. We hypothesize that the mentioned personas can benefit from a tool which assists them in evolving Context Maps. During this thesis we applied action research to validate our concepts and improve the prototype iteratively. With case studies such as the Lakeside Mutual microservice project and our own framework architecture we validated the usefulness and effectiveness of the suggested modeling framework. The conducted validation activities indicate that the hypothesis above holds true.

Acknowledgements

I would like to express my deep gratitude to my supervisor Prof. Dr. Olaf Zimmermann for his outstanding support during this thesis and throughout my Master's. I highly appreciate his guidance, insight, and expertise which assisted my research projects greatly. By adopting and validating Context Mapper, using and presenting the tool in application architecture courses at HSR, and co-authoring a conference paper, he supported and challenged me during my studies beyond his supervision duties. It has been a great privilege to work and study under his guidance.

I am grateful to everyone who gave me the opportunity to present my project in meetups, especially Vaughn Vernon for the possibility to demonstrate Context Mapper in one of his IDDD workshops. I am also thankful to the students and all those who used the tool and/or provided their valuable feedback that allowed us to validate the research concepts and improve the prototypic implementation of our modeling framework.

Last but not least I would like to thank my family and loved ones for being there for me and for supporting me in everything I do in my life. Many thanks to Brigita Okello for proofreading this thesis and all her love and support. I am extremely grateful to my mother and stepfather for their support and advice throughout my life, and for preparing me for my future. The completion of my Master's and this thesis would not have been possible without them.

Contents

Chapter 1

Introduction

1.1 Context and Problem

Decomposing a system into appropriately sized services is a challenging task which has gained much attention due to the trend towards (micro-)service-oriented architectures during the last years. Strategic Domain-driven Design (DDD) is a popular approach among practitioners to tackle this challenge. DDD adopters decompose a system by identifying so-called Bounded Contexts[1]. Context Map[1] [10] diagrams and the corresponding relationship patterns further allow them to model the relationships between the Bounded Contexts. However, modeling tools based on these DDD patterns rarely exist and there are different interpretations and opinions among practitioners regarding how these patterns can be applied and combined. Context Maps and DDD-based architecture models are created manually and without tool support so far.

It is our hypothesis that software architects, service designers and business analysts can benefit from a precise interpretation of the DDD patterns and a tool which supports them in expressing DDD-based architecture models in a rigorous and formal manner. Furthermore, we believe that software architecture and the models describing it evolve iteratively by applying transformations and Architectural Refactorings (ARs) [103].

In our previous work [46, 50] we realized Context Mapper [15], an open source project providing a Domain-specific Language (DSL) based on the strategic DDD patterns including editing, validation, and transformation tools. In our first term project [46] we conceptualized the Context Mapper DSL (CML) based on a meta-model clarifying and offering advice regarding how the DDD patterns can be combined. The second term project [50] proposed a series of ARs for such DDD-based architecture models derived from Decomposition Criteria (DCs). These ARs have been implemented as code refactorings for the CML language and allow transforming the architecture models in an agile [1] way. Besides the language and the refactorings, Context Mapper [15] provides generators to transform the CML Context Maps into other representations such as PlantUML [72] diagrams or Microservices Domain-Specific Language (MDSL) [102] (micro-)service contracts.

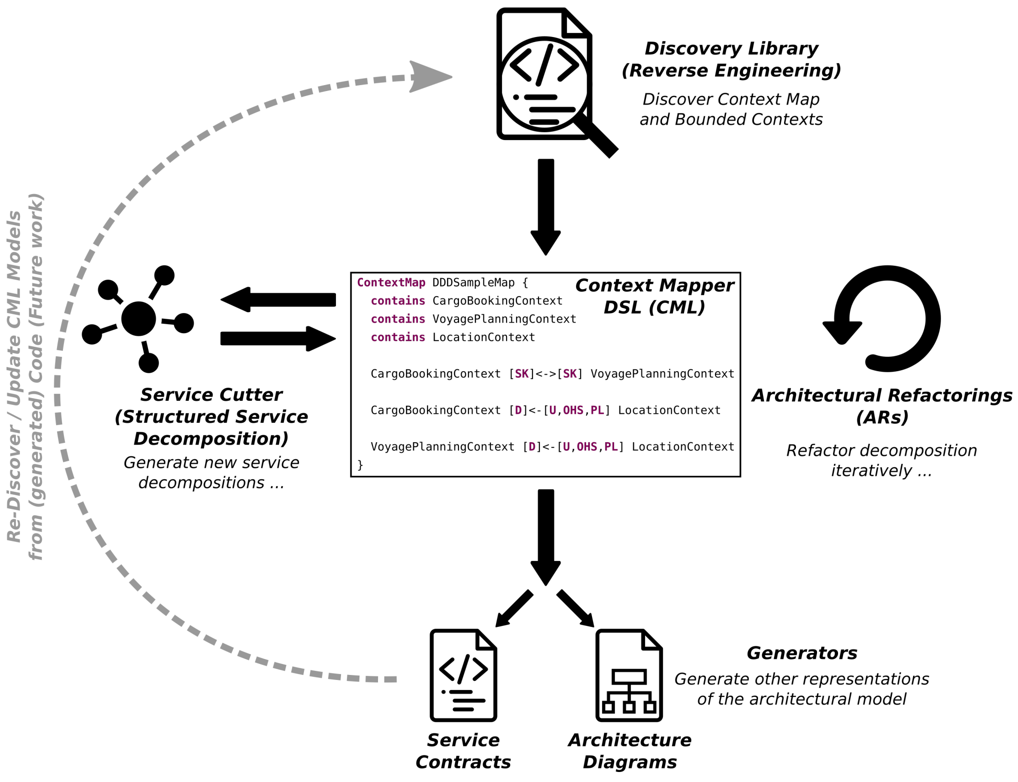
However, the conducted validation activities revealed potential enhancements and missing functionalities for certain user groups. Case studies in our previous project [50] have shown that users in brownfield projects prefer tools

[1]Please note that the DDD terms and patterns will be used throughout this thesis without a re-introduction. For an introduction to the Context Mapper and DDD concepts we refer to our previous work [46] and the DDD literature [24, 25, 98].

that support reverse engineering of architecture models. Other architecture modeling tools provide libraries to generate the models from existing source code. In addition, a structured and systematic approach which suggests new service decompositions on the basis of decoupling criteria was not yet available in the Context Mapper tool. The suggested ARs only supported transforming Bounded Contexts and Aggregates but not relationships between Bounded Contexts. Finally, a generator which produces a graphical representation of the Context Map was missing.

1.2 Vision

With this thesis we propose a framework architecture for Strategic DDD modeling tools such as Context Mapper [15]. The tool set realized so far must evolve towards a modular and extensible framework which fulfills the requirements of architects and business analysts applying DDD. Figure 1.1 illustrates the capabilities of the suggested framework. The CML language builds the core component the of the architecture, allowing users to model a system in terms of the DDD patterns.


Discovery Library
(Reverse Engineering)
Discover Context Map and Bounded Contexts
Context Mapper DSL (CML)
Service Cutter (Structured Service Decomposition)
Generate new service decompositions ...
Architectural Refactorings (ARs)
Refactor decomposition iteratively ...
Re-Discover / Update CML Models from (generated) Code (Future work)
Generators
Generate other representations of the architectural model
Service Contracts
Architecture Diagrams

```
ContextMap DDDSampleMap {
  contains CargoBookingContext
  contains VoyagePlanningContext
  contains LocationContext

  CargoBookingContext [SK]<->[SK] VoyagePlanningContext

  CargoBookingContext [D]<-[U,OHS,PL] LocationContext

  VoyagePlanningContext [D]<-[U,OHS,PL] LocationContext
}
```

FIGURE 1.1: DDD Modeling Framework Architecture Overview

With the discovery library we propose a strategy-based reverse engineering component to discover Bounded Contexts and Context Maps in existing source code. Therefore, brownfield projects are allowed to reverse engineer CML models of existing monolithic systems or entire Context Maps of systems exhibiting a (micro-) service-oriented architecture. Given an initial CML model, architects are able to refactor the decomposition with our ARs [50] iteratively. This set of already existing ARs must be extended with refactorings on the Context Map relationship level.

A structured and systematic service decomposition approach such as Service Cutter [36] shall be integrated into the framework. It allows generating new decomposition suggestions on the basis of (de-) coupling criteria [36, 50] and graph clustering algorithms. As illustrated in Figure 1.1, the input and output of the *Service Cutter* component shall be CML code. Finally, the framework allows to generate many other representations of the architectural models out of CML code. Context Mapper [15] can already generate PlantUML [72] diagrams and (micro-) service contracts. With this thesis another generator producing graphical Context Maps inspired by Vernon [98] and Brandolini [10] shall be incorporated.

As Figure 1.1 indicates, future projects may enhance the discovery library so that the generated CML models can be updated according to changes in the codebase or with respect to manual adjustments in generated artifacts such as the MDSL contracts. With such an update mechanism one can ensure that the models stay up-to-date and that the «model-code» gap [26] is closed.

Finally, the framework must be designed in an extensible fashion so that new discovery strategies, refactorings, and generators can be added easily.

1.3 Thesis Results

Based on the results of validation activities conducted during our previous work [46, 50] we improved the existing components of the Context Mapper tool [15] and realized the framework architecture as illustrated in Figure 1.1.

The CML language has been decoupled from the Eclipse Integrated Development Environment (IDE) and can now be used as *standalone* library in Java projects. This allows users to parse CML code and use our generators within their own applications. The existing MDSL service contract [102] and PlantUML [72] generators were further validated and improved during the thesis.

With the goal to fulfill the requirements of our target user group, we conceptualized a strategy-based Context Map discovery library for reverse engineering purposes. We researched which frameworks and technologies are popular for realizing microservice projects and selected the following discovery strategies to reverse engineer corresponding architectures. The library currently supports the discovery of CML Bounded Contexts from existing Spring Boot [71] services. The discovery strategy derives one Bounded Context for each Spring Boot application and Aggregates for each RESTful HTTP endpoint. The domain models within the Aggregates are derived from the Data Transfer Objects (DTOs) [28] of the operations within the endpoint. Moreover, the library is able to discover a Context Map, concretely relationships between Bounded Contexts, by analyzing Docker [20] configuration files (*compose* files). The selection of these discovery strategies for our prototype is based on the following criteria: *feature coverage* (discover complete CML models), *representativeness* (support popular frameworks that promise to be future-oriented), and *usability* (minimize manual work). We validated the approach with the Lakeside Mutual microservice project [61] by recovering its architecture based on the proposed discovery strategies.

The set of existing, structural ARs [50] was analyzed and enhanced by two new refactorings as illustrated in Figure 1.2. These two refactorings allow users to change the decomposition by adjusting Shared Kernel and Partnership relationships, hence striving for improved coupling and cohesion.

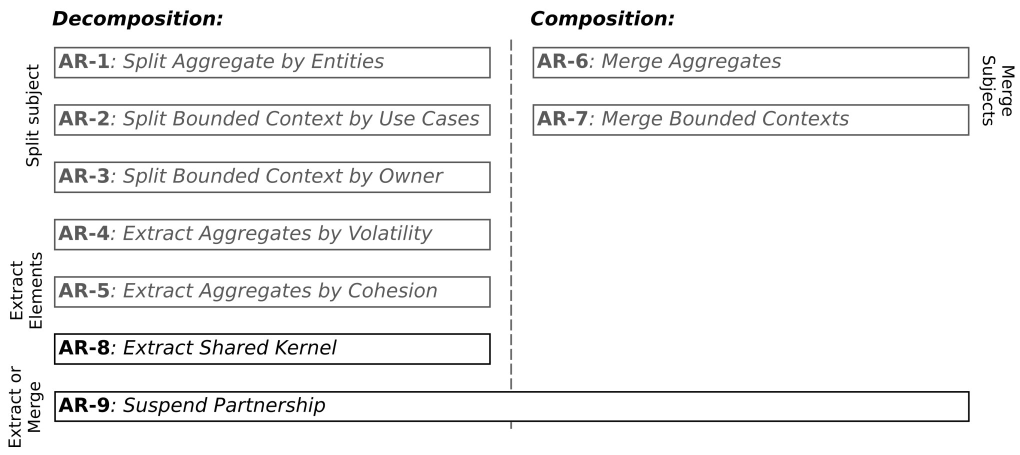

Decomposition:
Composition:
Split subject
AR-1: Split Aggregate by Entities
AR-2: Split Bounded Context by Use Cases
AR-3: Split Bounded Context by Owner
Extract Elements
AR-4: Extract Aggregates by Volatility
AR-5: Extract Aggregates by Cohesion
AR-8: Extract Shared Kernel
Extract or Merge
AR-9: Suspend Partnership
AR-6: Merge Aggregates
AR-7: Merge Bounded Contexts
Merge Subjects

FIGURE 1.2: New «Structural» Architectural Refactorings (ARs)

In addition, we created a new AR category for relationship refactorings. Figure 1.3 shows the two prototypic and already implemented ARs within this category. These ARs aim to provide simple tools to change the relationships between Bounded Contexts without changing the decomposition itself.

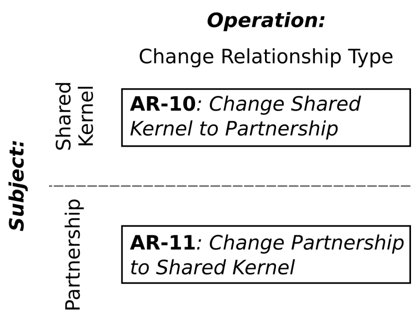

Operation:
Change Relationship Type
Subject:
Shared Kernel
AR-10: Change Shared Kernel to Partnership
Partnership
AR-11: Change Partnership to Shared Kernel

FIGURE 1.3: New Category of ARs: «Relationship» Refactorings

With Service Cutter [36] we integrated a structured and systematic service decomposition component into our framework. We implemented a new library version of the Service Cutter engine [35] first. The library is now used within Context Mapper to generate new Context Maps on the basis of Service Cutter's coupling criteria catalog [37].

Finally, we implemented another generator that produces graphical Context Maps. In our previous project reports [46, 50] we already used Context Map illustrations inspired by the notations of Vernon [98] and Brandolini [10]. The generator implemented as part of this thesis produces such illustrations automatically out of a CML Context Map. Figure 1.4 illustrates an example Context Map of our «fictitious insurance example»[2].

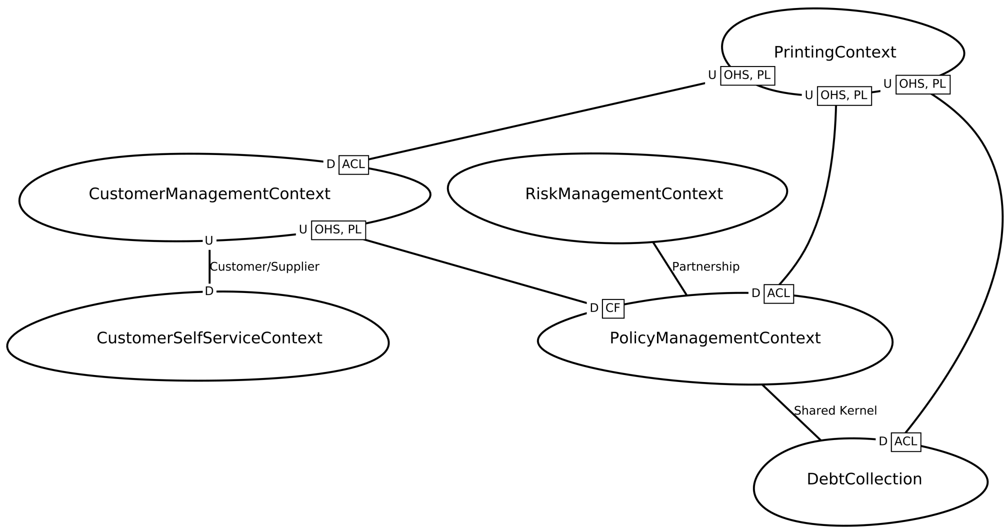


FIGURE 1.4: Example Context Map Generated by Context Mapper [15]

Besides the proposed modeling framework as the main result of this thesis, a few small changes to the CML language were made to address the user feedback. Moreover, we have enhanced our examples repository to provide the example models in different levels of detail. With these different stages we address the requirements of users without technical background and offer them simpler examples.

The remainder of this thesis is structured as follows. Chapter 2 discusses the user roles and scenarios for which the modeling framework has been designed. It further introduces the requirements in the form of user stories. Chapter 3 analyzes different realization approaches for the individual framework components and documents decisions and rationales. In Chapter 4 we present the design and implementation of the prototypes within the Context Mapper open source project [15]. It explains implementation details of the individual framework components and the interfaces between them. Chapter 5 evaluates the results of this thesis and discusses strengths and weaknesses of the approach. In Chapter 6 we discuss related work and compare the proposed framework with other approaches. Finally, Chapter 7 summarizes the thesis and outlines future work.

[2] `https://github.com/ContextMapper/context-mapper-examples`

Chapter 2

Framework Users and Requirements

This chapter discusses the user roles and requirements for the Context Mapper DSL (CML) language and all other components of the proposed framework. It presents the functional requirements for the language and the components for reverse engineering, structured service decomposition based on coupling criteria, the Architectural Refactorings (ARs), and the new graphical Context Map generator. On top of that, the end of the chapter exhibits the Non-Functional Requirements (NFRs) which have to be fulfilled by the implementation of our prototypes within the Context Mapper tool [15].

All functional requirements are specified with user stories based on the "Role-Feature-Reason" template invented 2001 by a team at Connextra in the UK [2]:

«As a <***who*** *wants to accomplish something*>,
I want to <***what*** *they want to accomplish*>
so that <***why*** *they want to accomplish that thing*>.» [2]

2.1 User Roles and Personas

In our previous work [46] we defined user stories and personas for our Domain-specific Language (DSL) and surrounding tools. Business analysts can use the language to analyse a domain and establish a so-called Ubiquitous Language [24], a common understanding and vocabulary about the domain. Software architects on the other hand can describe, evolve and communicate system architectures in terms of strategic Domain-driven Design (DDD). An architect can also describe Enterprise Application Integrations (EAIs) and how systems, applications, or development teams are connected. Adopters of (micro-)service-oriented architectures can use the language and our tools to model and evolve the decomposition into (micro-)services.

However, recent user feedback and validation activities have shown that the language and the provided examples[1] seem complex to users without technical background. We had focused more on the requirements of software architects and neglected those of business analysts. Therefore, the provided example models shall be improved so that not only software architects and engineers but also business analysts can be introduced to the language smoothly. With

[1] `https://github.com/ContextMapper/context-mapper-examples`

the following personas and user stories we want to clarify the difference between these two user roles. In Chapter 4 we explain how we addressed the problem and adjusted the example models concretely. Please note that the consecutive personas are based on our previous work [46] in which we already worked out the requirements for our CML language.

2.1.1 Martin Analyst

«Martin is a business analyst and domain expert with many years of experience within the domain of the project he is working on. Before he started working in the current software company as a domain expert he used to work for one of the customers using the software. Thus, he has a deep knowledge of the business and the domain.

Martin talks with the customer and brings the requirements into the team. His goal is to raise the knowledge about the domain within the team. However, Martin does not want to translate the business language into another, lets say *developer language*, since he knows from his experience that this always leads to misunderstandings. He insists on speaking the businesses language and wants to create models using the terms he knows.» [46]

Constraints [46]

We can derive requirements and constraints regarding a potential modeling tool from the role of a domain expert such as described above. The way of how a model is expressed should not require any programming skills. The language should be as similar as possible to the natural language. This type of user expects from the modeling tool that he or she can create models using the natural language. Furthermore, they do not want to learn a complex syntax such as typical programming languages may provide. They simply aim to focus on the core domain and its language.

2.1.2 Bob Architect

«Bob is a software architect and his goal is to influence architectural decisions of the development teams and ensure that these are justified and properly documented. He further coaches the teams regarding design and architecture issues that come up. He analyzes problems in existing architectures and tries to propose solutions. Models are the major tool for Bob to communicate with the other roles and stakeholders. With models he is able to illustrate potential architectural improvements or simply describe the actual state of a system. Since Bob communicates with different audiences he wants to create models in different representations and levels of abstraction.» [46]

Contraints [46]

A software architect may have to supervise many teams implementing different applications or Bounded Contexts. Therefore, he perhaps does not have the time and budget to familiarize himself with very complex tools at the beginning of a project. The creation of models must be efficient and ideally the tool allows the architect to adapt the level of abstraction for the target audience and generate other representations of the models. In comparison to the business analyst he additionally supports the development teams regarding technical details. Hence, the models must exhibit a certain technical depth. The domain models of Bounded Contexts are potentially worked out on a level of detail that allows to generate code.

From the personas above we can conclude that the level of detail in introductory examples must be different depending on the user role. A business analyst writes models which are understandable without programming knowledge. Engineers and architects on the other hand may write domain models from which code or database schemata could be derived. The examples provided by the Context Mapper tool [15] before this thesis did not fulfill this requirement, since the Bounded Context specifications were too detailed and the tactic DDD part of the DSL was too technical. The examples were splitted according to the two user roles introduced above. This change was guided by the following user stories which highlight the essences of the different requirements.

2.1.3 US-1: Analysing and Describing the Domain as Business Analyst

As a business analyst, I want to describe the problem domain and its subdomains in a natural language so that I can develop and communicate a common understanding as well as an ubiquitous language about the business concepts and their relationships in the given domain.

2.1.4 US-2: Describing Software Architecture and Design as Software Architect

As a software architect or engineer responsible for the design and implementation based on the results of the business analysis (US-1), I want to model the subsystems (Bounded Contexts) and components (Aggregates) of my architecture and how they interact so that I can evolve the architecture with tool support (refactorings and systematic service decomposition), communicate the architecture, and generate other representations of the models such as Unified Modeling Language (UML) diagrams, service contracts, or even code.

With the user stories and personas above we documented general requirements for our framework and the provided example models. The next sections discuss the functional requirements for the individual framework components specifically, followed by the NFRs the framework must fulfill.

2.2 Reverse Engineering Scenarios

The following user stories describe the requirements for brownfield project scenarios with existing systems. In such a case the users need a reverse engineering functionality to generate CML models automatically. The validation activities of our previous project [50] have shown that users in such projects are not willing to re-model big existing systems in a new modeling tool.

Both user stories describe reverse engineering scenarios. The user story US-3 however focuses on (micro-)service-oriented architectures where multiple Bounded Contexts (services) and a Context Map can be discovered, while US-4 describes a scenario with a monolithic system where one Bounded Context can be reverse engineered to decompose it afterwards.

2.2.1 US-3: Generate Bounded Contexts and Context Map Models for existing (Micro-)Service Architecture

As a software architect or engineer developing a (micro-)service-oriented system, I want to generate an architecture overview (i.e. a Context Map written in CML) for my existing system so that I can analyze, communicate, and improve the existing architecture as well as generate other representations of the model such as UML diagrams or a graphical Context Map.

2.2.2 US-4: Generate Bounded Context Model from existing Monolithic System

As a software architect or engineer who wants to decompose an existing monolithic system into (micro-)services, I want to generate Bounded Context descriptions written in CML out of my source code so that I can analyze it with systematic decomposition approaches such as Service Cutter and find a service decomposition in an iterative way by using architectural refactorings.

2.3 Structured Service Decomposition

Once a modeling framework user has a model of the system, reverse engineered or manually modeled, he or she may want to analyze the decomposition with respect to coupling and cohesion between the Bounded Contexts. Based on (de-)coupling criteria [37, 50] systematic service decomposition approaches such as Service Cutter [36] allow to generate new service decompositions which improve the coupling between the individual services and the cohesion within them.

2.3.1 US-5: Analyze a Strategic DDD Model using Coupling Criteria to find Better Service Decompositions

As a software architect or engineer who has modeled a system in terms of strategic DDD patterns with a tool such as Context Mapper, I want to let the tool analyze my model on the basis of given coupling criteria and let it suggest service decompositions so that I am able to improve the coupling between my Bounded Contexts and the cohesion within them.

2.4 Agile Context Map Evolution with Architectural Refactorings (ARs)

In our previous term project [50] we presented seven ARs which were already implemented within the Context Mapper tool before this thesis. They support the user in evolving and improving the architecture in an iterative way. The following generic user story describes the idea behind our ARs generally:

> «As a software architect or engineer, I would like to model strategic DDD Context Maps rapidly and use supporting transformation tools to revise and refine the architecture drafts, so that the DDD models can be crafted and evolved in an agile way.» [50]

During validation and testing activities we determined that all provided ARs focus on the decomposition of Bounded Contexts and Aggregates. There were no refactorings allowing to transform the Context Map relationships. Thus, the conceptualization of at least two relationship-related refactorings as a prototype was a requirement towards this thesis. Changing the type of a relationship between two contexts manually can be error-prone and takes more time in comparison with applying an automated refactoring. The following user story shall express the need for such transformations.

2.4.1 US-6: Change the Relationship between Bounded Contexts

As a business analyst, software architect, or engineer who models the relationships between Bounded Contexts, I want to revise and change the type of a relationship between two Bounded Contexts rapidly and with tool support, so that the DDD Context Map relationships can be crafted in an agile way and errors due to manual code adjustments are avoided.

2.5 Generating other Representations

Besides decomposing and evolving the architecture model a user must be able to generate other representations out of the model. With our previous projects [46, 50] we already provided a PlantUML [72] diagram generator, a Microservices Domain-Specific Language (MDSL) (micro-)service contract generator, and Service Cutter [35] input file generators.

2.5.1 Graphical Context Map Generation

With this thesis we add another generator producing graphical Context Maps to the framework. Users working with strategic DDD models are familiar with the Context Map visualizations by Vernon [98] and Brandolini [10] and probably expect the generation of such graphics from a framework like ours.

US-7: Generate Graphical Context Map

As a user of a strategic DDD modeling tool such as Context Mapper, I want to generate a graphical representation of the DDD Context Map automatically so that I have a better representation for communication and documentation purposes.

2.5.2 Existing Generators

Our validation activities also resulted in new requirements (enhancements) for the existing PlantUML [72] and MDSL [102] generators. However, the user stories for those existing generators are already part of the previous project reports [46, 50]. Therefore, we will not repeat the requirements for them within this thesis although they are still valid. In Chapter 4 we list the realized improvements in these generators in detail.

2.6 Non-Functional Requirements (NFRs)

This section presents the NFRs all the components of our modeling framework have to fulfill. Please note that some of these requirements were already defined in our previous projects [46, 50] and are taken from these project reports since they are still valid and must be fulfilled by the new components of the framework as well.

2.6.1 General Framework NFRs

We start with general NFRs which have to be fulfilled by all components and continue with individual requirements in the following sections.

Future-oriented Use of Tools and Frameworks

The tools and libraries used for the development of the framework components should be well established, open and sustainable. Libraries and frameworks with no activity/commits during the last year should be avoided. At least be sure that the tools can be replaced by using open and sustainable data formats.

Reliability

The developed tools should work reliably having no crashes and/or data losses. To achieve these goals the tools have to be implemented in an resilient fashion and should be tested well (Unit Tests, Integration Tests and manual User Tests).

Extensibility

The individual framework modules, namely the reverse engineering component, the ARs, the Service Cutter [36] integration, and the generators shall be developed in an extensible fashion. Adding new reverse engineering strategies, ARs, service cutting algorithms, and generators must be possible without substantial changes to the framework itself.

Licences

Since the project is open source, licences such as «Apache License 2.0» and «Eclipse Public License 1.0» are prefered. Libraries or frameworks under «GNU General Public License (GNU GPL)» must not be used.

Supportability and Maintainability

The project's code quality should be kept at a good level. We have to set up appropriate tools and mechanisms to support this goal (updating *master* only by pull request, use Continuous Integration (CI) build server, measure test coverage and keep it high). The code should be clean and understandable, also for a junior software engineer. Do not use very special (not well-known) language features and create a documentation if it is needed for more complex components.

Documentation

All new features added to the Context Mapper tool [15] shall be documented on the project's documentation website[2].

Examples

The online documentation and the Context Mapper examples repository[3] must provide meaningful examples for all developed language features and tools.

2.6.2 Reverse Engineering Framework

The following requirement concerns the reverse engineering part of the framework only.

Technology Neutrality and Extensibility

This component discovers Bounded Contexts and their relationships (Context Map) by analyzing existing codebases. Since the systems to be analyzed can be written in different languages and realized with different technologies, the framework must provide the possibility to implement discovery strategies for any language or technology. Note that we only implement two strategies as a proof-of-concept as part of this thesis. However, the framework must be designed such that new discovery strategies for other languages and technologies can be added easily without changing the framework itself.

2.6.3 CML Modeling Language (Core)

If changes to the CML language are implemented, they must be designed with respect to the following requirements. These DSL-related requirements have been developed during our previous projects [46, 50] on the basis of our own

[2] `https://contextmapper.org/docs`
[3] `https://github.com/ContextMapper/context-mapper-examples`

experience with programming languages and DSLs, and the input of the supervisor of this thesis.

Simplicity of the DSL

The DSL grammar should satisfy the requirements but still be designed as simple as possible. A software architect or engineer knowing the concepts of DDD should be able to understand introductory examples written in the DSL within 15 to 20 minutes. With provided examples and tutorials one should also be ready to start creating an own model within at most one hour.

Quickly Writable without Redundancy

A Context Map must stay easy and quickly to write. The syntax of the definitions shall not exhibit any redundancies or ambiguities. With the help of provided examples a user must be able to define a new Context Map with three to four Bounded Contexts within 30 minutes.

Well Readable

The definitions in a CML model shall be well readable. A user which is familiar with our DDD meta-model[4] [46] must be able to identify corresponding concepts and patterns on an existing CML Context Map within 5 minutes.

Consistent

The syntax of the DSL shall ensure that the definitions are always consistent with our interpretation [46] of the possibilities regarding the strategic DDD patterns. Semantic validators must be implemented to identify deviations if needed.

Parsable by the Tool (Xtext)

The defined syntax must be realizable and easily parsable with the used language framwork, namely Xtext [23].

2.6.4 Architectural Refactorings (ARs)

The following requirements must be fulfilled by all implemented ARs.

Transformations must result in valid Models

All ARs implemented in the Context Mapper tool must always result in valid models according to the grammar of CML, which is the advantage of such tools in comparison with applying the changes manually. If a transformation leads to conflicts or necessary changes in other parts of the CML model, the AR must solve them automatically.

[4]`https://contextmapper.org/docs/language-model/`

Performance

The execution of an AR must not take longer than two to three seconds at most. The performance shall be tested with the Context Mapper example models[5].

2.6.5 Service Cutter Integration

The integration of the Service Cutter [36] engine shall be done with respect to the requirement below.

Algorithm Exchangeability

The Service Cutter integration which shall propose new Context Maps on the basis of the Service Cutter criteria catalog [37] shall be implemented in a way which allows us to replace the algorithm behind the cutting mechanism. Service Cutter already provides two graph clustering algorithms and new algorithms may be evaluated as part of this thesis or future projects. Replacing the algorithm shall be as easy as possible.

We close this chapter with the introduced NFRs. All requirements our modeling framework must fulfill have been presented in the last sections. The next chapter documents our analysis and research activities leading to decisions and rationales regarding how the individual components can be realized. It discusses strategies how the reverse engineering component can be designed, outlines potential structured service decomposition algorithms, selects new ARs for Context Map relationships, and evaluates potential tools to realize the graphical Context Map generator.

[5]`https://github.com/ContextMapper/context-mapper-examples`

Chapter 3

Analysis of the Modeling Framework Components

In this chapter we research and analyze strategies and approaches for the implementation of the individual components, or Bounded Contexts, of the modeling framework proposed by this thesis. Figure 3.1 illustrates a corresponding Context Map of our framework. As many other illustrations within this thesis Figure 3.1 is generated with Context Mapper, since we model the architecture of our framework with our own tool. The «language core» context represents the Context Mapper DSL (CML) modeling language including the Xtext-based [23] Eclipse plugin. The Published Language (PL) of the core component is the semantic meta-model behind the CML language which we proposed in our first term project [46]. The meta-model is introduced again in Chapter 4 in order to explain the interfaces between the components in detail.

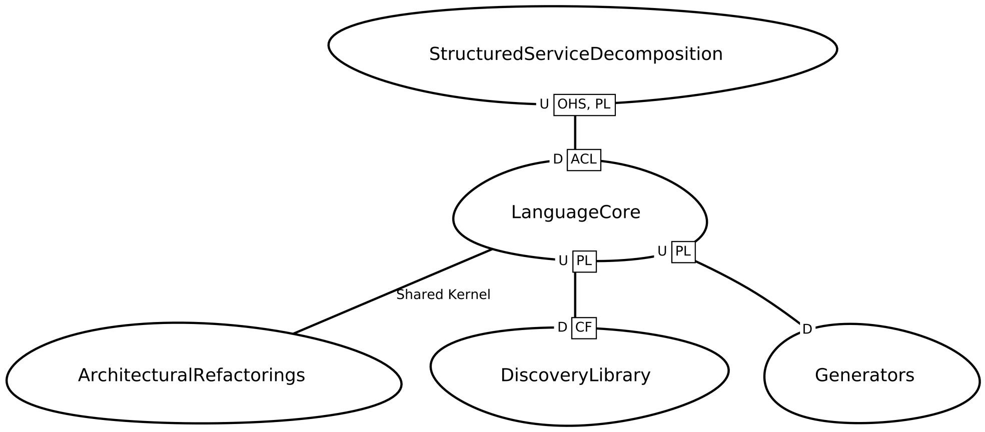


FIGURE 3.1: Strategic DDD Modeling Framework Context Map (Generated by Context Mapper [15])

The Architectural Refactorings (ARs) presented in our second term project [50] depend on the language and its meta-model as well. Thus, this Bounded Context has a close relationship with the «core» and as illustrated in Figure 3.1, the language meta-model can be seen as a Shared Kernel between these two contexts.

The «generators» Bounded Context includes all transformation tools allowing to generate other representations of the CML models or input files for other tools. It currently encompasses the PlantUML [72] diagram, Microservices

Domain-Specific Language (MDSL) [102], and Service Cutter input files [35] generators.

The remaining Bounded Contexts in our Context Map in Figure 3.1 are new and part of the evolution of the Context Mapper tool towards the modeling framework proposed by this thesis. With the «discovery library» Bounded Context the framework shall offer an extensible tool to generate CML models from existing source code. As illustrated, the discovery context uses the meta-model (PL) of the core component to generate the models. In addition, it acts as a Conformist (CF) and thus conforms to the published language of the core component. The «structured service decomposition» context complements the modeling framework with a tool that supports the generation of service decompositions automatically and based on defined Decomposition Criteria (DCs). This context is implemented as a separate library which provides an Open Host Service (OHS) (open API) that is used by the language core to decompose CML models in a systematic manner.

The following sections discuss potential implementation approaches and derive corresponding decisions on how the individual components of the framework shall be realized or enhanced. Concretely, we analyze and select reverse engineering strategies, new ARs to improve the refactoring possibilities, potential algorithms for the structured service decomposition integration, and we evaluate a library to realize a graphical strategic Domain-driven Design (DDD) Context Map generator.

3.1 Context Map and Bounded Context Discovery Strategies

With the discovery library we want to simplify the usage of the framework for users who work on existing projects by providing a reverse engineering functionality. The component shall generate CML models containing Bounded Contexts and Context Maps from existing source code. By analyzing multiple projects and their dependencies, the Context Map for existing (micro-)service-oriented projects shall be generated. For projects with existing monolithic systems, the library must be able to generate a Bounded Context definition of a system which can then be analyzed and decomposed within the framework towards a (micro)-service-oriented architecture.

3.1.1 Discovery Strategies

The discovery library must support different languages and technologies in order to gain many users. Moreover, microservice architectures typically consist of services implemented with different technologies. It is therefore not sufficient to concentrate on one programming language and/or technology. The library must be extensible and support different implementations of «discovery strategies». Based on different languages and technologies these discovery strategies shall find Bounded Contexts and their relationships so as to generate the CML model for the Context Mapper tool [15].

With this thesis we conceptualize the basic design for the library implementation and aim to provide example discovery strategies as a «proof of concept». In order to generate a CML model we must discover (micro-)services that are mapped to Bounded Contexts and their relationships that lead to a Context Map. If the user wants to decompose the resulting model with decomposition approaches such as Service Cutter [36] or ARs later, the tool must discover the Aggregates and domain objects (Entities, Value objects, etc.) within the Bounded Contexts. Only by discovering the Bounded Context on that level of detail, systematic service decomposition approaches are able to analyze the coupling between the Bounded Contexts and the cohesion within them. Generally, we distinguish between the following two discovery mechanisms:

1. **Bounded Context Discovery**: Simple discovery strategies can just discover Bounded Contexts without the containing domain model, since this is sufficient to produce a Context Map. More advanced strategies have to reverse engineer the domain model within the Bounded Contexts and generate the Aggregates and domain objects in CML.

2. **Relationship Discovery**: Discover relationships between the Bounded Contexts or services.

As part of our research we analyzed open source projects developed in a microservices architecture on the basis of the list provided by Taibi [92][1]. Based on the research results we derived potential strategies how to realize these two mechanisms. We evaluated which programming languages, technologies and frameworks are used within all these projects and how Bounded Contexts and Context Maps could be discovered accordingly. Table 3.1 lists potential approaches to discover Bounded Contexts (1) based on this evaluation.

TABLE 3.1: Potential Bounded Context Discovery Approaches

#	Approach	Description
1	Framework-based discovery	With this approach, services can be detected by the framework used to implement them. For example: Microservices implemented in Java often use the Spring Boot [71] framework. By searching for their *@SpringBootApplication*[2] annotation it is possible to detect such services. To reverse engineer Aggregates and domain objects the strategy could search for *@RequestMapping*[3] annotations and discover RESTful HTTP endpoints. From the DTOs [28] within these endpoints it is possible to derive a domain model. However, this approach needs a strategy for each framework which shall be supported to discover Bounded Contexts.

[1] `https://github.com/davidetaibi/Microservices_Project_List` (State: October 5, 2019)

[2] `https://docs.spring.io/spring-boot/docs/current/reference/html/using-spring-boot.html`

[3] `https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html`

TABLE 3.1: Potential Bounded Context Discovery Approaches (continued)

#	Approach	Description
2	Deployment descriptor based discovery (e.g. *Docker* [20])	Having services that are independently deployable is one major characteristic of a microservice architecture [78]. Richardson further recommends to deploy one service per Docker container [77, 80]. According to our analysis of the mentioned microservices list, these patterns are followed by the community and many projects create one *Dockerfile* [20] per (micro-)service. Thus, we could simply detect those services and our Bounded Contexts by searching for these files. However, this strategy does not allow to reverse engineer the domain model of the Bounded Context.
3	Custom annotation based discovery	Letting the user annotate the objects which should be discovered would be another approach. By providing a library with annotations which must be placed within the code of the application it would be possible to detect Bounded Contexts, Aggregates, and domain objects easily. For example, we could provide annotations such as *@BoundedContext* and *@Aggregate*. If the users annotate their corresponding classes, we could scan the code for these annotations. Marker interfaces such as the ones used by the Lakeside Mutual project[4] [61] would be another possible solution. However, the additional effort required by the user of the tool is a huge disadvantage of this approach.

Besides the discovery of Bounded Contexts we analyzed how their relationships could be reverse engineered. This is a more challenging problem and in some cases impossible to tackle by analyzing code. Table 3.2 lists potential approaches.

TABLE 3.2: Potential Context Map Discovery Approaches

#	Approach	Description
1	Framework configuration based discovery	Many microservice projects implemented with frameworks such as Spring Boot [71] configure the endpoints of other services within their configuration files (at least for the development environment where they all run on the same machine). These are typically URLs containing the port allowing to access the other service. Analyzing all the property, XML, or YAML files and derive the relationships between the services by using these endpoints would be a possible approach.

[4] `https://github.com/Microservice-API-Patterns/DDD-Library`

TABLE 3.2: Potential Context Map Discovery Approaches
(continued)

#	Approach	Description
2	Deployment descriptor based discovery (such as *docker-compose* files [20])	Our analysis of the open source microservice projects has shown that many projects using Docker [20] also use Docker Compose (*docker-compose.yml* file). Compose [19] is a tool which allows to configure how all containers of the application shall be deployed and to start them all together with one command. Such a configuration contains the dependencies between the individual containers (services) and therefore allows to derive the relationships between them.
3	No discovery	Lastly, there are many projects which use service discovery tools such as kubernetes[5], Consul[6] or Eureka[7]. In these cases other services are simply resolved by a name which is often configured somewhere in the source code of the microservice. This makes it practically impossible to detect the relationships from the codebase. A dynamic discovery at runtime goes beyond the scope of this work. Context Mapper users with such a case will only be able to reverse engineer the Bounded Contexts. The relationships must be modeled manually.

Although the discovery component must allow us to implement all possible approaches, we can only provide a «proof of concept» with this thesis project. In the following we discuss our selection criteria and the selected approaches for the implementation of our prototype.

3.1.2 Discovery Strategy Selection

To limit the scope of the project and the amount of discovery strategies to be implemented we defined the following selection criteria:

1. *Feature Coverage*: The prototype should implement discovery strategies for both cases, Bounded Contexts and Context Map. At least one Bounded Context discovery strategy shall reverse engineer Aggregates and domain objects.

2. *Representativeness*: Choose technologies which are popular and often used to implement microservices.

3. *Usability*: The strategies should reduce the effort needed from the user to a minimum. The goal of this framework is to simplify the application of the tool for users in brownfield projects. Strategies which still need manual work lower the benefits we aim for.

[5]`https://kubernetes.io`

[6]`https://www.consul.io/`

[7]`https://github.com/Netflix/eureka`

Based on the selection criteria listed above we decided to include the following discovery strategies in our prototype:

1. *Bounded Context discovery*: Use Spring Boot [71] annotations to detect the Bounded Contexts, Aggregates, and domain objects. The Aggregates and domain objects shall be discovered by using the request mapping annotations and the DTOs [28] in the corresponding methods.
2. *Relationship discovery*: Use *docker-compose* [19] files and their dependency declarations to find relationships between the Bounded Contexts.

With this selection we fulfill the criterion *feature coverage (1)* since it allows us to detect Bounded Contexts, Context Map relationships, and Aggregates with domain objects inside a Bounded Context. With the decision for the Spring Boot [71] framework we implicitly choose Java to be the supported programming language in our prototype. The language fulfills the criterion *representativeness (2)* with respect to our analysis of open source projects implementing a microservice architecture (73 percent). Furthermore, Java is the most popular programming language in 2019 [94]. 78 percent of more than 40 analyzed microservice applications [92] use Spring Boot or contain at least one service implemented using the framework. It is a very popular framework among Java developers and therefore fulfills the *representativeness (2)* as well. Finally, we avoided approaches which require additional effort from the users to fulfill the criterion *usability (3)*.

3.1.3 Merging Heuristics

The discussed approach of separating Bounded Context and Context Map (relationships) discovery requires some heuristics to match divergent Bounded Context names. For example, the Bounded Context name derived from a Spring Boot application is probably not equal to the container name mentioned in the *docker-compose* file. Since relationships involve knowledge about Bounded Contexts as well, the two strategies have to share information and heuristics must be applied to match the different names for the Bounded Contexts. The same is true if multiple Bounded Context discovery strategies are involved and duplicates have to be merged. However, we consider this issue to be a design decision and discuss the implemented solution in Chapter 4.

After an architecture model written in CML exists, a user may want to evolve and improve the described architecture. This is where the two Bounded Contexts or framework components for «Structured Service Decomposition» and «ARs» according to the Context Map in Figure 3.1 come into play. These framework parts support the user in evolving the decomposition based on DCs or by applying refactorings iteratively. The next section discusses the integration of Service Cutter [35] into Context Mapper [15] as our approach towards systematic service decomposition.

3.2 Structured Service Decomposition Algorithms

With this framework component a CML model shall be analyzed with respect to DCs to suggest new service decompositions automatically. As part of our first term project [46] we already generated the input files for the Service Cutter tool [35]. Service Cutter [36] generates service decompositions on the basis of 16 coupling criteria [37] and graph clustering algorithms. Not many other structured and systematic service decomposition tools such as Service Cutter exist. We discuss corresponding related work in Chapter 6. However, there are no other approaches which provide an open source tool that are publicly available besides Service Cutter.

Apart from the integration of the Service Cutter engine into our framework, we researched whether there are new graph clustering algorithms available which could enhance Service Cutter's algorithm support. The tool currently includes the «Epidemic Label Propagation» algorithm by Leung [56] and the algorithm of Newman and Girvan [63]. Since the implementation of the algorithm of Newman and Girvan is part of the Gephi tool [5] and licensed under the GNU General Public License Version 3 (GNU GPLv3) we can not integrate it into Context Mapper. It does not fulfill our licensing requirements documented in Chapter 2. The algorithm of Leung [56] is therefore the only one already integrated in Service Cutter which can be used in Context Mapper.

3.2.1 Algorithms Evaluated by Gysel et al. [34]

First we checked the algorithms which were already evaluated by Gysel et al. [34] at the time they realized Service Cutter. We re-evaluated whether there are new implementations of algorithms which could not be integrated at that time. Table 3.3 lists these graph clustering algorithms and our research results. The criteria used to assess the algorithms are listed below:

- *Distinct clusters* [34]: Every nanoentity is contained once and only once in a cluster. This is a criterion already used by Gysel et al. [34]. An integration into the Service Cutter implementation would be difficult if this criterion is not fulfilled.
- *Java implementation*: Our framework and tools are written in Java. To be able to integrate the algorithm easily we require a Java implementation correspondingly.
- *License*: The implementation license of the algorithm must fulfill our requirements mentioned in Chapter 2 and be compatible with the Apache 2 license. GNU General Public License (GNU GPL) licenses are ruled out.

TABLE 3.3: Graph Clustering Algorithms Evaluated by Gysel et al. [34]

Algorithm	**Candidate**	**Research Results**
Girvan-Newman [63]	no	We checked if another implementation of the algorithm of Newman and Girvan exists but have found none.

TABLE 3.3: Graph Clustering Algorithms Evaluated by Gysel et al. [34] (continued)

Algorithm	Candidate	Research Results
Markov Cluster Algorithm (MCL) [21]	no	The implementation in Gephi [5] still seems to be the only stable one written in Java. Since GNU GPLv3 is no option we cannot integrate this algorithm.
Highly Connected Subgraphs (HCS) [39] and CLICK Clustering Algorithm [84]	no	For these two algorithms there are still no Java implementations available.
K-mean algorithm [38]	no	As already mentioned by Gysel et al. [34], there is no simple way to transform the problem from a graph to a vector-based representation available.
Epidemic Label Propagation [56]	yes	This algorithm by Leung [56] is already integrated into Service Cutter.
Apiacoa[8] [81]	no	This algorithm could work for our problem. However, the authors do not mention any licensing information and the project is not open source (only binaries available). It is therefore difficult to evaluate if the library is stable and reliable.

3.2.2 Other Potential Graph Clustering Algorithms

The research results in Table 3.3 show that there are no new candidates among the algorithms already assessed by Gysel et al. [34]. As our assessement led to the same results, we did some research to find new graph clustering algorithm implementations not yet considered by Gysel et al. Table 3.4 lists the potential algorithms and candidates as a result of this research activity.

TABLE 3.4: Potential new Graph Clustering Algorithms for Service Cutter [35]

Algorithm	Candidate	Research Results
Chinese Whispers[9] [6]	yes	«Chinese Whispers» is a randomized graph-clustering algorithm which seems to fulfill our requirements. The implementation[9] is provided in Java and licensed under Apache 2.
CFinder [13]	no	CFinder is a tool to find clusters and communities within networks. The approach is however not suitable since they find *overlapping* and non-*distinct* groups of nodes.

[8] `https://apiacoa.org`

[9] Implemented in `https://github.com/nlpub/watset-java`

TABLE 3.4: Potential new Graph Clustering Algorithms for Service Cutter [35] (continued)

Algorithm	Candidate	Research Results
Louvain and Leiden[10] [95]	yes	The community detection algorithms Louvain and/or Leiden seem suitable for solving the problem we have and the Java implementation[10] is licensed under the MIT license. The MIT license is very permissive and would be compatible with our Apache 2 license.
Power Iteration Clustering[11] [57]	no	The «Power Iteration Clustering» algorithm by Lin and Cohen [57] seems to be a suitable solution for the problem. However, the only implementation is provided by Apache Spark[11] which is a huge framework coming with many dependencies. We would rather avoid integrating such a huge framework into our tool.

As Table 3.4 shows, we found two algorithm implementations which can be tested within Service Cutter. Nevertheless, during the progress of this thesis project we decided to integrate the current state of Service Cutter with the algorithm of Leung [56]. The approach can be validated with the existing algorithm. If the validation activities confirm that this is a good approach, new algorithms can be integrated to improve the component later. Despite this, implementing our own algorithm with a slightly different approach could turn out to be a better solution as the next section explains.

3.2.3 Implementing own Approach

Regarding this specific framework component more validation results have to be conducted in future projects. The integration of Service Cutter as explained later in Chapter 4 is a first step towards structured and systematic service decomposition in Context Mapper. Nonetheless, other approaches besides generating completely new service decompositions might have to be evaluated as well. A new algorithm could just suggest small extraction steps by identifying parts within a model which should build a new Bounded Context with respect to coupling and cohesion. This would allow us to provide refactoring suggestions and guide the user in the process of decomposing the system iteratively. Such an approach might be more consistent with agile practices [1] and the results would probably be more comprehensible for the users.

3.3 Architectural Refactoring (AR) Selection

As documented in the requirements in Chapter 2 the set of ARs has been extended. They support the user in evolving and improving the architecture

[10]Implemented in `https://github.com/CWTSLeiden/networkanalysis`

[11]Implemented in `https://spark.apache.org`

model iteratively. For example on the basis of the knowledge gained through decompositions generated with Service Cutter [36] as described in the previous section, the architect can change the model by applying ARs. In our previous work [50] we already proposed seven ARs to *split* Bounded Contexts, *extract* Aggregates, and *merge* Bounded Contexts or Aggregates. Figure 3.2 shows these existing refactorings.

Our validation activities and the received user feedback revealed that additional ARs to refactor Context Map relationships would be useful.

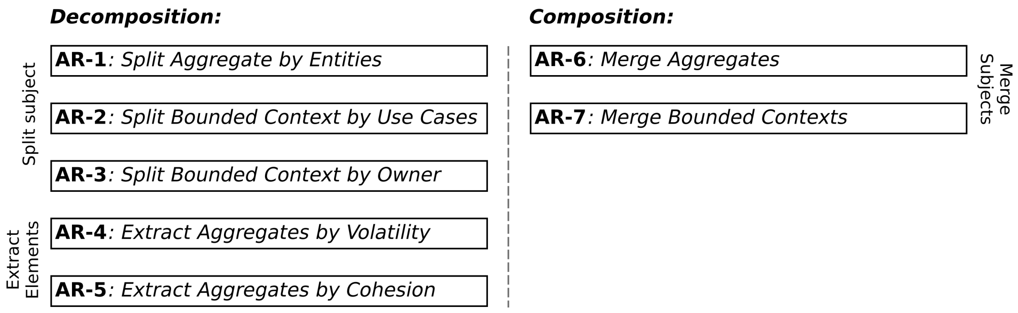

Decomposition:
Composition:
Split subject
Extract Elements
Merge Subjects
AR-1: Split Aggregate by Entities
AR-2: Split Bounded Context by Use Cases
AR-3: Split Bounded Context by Owner
AR-4: Extract Aggregates by Volatility
AR-5: Extract Aggregates by Cohesion
AR-6: Merge Aggregates
AR-7: Merge Bounded Contexts

FIGURE 3.2: Existing ARs organized by *decomposition vs. composition* and their operations *split*, *extract*, and *merge* [50].

Within the next sections we document potential candidates and the selection of ARs to be realized as part of this thesis.

3.3.1 New AR Categorization

In regard to ARs with Context Map relationships as subjects we identified two different kinds of refactorings. Simple ARs can change just the type of a relationship without altering the structure of the decomposition (Bounded Contexts). For example, changing a generic Upstream-Downstream relationship to a Customer-Supplier relationship has no impacts regarding structure. More complex relationship ARs on the other hand can change the structure of the decomposition. For example, if the goal of a refactoring is to remove a Shared Kernel relationship, the domain model of the Shared Kernel can lead to a new Bounded Context. Thus, the refactorings in our modeling framework and in Context Mapper are now categorized into the following two categories:

- **Structural refactorings**: These ARs change the structure of the decomposition.
- **Relationship refactorings**: ARs of this category change the type of a relationship only. The structure of the decomposition is not changed.

The existing refactorings illustrated in Figure 3.2 are all *structural refactorings*.

3.3.2 AR Candidates

We derived candidates for new relationship ARs from the available user feedback and our own experience with the current version of the Context Mapper tool. The following Table 3.5 lists *structural* refactoring candidates first.

TABLE 3.5: Structural AR Candidates for Context Map Relationships

AR Name	Parameters	Description
Extract Shared Kernel	Reference to Shared Kernel relationship	Removes the Shared Kernel relationship, creates a new Bounded Context for the Shared Kernel, and adds two new Upstream-Downstream relationships (with the new context as upstream and the existing two contexts as downstream).
Inline Bounded Context to Shared Kernel	Reference to Bounded Context	This would be the inverse refactoring to *Extract Shared Kernel*. Instead of extracting the Shared Kernel into a new Bounded Context, this AR would create a Shared Kernel from an existing Bounded Context that is upstream within two Upstream-Downstream relationships.
Suspend Partnership	Reference to Partnership relationship	The Partnership pattern describes an intimate relationship requiring coordination with respect to planning and development between the two involved teams. In order to achieve increased autonomy between the individual teams, which is often an important goal when applying strategic DDD [96], it may be desirable to eliminate a Partnership. This AR shall be able to suspend a Partnership by offering three options: a) merge the Bounded Contexts, b) extract a new Bounded Context with common parts and establish Upstream-Downstream relationships, or c) move commonalities to one of the two Bounded Contexts and create an Upstream-Downstream relationship between the two.
Inline Bounded Context to Partnership	Reference to Bounded Context	Analog to *Inline Bounded Context to Shared Kernel* this would be the inverse refactoring to option b) of *Suspend Partnership*. Instead of extracting the Partnership's commonalities into a new Bounded Context, this AR would create a Partnership from an existing Bounded Context which is upstream within two Upstream-Downstream relationships.

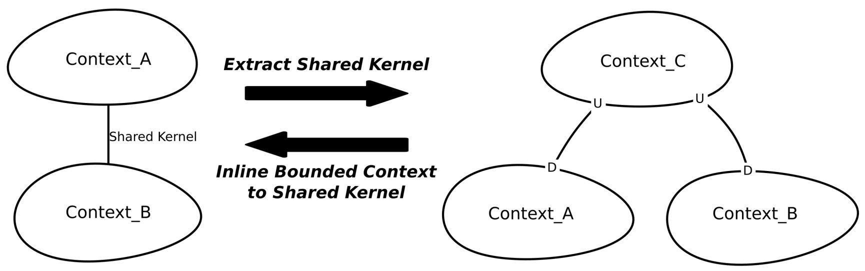


FIGURE 3.3: *Extract Shared Kernel* and *Inline Bounded Context to Shared Kernel*

Figure 3.3 and Figure 3.4 illustrate the idea behind the described AR candidates in Table 3.5 graphically. If the user selects option b) in *Suspend Partnership*, the two scenarios with *extract* and *inline* are basically the same for the Shared Kernel and the Partnership case.

The cases a) and b) of *Suspend Partnership* are not illustrated graphically at this point since they are trivial (illustration available in AR summary later in this chapter). Case a) would lead to the same result as applying the already existing AR *Merge Bounded Contexts*. Case c) on the other hand would simply replace the Partnership relationship with an Upstream-Downstream relationship.

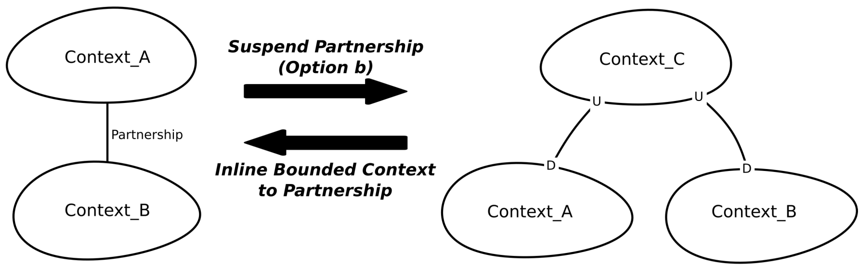

Context_A
Partnership
Context_B
Suspend Partnership (Option b)
Inline Bounded Context to Partnership
Context_C
U
U
D
D
Context_A
Context_B

FIGURE 3.4: *Suspend Partnership* and *Inline Bounded Context to Partnership*

These AR candidates would not only transform Context Map relationships but also change the structure of the decomposition as Figure 3.3 and Figure 3.4 clearly illustrate. Thus, they would belong to our *structural refactorings* category. Table 3.6 on the other hand presents *relationship refactorings* which do not change the decomposition.

TABLE 3.6: Relationship AR Candidates

AR Name	**Parameters**	**Description**
Change Shared Kernel to Partnership	Reference to Shared Kernel	This AR changes the type of a relationship from Shared Kernel to Partnership.
Change Partnership to Shared Kernel	Reference to Partnership	This AR changes the type of a relationship from Partnership to Shared Kernel.
Change Upstream-Downstream to Customer-Supplier	Reference to Upstream-Downstream relationship	This AR changes the type of a relationship from Upstream-Downstream to Customer-Supplier. Note that the AR has to ensure that there are no conflicts with the relationship patterns and our semantic rules [50]. For example, the OHS pattern is not allowed in a Customer-Supplier relationship according to our semantics.

TABLE 3.6: Relationship AR Candidates (continued)

AR Name	Parameters	Description
Change Customer-Supplier to generic Upstream-Downstream	Reference to Customer-Supplier relationship	This AR changes the type of a relationship from Customer-Supplier to a generic Upstream-Downstream relationship.

Although the refactorings listed by Table 3.6 may appear to be simple, the productivity of our modeling framework users can still be improved if such minor changes are automated. Transforming the relationships manually can be error-prone and take more time. Additional refactorings could be proposed to change the relationship patterns (OHS, PL, Anticorruption Layer (ACL), CF) and increase the refactoring support even more. Since the amount of refactorings which we could implement during this thesis project was limited, we had to concentrate on the refactorings listed above. Adding and removing the mentioned relationship patterns in CML is quite simple, even when done manually.

3.3.3 AR Selection

Because our implementation capacities were limited we had to select three to four refactorings for our prototypic implementation in Context Mapper. Other refactorings could be realized in future projects. The following criteria determined our AR selection.

- *Relevance in practice*: ARs which are likely to be applied more often than others should be prefered.
- *Generality*: Both our AR categories shall be covered. In addition, if the selected ARs for our prototype have been implemented, the remaining ARs can be realized in a similar way.
- *Usability*: We chose ARs which increase the usability for users the most. This means that we realize those ARs for which the manual changes are the most complex.

According to the criteria listed above we decided to realize the following ARs as part of this thesis project:

- AR-8: Extract Shared Kernel (*structural*)
- AR-9: Suspend Partnership (*structural*)
- AR-10: Change Shared Kernel to Partnership (*relationship*)
- AR-11: Change Partnership to Shared Kernel (*relationship*)

Considering that our scope allowed us to choose three to four ARs it made sense to realize at least two structural refactorings due to the higher complexity. The relationship refactorings are all similar with respect to implementation (*generality*). Out of the structural candidates in Table 3.5 we decided to realize *Extract Shared Kernel* and *Suspend Partnership* and not the corresponding inverses. This decision is based on the *relevance in practice* criterion. Since the Shared Kernel und Partnership relationships are very close relationships and the coupling between these contexts is typically higher in comparison to Upstream-Downstream relationships, we believe that it is more likely that architects want to reduce this kind of relationships rather than introduce them. These two refactorings tend to reduce coupling whereas the inverses do the opposite.

The selection of relationship refactorings out of the candidates in Table 3.6 is more difficult since we assessed them all equally with respect to the criteria mentioned above. We finally selected AR-10 and AR-11 since they refactor Shared Kernels and Partnerships as well. The selection results in a suitable set of ARs which offers more refactoring possiblities for the user. Especially, changing a Shared Kernel to a Partnership relationship opens the possibility to apply all three options (a, b and c) of *Suspend Partnership* to get rid of the tight relationship and reduce the coupling. Overall we assumed that removing tight relationships and therefore refactor Shared Kernels and Partnerships suits both the *relevance in practice* as well as the *usability* criterion.

3.3.4 AR Summaries

We finish this section about the ARs by providing summaries of the selected ones which have been implemented as part of this thesis. Note that the pattern summaries providing context, motivation, solution, and effects of all ARs in Context Mapper can be found online[12] as well as in Appendix B.

AR-8: Extract Shared Kernel

Context: A Shared Kernel describes an intimate relationship in which two Bounded Contexts share a part of their domain model. This shared model part is typically implemented in a shared library used by both Bounded Contexts. This kind of relationship leads to a higher coupling in comparison with other relationship types such as Upstream-Downstream.

Motivation: A Shared Kernel leads to interdependencies between two teams and may come with undesired coupling. Changes within the shared model parts may influence both development teams in the relationship. If we strive for clear responsibilities [74] and team autonomy [64], we may want to reduce the coupling between teams. This AR can be applied in such a case if the Shared Kernel reaches a size where the common model part can also be seen as a separate Bounded Context.

Solution and Effect: If the Shared Kernel model part between two teams is getting big and costly to maintain, it might be a solution to build a separate Bounded Context and team for this domain model. This AR creates a new Bounded Context for the Shared Kernel and establishes Upstream-Downstream relationships between the new context and the existing ones. The resulting Upstream-Downstream relationships replacing the Shared Kernel may improve

[12]`https://contextmapper.org/docs/architectural-refactorings/`

the coupling between the contexts and the cohesion within them. Figure 3.5 illustrates the transformation.

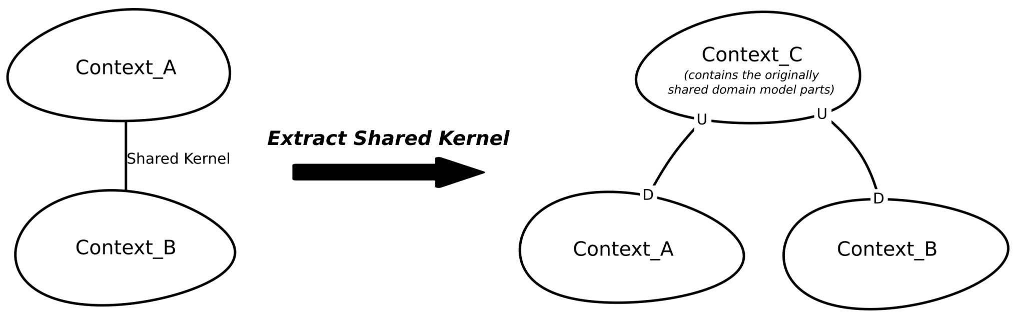


FIGURE 3.5: AR-8: Extract Shared Kernel

Inverse ARs: The inverse AR to *AR-8: Extract Shared Kernel* has not been implemented yet. Previously in this chapter we briefly described how the inverse refactoring would work.

AR-9: Suspend Partnership

Context: A Partnership relationship describes another intimate relationship between two Bounded Contexts and/or development teams. In comparison to the Shared Kernel however, it does not have to be the case that the teams share parts of their domain models. The intimacy in a Partnership is defined by interdependent features and a joint management of planning and integration. The organizational and feature-related interdependencies between the two teams leads to a situation where both teams can either only fail or succeed together. New developments and releases must always be coordinated between the teams.

Motivation: The interdependencies between the two teams in a Partnership may lead to an undesired coupling and increased inter-team coordination and communication to keep the product stable. To develop clear responsibilities [74] and team autonomy [64] it might be necessary to decouple the teams and suspend the Partnership. This AR can be applied if the coupling between two teams is getting cumbersome and the Partnership must be suspended.

Solution and Effect: If the coupling between two teams in a Partnership relationship must be reduced the *Suspend Partnership* refactoring can be applied. This AR offers three options to get rid of the Partnership:

(a) Merge the two Bounded Contexts: if the teams are small enough and the coupling very high, merging the Bounded Contexts might be the right solution. This option corresponds to *AR-7: Merge Bounded Contexts*.

(b) Extract a new Bounded Context for tightly coupled model parts and establish Upstream-Downstream relationships: if the Partnership is mainly defined by a common part of the domain models, the Partnership can be suspended in the same way as *AR-8: Extract Shared Kernel* works.

(c) Replace the Partnership with an Upstream-Downstream relationship: another solution might be that one of the two teams takes over the responsibilities of the common or highly coupled parts. This way one can establish an Upstream-Downstream relationship. Note that this option may move responsibilities and reduce the influence of the new downstream team.

Figure 3.6 illustrates the three possible solutions to suspend a Partnership offered by this AR.

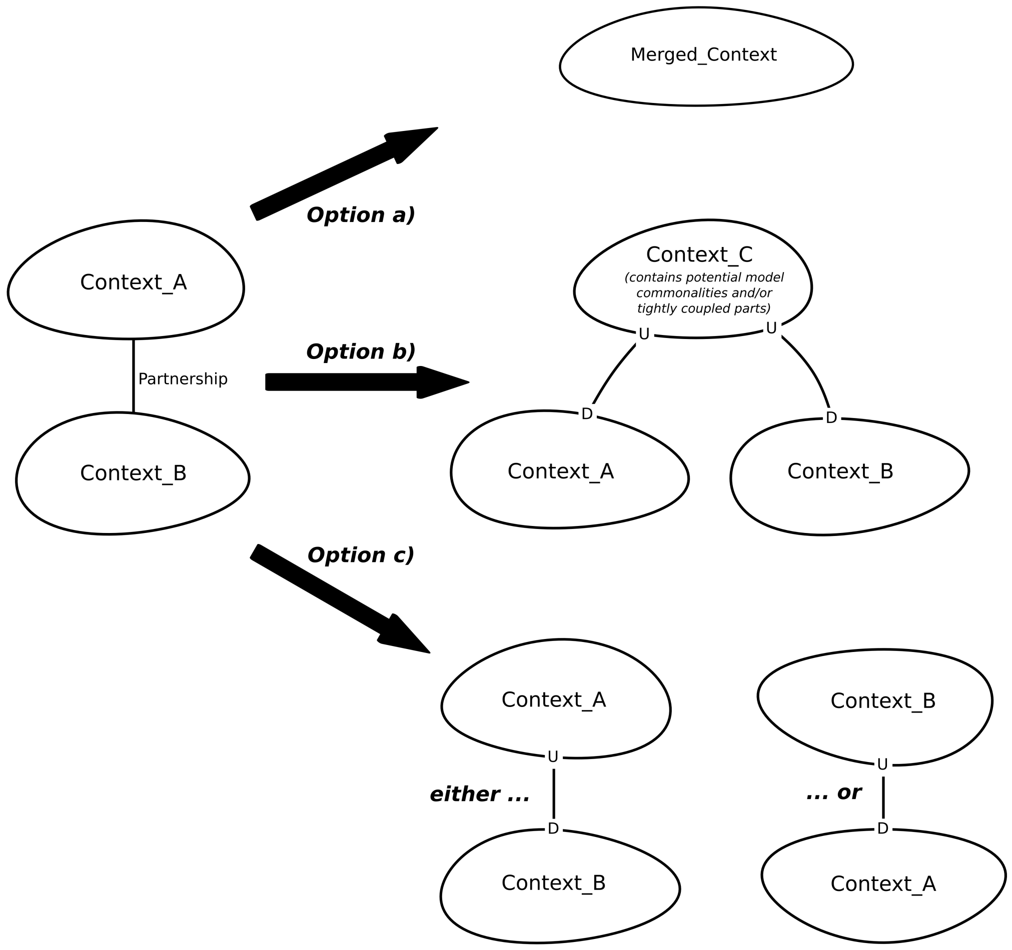


FIGURE 3.6: AR-9: Suspend Partnership

Inverse ARs: Although we briefly described the inverse AR to *AR-9: Suspend Partnership* previously in this chapter, it has not yet been implemented.

Since the *relationship* refactorings are very simple in comparison to the structural ones, we only provide short summaries for them.

AR-10: Change Shared Kernel to Partnership

Our relationship refactorings allow the user/modeler to change the type of a relationship on a Context Map easily without manual work. The symmetric relationships according to our semantic model [46], Shared Kernel and Partnership, are interchangeable without impacts to the structure of the decomposition. This refactoring changes a Shared Kernel relationship to a Partnership relationship.

Inverse ARs: The result of *AR-10: Change Shared Kernel to Partnership* can be inverted by applying *AR-11: Change Partnership to Shared Kernel*.

AR-11: Change Partnership to Shared Kernel

Similar to AR-10, Shared Kernels and Partnerships are interchangeable without impacts to the structure of the decomposition. This refactoring changes a Partnership relationship to a Shared Kernel relationship.

Inverse ARs: The inverse AR of *AR-11: Change Partnership to Shared Kernel* is *AR-10: Change Shared Kernel to Partnership*.

After having conceptualized the additional ARs proposed by this thesis, we will discuss graphical tools for the Context Map generator in the next section and decide which one shall be used in the prototypic implementation.

3.4 Graphical DDD Context Map Tool Evaluation

The modeling framework has been extended by a Context Map generator that transforms a CML Context Map into a graphical representation. We already created graphical Context Maps to illustrate the examples in our previous work [46] and in our examples repository[13] manually. Figure 3.7 shows the Context Map of our fictitious insurance example [46].

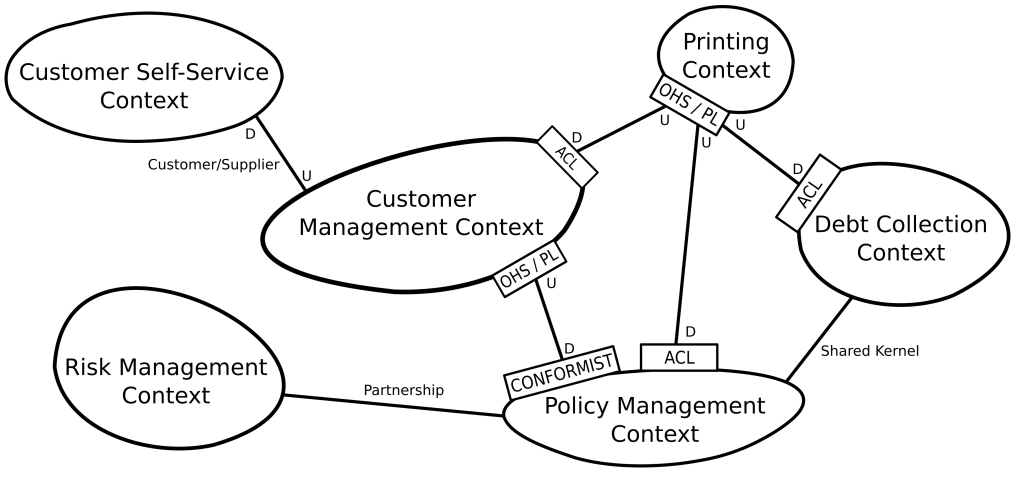


FIGURE 3.7: Graphical Context Map Example [46]

This graphical representation of a Context Map is inspired by the illustrations of Vernon [98] and Brandolini [10]. The Context Map generator of our DDD modeling framework produces such graphical representations automatically. A graphical library or tool had to be evaluated in order to realize the generator.

[13]https://github.com/ContextMapper/context-mapper-examples

3.4.1 Tool Evaluation

Since a Context Map such as the one illustrated in Figure 3.7 is basically a graph and one of our objectives was to avoid dealing with layout algorithms, we focused on graph visualization tools. Some visualization and diagram tools require to create the layout manually. Implementing our own layout algorithm would have exceeded the scope of this thesis. Table 3.7 lists the potential tool candidates for implementing our generator. Since Context Mapper [15] is implemented in Java we only list libraries which provide an Application Programming Interface (API) for this programming language.

TABLE 3.7: Graph Visualization Tools

Tool	Description	License
yFiles[14] [101]	yFiles provides a set of powerful diagramming libraries which allow to create diagrams with the help of automatic layout algorithms. Besides Java it supports the creation of their diagrams with .NET and HTML.	Commercial
Gephi[15] [5]	Gephi is an open source tool for visualization and exploration of graphs. Its goal is to help data analysts and scientists to analyze graph-based data.	CDDL-1.0 and GNU GPLv3
Graphviz[16] [30]	Graphviz is an open source graph visualization software. In comparison with others it is not only used to vizualize simple graphs and networks, it also has applications in software engineering tools. PlantUML [72] is one example using Graphviz to create its UML diagrams. Other architecture visualization tools such as Structurizr [91] export diagrams to the Graphviz format as well.	CPL-1.0
JUNG[17] [44]	JUNG provides a language for the modeling, analysis, and visualization of data that can be represented as a graph or network.	BSD license
Prefuse[18] [40]	Prefuse calls itself a software tool for creating rich interactive data visualizations in the Java programming language. It supports data structures such as tables, graphs, and trees. It further provides layout and visual encoding techniques.	BSD license

Note that there are many more libraries for web technologies and browser applications which we have not listed due to our Java context. We further omitted

[14]https://www.yworks.com/products/yfiles
[15]https://gephi.org/
[16]http://www.graphviz.org/
[17]http://jung.sourceforge.net/
[18]https://github.com/prefuse/Prefuse

tools which only support graph representations in a mathematical sense and do not provide the needed graphical features.

We finally decided to use the Graphviz [30] tool based on *graphical features*, *future-orientation*, and *licences* as our main decision drivers. yFiles [101] would be a very powerful alternative but it is a commercial tool. Since we work on an open source tool and want to use open libraries, yFiles is not an option. Gephi's [5] features focus on network visualization and offer less graphical features for diagrams like we aim to produce in comparison with Graphviz. JUNG [44] and Prefuse [40] both seem to be inactive projects with the last releases in 2010 and 2007. They do not fulfill our requirements regarding *future-orientation* (also see Non-Functional Requirements (NFRs) in Chapter 2).

Graphviz is a very active project, licenced under the *The Common Public License 1.0*[19], and offers enough features to generate a Context Map which at least comes close to the graphical representation in Figure 3.7. It is further used by many other open source software engineering tools, one of which is PlantUML [72]. In addition, its *DOT language*[20] is supported by other visualization tools and web-based graph editors.

Within this chapter we analyzed the individual Bounded Contexts (components) of our modeling framework, researched potential realization strategies, and documented the required design and scope decisions for the implementation of the prototypes in Context Mapper. The next chapter discusses the design and implementation of the individual components in Context Mapper in detail. It further explains how the interfaces between the framework components are designed and implemented.

[19]`http://www.graphviz.org/license/`
[20]`https://www.graphviz.org/doc/info/lang.html`

Chapter 4

Context Mapper: Design and Implementation

The Context Mapper open source project[1] [15] represents the prototypic implementation of the modeling framework proposed by this thesis. The Context Mapper DSL (CML) language, which is the core of the modeling framework, has been developed as part of the first term project [46]. In addition, the PlantUML [72] generator and a Service Cutter [35] input file generator were already exising before this thesis. With the second term project [50] we added the first Architectural Refactorings (ARs) and the Microservices Domain-Specific Language (MDSL) generator to the tool.

This chapter discusses all changes and newly implemented framework components which are part of this thesis in detail. It explains the design and implementation of new components and documents improvements realized in existing ones according to our validation activities.

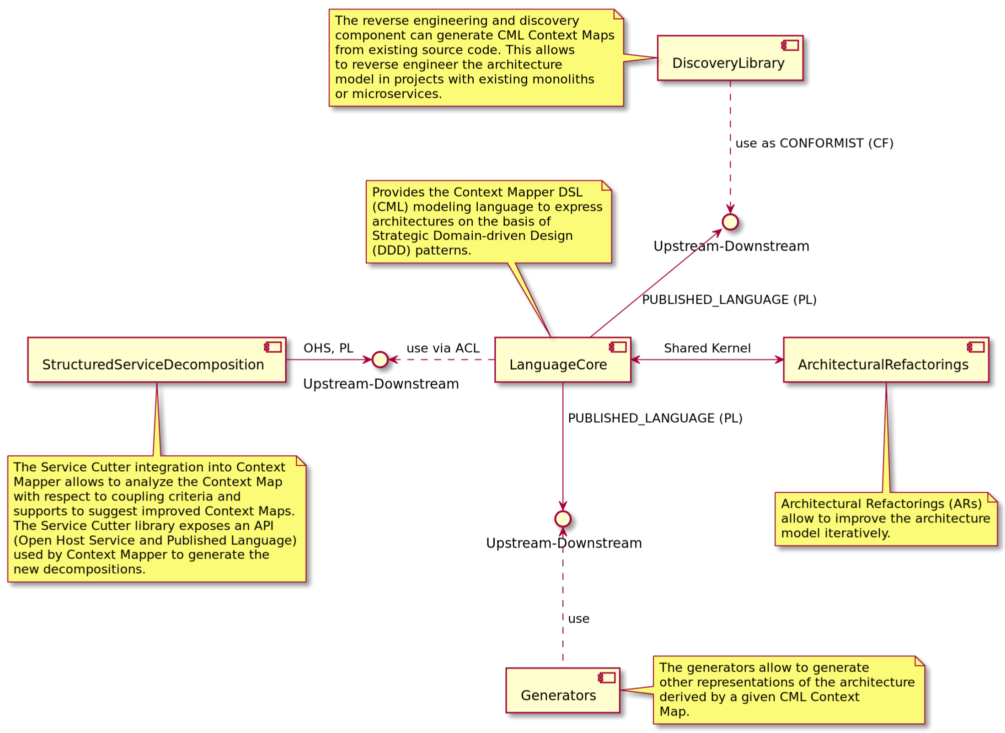
4.1 Component Overview

In Chapter 1, we already provided an overview of the framework components and introduced the functionalities. Before we explain the implementation details of the individual components, Figure 4.1 shows them again as a Unified Modeling Language (UML) component diagram.

The core component exposes the semantic model of the Domain-specific Language (DSL) [27] as a *Published Language (PL)* [24, 25]. The other components use this model to process, generate, or transform CML models. The discovery library generates CML models out of existing source code. It generates code that always *conforms* with the model of the core. The structured service decomposition component based on Service Cutter [36] uses CML models as input and output. It generates other Context Maps (decompositions) of existing models and therefore uses the exposed model of the core to read and write CML models. The ARs which are implemented as model transformations for the DSL [49] are technically tightly coupled to the language model. Thus, the semantic model of the language can be seen as a *Shared Kernel* in this case. The generators consume the exposed model of the core to generate other representations, currently PlantUML [72] diagrams, graphical Context Maps, Service Cutter [35] input files, and MDSL contracts, out of the CML models. While the

[1] `https://contextmapper.org/`

discovery library simply *conforms* to the language model, the structured service decomposition and generator components have minimal influence to the design of the core language.



The reverse engineering and discovery component can generate CML Context Maps from existing source code. This allows to reverse engineer the architecture model in projects with existing monoliths or microservices.
DiscoveryLibrary
use as CONFORMIST (CF)
Provides the Context Mapper DSL (CML) modeling language to express architectures on the basis of Strategic Domain-driven Design (DDD) patterns.
Upstream-Downstream
PUBLISHED_LANGUAGE (PL)
StructuredServiceDecomposition
OHS, PL
use via ACL
Upstream-Downstream
LanguageCore
Shared Kernel
ArchitecturalRefactorings
PUBLISHED_LANGUAGE (PL)
The Service Cutter integration into Context Mapper allows to analyze the Context Map with respect to coupling criteria and supports to suggest improved Context Maps. The Service Cutter library exposes an API (Open Host Service and Published Language) used by Context Mapper to generate the new decompositions.
Architectural Refactorings (ARs) allow to improve the architecture model iteratively.
Upstream-Downstream
use
Generators
The generators allow to generate other representations of the architecture derived by a given CML Context Map.

FIGURE 4.1: Context Mapper Framework Overview
(UML Component Diagram Generated by Context Mapper)

The next sections explain the design and implementation of all modeling framework components as they are realized in the Context Mapper open source project.

4.2 Core: Context Mapping DSL (CML)

Please note that this report does not explain the CML language syntax in detail. We refer to our previous project reports [46, 50], the online documentation[2], and the language reference in Appendix A for syntax details.

Since the semantic model of the language [46] is important for all framework components, we briefly introduce it again. Figure 4.2 illustrates this model behind the CML language. A CML model contains a Context Map with all relevant Bounded Contexts. A Bounded Context implements parts of one or many Subdomains. Moreover, the Context Map models the relationships between the Bounded Contexts. We distinguish between symmetric and Upstream-Downstream (asymmetric) relationships. The Domain-driven Design (DDD) patterns, *Partnership* and *Shared Kernel*, represent symmetric relationships whereas *Customer-Supplier* and *Generic Upstream-Downstream* relationships are asymmetric. The relationship patterns *Published Language (PL)*

[2]`https://contextmapper.org/docs/`

and *Open Host Service (OHS)* can be used on the upstream side in Upstream-Downstream relationships. *Anticorruption Layer (ACL)* and *Conformist* on the downstream side respectively.

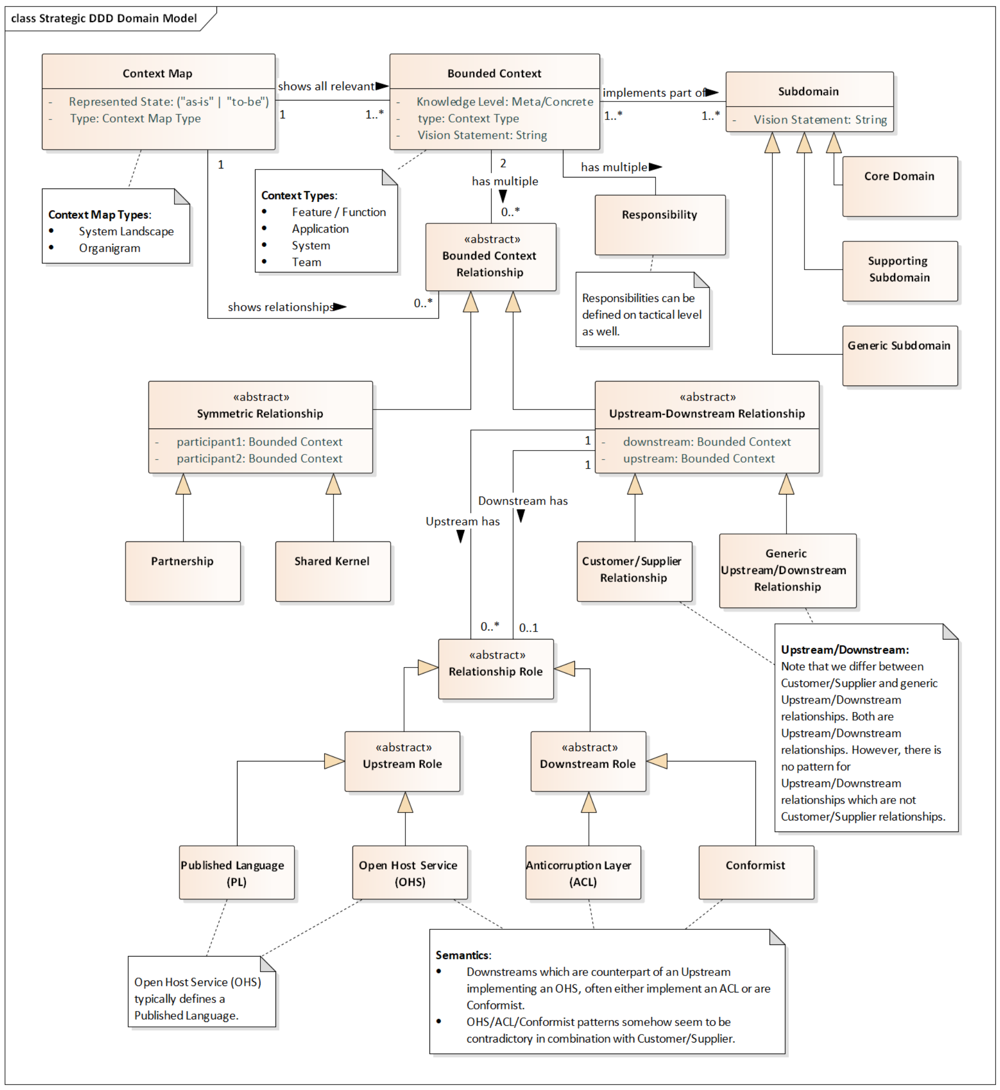

class Strategic DDD Domain Model
Context Map
- Represented State: ("as-is" | "to-be")
- Type: Context Map Type
shows all relevant
1
1..*
Bounded Context
- Knowledge Level: Meta/Concrete
- type: Context Type
- Vision Statement: String
implements part of
1..*
1..*
Subdomain
- Vision Statement: String
Core Domain
Supporting Subdomain
Generic Subdomain
1
2
has multiple
0..*
has multiple
Responsibility
Responsibilities can be defined on tactical level as well.
Context Map Types:
• System Landscape
• Organigram
Context Types:
• Feature / Function
• Application
• System
• Team
«abstract» Bounded Context Relationship
shows relationships
0..*
«abstract» Symmetric Relationship
- participant1: Bounded Context
- participant2: Bounded Context
«abstract» Upstream-Downstream Relationship
1
1
- downstream: Bounded Context
- upstream: Bounded Context
Partnership
Shared Kernel
Downstream has
Upstream has
Customer/Supplier Relationship
Generic Upstream/Downstream Relationship
0..*
0..1
«abstract» Relationship Role
Upstream/Downstream:
Note that we differ between Customer/Supplier and generic Upstream/Downstream relationships. Both are Upstream/Downstream relationships. However, there is no pattern for Upstream/Downstream relationships which are not Customer/Supplier relationships.
«abstract» Upstream Role
«abstract» Downstream Role
Published Language (PL)
Open Host Service (OHS)
Anticorruption Layer (ACL)
Conformist
Open Host Service (OHS) typically defines a Published Language.
Semantics:
• Downstreams which are counterpart of an Upstream implementing an OHS, often either implement an ACL or are Conformist.
• OHS/ACL/Conformist patterns somehow seem to be contradictory in combination with Customer/Supplier.

FIGURE 4.2: Context Mapper DSL (CML) Semantic Model [46]

The language further allows to specify the domain model within a Bounded Context on the basis of the Sculptor [83] syntax. However, we will introduce this part of the language model later.

4.2.1 Language Core Changes

The following Table 4.1 lists all changes to the core component which have been implemented during this thesis. These improvements have been identified by the validation activities during the previous project [50] and action research [4] during this thesis.

TABLE 4.1: Changes in the Core Component (Modeling Language)

#	Change	Description
1	Standalone DSL usage	The DSL is now published as standalone library as well. Therefore it cannot only be used within our Eclipse plugin, but can be integrated in any Java application.
2	Adjusted relationship grammar	The relationship syntax of CML has been adapted so that the *U* for *upstream* and the *D* for *downstream* or complete brackets in Upstream-Downstream relationships can be omitted.
3	Limited scope to single file	The scoping regarding object references in CML models was not correct if multiple files were used (referencing objects in other CML files). Because of this, we limited the scope of a single CML model to references within the same file for now.
4	Refining Bounded Contexts	A new keyword *«refines»* has been introduced which allows to specify that a Bounded Context refines another one (like an inheritance mechanism).
5	Implementing domains	Bounded Contexts can implement a complete domain now. So far the user had to reference a list of subdomains.
6	Domain Vision Statement on domain	Not only subdomains can have Domain Vision Statements, but the surrounding domain as well.
7	Optional equality sign	The equality sign («=») which is used to assign attribute values is optional now.

The following sections explain the two major changes from the list above (*#1 and #2*) in detail. The other minor changes are further documented in the online documentation[3] or the language reference in Appendix A.

Standalone CML Usage (1)

Up to now, we provided the usage of the CML language within our Eclipse plugin only. With this project we have adjusted the publication process so that the language core is published as standalone Java library to the Maven central[4] as well. The library also includes the ARs and generators. Consequently, it is possible to parse, read, and refactor CML models in any Java application. On top of that, it is possible to use our generators with the library. It can also be used as a foundation for the implementation of other Integrated Development Environment (IDE) plugins or Command Line Interfaces (CLIs).

The following Listing 1 illustrates how the standalone library can be used to parse CML code and read data from a model.

[3]`https://contextmapper.org/docs/`

[4]`https://repo1.maven.org/maven2/org/contextmapper/context-mapper-dsl/`

```java
public static void main(String[] args) throws IOException {
  // Setup and loading CML file:
  ContextMappingDSLStandaloneSetup.doSetup();
  Resource resource = new ResourceSetImpl()
    .getResource(URI.createURI("./src/main/cml/Example-Model.cml"), true);
  ContextMappingModel model = (ContextMappingModel) resource.getContents().get(0);

  // We search for a Bounded Context ... (read model)
  BoundedContext customerManagementBC = model.getBoundedContexts().stream()
      .filter(bc -> bc.getName().equals("CustomerManagementContext")).findFirst().get();

  // ... and add a new Aggregate to it (example change to the model)
  Aggregate newAggregate = ContextMappingDSLFactory.eINSTANCE.createAggregate();
  newAggregate.setName("New_Aggregate");
  customerManagementBC.getAggregates().add(newAggregate);

  // unparse model: saving changes in CML file
  resource.save(SaveOptions.defaultOptions().toOptionsMap());
}
```

LISTING 1: Standalone CML Library Usage in Java

The example further shows how one can change the model and serialize the changes back to the CML file. The additional Listing 2 demonstrates how a generator, in this case the PlantUML generator, can be used on a given CML resource.

```java
public static void main(String[] args) {
  // Setup and loading CML file:
  ContextMappingDSLStandaloneSetup.doSetup();
  Resource resource = new ResourceSetImpl()
    .getResource(URI.createURI("./src/main/cml/Example-Model.cml"), true);

  // Create the PlantUML generator
  PlantUMLGenerator generator = new PlantUMLGenerator();

  // Generate the UML diagrams into 'src-gen'
  JavaIoFileSystemAccess javaIoFileSystem = FileSystemHelper.getFileSystemAccess();
  javaIoFileSystem.setOutputPath("./src-gen");
  generator.doGenerate(resource, javaIoFileSystem, new GeneratorContext());
}
```

LISTING 2: Standalone CML Library: Generator Example

An extensive set of examples on how to use CML standalone can be found in our *standalone example project*[5]. It further shows how to use the Xtext [23] build plugins in Gradle[6] or Maven[7] in order to compile and validate CML models during the build of a project.

Adjusted Relationship Grammar (2)

The second major change in the core component is an adjustment of the relationship grammar. Listing 3 illustrates a CML Context Map with an Upstream-Downstream relationship.

[5]https://github.com/ContextMapper/context-mapper-standalone-example
[6]https://github.com/xtext/xtext-gradle-plugin/
[7]https://github.com/xtext/maven-xtext-example/

This example corresponds to the syntax as implemented in our previous projects [46, 50]. The declaration of the relationship is still valid with the current version of CML.

```
ContextMap InsuranceExampleContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  /* Add bounded contexts to this context map: */
  contains PrintingContext
  contains PolicyManagementContext

  /* Define the context relationships: */
  PrintingContext [U,OHS,PL]->[D,ACL] PolicyManagementContext
}
```

LISTING 3: CML: Upstream-Downstream Example

However, users who worked with the language repeatedly mentioned the redundancy of the *upstream* and *downstream* declaration in this syntax. To simplify this example Listing 4 shows the same relationship declaration without the relationship patterns (OHS, PL, ACL).

```
PrintingContext [U]->[D] PolicyManagementContext
```

LISTING 4: CML: Upstream-Downstream Example Simplified

The redundancy in this declaration is given by the fact that the arrow must always point from the *upstream* towards the *downstream*, since it represents the *influence flow* [73]. Although we believe this example may be easier to read for users, it is clear that the declaration does not lose information if the *U* and the *D* are omitted. The arrow already specifies which Bounded Context is the *upstream* and which one is the *downstream*.

Thus, we decided to adjust the grammar and make the *U* and the *D* optional. The declaration in Listing 5 can be parsed as well and is semantically the same declaration as the one in Listing 4.

```
PrintingContext -> PolicyManagementContext
```

LISTING 5: CML: Upstream-Downstream Adjusted Syntax (1)

Note that this declaration requires a higher awareness regarding the meaning of the arrow among new users or model readers. A user who is aware of the meaning of the *U* and *D* was able to derive the meaning of the arrow in the original version in Listing 3. With this adjustment we reduce redundancy but a user must have a clear understanding what the arrow means now. It could be misinterpreted as «*the PrintingContext uses/calls the PolicyManagementContext*», which is not the case here.

If we add the relationship patterns from the original Listing 3 again and still ommit the *U* and *D* we get the declaration in Listing 6.

```
PrintingContext [OHS,PL]->[ACL] PolicyManagementContext
```

LISTING 6: CML: Upstream-Downstream Adjusted Syntax (2)

This situation required another adjustment of the grammar behind Upstream-Downstream relationships since a user may want to add patterns on one side of the arrow only. Therefore, it is now possible to add only one bracket beside the arrow as the examples in Listing 7 illustrate.

```
PrintingContext [OHS,PL]-> PolicyManagementContext

PrintingContext ->[ACL] PolicyManagementContext
```

LISTING 7: CML: Upstream-Downstream Adjusted Syntax (3)

This flexibility made it necessary to limit the possibilities regarding the placement of the brackets, otherwise it would have become too difficult to parse the relationship declarations. Before the adjustment it was possible to place the brackets either on the left side or on the right side of the Bounded Context name. Therefore all alternatives illustrated in Listing 8 were allowed.

```
PrintingContext [U]->[D] PolicyManagementContext

// deprecated:
[U]PrintingContext -> [D]PolicyManagementContext

// deprecated:
PrintingContext[U] -> PolicyManagementContext[D]

// deprecated:
[U]PrintingContext -> PolicyManagementContext[D]
```

LISTING 8: CML: Upstream-Downstream Deprecated Bracket Placements

As the listing points out, only the first alternative with the brackets immediately on the left and/or on the right side of the arrow is still supported. The other options introduced in our previous project [50] no longer compile in our latest releases. The same changes apply to Customer-Supplier relationships.

Note that it is still possible to use the *U* and the *D* within the brackets, as it was supported before. From our perspective it can increase the readability of the models for users which are new to the language.

After the overview over the changes within the core component the next sections introduces the new components of the proposed modeling framework implemented during this thesis.

4.3 Discovery Library for Reverse Engineering

With the discovery library we aim to provide an extensible tool to reverse engineer CML models from existing source code. Users working in brownfield projects shall be able to generate CML models of an existing system without manual modeling work. As already explained in Chapter 3, the discovery library is designed in an extensible way so that different discovery strategies can be implemented. Thus, it is possible to support various frameworks and technologies.

4.3.1 Library Design

The UML class diagram illustrated in Figure 4.3 shows the fundamental design of the library. The *ContextMapDiscoverer* is the main class to discover the model from user perspective.

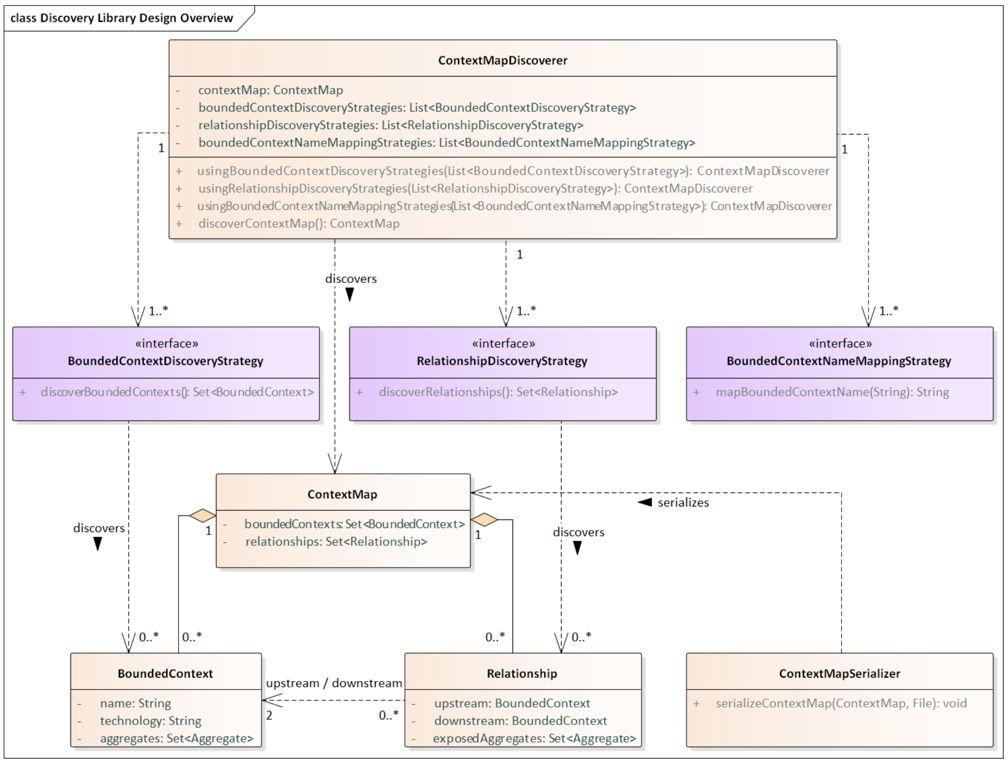

class Discovery Library Design Overview
ContextMapDiscoverer
- contextMap: ContextMap
- boundedContextDiscoveryStrategies: List<BoundedContextDiscoveryStrategy>
- relationshipDiscoveryStrategies: List<RelationshipDiscoveryStrategy>
- boundedContextNameMappingStrategies: List<BoundedContextNameMappingStrategy>
+ usingBoundedContextDiscoveryStrategies(List<BoundedContextDiscoveryStrategy>): ContextMapDiscoverer
+ usingRelationshipDiscoveryStrategies(List<RelationshipDiscoveryStrategy>): ContextMapDiscoverer
+ usingBoundedContextNameMappingStrategies(List<BoundedContextNameMappingStrategy>): ContextMapDiscoverer
+ discoverContextMap(): ContextMap
1
1..*
discovers
«interface» BoundedContextDiscoveryStrategy
+ discoverBoundedContexts(): Set<BoundedContext>
«interface» RelationshipDiscoveryStrategy
+ discoverRelationships(): Set<Relationship>
«interface» BoundedContextNameMappingStrategy
+ mapBoundedContextName(String): String
ContextMap
- boundedContexts: Set<BoundedContext>
- relationships: Set<Relationship>
serializes
0..*
BoundedContext
- name: String
- technology: String
- aggregates: Set<Aggregate>
upstream / downstream
2
Relationship
- upstream: BoundedContext
- downstream: BoundedContext
- exposedAggregates: Set<Aggregate>
ContextMapSerializer
+ serializeContextMap(ContextMap, File): void

FIGURE 4.3: Discovery Library Design Overview

By using the method *discoverContextMap* it reverse engineers a *ContextMap* which consists of *BoundedContexts* and *Relationships*. The *ContextMapDiscoverer* uses a set of *BoundedContextDiscoveryStrategy* implementations to discover Bounded Contexts and a set of *RelationshipDiscoveryStrategy* implementations to discover the relationships between them. These two interfaces provide the extension point to inject individual discovery strategies. Already implemented strategies are introduced later in this chapter.

The *BoundedContextNameMappingStrategy* interface offers another extension point to apply heuristics allowing to map dissimilar Bounded Context names.

These mapping strategies are used by the implemented discovery logic to assign relationships to the correct Bounded Contexts. For example, a *BoundedContextDiscoveryStrategy* may discover two Bounded Contexts named *Context1* and *Context2* while the *RelationshipDiscoveryStrategy* finds the relationship between them with the names *context-1* and *context-2*. By implementing a corresponding *BoundedContextNameMappingStrategy* it is possible to map these names and discover the relationship correctly.

The *ContextMapSerializer* finally allows to persist the discovered model in a CML file. Therefore, a user can simply reverse engineer a CML model with the following three steps:

1. Create and configure a *ContextMapDiscoverer*.
2. Call the *discoverContextMap()* method.
3. Create and persist a CML model by using the *ContextMapSerializer*.

Listing 9 illustrates a simple example how a library user can implement this in Java.

```java
public static void main(String[] args) throws IOException {
  // configure discoverer with discovery strategies (1)
  ContextMapDiscoverer discoverer = new ContextMapDiscoverer()
    .usingBoundedContextDiscoveryStrategies(
      new SpringBootBoundedContextDiscoveryStrategy("org.example.app"))
    .usingRelationshipDiscoveryStrategies(
      new DockerComposeRelationshipDiscoveryStrategy(new File("/source/example-app/")))
    .usingBoundedContextNameMappingStrategies(
      new DefaultNameMappingStrategy());

  // discover context map (2)
  ContextMap contextmap = discoverer.discoverContextMap();

  // save CML file (3)
  new ContextMapSerializer().serializeContextMap(contextmap,
    new File("./src-gen/example-model.cml"));
}
```

LISTING 9: Discovery Library: Simple Usage Example

4.3.2 Discovery Model and Mapping to CML

A *BoundedContextDiscoveryStrategy* can discover much more than just a Bounded Context with a name. The implemented discovery model allows to derive Aggregates with Entities and Value Objects as well. Those domain objects can contain attributes, references to other domain objects, and methods. With our approach, it depends on the implementation of the *BoundedContextDiscoveryStrategy* how much details of the Bounded Contexts are discovered. Figure 4.4 illustrates the current version of the discovery model. The discovery strategy implementations return the discovered data as an instance of this meta-model.

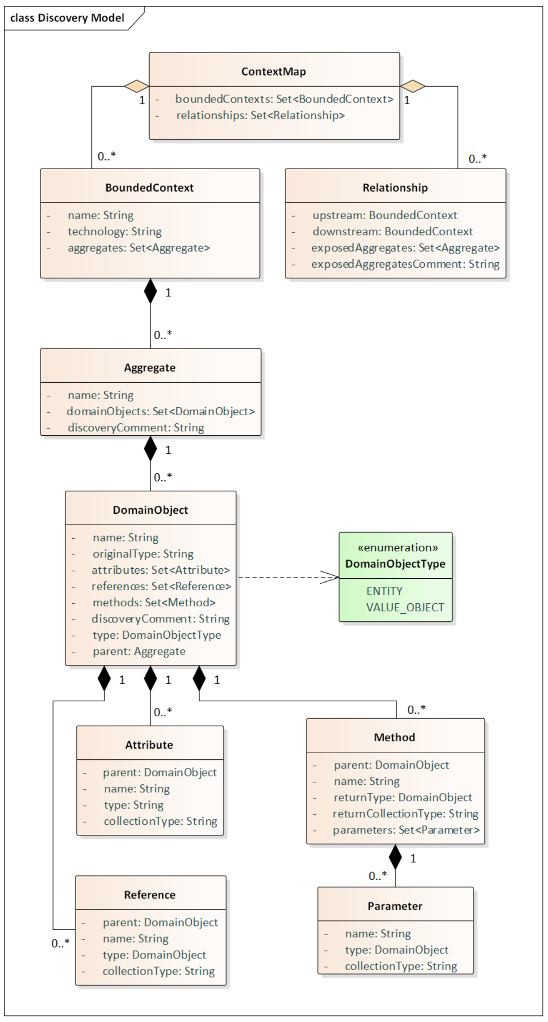


FIGURE 4.4: Discovery Model: Information Reverse Engineered by Discovery Strategies

The *ContextMapSerializer* implementation finally maps this model to the CML language model. As already mentioned ealier, the grammar and semantic model of the tactic DDD part within CML is based on Sculptor [83]. The following Figure 4.5 shows both models, the discovery model and the tactic CML model, and illustrates how the *ContextMapSerializer* of the discovery library maps the elements. Note that the CML model in Figure 4.5 is extremely simplified due to space limitations.

Since the model of the discovery library is already based on the DDD patterns, many objects such as *ContextMap*, *BoundedContext*, and *Aggregate* map naturally to their counterparts. However, within the discovery library we only detect asymmetric Upstream-Downstream relationships, since the detection of *Partnerships* or *Shared Kernels* would be difficult. Thus, the *Relationship* of the discovery model maps to the *UpstreamDownstreamRelationship* in CML. The tactical part to detect the domain model within a Bounded Context is kept simple within the discovery model. With the current version, it is possible to detect *Entities* and *Value Objects* only. Many other domain object types inheriting from

DomainObject in CML are not depicted in Figure 4.5 since we do not support the detection.

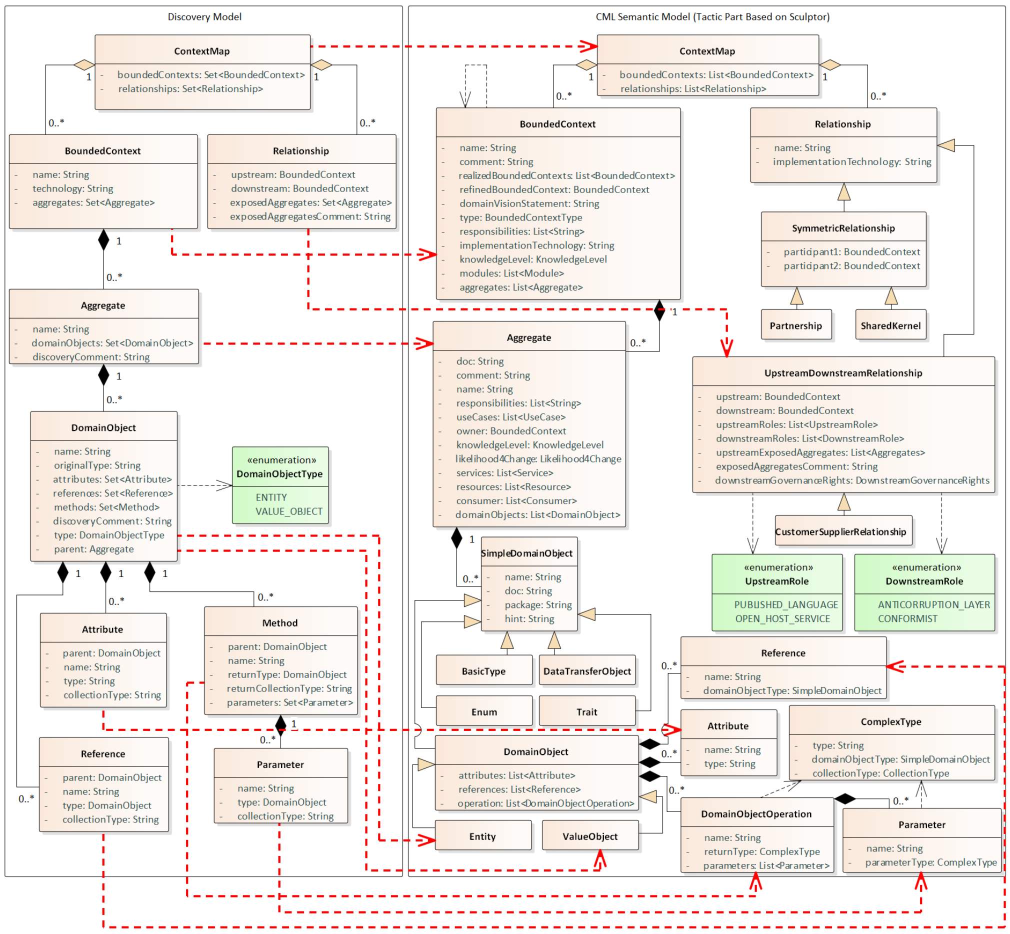


FIGURE 4.5: Discovery Model to CML Mapping

DomainObject's of the types *ENTITY* or *VALUE_OBJECT* naturally map to the corresponding types in CML. Just like the CML meta-model, the discovery library supports *Attribute*'s, *Reference*'s, and *Method*'s (*DomainObjectOperation*'s in CML) within the domain objects. Thus, the mapping between these elements is trivial. A discovered CML code example is presented later in our case study (Section 4.3.4).

4.3.3 Discovery Strategies

After having seen what output the discovery strategies produce (model explained above) this section discusses the already implemented strategies. In Chapter 3 we explained the selection process and decided to realize the following two strategies as part of the prototype implemented during this thesis:

1. Bounded Context discovery including domain model for Spring Boot [71] applications.

2. Relationship discovery on the basis of Docker «Compose» [19].

Figure 4.6 illustrates the implemented strategies as a UML class diagram.

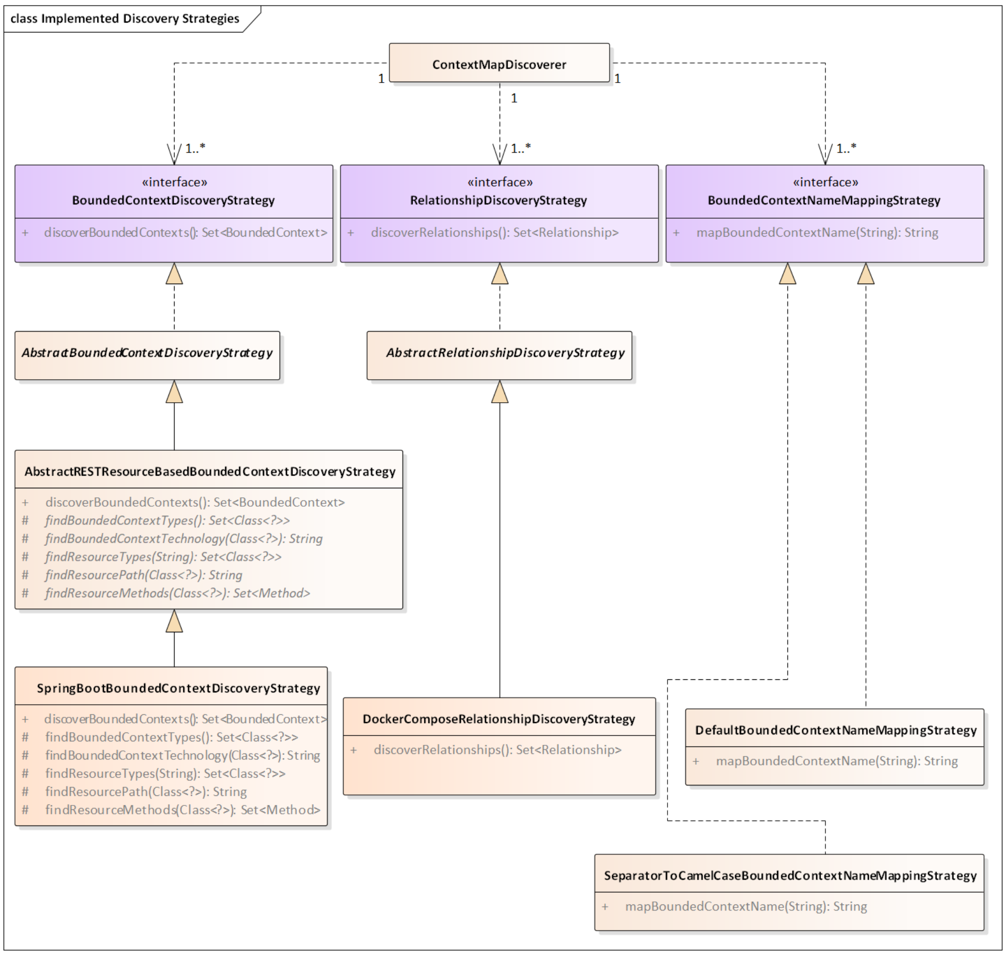

class Implemented Discovery Strategies
ContextMapDiscoverer
1
1
1
1..*
1..*
1..*
«interface» BoundedContextDiscoveryStrategy
+ discoverBoundedContexts(): Set<BoundedContext>
«interface» RelationshipDiscoveryStrategy
+ discoverRelationships(): Set<Relationship>
«interface» BoundedContextNameMappingStrategy
+ mapBoundedContextName(String): String
AbstractBoundedContextDiscoveryStrategy
AbstractRelationshipDiscoveryStrategy
AbstractRESTResourceBasedBoundedContextDiscoveryStrategy
+ discoverBoundedContexts(): Set<BoundedContext>
# findBoundedContextTypes(): Set<Class<?>>
# findBoundedContextTechnology(Class<?>): String
# findResourceTypes(String): Set<Class<?>>
# findResourcePath(Class<?>): String
# findResourceMethods(Class<?>): Set<Method>
SpringBootBoundedContextDiscoveryStrategy
+ discoverBoundedContexts(): Set<BoundedContext>
# findBoundedContextTypes(): Set<Class<?>>
# findBoundedContextTechnology(Class<?>): String
# findResourceTypes(String): Set<Class<?>>
# findResourcePath(Class<?>): String
# findResourceMethods(Class<?>): Set<Method>
DockerComposeRelationshipDiscoveryStrategy
+ discoverRelationships(): Set<Relationship>
DefaultBoundedContextNameMappingStrategy
+ mapBoundedContextName(String): String
SeparatorToCamelCaseBoundedContextNameMappingStrategy
+ mapBoundedContextName(String): String

FIGURE 4.6: Implemented Discovery Strategies (Prototype)

Spring Boot Bounded Context Discovery

With the *SpringBootBoundedContextDiscoveryStrategy* we provide a first prototype to discover Bounded Contexts including Aggregates, Entities, Value Objects, and corresponding attributes and methods. Many microservice projects use Spring Boot [71] to realize individual services [92]. Since one service in a microservice architecture corresponds to the idea of a Bounded Context on a DDD Context Map, we create one Bounded Context for each discovered Spring Boot [71] application accordingly.

This discovery strategy is based on Java runtime metadata analysis with the reflections library[8] and the Java reflection Application Programming Interface (API)[9].

A Spring Boot [71] application can be identified by the *SpringBootApplication* annotation. By using the mentioned reflections library we scan for those annotations on the classpath to find the individual Bounded Contexts. Note that the user who wants to analyze a microservice application based on Spring

[8]`https://github.com/ronmamo/reflections`

[9]`https://www.oracle.com/technical-resources/articles/java/javareflection.html`

Boot [71] must add all corresponding service classes to the Java classpath of the analysis application using our discovery library.

Table 4.2 summarizes how we detect the elements of the discovery model by using the Spring annotations. The following examples illustrate the behavior in detail.

TABLE 4.2: Spring Annotations to Discovery Model Mapping

Annotation / Code Element	**Resulting Object in Discovery Model**
@SpringBootApplication	Bounded Context
@RequestMapping on class	Aggregate with one «aggregate root» Entity
@RequestMapping, @GetMapping, @PostMapping, @PutMapping, @PathMapping, and @DeleteMapping on methods	Method in «aggregate root» Entity
Parameters in discovered methods above	Domain object of type *Value Object*
Return type in discovered methods above	Domain object of type *Value Object*

Listing 10 shows an example Java class annotated with the *SpringBootApplication* annotation.

```java
@SpringBootApplication
public class DiscoveryExampleApplication {
  public static void main(String[] args) {
    SpringApplication.run(DiscoveryExampleApplication.class, args);
  }
}
```

LISTING 10: Example Spring Boot Application

Having this class on the classpath, our library would create a Bounded Context called «DiscoveryExample». The name of the Bounded Context is derived from the annotated class. If the class name ends with «Application», we remove this ending from the Bounded Context name.

Once we have discovered the application, we scan the given Java package for RESTful HTTP resources. These resources can be found by searching for Spring annotations again. Listing 11 and Listing 12 show an example class representing such a resource implemented for the Spring framework.

```java
@RestController
@RequestMapping({"/customers"})
public class CustomerInformationHolder {

  @PutMapping({"/{customerId}/address"})
  public ResponseEntity<Address> changeAddress(@PathVariable CustomerId customerId,
    @RequestBody Address requestDto) {
```

LISTING 11: Example Spring RESTful HTTP Resource (1)

```
    // just an example operation...
  }

  @GetMapping({"/{customerId}/"})
  public ResponseEntity<Customer> getCustomer(@PathVariable CustomerId customerId) {
    // just an example operation...
  }

  @GetMapping({"/{customerIds}/"})
  public ResponseEntity<List<Customer>> getCustomers(
    @PathVariable List<CustomerId> customerIds) {
    // just an example operation...
  }

}
```

LISTING 12: Example Spring RESTful HTTP Resource (2)

The example illustrates the Java annotations used to discover resources and operations. The *RequestMapping* annotation on the class allows us to find the resource for which we would create a «customers» Aggregate (name is derived from resource path). By searching for *RequestMapping*, *GetMapping*, *PostMapping*, *PutMapping*, *PatchMapping*, and *DeleteMapping* annotations on methods, we find the operations within the resource. A «root» Entity is created within each Aggregate derived from the resources. This «root» Entity will contain methods derived from the operations of the resource. From the example in Listing 11 and Listing 12 we would derive a «root» Entity with the methods *changeAddress*, *getCustomer*, and *getCustomers*.

In addition, this discovery strategy derives Value Objects for all Data Transfer Objects (DTOs) [28] appearing in the operation parameters and return types (currently non-recursive). Listing 13 and Listing 14 show example CML code which would be discovered assuming the Spring Boot [71] application in Listing 10 contains the RESTful HTTP resource illustrated in Listing 11 and Listing 12 within the same package or a sub-package.

```
ContextMap {
  contains DiscoveryExample

}

BoundedContext DiscoveryExample {
  implementationTechnology "Spring Boot"

  // This Aggregate has been created on the basis of the RESTful HTTP controller
  // org.contextmapper.discovery.example.CustomerInformationHolder.
  Aggregate customers {
    // This value object has been derived from the class
    // org.contextmapper.discovery.example.Address.
    ValueObject Address {
      String city
      int plz
      String street
      List<String> arrayTest
    }
```

LISTING 13: Example Discovery Result in CML (1)

```
    // This value object has been derived from the class
    // org.contextmapper.discovery.example.CustomerId.
    ValueObject CustomerId {
      String id
    }
    // This value object has been derived from the class
    // org.contextmapper.discovery.example.Customer.
    ValueObject Customer {
      - List<Address> addressList
      - CustomerId id
    }
    Entity customers_RootEntity {
      aggregateRoot
      def @Customer getCustomer(@CustomerId arg0);
      def @Address changeAddress(@CustomerId arg0, @Address arg1);
      def List<@Customer> getCustomers(List<@CustomerId> arg0);
    }
  }
}
```

LISTING 14: Example Discovery Result in CML (2)

The discovered Bounded Context contains the «customers» Aggregate with a «root» Entity. The Entity contains the three methods which are derived from the resource operations illustrated in Listing 11 and Listing 12. The parameter and return types are modeled as Value Objects. The generated comments help the user to understand from which classes in the original source code the individual Aggregates and Value Objects have been derived.

In this example we used only the *SpringBootBoundedContextDiscoveryStrategy* to reverse engineer a Bounded Context. Thus, the Context Map does not contain any relationships between contexts. The following section presents our prototypic discovery strategy to reverse engineer such relationships between (micro-)services (Bounded Contexts).

Docker «Compose» Relationship Discovery

As discussed in Chapter 3, Docker «Compose» [19] is often used in microservice projects to run all the services within Docker containers [20]. To start and manage the services correctly, Docker «Compose» [19] must know the dependencies between the services. They are configured in a *docker-compose.yml* file in the YAML Ain't Markup Language (YAML) format.

With the *DockerComposeRelationshipDiscoveryStrategy* we provide a prototypic implementation for a Context Map relationship discovery in projects using Docker «Compose» [19]. The discovery library always applies *BoundedContextDiscoveryStrategy*'s first. After that, all *RelationshipDiscoveryStrategy*'s are applied. The relationship strategies can only find relationships between Bounded Contexts which have been previously discovered by the Bounded Context strategies. For example, we may discover Bounded Contexts based on Spring Boot [71] and then find the relationships between these contexts with the Docker «Compose» [19] configuration.

The following small example will illustrate how the strategy works. Assume there are two Spring Boot [71] applications already discovered with the strategy explained before. The two applications in Listing 15 (simplification) lead to the Bounded Contexts *Microservice1* and *Microservice2*.

```
@SpringBootApplication
public class Microservice1 {
}

@SpringBootApplication
public class Microservice2 {
}
```

LISTING 15: Spring Boot Microservices (Simplified Example)

A *docker-compose.yml* file for this scenario could look like the one illustrated in Listing 16. Important here is the *depends_on* attribute in the configuration of *microservice2*.

```
version: "3"
services:
  microservice1:
    build: microservice1
    image: example/microservice1
    ports:
      - "8080:8080"
  microservice2:
    build: microservice2
    image: example/microservice2
    depends_on:
      - microservice1
    ports:
      - "8090:8090"
```

LISTING 16: *docker-compose.yml* Example File

The *DockerComposeRelationshipDiscoveryStrategy* will reverse engineer Upstream-Downstream relationships in CML based on these dependency declarations. Running the discovery with the two presented strategies and the inputs of Listing 15 and Listing 16 generates the following CML code shown in Listing 17.

```
ContextMap {
  contains Microservice1
  contains Microservice2

  Microservice1 -> Microservice2
}

BoundedContext Microservice1 {
  implementationTechnology "Spring Boot"
}

BoundedContext Microservice2 {
  implementationTechnology "Spring Boot"
}
```

LISTING 17: Discovered Relationships Example (CML)

The listing shows the discovered relationship with *Microservice1* as upstream context and *Microservice2* as downstream context. Remember that the relationship arrow in CML illustrates the *influence flow* [73]. The current version of our prototype simply discovers Upstream-Downstream relationships as illustrated above. The discovery of additional relationship patterns such as Open Host Service (OHS), Published Language (PL), Anticorruption Layer (ACL), or Conformist (CF) is not yet supported. The discovery library may be enhanced in future projects to support the detection of such patterns.

The careful reader may have already noticed that the Bounded Context names in the example above cannot be matched automatically. While the names discovered by the Bounded Context strategy are *Microservice1* and *Microservice2*, the corresponding names in the *docker-compose.yml* file are *microservice1* and *microservice2* (upper vs. lower case). This leads us to the last extension point of our discovery framework which allows to inject name mapping strategies flexibly.

Bounded Context Name Mapping Strategies

The different discovery strategies in a reverse engineering process with our discovery library may find the same Bounded Context with different names. The example in the last section illustrates such a scenario. Different technologies, in the case above Spring Boot [71] and Docker «Compose» [19], may configure and represent the same service with varying names.

The *BoundedContextNameMappingStrategy* interface offers an extension point to inject various name mapping strategies which help the *ContextMapDiscoverer* to identify these Bounded Contexts. If you provide multiple name mapping strategies, the current implementation will apply them all in the order of configuration until a Bounded Context is found. If a relationship discovery strategy cannot find a Bounded Context in the list of previously discovered contexts it will ignore the relationship.

Table 4.3 explains the two already provided name mapping strategies implemented as part of our prototype (see Figure 4.6). The second strategy has been used in the example of the previous section to match the names that differ with reference to capitalization of the first letter.

TABLE 4.3: Implemented Name Mapping Strategies (Discovery Library)

Strategy	**Description**
DefaultBoundedContext-NameMappingStrategy	This is the default strategy used by the discovery library. If no other strategy to map Bounded Context names has been registered, this one will be applied. This strategy maps names as they are (basically no mapping).
SeparatorToCamelCase-BoundedContext-NameMappingStrategy	This strategy maps lower case strings which use a separation character to camel case strings. Example: The name *my-example-service* can be mapped to *MyExampleService* by using the minus sign («-») as separator.

4.3.4 Lakeside Mutual Case Study

We validated the discovery library by applying it on a microservice project. The Lakeside Mutual[10] project is a fictitious insurance company which serves as a sample application to demonstrate microservices and the application of Microservice API Patterns (MAP) [105]. We chose the project since it contains several services based on Spring Boot [71] and also uses Docker «Compose» [19]. Thus, it is suitable to validate the current state of the discovery library with the provided discovery strategies.

Architecture Overview

Figure 4.7 shows the architecture of the Lakeside Mutual application as documented in their GitHub repository[10].

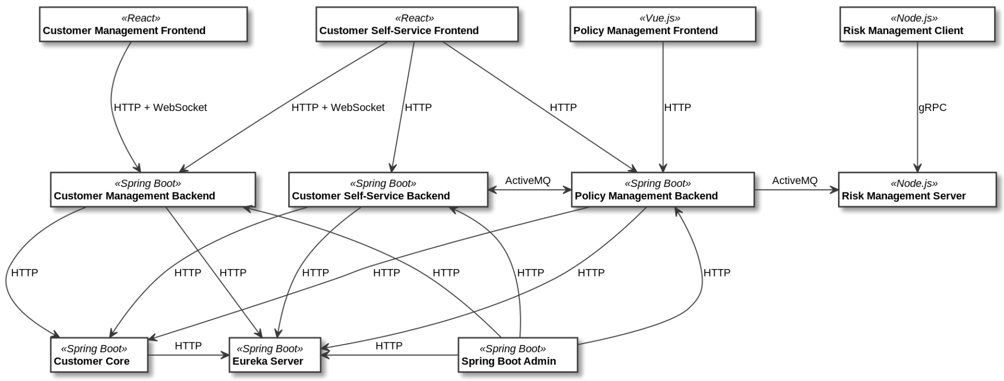

«React» Customer Management Frontend
«React» Customer Self-Service Frontend
«Vue.js» Policy Management Frontend
«Node.js» Risk Management Client
HTTP + WebSocket
HTTP
gRPC
ActiveMQ
«Spring Boot» Customer Management Backend
«Spring Boot» Customer Self-Service Backend
«Spring Boot» Policy Management Backend
«Node.js» Risk Management Server
«Spring Boot» Customer Core
«Spring Boot» Eureka Server
«Spring Boot» Spring Boot Admin

FIGURE 4.7: Lakeside Mutual Architecture Overview [61]

From the components illustrated on this overview, we built and added the following to our classpath:

- Customer Core
- Customer Management Backend
- Customer Self-Service Backend
- Policy Management Backend

Frontends and services implemented with other technologies than Spring Boot [71] can not be discovered at the moment. Thus, the architecture we are able to discover with the current prototype simplifies to the one illustrated in Figure 4.8. Note that all relationships will be discovered as asymmetric Upstream-Downstream relationships as specified in the Docker configuration. The ActiveMQ[11] messaging channel between the *Customer Self-Service Backend* and the *Policy Management Backend* might actually be a symmetric relationship as indicated in Figure 4.8. To discover this detail we will have to implement another discovery strategy specifically for this technology in future projects.

[10]`https://github.com/Microservice-API-Patterns/LakesideMutual`
[11]`https://activemq.apache.org/`

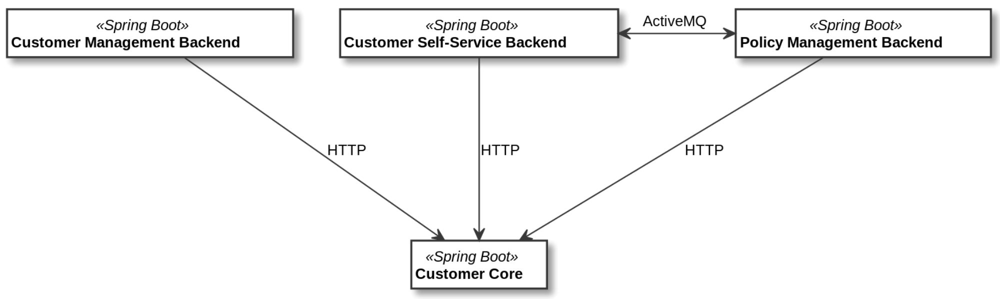


FIGURE 4.8: Lakeside Mutual: Discoverable Components

Discovery and Resulting CML Model

We used our discovery library to reverse engineer the components as illustrated above. The required source code to apply the library on the Lakeside Mutual project is shown in Listing 18.

The *SpringBootBoundedContextDiscoveryStrategy* is used to detect Bounded Contexts, while the *DockerComposeRelationshipDiscoveryStrategy* discovers the relationships. For the name mapping we used an adjusted *SeparatorToCamelCaseBoundedContextNameMappingStrategy*. Since only the Docker «Compose» [19] file uses the endings «Backend», we remove this part to map the names with the Spring Boot [71] application names correctly.

```java
public class LakesideMutualContextMapDiscoverer {
  public static void main(String[] args) throws IOException {
    ContextMapDiscoverer discoverer = new ContextMapDiscoverer()
        .usingBoundedContextDiscoveryStrategies(
            new SpringBootBoundedContextDiscoveryStrategy("com.lakesidemutual"))
        .usingRelationshipDiscoveryStrategies(
            new DockerComposeRelationshipDiscoveryStrategy(
                new File(System.getProperty("user.home") + "/source/LakesideMutual/")))
        .usingBoundedContextNameMappingStrategies(
            new SeparatorToCamelCaseBoundedContextNameMappingStrategy("-") {
              @Override
              public String mapBoundedContextName(String s) {
                // remove the "Backend" part of the Docker service names ...
                String name = super.mapBoundedContextName(s);
                return name.endsWith("Backend") ?
                    name.substring(0, name.length() - 7) : name;
              }
            });

    ContextMap contextmap = discoverer.discoverContextMap();
    new ContextMapSerializer().serializeContextMap(contextmap,
      new File("./src-gen/lakesidemutual.cml"));
  }
}
```

LISTING 18: Java Code to Apply Discovery Library on Lakeside Mutual Project

The complete source code and CML result of this case study can be found in the GitHub repository of our discovery library[12] under the examples[13]. Listing

[12]https://github.com/ContextMapper/context-map-discovery
[13]https://github.com/ContextMapper/context-map-discovery/tree/master/Examples

19 shows a shortened version of the discovered CML output without domain models within the Bounded Contexts. The complete model would take too much space within this thesis report. Note that all relationships illustrated in Figure 4.8 have been discovered. As already mentioned the asymmetry between *PolicyManagement* and *CustomerSelfService* might not be correct.

```
ContextMap {
  contains PolicyManagement, CustomerManagement, CustomerSelfService, CustomerCore

  CustomerCore -> PolicyManagement

  CustomerCore -> CustomerManagement

  PolicyManagement -> CustomerSelfService

  CustomerCore -> CustomerSelfService
}

// note: removed domain models from Bounded Contexts to save space:
BoundedContext PolicyManagement {
  implementationTechnology "Spring Boot"
}
BoundedContext CustomerManagement {
  implementationTechnology "Spring Boot"
}
BoundedContext CustomerSelfService {
  implementationTechnology "Spring Boot"
}
BoundedContext CustomerCore {
  implementationTechnology "Spring Boot"
}
```

LISTING 19: Lakeside Mutual Case Study:
Discovered CML Model

However, after having the model reverse engineered, the user is able to correct such details manually as well. The accuracy of the resulting CML model for this case study could be further improved by implementing additional discovery strategies in future projects.

4.3.5 Known Limitations

The validation of our discovery approach with the Lakeside Mutual project has shown that especially the reverse engineering of the domain models within the Bounded Contexts is challenging. Currently all DTOs [28] are simply mapped to Value Objects which might not conform with the reality. In fact, many of the discovered domain objects are actually Entities. However, with the current approach it is not easily possible to differentiate these types on the basis of the DTOs. In future projects we may find improved strategies for this problem, although correcting such mistakes within the resulting model is still easier than creating the model manually. Thus, users with existing projects already benefit from this approach.

In addition, the current approach creates duplicate domain objects in case one is used in multiple RESTful HTTP endpoints. We currently create Value Objects within an Aggregate for all discovered DTOs in the corresponding endpoints. If the same DTO occurs in two endpoints, we create two separate Value

Objects, one for each resulting Aggregate. This is for example problematic if the created model is later decomposed using the structured service decomposition component. The duplicate domain objects will hide potential coupling between objects that influences the resulting decomposition.

Another issue not yet solved is the detection of *exposed* Aggregates. In the complete resulting CML model[14] of the case study one can see that we currently add all discovered Aggregates to the *exposedAggregates* attribute of the corresponding relationships. This might not be correct in all cases since there can be Aggregates which are only used by some consumers but not all. Detecting which Aggregates (or originally RESTful HTTP resources) are actually used by which consumers is another challenging issue for future releases of the library.

4.4 Service Decomposition with Service Cutter

With the Service Cutter [36] integration into our framework we provide another approach to decompose an existing DDD Context Map. While the ARs offer manual steps to (de-)compose the model iteratively, this structured approach generates new decompositions based on Decomposition Criteria (DCs) automatically. The Service Cutter tool [35] realizes this with graph clustering algorithms and a scoring system based on its coupling criteria catalog [37] as already explained in Chapter 3.

4.4.1 Integration

The original Service Cutter tool [35] is a web application based on Spring Boot [71] and built with JHipster [43]. To integrate the approach into Context Mapper we created a fork from the original project and realized our own Service Cutter library[15] as part of this thesis project. The library offers an API to generate service decompositions used by Context Mapper as illustrated by the extract of the framework component diagram in Figure 4.9.

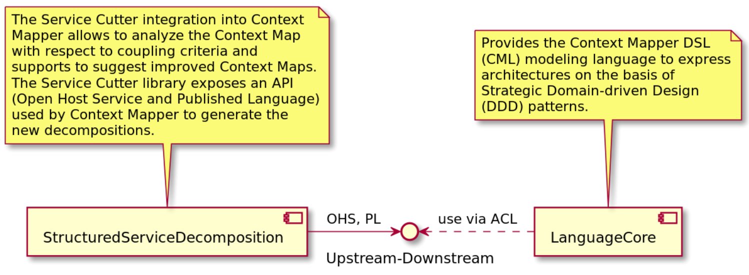

The Service Cutter integration into Context Mapper allows to analyze the Context Map with respect to coupling criteria and supports to suggest improved Context Maps. The Service Cutter library exposes an API (Open Host Service and Published Language) used by Context Mapper to generate the new decompositions.
Provides the Context Mapper DSL (CML) modeling language to express architectures on the basis of Strategic Domain-driven Design (DDD) patterns.
StructuredServiceDecomposition
OHS, PL
use via ACL
Upstream-Downstream
LanguageCore

FIGURE 4.9: Service Cutter Library Framework Interface

With the Service Cutter library we implemented an API for service cutting which could be used in any Java application and not only in Context Mapper. The API is based on the same data structures which have to be provided in the original Service Cutter tool [35] as JavaScript Object Notation (JSON) files.

[14]https://github.com/ContextMapper/context-map-discovery/tree/master/Examples
[15]https://github.com/ContextMapper/service-cutter-library

We use the MDSL contract language[16] [102] to present the service cutting API here. This way we not only provide a service contract but also validate our own tool at the same time since we generated the MDSL contract with Context Mapper. It is important to note that the contract has been simplified for the presentation in this report. We shortened some data types since not all details are necessary to understand the basic structure. The complete CML model and MDSL contract can be found in our examples repository[17].

```
// input model (Entity Relation Diagram & User Representations)
data type EntityRelationshipDiagram {
  "name":V<string>,
  "entities":{ "name":V<string>, "nanoentities":V<string>* }*,
  "relations":{ "origin":{ "name":V<string>, "nanoentities":V<string>* },
              "destination":{ "name":V<string>, "nanoentities":V<string>* },
              "relationType":RelationType }* }
data type UserRepresentations {
  /* removed user representation details to save space (see Figure 4.10) */ }
data type ERDAndUserRepresentationsContainer {
  "erd":EntityRelationshipDiagram, "representations":UserRepresentations }
data type RelationType { "AGGREGATION" | "COMPOSITION" | "INHERITANCE" }
```

LISTING 20: Service Cutter API Data Types (MDSL)

Listing 20 and Listing 21 show the datatypes of the API first. The initial block in Listing 20 corresponds to the input model describing the system required by Service Cutter [35]. It consists of an entity relation diagram and optional user representations such as use cases. Figure 4.10 illustrates this input model as a UML class diagram.

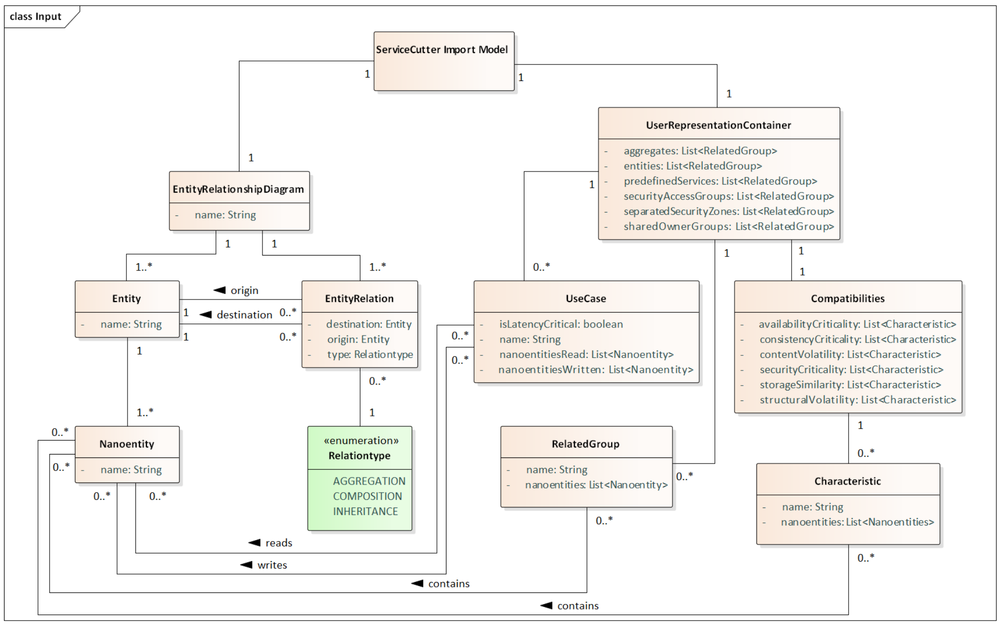

class Input
ServiceCutter Import Model
UserRepresentationContainer
- aggregates: List<RelatedGroup>
- entities: List<RelatedGroup>
- predefinedServices: List<RelatedGroup>
- securityAccessGroups: List<RelatedGroup>
- separatedSecurityZones: List<RelatedGroup>
- sharedOwnerGroups: List<RelatedGroup>
EntityRelationshipDiagram
- name: String
Entity
- name: String
EntityRelation
- destination: Entity
- origin: Entity
- type: Relationtype
origin
destination
UseCase
- isLatencyCritical: boolean
- name: String
- nanoentitiesRead: List<Nanoentity>
- nanoentitiesWritten: List<Nanoentity>
Compatibilities
- availabilityCriticality: List<Characteristic>
- consistencyCriticality: List<Characteristic>
- contentVolatility: List<Characteristic>
- securityCriticality: List<Characteristic>
- storageSimilarity: List<Characteristic>
- structuralVolatility: List<Characteristic>
Nanoentity
- name: String
«enumeration» Relationtype
AGGREGATION
COMPOSITION
INHERITANCE
RelatedGroup
- name: String
- nanoentities: List<Nanoentity>
Characteristic
- name: String
- nanoentities: List<Nanoentities>
reads
writes
contains
contains
1
1..*
0..*

FIGURE 4.10: Service Cutter Library Input Model for System Description [46]

[16] All MDSL code snippets in this thesis and the current state of the MDSL generator are compatible with the MDSL version 1.0.2 (released in October 2019).

[17] `https://github.com/ContextMapper/context-mapper-examples`

The following MDSL block describes the complete *context* value object needed to call the cutting service.

```
// Service cutting context built by input and coupling criteria
data type ServiceCutterContext {
  "systemName":V<string>, "criteriaCatalog":CouplingCriteriaCatalog,
  "couplingInstances":CouplingInstance*, "nanoEntities":V<string>*,
  "solverConfiguration":SolverConfiguration }
data type CouplingCriteriaCatalog P // not specified in detail to save space
data type CouplingInstance P // not specified in detail to save space

// Solver configuration provided by user (algorithm and priorities)
data type SolverConfiguration {
  "algorithm":V<string>, "solverPriorities":{ "criterion":V<string>,
                                             "priority":SolverPriority }* }
data type SolverPriority { "IGNORE" | "XS" | "S" | "M" | "L" | "XL" | "XXL" }

// Cutting result provided by Service Cutter
data type SolverResult {
  "services":{ "nanoentities":V<string>*, "id":char }*,
  "relations":{ "serviceA":V<string>, "serviceB":V<string>,
               "sharedEntities":V<string>*, "direction":Direction }* }
data type Direction { "OUTGOING" | "INCOMING" | "BIDIRECTIONAL" }
```

LISTING 21: Service Cutter API Data Types (MDSL, continued)

It contains the following: the coupling criteria catalog [37], *coupling instances* which describe the coupling between nanoentities for a specific coupling criterion, all *nanoentities*, and the solver configuration. An Entity in Service Cutter contains a set of so-called *nanoentities* [36] which are basically the attributes of the Entities in our case.

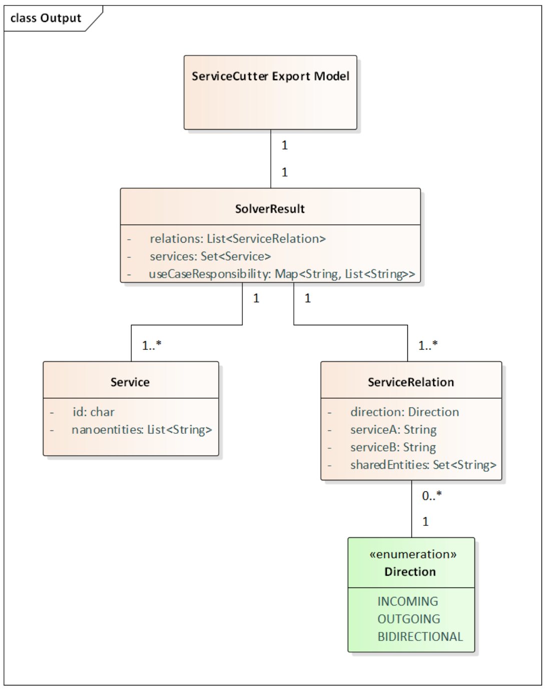


FIGURE 4.11: Service Cutter Library Output Model Describing the Suggested Service Cut [46]

With the solver configuration the user of the API is able to control which graph clustering algorithm is used and how the individual coupling criteria are prioritized. The prioritization influences the scoring system used in Service Cutter and therefore has a corresponding effect on how the services are cut. The last block of the MDSL data types in Listing 21 describes the output model which is returned by the cutting process. It contains the resulting services as a set of *nanoentities* and the suggested relations between the services. Figure 4.11 illustrates the output model as UML class diagram again.

As the endpoints of the MDSL contract in Listing 22 illustrate, there are three API types (endpoints) to be used.

```
endpoint type ServiceCutterContextBuilder // stateful type
  exposes
    operation ServiceCutterContextBuilder // constructor
      expecting
        payload EntityRelationshipDiagram
    operation withUserRepresentations // provide user representations (optionally)
      expecting
        payload UserRepresentations
    operation withCustomSolverConfiguration // change solver configuration (optionally)
      expecting
        payload SolverConfiguration
    operation build // build the context
      delivering
        payload ServiceCutterContext

endpoint type SolverConfigurationFactory
  exposes
    operation createDefaultConfiguration // create initial/default solver configuration
      delivering
        payload SolverConfiguration

endpoint type ServiceCutter
  exposes
    operation generateDecomposition // decompose the system with the provided context
      expecting
        payload ServiceCutterContext
      delivering
        payload SolverResult
```

LISTING 22: Service Cutter API Operations (MDSL)

The first one is used to construct the *ServiceCutterContext* object by providing the input models and the solver configuration. The second *SolverConfigurationFactory* is used to create an initial solver configuration which can be changed by the user. The decomposition is generated with the third type. Listing 23 and Listing 24 illustrate the usage from a user's perspective in Java.

```
EntityRelationDiagram erd = /* construct the ERD model */;
UserRepresentationContainer userRepresentations = /* construct user representations */;

// create and maybe adjust solver configuration
SolverConfiguration solverConfiguration = new SolverConfigurationFactory()
    .createDefaultConfiguration();
```

LISTING 23: Service Cutter API Usage Example (Java)

```java
// create context
ServiceCutterContext context = new ServiceCutterContextBuilder(erd)
    .withUserRepresentations(userRepresentations)
    .withCustomSolverConfiguration(solverConfiguration)
    .build();

// generate decomposition
ServiceCutter serviceCutter = new ServiceCutter(context);
SolverResult result = serviceCutter.generateDecomposition();
```

LISTING 24: Service Cutter API Usage Example (Java)

Finally, the MDSL contract specifies the provider and the client of the API as illustrated in Listing 25.

```
API provider ServiceCutterLibrary
  offers ServiceCutterContextBuilder
  offers SolverConfigurationFactory
  offers ServiceCutter
API client LanguageCore
  consumes ServiceCutterContextBuilder
  consumes SolverConfigurationFactory
  consumes ServiceCutter
```

LISTING 25: Service Cutter API Provider and Client (MDSL)

This is equivalent to the component diagram already shown in Figure 4.9 at the beginning of this section. Our Service Cutter library provides the three API types (endpoints) which are then used by the core component to generate new service decompositions.

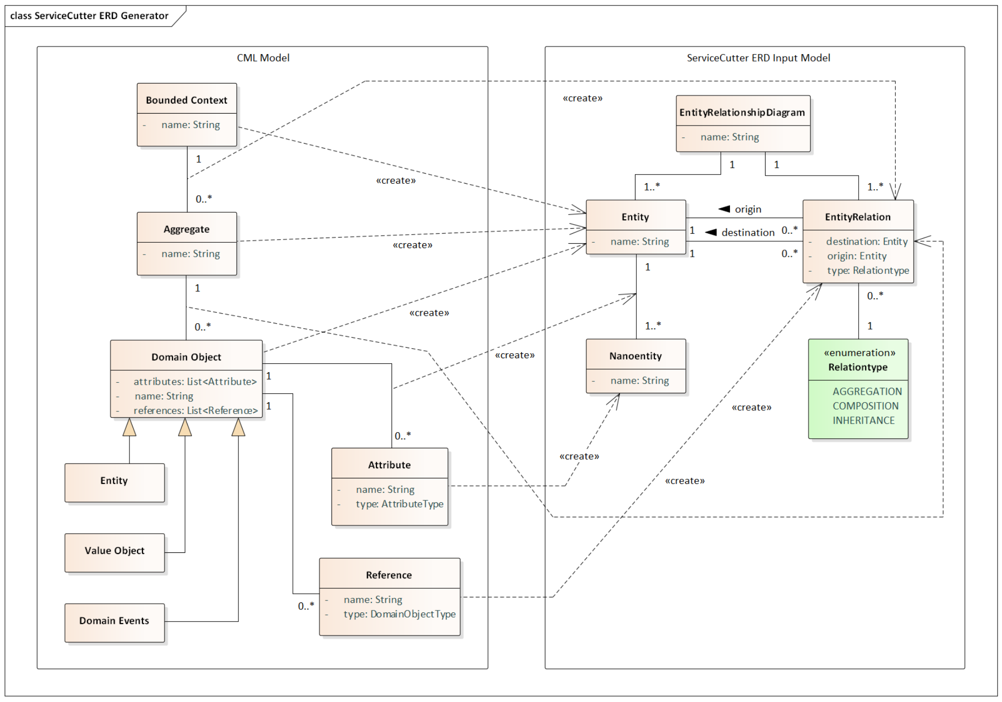


FIGURE 4.12: Service Cutter Entity Relation Input Mapping [46]

As indicated in Figure 4.9 our core component uses a translation and ACL layer to convert between the Service Cutter library model and the CML model. The mapping between these models has already been defined in our first term project [46]. At this time we already provided generators producing the JSON input files for the original Service Cutter tool [35]. The input mapping is illustrated in Figure 4.12.

Bounded Contexts, Aggregates, and all domain objects in the CML model are represented as an Entity in the Service Cutter model. We further create *nanoentities* for all attributes of domain objects in CML. References between domain objects are mapped to corresponding entity relations in Service Cutter.

From the result model illustrated in Figure 4.10 we produce CML code again. Listing 26 shows an example of a model (DDD sample application [24]) decomposed with the Service Cutter library. Resulting services are mapped to Bounded Contexts. Each Bounded Context further contains an Aggregate with the corresponding Entities and attributes derived from the *nanoentities*. Note that we currently do not reconstruct the original types of the attributes. The types are lost within the process and are not provided by Service Cutter. Thus, the type of all attributes is set to *UnknownType* at the moment.

```
ContextMap {
  contains ServiceA, ServiceB, ServiceC

  ServiceA [SK]<->[SK] ServiceB
  ServiceC [U]->[D] ServiceB
}
BoundedContext ServiceA {
  Aggregate AggregateA {
    Entity A_CarrierMovement {
      UnknownType departureTime
      UnknownType arrivalTime
    }
    Entity A_VoyageNumber {
      UnknownType number
    }
  }
}
BoundedContext ServiceB {
  Aggregate AggregateB {
    Entity B_Cargo {
      UnknownType trackingId
    }
    /* other entities removed here (saving space) */
  }
}
BoundedContext ServiceC {
  Aggregate AggregateC {
    Entity C_UnLocode {
      UnknownType unLocode
    }
  }
}
```

LISTING 26: Service Cutter Integration: CML Result Code Example [46]

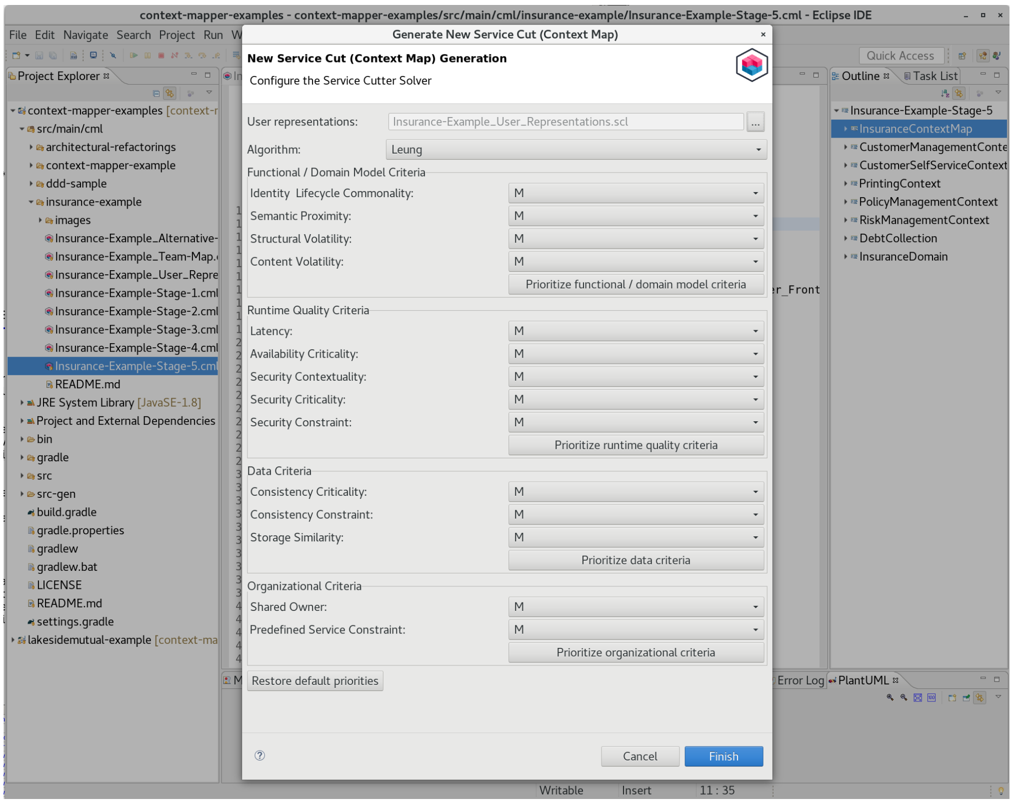
In addition, Listing 26 illustrates that service relations of the types *INCOMING* and *OUTGOING* are mapped to Upstream-Downstream relationships while *BIDIRECTIONAL* relations result in Shared Kernels.

4.4.2 Eclipse Plugin Integration

The functionality to generate new Context Maps with the Service Cutter library has been integrated into our Context Mapper Eclipse plugin. A dialog similar to the web User Interface (UI) of the original Service Cutter tool [35] allows the user to change the solver configuration. Figure 4.13 shows this UI within our plugin. It allows specifying the algorithm to be used and the priorities for all coupling criteria. In addition to that, the file with the user representations can be referenced.

The specification of the user representations is done in a separate DSL called Service Cutter DSL (SCL). This DSL has already been developed during our previous work [46]. For details about this configuration we refer to the corresponding project report [46] or the online documentation[18].

Finishing the dialog as shown in Figure 4.13 generates a new decomposition using the presented library and creates a new CML file such as in the example in Listing 26.


context-mapper-examples - context-mapper-examples/src/main/cml/insurance-example/Insurance-Example-Stage-5.cml - Eclipse IDE
File Edit Navigate Search Project Run
Project Explorer
Outline
Task List
Quick Access
Insurance-Example-Stage-5
InsuranceContextMap
CustomerManagementConte
CustomerSelfServiceContext
PrintingContext
PolicyManagementContext
RiskManagementContext
DebtCollection
InsuranceDomain
Error Log
PlantUML
Writable
Insert
11 : 35

Generate New Service Cut (Context Map)

New Service Cut (Context Map) Generation

Configure the Service Cutter Solver

User representations: Insurance-Example_User_Representations.scl

Algorithm: Leung

Functional / Domain Model Criteria

Identity Lifecycle Commonality:	M
Semantic Proximity:	M
Structural Volatility:	M
Content Volatility:	M

Prioritize functional / domain model criteria

Runtime Quality Criteria

Latency:	M
Availability Criticality:	M
Security Contextuality:	M
Security Criticality:	M
Security Constraint:	M

Prioritize runtime quality criteria

Data Criteria

Consistency Criticality:	M
Consistency Constraint:	M
Storage Similarity:	M

Prioritize data criteria

Organizational Criteria

Shared Owner:	M
Predefined Service Constraint:	M

Prioritize organizational criteria

Restore default priorities

Cancel Finish

FIGURE 4.13: Service Cutter Integration into Context Mapper (UI)

4.4.3 Validation

We have already validated the Service Cutter approach as part of the first term project [46]. With our fictitious insurance example[19] we showed that the approach produces reasonable service cuts. As part of this thesis we tested the

[18]https://contextmapper.org/docs/service-cutter/

[19]https://github.com/ContextMapper/context-mapper-examples

integration of the library version into Context Mapper with the same example again. Since we have not changed the cutting engine the results stay the same. More validation with other algorithms has to be done in future projects.

4.4.4 Cutting Algorithms

In Chapter 3 we presented candidates for additional graph clustering algorithms which could be integrated. Currently the Leung [56] algorithm is the only one supported in Context Mapper. During the process of integrating Service Cutter and validating the approach within our tool, we decided to not integrate other graph clustering algorithms during this thesis. More validation in future projects shall first show that this is the approach to continue with. Implementing our own algorithm, which only suggests potential extraction candidates (Bounded Contexts) instead of complete decompositions, might be a valid alternative. This would allow us to suggest AR applications instead of creating complete new CML models and would fit better into an iterative decomposition process.

4.4.5 Known Limitations

As already indicated, some information within a CML model such as data types are currently lost within the service cutting process. This means that the original CML model contains information which are no longer present in the resulting CML files including new decompositions. This is a limitation which influences the *usability* in a negative way. The produced decompositions can be used as suggestions and provide hints regarding the coupling between Entities, but the user is not able to continue the modeling process on the basis of the produced files. To avoid data loss, the user has to re-model the intended decomposition steps within the original model manually.

In addition, the tool currently only supports the algorithm of Leung [56] which is non-deterministic. This means that every execution of the algorithm potentially produces a new result. If this approach is pursued in future projects, the integration of more stable algorithms may be necessary.

Finally, note that the application of the Service Cutter tool within Context Mapper only produces reasonable results if the specified Bounded Contexts include Aggregates, Entities, and attributes. Without attributes within the Entities of the domain models, it is not possible to derive corresponding *nanoentities* which are required for the approach.

4.5 Architectural Refactorings (ARs)

As already introduced in previous chapters, the ARs allow the users of our framework to evolve the architecture model iteratively. Since we already discussed how these model transformations [49] are implemented on the technical level in our previous project [50], this section focuses on providing examples for the newly conceptualized ARs.

From the framework perspective the AR component interacts with the language core component closely. As illustrated by the extract of the framework component diagram in Figure 4.14 the two components are in a *Shared Kernel* relationship in terms of DDD.

Both components share the same domain model which is the semantic model [27] of the CML language introduced in Section 4.2 (Figure 4.2). The language core parses the CML code and provides the modeled system as an Eclipse Modeling Framework (EMF) model [89] whereas the ARs perform transformations on that model. The parsing and serialization back to CML code is realized with the Xtext framework [23].

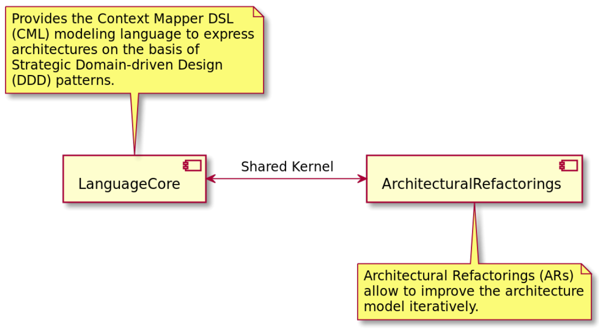

Provides the Context Mapper DSL (CML) modeling language to express architectures on the basis of Strategic Domain-driven Design (DDD) patterns.
LanguageCore
Shared Kernel
ArchitecturalRefactorings
Architectural Refactorings (ARs) allow to improve the architecture model iteratively.

FIGURE 4.14: Architectural Refactorings (ARs) Framework Interfaces

4.5.1 Additional Refactorings

As explained in Chapter 3, four ARs have been added to Context Mapper as part of this thesis project. These new refactorings focus on Context Map relationships while the existing ARs from our previous work [50] focus on the decomposition of Bounded Contexts and Aggregates.

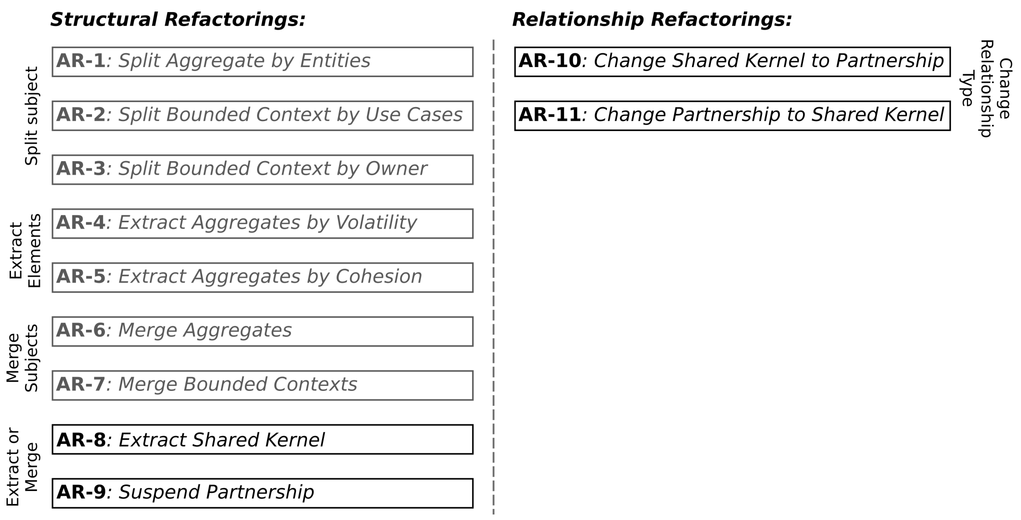

Structural Refactorings:
Split subject
AR-1: Split Aggregate by Entities
AR-2: Split Bounded Context by Use Cases
AR-3: Split Bounded Context by Owner
Extract Elements
AR-4: Extract Aggregates by Volatility
AR-5: Extract Aggregates by Cohesion
Merge Subjects
AR-6: Merge Aggregates
AR-7: Merge Bounded Contexts
Extract or Merge
AR-8: Extract Shared Kernel
AR-9: Suspend Partnership
Relationship Refactorings:
AR-10: Change Shared Kernel to Partnership
AR-11: Change Partnership to Shared Kernel
Change Relationship Type

FIGURE 4.15: Architectural Refactorings (ARs) by Category

Figure 4.15 shows a categorization that includes the new ARs introduced with this thesis. In the following we will explain the four added ARs (AR-8, AR-9, AR-10, and AR-11) with corresponding examples.

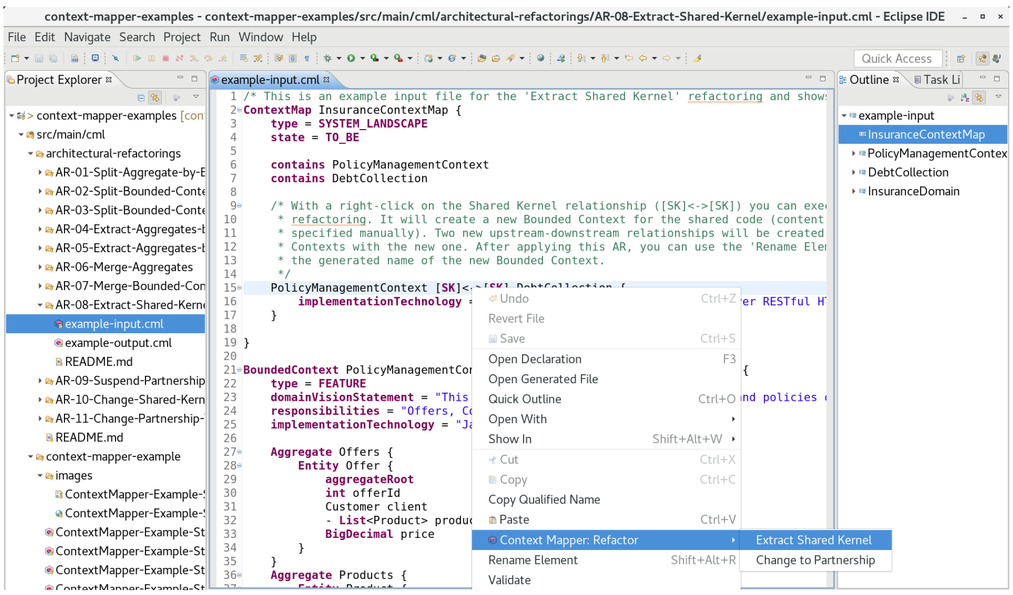
4.5.2 AR-8: Extract Shared Kernel

Extract Shared Kernel offers the possibility to suspend an existing Shared Kernel relationship by moving the Shared Kernel into a new Bounded Context. We already explained the details about the refactoring in Chapter 3. Listing 27 illustrates a CML Context Map on which this refactoring can be applied.

```
ContextMap InsuranceContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  contains PolicyManagementContext, DebtCollection

  PolicyManagementContext [SK]<->[SK] DebtCollection
}

BoundedContext PolicyManagementContext

BoundedContext DebtCollection
```

LISTING 27: AR-8: Extract Shared Kernel Example (Input)

Note that all examples here are simplified and contain only the necessary parts to illustrate the transformation. The complete examples can be found in our GitHub repository[20].



context-mapper-examples - context-mapper-examples/src/main/cml/architectural-refactorings/AR-08-Extract-Shared-Kernel/example-input.cml - Eclipse IDE

```
/* This is an example input file for the 'Extract Shared Kernel' refactoring and shows
ContextMap InsuranceContextMap {
    type = SYSTEM_LANDSCAPE
    state = TO_BE

    contains PolicyManagementContext
    contains DebtCollection

    /* With a right-click on the Shared Kernel relationship ([SK]<->[SK]) you can exe
     * refactoring. It will create a new Bounded Context for the shared code (content
     * specified manually). Two new upstream-downstream relationships will be created
     * Contexts with the new one. After applying this AR, you can use the 'Rename Ele
     * the generated name of the new Bounded Context.
     */
    PolicyManagementContext [SK]<
        implementationTechnology =                                  er RESTful H
    }

}

BoundedContext PolicyManagementCo                                   {
    type = FEATURE
    domainVisionStatement = "This                                   nd policies 
    responsibilities = "Offers, Co
    implementationTechnology = "Ja

    Aggregate Offers {
        Entity Offer {
            aggregateRoot
            int offerId
            Customer client
            - List<Product> produc
            BigDecimal price
        }
    }
    Aggregate Products {
```

FIGURE 4.16: Architectural Refactoring Example in Context Mapper Eclipse Plugin

Within our IDE (Eclipse plugin) a user can refactor a Shared Kernel relationship with a right-click on it as shown in Figure 4.16.

Applying *Extract Shared Kernel* to the example illustrated in Listing 27 produces the result shown in Listing 28. The refactoring creates a new Bounded

[20]`https://github.com/ContextMapper/context-mapper-examples`

Context for the Shared Kernel and two new Upstream-Downstream relationships. Since the shared domain model has been used by both original Bounded Contexts we assume that both need a relationship with the new generated context. The generated Bounded Context contains an Aggregate and a «root» Entity acting as empty placeholders.

```
ContextMap InsuranceContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  contains PolicyManagementContext, DebtCollection
  contains PolicyManagementContext_DebtCollection_SharedKernel

  PolicyManagementContext_DebtCollection_SharedKernel [U]->[D] PolicyManagementContext

  PolicyManagementContext_DebtCollection_SharedKernel [U]->[D] DebtCollection
}

BoundedContext PolicyManagementContext

BoundedContext DebtCollection

BoundedContext PolicyManagementContext_DebtCollection_SharedKernel {
  Aggregate SharedKernelAggregate {
    Entity SharedKernelRoot {
      aggregateRoot
    }
  }
}
```

LISTING 28: AR-8: Extract Shared Kernel Example (Output)

We currently can not generate the domain model of the Shared Kernel since the CML language does not support to specify which parts of the models actually belong to the kernel. This might be an improvement for future releases.

4.5.3 AR-9: Suspend Partnership

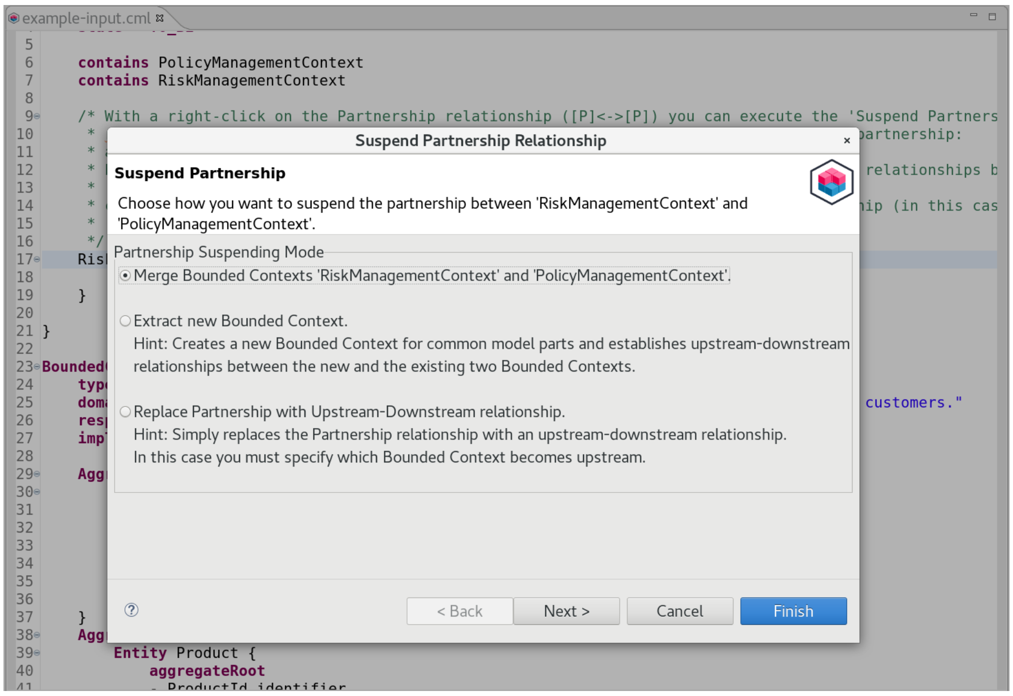
Suspend Partnership is a similar refactoring to *Extract Shared Kernel*, but suspending a Partnership currently provides more options.

```
ContextMap InsuranceContextMap {
  contains PolicyManagementContext, RiskManagementContext

  RiskManagementContext [P]<->[P] PolicyManagementContext
}

BoundedContext PolicyManagementContext

BoundedContext RiskManagementContext
```

LISTING 29: AR-9: Suspend Partnership Example (Input)

Listing 29 shows an example model on which the refactoring can be applied. Applying this refactoring gives the user a choice of three options to suspend the Partnership. Figure 4.17 shows the options within our plugin.

The first one allows the user to merge the two Bounded Contexts. In this case the wizard will ask which context shall remain. The other context will then be merged into the remaining one. If applied to the model in Listing 29, this option leads to the result illustrated in Listing 30.


Suspend Partnership Relationship

Suspend Partnership

Choose how you want to suspend the partnership between 'RiskManagementContext' and 'PolicyManagementContext'.

Partnership Suspending Mode

- [✓ Merge Bounded Contexts 'RiskManagementContext' and 'PolicyManagementContext'.]
- Extract new Bounded Context.
  Hint: Creates a new Bounded Context for common model parts and establishes upstream-downstream relationships between the new and the existing two Bounded Contexts.
- Replace Partnership with Upstream-Downstream relationship.
  Hint: Simply replaces the Partnership relationship with an upstream-downstream relationship.
  In this case you must specify which Bounded Context becomes upstream.

FIGURE 4.17: AR-9: Suspend Partnership Options

In the case of such a simple model with only two Bounded Context the resulting CML file will only contain one context without any relationships.

```
ContextMap InsuranceContextMap {
        type = SYSTEM_LANDSCAPE
        state = TO_BE

        contains RiskManagementContext
}

BoundedContext RiskManagementContext
```

LISTING 30: AR-9: Suspend Partnership Example
(Output Option 1)

The second option would be similar to *Extract Shared Kernel*. In this case we assume the Partnership is defined by some commonalities which can be extracted to a new Bounded Context in order to suspend the Partnership. By creating a new Bounded Context the Partnership relationship can be replaced with two Upstream-Downstream relationships. Listing 31 illustrates the resulting CML model for this option. The refactoring further creates an Aggregate and a «root» Entity as empty placeholders within the new Bounded Context.

```
ContextMap InsuranceContextMap {
  contains PolicyManagementContext, RiskManagementContext
  contains RiskManagement_PolicyManagement_Partnership

  RiskManagement_PolicyManagement_Partnership [U]->[D] RiskManagementContext

  RiskManagement_PolicyManagement_Partnership [U]->[D] PolicyManagementContext
}

BoundedContext PolicyManagementContext

BoundedContext RiskManagementContext

BoundedContext RiskManagement_PolicyManagement_Partnership {
  Aggregate CommonModelAggregate {
    Entity CommonModelPartRoot {
      aggregateRoot
    }
  }
}
```

LISTING 31: AR-9: Suspend Partnership Example (Output Option 2)

A third simple option to suspend the Partnership is to replace the relationship only. This option does not change the model structurally but replaces the existing relationship with an Upstream-Downstream relationship. This could for example be the case if two development teams decide that one of the two teams includes all commonalities into their own Bounded Context. With that the team taking over the common parts becomes the upstream and the other team the downstream within the new relationship. Listing 32 illustrates the according result written in CML. The user must manually select in the wizard which Bounded Context becomes upstream.

```
ContextMap InsuranceContextMap {
  contains PolicyManagementContext, RiskManagementContext

  PolicyManagementContext [U]->[D] RiskManagementContext
}

BoundedContext PolicyManagementContext

BoundedContext RiskManagementContext
```

LISTING 32: AR-9: Suspend Partnership Example (Output Option 3)

The two presented ARs both concern Context Map relationships which also affect the decomposition structurally. The next refactorings, AR-10 and AR-11, have no impact to the structure of the decomposition.

4.5.4 AR-10: Change Shared Kernel to Partnership

This refactoring simply changes a Shared Kernel relationship to a Partnership relationship. Applied to the relationship illustrated in Listing 33 results in the changed relationship shown in Listing 34.

```
ContextMap InsuranceContextMap {
  contains PolicyManagementContext, DebtCollection

  PolicyManagementContext [SK]<->[SK] DebtCollection
}

BoundedContext PolicyManagementContext

BoundedContext DebtCollection
```

LISTING 33: AR-10: Shared Kernel to Partnership Example (Input)

```
ContextMap InsuranceContextMap {
  contains PolicyManagementContext, DebtCollection

  PolicyManagementContext [P]<->[P] DebtCollection
}

BoundedContext PolicyManagementContext

BoundedContext DebtCollection
```

LISTING 34: AR-10: Shared Kernel to Partnership Example (Output)

4.5.5 AR-11: Change Partnership to Shared Kernel

We do not show another example for AR-11 since it is trivially the inverse transformation of the already explained AR-10 above. Applying *Change Partnership to Shared Kernel* to the model in Listing 34 would transform it back to Listing 33.

With the last two sections we discussed the implementation details of the two framework components to decompose the CML models, namely the ARs and the Service Cutter integration. The next section introduces the last part of the modeling framework: the generators.

4.6 Generators

The generators component of the proposed modeling framework offers users the possibility to transform the CML models into other representations. These can be graphical representations such as UML diagrams or formats to integrate with other tools. As the extract of the framework component diagram in Figure 4.18 shows, the generators depend on the CML core component. They all read the data from existing models and generate other artifacts based on these information.

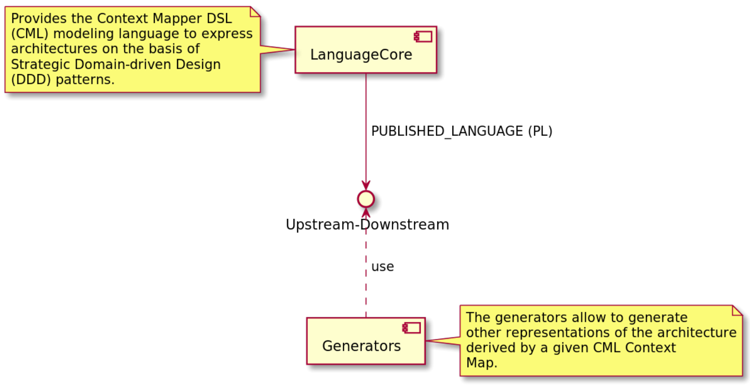

Provides the Context Mapper DSL (CML) modeling language to express architectures on the basis of Strategic Domain-driven Design (DDD) patterns.
LanguageCore
PUBLISHED_LANGUAGE (PL)
Upstream-Downstream
use
Generators
The generators allow to generate other representations of the architecture derived by a given CML Context Map.

FIGURE 4.18: Generators Framework Interface

With our previous projects [46, 50] we already implemented generators producing PlantUML [72] diagrams, MDSL contracts, and Service Cutter [35] input files (JSON). The PlantUML and MDSL generators have been improved during this thesis project according to the feedback collected with our validation activities. In addition to this, we added a new graphical Context Map generator. The illustrated interface in Figure 4.18 is mainly defined by the CML meta-model introduced in the beginning of this chapter.

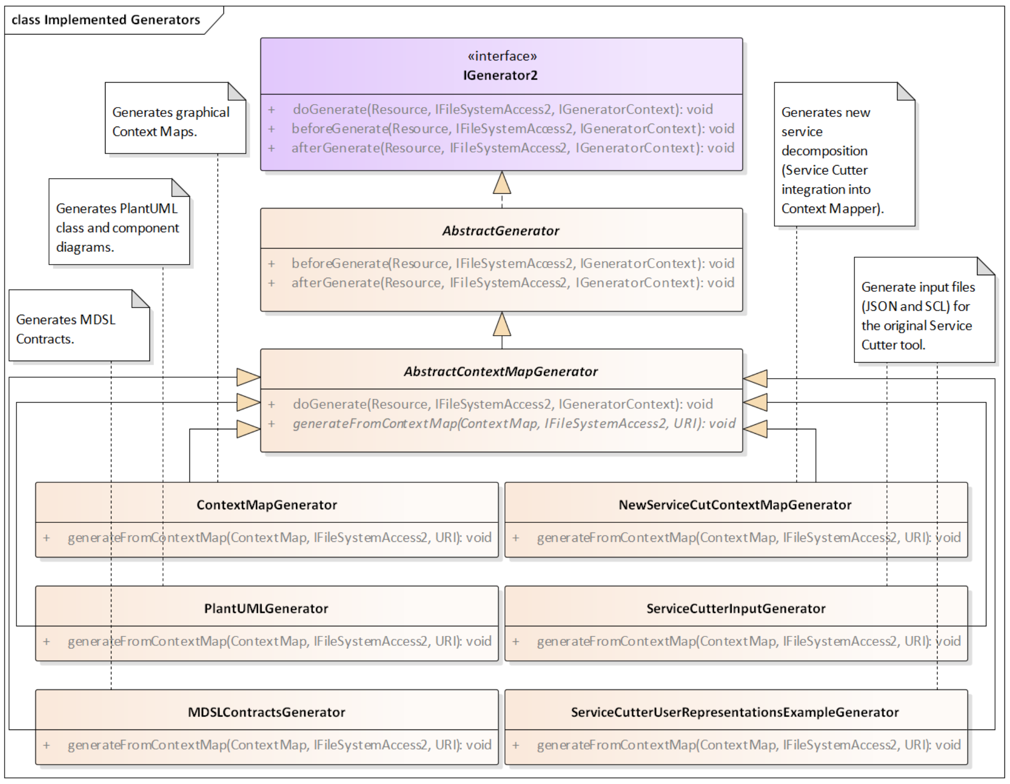

class Implemented Generators
«interface»
IGenerator2
+ doGenerate(Resource, IFileSystemAccess2, IGeneratorContext): void
+ beforeGenerate(Resource, IFileSystemAccess2, IGeneratorContext): void
+ afterGenerate(Resource, IFileSystemAccess2, IGeneratorContext): void
AbstractGenerator
+ beforeGenerate(Resource, IFileSystemAccess2, IGeneratorContext): void
+ afterGenerate(Resource, IFileSystemAccess2, IGeneratorContext): void
AbstractContextMapGenerator
+ doGenerate(Resource, IFileSystemAccess2, IGeneratorContext): void
+ generateFromContextMap(ContextMap, IFileSystemAccess2, URI): void
ContextMapGenerator
+ generateFromContextMap(ContextMap, IFileSystemAccess2, URI): void
NewServiceCutContextMapGenerator
+ generateFromContextMap(ContextMap, IFileSystemAccess2, URI): void
PlantUMLGenerator
+ generateFromContextMap(ContextMap, IFileSystemAccess2, URI): void
ServiceCutterInputGenerator
+ generateFromContextMap(ContextMap, IFileSystemAccess2, URI): void
MDSLContractsGenerator
+ generateFromContextMap(ContextMap, IFileSystemAccess2, URI): void
ServiceCutterUserRepresentationsExampleGenerator
+ generateFromContextMap(ContextMap, IFileSystemAccess2, URI): void
Generates graphical Context Maps.
Generates PlantUML class and component diagrams.
Generates MDSL Contracts.
Generates new service decomposition (Service Cutter integration into Context Mapper).
Generate input files (JSON and SCL) for the original Service Cutter tool.

FIGURE 4.19: Implemented Generators (UML Class Diagram)

Figure 4.19 shows the implemented generators in a UML class diagram with the common interface they implement.

All generators implement the *IGenerator2* interface provided by the Xtext [23] framework. The *AbstractGenerator* class is provided by Xtext as well. The *AbstractContextMapGenerator* is our own abstract class used to generate outputs with a Context Map as input. Generators implementing this abstract class expect the given EMF *Resource* to contain a CML Context Map. All generators process the Context Map by implementing the *generateFromContextMap* method.

With the common interface the usage of a generator is always implemented in the same way from user perspective. Listing 35 shows how it is done for the PlantUML example. To use another one, only line 5 has to be replaced with the instantiation of the preferred generator.

```java
1   ContextMappingDSLStandaloneSetup.doSetup();
2   Resource resource = new ResourceSetImpl().getResource(URI.createURI("input.cml"), true);
3
4   // Create the generator (can be any other generator of our framework as well)
5   IGenerator2 generator = new PlantUMLGenerator();
6
7   // Generate the output files into 'src-gen'
8   JavaIoFileSystemAccess javaIoFileSystemAccess = FileSystemHelper.getFileSystemAccess();
9   javaIoFileSystemAccess.setOutputPath("./src-gen");
10  generator.doGenerate(resource, javaIoFileSystemAccess, new GeneratorContext());
```

LISTING 35: Generators Usage in Java (Standalone)

Note that this example illustrates the usage in a standalone scenario outside Eclipse. The usage in Eclipse is basically the same but without calling the *doSetup* method of the *ContextMappingDSLStandaloneSetup* class. The *IFileSystemAccess2* and *IGeneratorContext* instances are initialized in a different way as well.

4.6.1 Graphical Context Map Generator

As already mentioned, the implementation of a graphical Context Map generator is part of the scope of this thesis. The selection of the graphical tool to implement the generator, Graphviz [30], is documented in Chapter 3. Note that this generator is the only one with system preconditions since Graphviz must be installed on the users computer to execute it.

To realize this generator we first implemented a Java library[21] to generate such Context Maps based on Graphviz which is independent of our framework. The generator implementation within our framework uses this library. Therefore, we provide a Context Map generator which could be used in any other frameworks or modeling tools realized in Java as well. The API is very straight forward as shown in Listing 36. The graphical illustration in Figure 4.20 is automatically generated with this piece of Java code.

[21]`https://github.com/ContextMapper/context-map-generator`

```java
public static void main(String[] args) throws IOException {
  BoundedContext cargoBookingContext = new BoundedContext("Cargo Booking Context");
  BoundedContext voyagePlanningContext = new BoundedContext("Voyage Planning Context");
  BoundedContext locationContext = new BoundedContext("Location Context");

  ContextMap contextMap = new ContextMap()
    .addBoundedContext(cargoBookingContext)
    .addBoundedContext(voyagePlanningContext)
    .addBoundedContext(locationContext)
    .addRelationship(new SharedKernel(cargoBookingContext, voyagePlanningContext))
    .addRelationship(new UpstreamDownstreamRelationship(locationContext,
                                                         cargoBookingContext)
      .setUpstreamPatterns(OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE))
    .addRelationship(new UpstreamDownstreamRelationship(locationContext,
                                                       voyagePlanningContext)
      .setUpstreamPatterns(OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE));

  new ContextMapGenerator().setLabelSpacingFactor(4)
    .generateContextMapGraphic(contextMap, Format.PNG, "/home/user/context-map.png");
}
```

LISTING 36: Context Map Generator Library:
Java Usage Example

The first two code blocks construct the example Context Map by defining the Bounded Contexts and the Context Map relationships. The model corresponds to our meta-model introduced in the beginning of this chapter. The last two lines generate the image. The *label spacing factor* parameter has been introduced to solve label overlapping problems. Sometimes Graphviz produces graphics with overlapping labels. By increasing the mentioned factor the distance between graph edges is increased which solves the problem almost always. We currently support the generation of the Context Maps in the following three formats: Scalable Vector Graphics (SVG), Portable Network Graphics (PNG), and Graphviz's Graph Description Language (DOT).

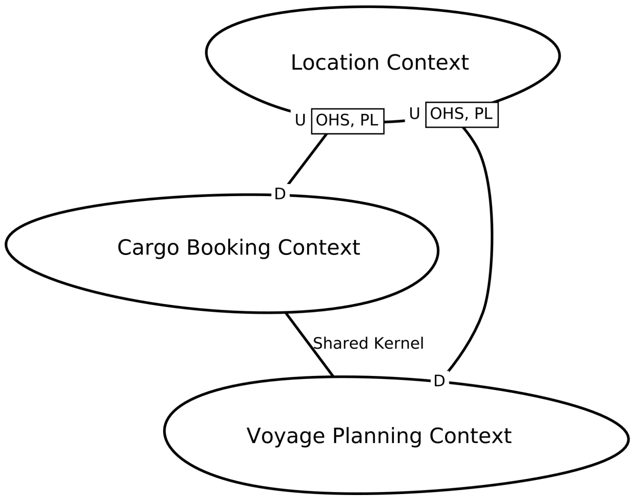
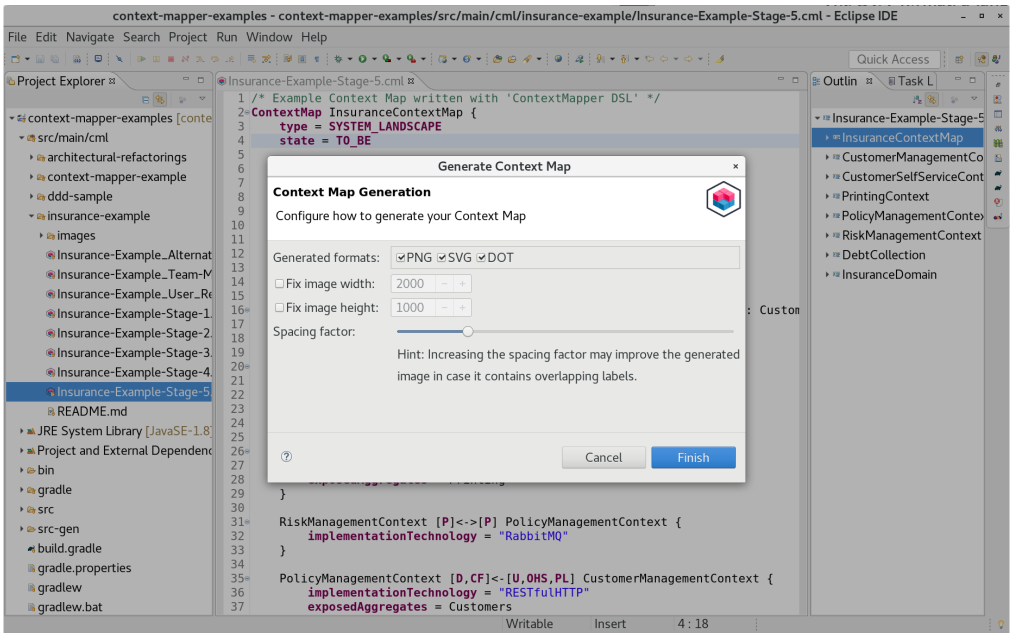


FIGURE 4.20: Output Image (Context Map) for Example in
Listing 36

The DOT format, a language specifically used to describe graphs, is also supported by other graph tools apart from Graphviz. This allows to continue editing the generated Context Maps in other tools. Of course this is also possible with SVG files.

The user does not have to write code to generate the Context Map with the generator implementation in our core component. It is automatically constructed and generated from CML Context Maps. The user can parameterize the generator with the UI shown in Figure 4.21 within our Eclipse plugin easily.


context-mapper-examples - context-mapper-examples/src/main/cml/insurance-example/Insurance-Example-Stage-5.cml - Eclipse IDE
File Edit Navigate Search Project Run Window Help
Quick Access
Project Explorer
context-mapper-examples [conte
src/main/cml
architectural-refactorings
context-mapper-example
ddd-sample
insurance-example
images
Insurance-Example_Alternat
Insurance-Example_Team-M
Insurance-Example_User_Re
Insurance-Example-Stage-1.
Insurance-Example-Stage-2.
Insurance-Example-Stage-3.
Insurance-Example-Stage-4.
Insurance-Example-Stage-5.
README.md
JRE System Library [JavaSE-1.8]
Project and External Dependenc
bin
gradle
src
src-gen
build.gradle
gradle.properties
gradlew
gradlew.bat
Insurance-Example-Stage-5.cml
Outlin
Task L
Insurance-Example-Stage-5
InsuranceContextMap
CustomerManagementCo
CustomerSelfServiceCont
PrintingContext
PolicyManagementContex
RiskManagementContext
DebtCollection
InsuranceDomain
Writable
Insert
4 : 18

```
/* Example Context Map written with 'ContextMapper DSL' */
ContextMap InsuranceContextMap {
    type = SYSTEM_LANDSCAPE
    state = TO_BE
                                                              : Custom
}

RiskManagementContext [P]<->[P] PolicyManagementContext {
    implementationTechnology = "RabbitMQ"
}

PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerManagementContext {
    implementationTechnology = "RESTfulHTTP"
    exposedAggregates = Customers
```

Generate Context Map

Context Map Generation

Configure how to generate your Context Map

Generated formats: [✓ PNG] [✓ SVG] [✓ DOT]

☐ Fix image width: 2000

☐ Fix image height: 1000

Spacing factor:

Hint: Increasing the spacing factor may improve the generated image in case it contains overlapping labels.

Cancel Finish

FIGURE 4.21: Context Map Generation Parameters in Context Mapper

The parametrization further allows to control the output image resolution by providing a fixed width or height.

Validation and Examples

We tested the Context Map generator with the models in our examples repository[22] and the Lakeside Mutual [61] case study reverse engineered with our discovery library. The results were positive although we do not have an influence to the layout algorithms of Graphviz.

Figure 4.22 illustrates the Context Map for the Lakeside Mutual project [61]. It is generated out of the CML model which has been reverse engineerd as explained in Section 4.3 of this chapter. This example does not contain any relationship patterns such as PL or OHS since the discovery library cannot detect such patterns yet.

Another example of a Context Map generated with our tool has already been presented in Chapter 3. The Context Map illustrating our own framework's Bounded Contexts was generated with Context Mapper. The output for the DDD «Cargo» sample application which is also modeled in our examples

[22]`https://github.com/ContextMapper/context-mapper-examples`

repository has already been shown in Figure 4.20 of this section. Note that regenerating Context Maps with the same CML input always results in slightly different results since the shapes of the Bounded Contexts are computed randomly.

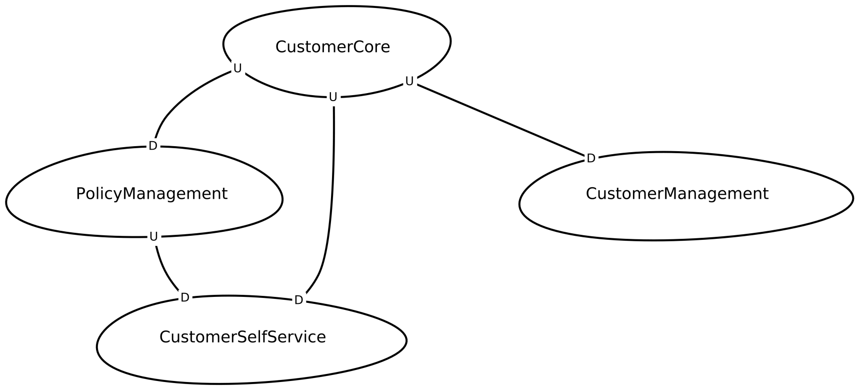


FIGURE 4.22: Lakeside Mutual [61] Context Map
(Generated with Context Mapper)

An example which incorporates all patterns supported by CML is shown in Figure 4.23. It is the Context Map for our fictitious insurance scenario. The Context Map features all types of relationships: Partnership, Shared Kernel, Upstream-Downstream, and Customer-Supplier. With PL, OHS, ACL and CF, it also contains all supported relationship patterns.

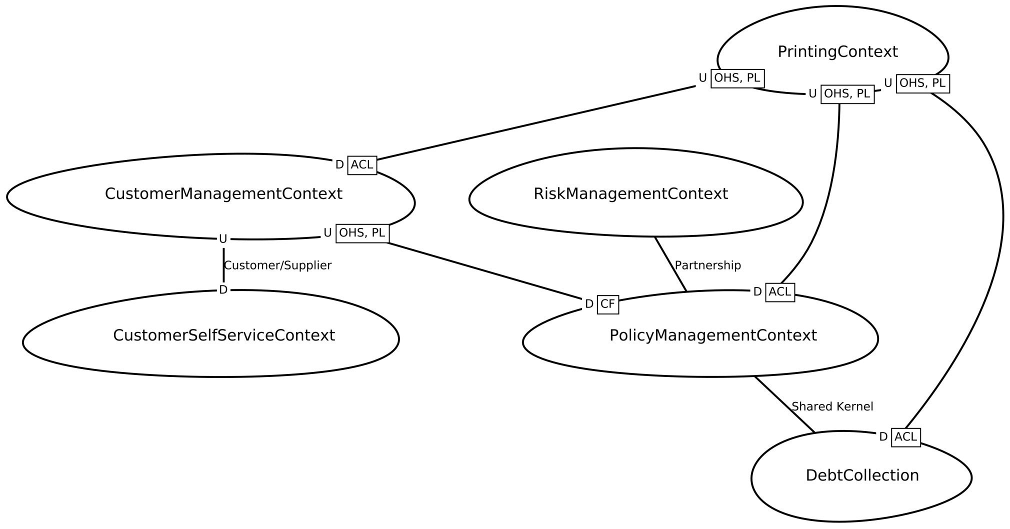


FIGURE 4.23: Fictitious Insurance Scenario Context Map
(Generated with Context Mapper)

The generator produced good results for all our Context Maps although we have no influence on the layout. In addition, the tool was tested with Context Maps containing more than twenty Bounded Contexts and still produced readable results. However, if one Bounded Context has many relationships with other contexts, the generator may produce graphics with overlapping relationship pattern labels. This can not be avoided completely.

4.6.2 PlantUML Generator

Existing generators including the PlantUML [72] generator developed during the first term project [46] have been improved during this thesis according to the collected user feedback. Table 4.4 summarizes the PlantUML generator changes realized during this project.

TABLE 4.4: PlantUML Generator Improvements

Change	Description
Class diagrams for subdomains	CML already supported the usage of Entities within subdomains to specify which domain objects are part of such a subdomain. With this change the PlantUML generator not only produces class diagrams for Bounded Contexts but for subdomains as well (as long as a subdomain contains Entities).
Adjustments in interface labels	The labels of interface exposures in component diagrams state which Aggregates are exposed concretely (exposed Aggregates must be specified in CML).
Support *nullable* attributes	Attributes which are specified as *nullable* in CML are now marked as such in the generated class diagrams.
Support inheritance	The tactic DDD syntax based on Sculptor [83] supports to model inheritance. With this change the PlantUML class generator respects these inheritance hierarchies and represents them in the UML diagrams correctly.
Customer-Supplier relationship labels	The labels for Customer-Supplier relationships in component diagrams indicate which component is the customer and which one is the supplier now.
Improved domain object reference labels	The labels of references to other domain objects in class diagrams now show the corresponding attribute name (source of reference).

Since this generator already existed before this thesis project we do not show any examples or more details in this report. Readers interested in the generator and example diagrams may consult our online documentation[23].

4.6.3 MDSL Service Contract Generator

The second generator which has been improved during this thesis is the MDSL service contract generator. Table 4.5 lists and documents the improvements and changes realized.

TABLE 4.5: MDSL Generator Improvements

Change	Description
Support *nullable* attributes	Attributes which are specified as *nullable* in CML are mapped to the corresponding concept in MDSL correctly.

[23] `https://contextmapper.org/docs/plant-uml/`

TABLE 4.5: MDSL Generator Improvements (continued)

Change	Description
Added Domain Vision Statements and relationship patterns to MDSL output	The Domain Vision Statements of Bounded Contexts and the relationship patterns (PL, OHS, ACL, CF) are now added as comments to the generated MDSL output.
Generate *usage context*	The attribute *usage context* of MDSL is now generated as well.
Protected regions [32]	The generator now supports protected regions and re-generation. This means that the generation of a contract can be executed multiple times to override the existing contract with new changes modeled in CML. Protected regions which are marked with predefined comments are not changed if a contract is overwritten. With this change we allow the user to adjust generated contracts and support re-generation at the same time. Protected regions are one approach to integrate handwritten and generated code [32].
Generate resource roles and responsibilities	MDSL supports the specification or resource roles (*information holder*, *processing resource*, etc.) on endpoints and operation responsibilities (*computation function*, *retrieval operation*, etc.) on operations. The generator extracts the corresponding MAP pattern names [105] from CML documentation strings (supported by Sculptor [83] syntax) and generates the attributes in MDSL now.
Support enums	Enums in CML are now mapped to MDSL as well.
Resolve inherited attributes	If a domain object inherits attributes from other objects (*extends* mechanism), these attributes are resolved and respected in the generated parameter trees [105] in MDSL.

Note that we do not introduce MDSL language details in this report. A documentation of the language can be found online[24]. We already provided an example for a concrete MDSL application with the Service Cutter interface documentation in Section 4.4. More information about the generator can be found online[25] as well.

With the generators we discussed the design and implementation of the last component of the proposed framework architecture. The next and last section of this chapter discusses changes in our CML example models briefly.

[24]`https://socadk.github.io/MDSL/`

[25]`https://contextmapper.org/docs/mdsl/`

4.7 Context Mapper DSL (CML) Example Models

As already mentioned in Chapter 2, our validation activities and user feedbacks showed that our example CML models[26] were too detailed for certain user groups. According to our own requirements, these example models had to be enhanced so that the language can be introduced to all target user roles. Users such as business analysts without technical background as well as software architects must find example models which suit them.

This issue has been solved by providing and evolving the models in multiple stages. Each stage adds additional details. Users without technical background may be interested in the first stages while software engineers or architects might prefer the last stages with the detailed domain models. Table 4.6 summarizes the stages we currently provide for the CML models in our examples repository.

TABLE 4.6: CML Example Model Stages

Stage	Description

Stage 1	The first stage models a Context Map in its simplest form. There are no relationship patterns yet and the Bounded Contexts do not contain any details (Aggregates or domain objects). The following example illustrates a Context Map in this stage:
	`ContextMap DDDSampleMap {`
	`  contains CargoBookingContext, LocationContext`
	
	`  CargoBookingContext <- LocationContext`
	`}`
	`BoundedContext CargoBookingContext`
	`BoundedContext LocationContext`

Stage 2	The second stage refines the Context Map relationships of the examples and adds relationship patterns (PL, OHS, ACL, CF). The relationship of the example in stage one could look like this in stage two: `CargoBookingContext [D]<-[U,OHS,PL] LocationContext`

Stage 3	Stage three starts to increase the level of detail with respect to Bounded Contexts. The Aggregates are added, but still no domain objects. The following example shows a Bounded Context in stage three:
	`BoundedContext CargoBookingContext {`
	`  Aggregate CargoItineraryLegDeliveryRouteSpecification`
	`  Aggregate Handling`
	`}`

Stage 4	This stage adds simple domain object (Entities, Value Objects, etc.) to the Aggregates. The domain objects do not contain any attributes. Thus, this stage just expresses which objects are part of the Bounded Context domain models but does not specify the objects in detail (attributes, methods, etc.). This stage may still be understandable for many users without technical background. For example, the *CargoItineraryLegDeliveryRouteSpecification* Aggregate of the previous stage might be enriched as follows:

[26] https://github.com/ContextMapper/context-mapper-examples

TABLE 4.6: CML Example Model Stages (continued)

Stage	Description

Stage 4 (cont.)	`Aggregate CargoItineraryLegDeliveryRouteSpecification {`
	`  Entity Cargo`
	`  ValueObject Delivery`
	`  ValueObject Itinerary`
	`}`

Stage 5	Stage five finally adds all details within the Aggregates and domain objects. Attributes and methods are specified within the domain objects and the Aggregates can contain services, repositories, and so on. This stage clearly addresses users with technical and programming background. The Aggregate in the previous example could look as follows in this stage:
	`Aggregate CargoItineraryLegDeliveryRouteSpecification {`
	`  Entity Cargo {`
	`    aggregateRoot`
	
	`    TrackingId trackingId`
	`    - LocationShared origin`
	`    - RouteSpecification routeSpecification`
	`    - Itinerary itinerary`
	`    - Delivery delivery`
	
	`    Repository CargoRepository {`
	`      @Cargo find(TrackingId tId) throws CargoNotFoundException;`
	`      List<@Cargo> findAll;`
	`      store(@Cargo cargo);`
	`      TrackingId nextTrackingId();`
	`    }`
	`  }`
	`  Service RoutingService {`
	`    List<@Itinerary> fetchRoutesForSpec(@RouteSpecification rs);`
	`  }`
	`}`

More user feedback regarding this approach has to be collected in future projects to evaluate whether it solved the mentioned problem. More stages could be developed in the future as well. Stage one already contains the subdomains which are realized by the Bounded Contexts (not shown above but in our examples repository). A stage zero without subdomains might further simplify the introduction to the CML language for new users.

With the improvement of our example models we conclude this chapter documenting the design and implementation work of this thesis. The next chapter evaluates the results and discusses the benefits and potential liabilities of the proposed framework.

Chapter 5

Evaluation and Discussion

This chapter discusses our validation activities and evaluates the fulfillment of the requirements presented in Chapter 2 and the defined project goals. It further discusses strengths and potential weaknesses of the proposed approach.

5.1 Validation via Prototyping, Action Research and Case Studies

In this section we discuss the validation goals and techniques. We further outline the applied validation activities for the individual framework components and the corresponding results.

5.1.1 Validation Goals

Research contributions such as the framework proposed by this thesis must be validated. In Chapter 1 we hypothesized that software architects, service designers, and business analysts applying Domain-driven Design (DDD) can benefit from a modeling framework such as ours. The main goal of our validation activities was to demonstrate the *usefulness* of the framework and that it is indeed *beneficial* for the target user group. Validating the *feasibility* of the proposed concepts and the *usability* of the tools were other important objectives.

5.1.2 Validation Techniques

The prototypic implementation of the framework in the Context Mapper open source project [15], the application of the tool in case studies, and action research [4] were our primary validation activities. In addition, we conducted self-experiments and used the tool for teaching at our institution.

The implementation combined with action research allowed us to improve the framework and its tools iteratively and with short feedback cycles. The supervisor of this thesis applied the tool in projects and provided feedback based on experience on a regular basis. Case studies such as the Lakeside Mutual project [61] further allowed us to validate the *practicability* of individual framework components in specific use cases. Besides that, we conducted a self-experiment by using our own tool to model and document the architecture of the framework itself. All these activities supported us in validating the *usefulness* and *usability* of the tools. Table 5.1 provides an overview over all validation activities for the individual framework components.

TABLE 5.1: Validation Activities Overview

Component (or Use Case)	**Validation Activities**
Complete Modeling Framework	Prototyping (the Context Mapper tool represents the prototype)
Core: CML modeling language	Action research, case studies, teaching, self-experiment
Discovery library (reverse engineering)	Lakeside Mutual [61] case study
ARs	Action research, small case studies (Context Mapper examples[1])
Systematic service decomposition (Service Cutter library)	Fictitious insurance case study[1]
Generators	Action research, case studies, self-experiment

The applied validation approach corresponds to Shaw's validation type *experience* [85]. This type of validation suits the kind of problem we address in this thesis, in contrast to analysis based on rigorous proofs and controlled experiments. It has the goal to demonstrate *correctness*, *usefulness*, and *effectiveness* of the proposed concepts, framework, and tools.

5.1.3 Validation Results

In the following we discuss the validation activities and their results for each individual framework component.

Core: CML Modeling Language

The CML language as core component of the modeling framework has evolved iteratively by applying action research. We further validated the language with the case studies in our examples repository[1]. In addition, we modeled our own framework architecture in CML (self-experiment) as part of this thesis. Different versions of the modeling language were further used as part of an application architecture course by 20 exercise participants at our institution. These computer science students and the advisor of this thesis modeled a real-world scenario from the oil industry [99] with the CML language. We questioned and evaluated whether they were able to understand the example models and write their own models within given time frames. The collected feedback was generally positive and we implemented corresponding improvements during this thesis.

By applying all these validation activities we were able to improve the *usability* of the language during our previous work [46, 50] and this thesis. The results support our hypothesis that a Domain-specific Language (DSL) to model DDD Context Maps in formal manner can be *beneficial* and *useful* for software architects and business analysts. The implementation in Context Mapper confirmed the *feasibility* of the model-driven approach based on a DSL.

[1]`https://github.com/ContextMapper/context-mapper-examples`

Discovery Library

The discovery library to reverse engineer CML models was validated with the Lakeside Mutual [61] case study. As explained in Chapter 4 we were able to recover the architecture of this microservice project using our library with a few lines of code. The resulting Context Map contains all Bounded Contexts besides the ones implemented in JavaScript. To discover the architecture completely, new discovery strategies have to be implemented in future projects. We validated the discovered Context Map by comparing it with the architecture overview provided by the authors of the project. In addition, we had the model checked by one of the authors. The case study indicates that our approach towards reverse engineering is *feasible* and *useful* for adopters in brownfield projects.

Architectural Refactorings (ARs)

So far, the ARs have only been validated by implementing them as code refactorings in Context Mapper and by applying them to our example models [14]. More validation has to be done in future projects to show their *usefulness* and *effectiveness*. A chain of ARs shall be applied to illustrate how a real world system can be decomposed stepwise.

Systematic Service Decomposition

We validated the systematic service decomposition component of our framework by applying it to our fictitious insurance case study [14]. Since we have not integrated a new algorithm into the Service Cutter [35] engine, the results with respect to the generated decompositions itself do not deviate from the results presented in our previous project [46]. In this thesis we focused on validating the *usefulness* and *effectiveness* of the Service Cutter integration into Context Mapper rather than the decomposition results itself. The validation has shown that the current algorithm is not practical to decompose a system iteratively. Future projects may focus on other approaches suggesting single AR steps instead of generating a decomposition in one single step. This would also allow integrating the systematic decomposition component with the ARs. In addition, the *usability* of the current prototype is decreased due to the fact that parts of the input model are lost during the decomposition process with Service Cutter. To obtain all information the user is forced to merge the original CML model with the generated decomposition manually. This merging process must be automated in future projects if the approach is pursued.

Generators

The generators component of the framework was validated during this thesis as well. First, we used the PlantUML [72], Microservices Domain-Specific Language (MDSL) [102], as well as the new graphical Context Map generator to document our own framework online[2] and in this report (self-experiment). We applied action research and improved the existing generators (PlantUML and

[2]`https://contextmapper.org/`

MDSL) according to the validation results conducted during this thesis immediately. Especially the MDSL generator was validated intensively, not only by the author of this work but also by the supervisor and creator of the MDSL language. The *usability* and the workflow when utilizing the generator have been improved by introducing *protected regions* [32] and other enhancements listed in Chapter 4. All generators were also applied to the modeled case studies (such as Lakeside Mutual [61]) and example models [14]. The results suggest that the transformations into other representations on different levels of abstraction are *useful* for users of the framework.

5.1.4 Summary and Conclusion

In summary, the validation results over the whole framework were positive and suggest that our hypothesis that software architects and business analysts can benefit from a modeling framework based on strategic DDD is true. We applied prototyping and action research [4] for all framework components. The reverse engineering component was validated with the Lakeside Mutual [61] case study. Additional validation activities included self-experiments and teaching. However, more validation has to be done in future projects, especially regarding the ARs and the systematic service decomposition approach.

5.2 Requirement Fulfillment Evaluation

In Chapter 1 of this report we established the goals of this thesis. A detailed list of required deliverables has been compiled in the project definition [48]. Chapter 2 further specified functional requirements in the form of user stories, as well as Non-Functional Requirements (NFRs). This section discusses the degree of fulfillment for all specified requirements.

5.2.1 Project Deliverables

Table 5.2 lists all project deliverables originally specified by our project definition [48] and how we assess their fulfillment.

TABLE 5.2: Project Deliverables: Fulfillment Evaluation

Deliverable	**Fulfillment**
Known issues and limitations which resulted from user feedback during the previous project should be prioritized and incorporated.	All issues in already existing components which we identified through our validation were respected during this project. The issues were discussed, prioritized, and solved according to the defined priorities. The points which have actually been solved during this thesis can be found in our GitHub issues[3]. Thus, we consider this requirement addressed sufficiently.

[3]`https://github.com/ContextMapper/context-mapper-dsl/issues`

TABLE 5.2: Project Deliverables: Fulfillment Evaluation (continued)

Deliverable	**Fulfillment**
A modular component architecture for strategic DDD modeling tools such as Context Mapper shall be conceptualized. The framework must cover architecture modeling, analysis, ARs, generators, and reverse engineering tools.	With this thesis we present a modeling framework covering the requested components. The CML modeling language conceptualized in our first term project [46] supports the creation of architecture models based on strategic DDD patterns (pattern selection documented in our previous work [46], Appendix A, and online[4]). In addition, we improved the language during this thesis as described in Section 4.2 of Chapter 4. As documented in Section 4.4 we realized a framework component that decomposes CML models in a structured and systematic manner (Service Cutter library). The component analyzes the models based on coupling criteria and suggests new Context Maps with improved coupling and cohesion. With our previous term project [50] and this thesis we implemented eleven ARs as code refactorings for the CML language. All refactorings are documented online[5] or in Appendix B. With the new graphical Context Map generator introduced in Section 4.6 of Chapter 4 we offer a set of four generators[6] now. Finally, the discovery library introduced in Section 4.3 allows users to reverse engineer CML models from existing source code. Thus, we covered all requested framework components and consider this point to be fulfilled.
A reverse engineering library to generate CML code from existing code bases shall be implemented. The solution should be extensible with different Context Map and Bounded Context discovery strategies. This «proof of concept» shall provide two discovery approaches, one for each case.	We proposed an extensible discovery library allowing to inject new Bounded Context and relationship discovery strategies in Section 4.3 of Chapter 4. The prototype includes a strategy to discover Bounded Contexts based on Spring Boot [71] and another one to discover relationships based on Docker Compose [19]. The design of the library allows us to add new discovery strategies in future projects easily. Therefore, this requirement has been fulfilled.

[4] `https://contextmapper.org/docs/language-reference/`

[5] `https://contextmapper.org/docs/architectural-refactorings/`

[6] `https://contextmapper.org/docs/generators/`

TABLE 5.2: Project Deliverables: Fulfillment Evaluation (continued)

Deliverable	**Fulfillment**
The Service Cutter engine shall be integrated into the Context Mapper tool to provide new Context Map suggestions directly within the tool.	We realized a new Service Cutter library and integrated it into the Context Mapper tool as explained in Section 4.4 of Chapter 4. It allows users to generate new Context Maps based on Service Cutter's coupling criteria [37] and a graph clustering algorithm. The library is designed in an extensible fashion which allows us to include new algorithmic solutions in future projects. Although more validation activities and probably new algorithms are needed, we delivered a first approach and prototype that is integrated in Context Mapper.
A new generator should produce graphical Context Maps in the notation inspired by Vernon [98] and Brandolini [10].	The graphical Context Map generator has been implemented with Graphviz [30] as documented in Section 4.6 of Chapter 4. The notation of the generated graphics which is also used in this thesis report is based on the Context Map illustrations by Vernon [98] and Brandolini [10]. Thus, this point is fulfilled as well.
At least two new ARs must allow the users to refactor relationships on the Context Map.	We implemented the following four ARs during this thesis: *Extract Shared Kernel*, *Suspend Partnership*, *Change Shared Kernel to Partnership*, and *Change Partnership to Shared Kernel* (documented in Chapter 4 and Appendix B). These four ARs are all designed to refactor Context Map relationships. This project requirement is thus met.
Adjusted and new model examples must distinguish between software engineering and business analysis users. With this we provide less detailed models which can be read easily by users without engineering background.	As explained in Chapter 4, we introduced five stages with different levels of detail for all our model examples to address this issue. The first stages do not contain technical details and are understandable for users without engineering background while stage four and five introduce the details required by software engineers and architects. We consider the point as fulfilled. However, we may further improve the documentation and the examples based on future user feedback.

TABLE 5.2: Project Deliverables: Fulfillment Evaluation (continued)

Deliverable	Fulfillment
The implemented language tools and refactorings must be easy to use, robust and validated with respect to the project requirements.	ARs can be applied easily and are executed within maximal two to three seconds in the CML editor in our Eclipse plugin, identical to the already existing ARs from our previous work [50]. With our quality practices we held the robustness of all tools as high as possible. The core component with the DSL, the ARs, and the generators have an overall test coverage of 93.86 percent. The test coverages of the discovery library, Service Cutter library and Context Map generator library are at 96.58 percent, 90.48 percent, and 98.82 percent[7]. Therefore, the requirements regarding the robustness of the tools are addressed. The fulfillment regarding the projects functional and non-functional requirements are discussed later in this chapter.
All features shall be documented on the Context Mapper website.	Every improvement or new feature has been mentioned in our release notes[8] and the online documentation[9] was always updated. This requirement is therefore met as well.

In summary, Table 5.2 shows that all requested deliverables from the project definition [48] were respected and at least a minimal prototype has been realized in Context Mapper [15]. The following sections discuss the results of this thesis with respect to the functional and non-functional requirements.

5.2.2 Functional Requirements

Chapter 2 introduced user stories representing the requirements for all framework components. In the following we discuss the actual results for each user story and point out open issues.

US-1: Analysing and Describing the Domain as Business Analyst

The core component of the modeling framework with its CML language allows describing a domain based on the strategic DDD patterns. The integrated Sculptor syntax [83] offers the possibility to describe domain objects (such as Entities, Value Objects, etc.) and therefore the Ubiquitous Language of the domain. With the newly introduced stages in our modeling examples [14] (see Section 4.7 in Chapter 4) we illustrated how a system or domain can be described on a level of detail which is understandable for business analysts or other user roles without technical background.

[7]The test coverages were measured before the submission of this thesis in January 2020.

[8]`https://github.com/ContextMapper/context-mapper-dsl/releases`

[9]`https://contextmapper.org/`

Future validation activities will have to elaborate whether the current presentation of the models fulfills the requirements of business analysts completely. Based on user feedback the amount of stages might be increased and the changes from one stage to the next one minimized. For example, we could evolve the subdomain definitions stepwise as well. The stages currently focus on the evolution of the Bounded Context definitions.

US-2: Describing Software Architecture and Design as Software Architect

The CML modeling language allows software architects to describe systems and their decomposition into (micro-)services in terms of DDD Bounded Contexts. With the Context Map and relationship patterns such as Published Language (PL), Open Host Service (OHS), Anticorruption Layer (ACL), and Conformist (CF) an architect can describe the relationships between components, sub-systems, services, or teams. Thus, the concepts also allow describing Enterprise Application Integration (EAI) solutions.

The generator component of the modeling framework supports the transformation into other representations such as Unified Modeling Language (UML) diagrams required by this user story. With the ARs and the structured service decomposition component we provide tool support to evolve the architecture models iteratively.

US-3: Generate Bounded Contexts and Context Map Models for existing (Micro-)Service Architecture

With the realized discovery library we provide a tool to generate CML Bounded Contexts and Context Maps from existing code bases of microservice projects. Our Lakeside Mutual [61] case study explained in Chapter 4 demonstrated how it works on a real microservice project. As this user story required, the library is able to discover relationships between the Bounded Contexts. The discovered decomposition can then be analyzed, communicated, and improved with the ARs. The generators allow the framework users to transform the discovered architecture into graphical representations such as Context Maps or PlantUML [72] diagrams.

US-4: Generate Bounded Context Model from existing Monolithic System

The discovery library is not only designed to identify Bounded Contexts and their relationships (Context Map), but to reverse engineer the domain models within those Bounded Contexts as well. Thus, users in brownfield projects are allowed to reverse engineer a single Bounded Context from a monolithic system and then decompose it by using the other components of our modeling framework. The ARs allow them to refactor the system iteratively. With the Service Cutter [35] integration a user can decompose the monolith systematically.

The prototypic discovery strategy based on Spring Boot [71] reverse engineers Aggregates, Entities, and Value Objects of the corresponding Bounded Contexts. However, the identification of the domain object type (Entity, Value Object, etc.) might currently not be correct in all cases since we map all Data

Transfer Objects (DTOs) [28] discovered in the RESTful HTTP endpoints to Value Objects. For example, domain objects could be mapped to Value Objects although they might be Entities. Future projects will have to improve the current solution in order to map these domain object types correctly (or at least generate a corresponding comment, if the type can not be discovered). In addition, the current solution produces duplicate domain objects in case the corresponding DTO was discovered in multiple Aggregates. The implementation must consider such situations in the future and consolidate the domain objects beyond Aggregate boundaries. With the current duplication of domain objects (creation of domain object in each Aggregate) we hide coupling between the Aggregates which exists in the real system.

Nevertheless, the current version of the discovery library in combination with the provided ARs and the systematic service decomposition component fulfills the requirements of this user story in principle.

US-5: Analyze a Strategic DDD Model using Coupling Criteria to find Better Service Decompositions

The structured service decomposition component, concretely the Service Cutter [35] integration, is meant to address the requirements of this user story. The tool is based on a coupling criteria catalog [37] and its goal is to generate service decompositions which improve coupling and cohesion.

With the current solution we provide a first «proof of concept» for this user story. However, our validation activities showed that we have to improve this part of the framework in future projects. The generated decompositions in the form of new CML models lose information which were provided by the input model, leading to unsatisfying usability. Because the generated output has to be merged with the original input model manually if the user does not want to lose the information, it is not well integrated into the workflow. Therefore, an architect is only able to use the generated decompositions as an inspiration on how to refactor the system. Furthermore, the results are often difficult to interpret and it is not clear whether and why coupling and cohesion is improved.

US-6: Change the Relationship between Bounded Contexts

With the four new ARs allowing to refactor Context Map relationships we addressed this user story and the user feedback we received regarding the existing ARs. While modeling a Context Map in CML these refactorings allow users to revise and change the type of a relationship between two contexts rapidly and with tool support. In total we offer eleven ARs now. Therefore we support crafting the Context Map in an agile way and reduce errors due to manual code changes.

US7: Generate Graphical Context Map

We introduced the graphical Context Map generator based on Graphviz [30] in Section 4.6 of Chapter 4. In addition, the Context Map illustrations throughout this thesis are created with the tool. The generator fulfills the requirements claimed by this user story. Users can generate the graphical representation of a Context Map inspired by Vernon [98] and Brandolini [10] for communication and documentation purposes automatically.

In summary, five out of seven user stories are fully addressed. *US-4* and *US-5* are satisfied partially but the concepts and implemented prototypes must be improved in future projects. The discovery of the domain models inside Bounded Contexts must be enhanced in order to map the types of the objects correctly and avoid duplicates (*US-4*). In addition, a new algorithmic solution for the systematic service decomposition component must be found (*US-5*). Moreover, additional validation activities are required especially regarding the usability for business analysis users and the usefulness of the ARs.

5.2.3 Non-Functional Requirements

Chapter 2 not only introduced functional requirements but listed NFRs for the framework components as well. Within this section we discuss the fulfillment of these NFRs. As in Chapter 2 we start with general framework requirements and continue with the discovery library, CML language, ARs, and finish with the Service Cutter [35] integration.

Future-oriented Use of Tools and Frameworks

We used only frameworks and libraries that promise to be *established*, *open*, and *sustainable*. In the discovery library we realized strategies for Spring Boot [71] and Docker Compose [19], both tools that exhibit the mentioned three attributes. The discovery library itself only uses few dependencies like the reflections library[10] or Apache Commons libraries[11] which fulfill the attributes as well.

With the new ARs, we have not introduced any new libraries or dependencies. We consider the Service Cutter tool [35] open and sustainable. However, it is still a research project which is not known to be used in the industry. Thus, one could question whether it can be said to be established or not. Nevertheless, the approach is acknowledged by the academic field and currently the best choice among tools for structured and systematic service decomposition. Related projects are only theoretical and no other open source projects realizing a similar approach exist.

For the implementation of the graphical Context Map generator we used Graphviz [30]. The library is used in many other projects and software diagram tools such as PlantUML [72]. Thus, it definitely exhibits the mentioned three attributes above.

Although the establishment of Service Cutter can be questioned we consider this NFR as mainly accomplished.

Reliability

As this NFR requests we were striving for a resilient implementation and tested all developed functionalities with unit tests, integration tests and manual user

[10]`https://github.com/ronmamo/reflections`
[11]`http://commons.apache.org/`

tests. As already mentioned in this chapter, we held the test coverage as high as possible. None of our own components have a test coverage lower than 90 percent. This NFR can hence be considered fulfilled.

Extensibility

The framework components are designed in an extensible fashion. New discovery strategies, ARs, service decomposition algorithms, and generators can be added without major changes to the framework and its design itself. Therefore, we consider this NFR to be met.

Licenses

This NFR is fulfilled since we only included third-party libraries with open licenses. No GNU General Public License (GNU GPL) licensed libraries are used within Context Mapper.

Supportability and Maintainability

The code quality has been kept as high as possible by applying corresponding quality measures. Mechanisms such as mandatory pull requests for updates on the *master* branch (code review), Continuous Integration (CI) build server executing unit and integration tests for each commit, and a high test coverage measured by a corresponding tool[12] have been established. The Java code is clean and understandable for a junior software engineer. Furthermore, we have not used any special or not well-known language features. This NFR can therefore be seen satisfied.

Documentation

Since we documented all new features on the project's online documentation website[13], this NFR is met as well.

Examples

We provide example CML models for all language features and ARs in our examples repository [14]. The Lakeside Mutual [61] case study for the discovery library can be found in the GitHub repository of the library itself[14] (*Examples* folder). Examples of graphical Context Maps generated with our tool can be found online[15] and in this thesis report. Therefore we provide examples for all new features and tools realized during this thesis.

[12]`https://codecov.io`

[13]`https://contextmapper.org`

[14]`https://github.com/ContextMapper/context-map-discovery`

[15]`https://contextmapper.org/docs/context-map-generator/`

Reverse Engineering: Technology Neutrality and Extensibility

With the strategy-based discovery library we designed the reverse engineering framework component in a style which is technology neutral and extensible. It allows adding arbitrary strategies supporting all kinds of programming languages, frameworks, and technologies. With our current prototype we focused on Java-related frameworks and Java Virtual Machine (JVM) languages but new programming languages could be supported by adding new strategies and without any changes to the framework itself. Thus, this NFR is fulfilled.

CML-Related NFRs

Chapter 2 lists the following non-functional requirements for the CML language: *Simplicity of the DSL*, *Quickly Writable without Redundancy*, *Well Readable*, *Consistent*, and *Parsable by the Tool (Xtext)*.

We consider all of those NFRs fulfilled since we did not have to realize major changes in the language grammar during this thesis. The collected user feedback was positive and only small adjustments regarding the language were requested. The changes as documented in Chapter 4 have not affected the language in a way that violates the requirements above.

ARs: Transformations must result in valid Models

The new ARs were tested carefully and transformations which lead to potential validation errors were considered. The model transformations should always result in valid CML models. We consider this NFR fulfilled, although it is not possible to guarantee that we covered all possible cases.

ARs: Performance

The application of an AR never took longer than three seconds with our case study models. Thus, this NFR is accomplished.

Service Cutter Integration: Algorithm Exchangeability

The Service Cutter [35] engine is already designed to support different clustering algorithms. Within our new library version of the tool we have not changed anything in this regard. We kept the new Application Programming Interface (API) which we call from Context Mapper independent of the cutting algorithms. This makes it possible to integrate new algorithmic solutions without changing the API and the Context Mapper ACL. Therefore we consider this NFR to be fulfilled.

5.3 Strengths and Weaknesses

We conclude this chapter by discussing the strengths or benefits, and weaknesses of the proposed modeling framework.

5.3.1 Consistency with DDD Patterns and Terminology

The approach mainly targets software architects and business analysts already applying strategic DDD. For this reason the CML language, from which the framework originally evolved, is shaped by the DDD patterns, terms and definitions [24, 25, 98]. This is an advantage regarding *understandability* and *usability* for all DDD practitioners which are familiar with the patterns. Our semantic rules and validators [46] may even lead to learning effects and increased awareness regarding possible pattern combinations among these practitioners. The consistency with the original DDD patterns allows them to understand and get started with the modeling language rapidly.

On the other hand, this can be a disadvantage if we want to introduce the framework to users without extensive knowledge of the DDD concepts. Certain terms like *Upstream-Downstream* or patterns such as *PL* are often not self-explanatory and require users to familiarize themselves with the patterns.

5.3.2 Increased Productivity for Context Mapping

The *context mapping* activity seems to be practiced by hand so far. In comparison to hand-drawn Context Maps our framework allows evolving the architecture models with little effort. Hand-drawn sketches have to be re-drawn to update them according to major architecture changes. With our ARs we simplify the application of such model transformations which can be applied iteratively. The refactorings can be a benefit in comparison with other modeling or diagramming tools where architectural changes have to be modeled manually.

5.3.3 (Non-)Conformance With Agile Practices

The support for iterative model evolution as mentioned above is why we believe that the approach also conforms to agile practices [1]. The tool can be used to evolve the architecture and «respond to changes» instead of create a Context Map once and «follow the plan». The Agile Modeling (AM) approach by Ambler [3] also supports our claim that agile and modeling practices can be combined.

Nevertheless, the model-driven architecture approach might be considered non-conforming with agile practices by others. Practitioners of agile methods are often equating architecture-centric methods and model-driven architecture with «high-ceremony processes emphasizing document production» [65]. From this perspective the formalization of the Context Maps could be seen as a weakness of the approach, non-conforming with «working software over comprehensive documentation» [1]. Practitioners of agile methods may prefer the informal and hand-drawn approach for *context mapping*.

5.3.4 Different Levels of Abstraction Supported by Generators

From our perspective, the possibility to generate other diagrams and representations of the same input model is one of the strengths of the proposed approach. The modeling framework does not focus on specific types of diagrams or representations. Communicating architecture always requires different levels of abstraction and different perspectives depending on the usage and the audience. This is an issue addressed in many other architecture modeling approaches such as UML, the «4+1» view model of software architecture by Kruchten [53], or the C4 model by Brown [11]. The advantage of the DSL and generators approach is that the architect does not have to create different diagrams for different perspectives manually. The CML language allows modelers to add as much detail to the architecture model as needed. The generators will then use the information on the level of abstraction it needs to produce the requested output. For example, a user can generate a high-level graphical Context Map illustrating relationships between Bounded Contexts as well as a detailed class diagram for a certain component or context out of the same CML model.

At the same time, maintaining all architectural perspectives within a single model can be a weakness since their complexity increases with the level of details. If the models have to be understandable for all user roles, business analysts as well as engineers and architects, this may lead to conflicts. To generate class diagrams the model has to be enriched with details which may be too technical for business analysts. The approach how we solved this in our examples repository[16], by providing different stages for the same model, leads to duplicate CML code. This might be acceptable for our illustration purposes, but is a disadvantage not negligible in a productive scenario. An extension mechanism allowing to add details in separate CML files might be a solution to tackle this problem in future projects.

5.3.5 The «Model-Code» Gap

Another liability of the model-driven approach, especially for detailed domain models, is the «model-code» gap [26]. If the domain models within the Bounded Contexts are modeled manually they are likely to become obsolete quickly. The strength of the proposed framework concerning this matter is the discovery library that allows to close this gap. If the domain models of the Bounded Contexts are generated from the source code automatically, it is possible to update the models frequently without manual work. However, if the user adjusts the discovered models it is not yet possible to update them without losing the manual changes.

5.3.6 IDE Support for DSL

Furthermore, we identified the provisioning of different Integrated Development Environments (IDEs) as potential weakness of the DSL-based approach. For a good *usability* the framework user needs an editor providing code completion, syntax highlighting, and validators which check that the models are syntactically and semantically correct. Unfortunately there is currently no alternative DSL framework to Xtext [23] available which is able to generate plugins for all popular IDEs. Since developers and software architects work with different IDEs it is difficult to reach all of them only by supporting Eclipse. Future projects have to address this issue by realizing a web-based approach

[16]`https://github.com/ContextMapper/context-mapper-examples`

such as the one suggested by Bünder [12] (Eclipse Theia[17]) or by implementing plugins for other IDEs.

5.3.7 Future Safety

Despite technical challenges as mentioned above, we consider the approach based on domain modeling and DDD to be future-proof. The independence from technology and architectural style is a strength of the concept. Even if the hype regarding (micro-)service-oriented architectures should be over in the future, domain modeling will always be relevant in practice.

5.3.8 Evolutionary Framework Design

Finally, we believe it to be a strength of the proposed modeling framework that it evolved from user requirements iteratively. During the previous projects [46, 50] and this thesis we validated the approach constantly and added framework components due to the collected user feedback. Thus, the design reflects actual user requirements and is not made up in advance.

In this chapter we outlined our validation activities and discussed their results which suggest that the target audience of our modeling framework can benefit from the proposed approach. We demonstrated that the requested project deliverables and goals are satisfied. The evaluation of the requirements further exhibited that five out of seven user stories are fully addressed and the fulfillment of the NFRs has been ensured. In addition, we discussed strengths and weaknesses of the proposed framework.

We conclude the chapter as follows: Although parts of the framework must be improved in future projects, the validation results are positive and encourage us to continue our work. The consistency with the DDD patterns and terminology (for DDD practitioners), increased productivity in *context mapping*, support for iterative (agile) decomposition through automated refactorings, and the generation of architecture diagrams on different levels of abstraction are important benefits of the proposed modeling framework. The initial usability and understandability for users without DDD knowledge, the maintenance of models for different user roles, the «model-code» gap, and the IDE support are current liabilities and issues for future projects.

The next chapter discusses related work and compares our framework with other architecture modeling tools and similar approaches.

[17]`https://theia-ide.org/`

Chapter 6

Comparison with Related Work

The decomposition of software systems into modules or services has gained attention within the last years, especially due to the trend towards microservice architectures [104]. Domain-driven Design (DDD) as one approach to tackle the challenge of decomposing a system has not only been applied by practitioners but has also been mentioned by the academic field [18, 41, 54, 59, 62, 70, 75].

6.1 Modeling Language

Not many modeling languages or tools based on DDD patterns already exist. The few projects implementing Domain-specific Languages (DSLs) based on DDD such as Sculptor [83], fuin.org's DDD DSL [29], or DSL Platform [22] are focused on the tactical patterns rather than the strategic ones. Tactic DDD focuses on the domain model within one Bounded Context, while our modeling framework concentrates on the strategic DDD patterns to model the relationships between Bounded Contexts. Domain-driven frameworks such as Apache Isis[1] or OpenXava[2] which are based on internal DSLs using conventions and annotations are designed to support the implementation of applications rather than modeling the architecture. Duc Minh Le et al. [55] proposed another annotation-based DSL for object-oriented software development.

Rademacher [76] proposed a Unified Modeling Language (UML) profile to model microservice architectures with DDD patterns. The profile is mentioned to be a foundation for validating domain models and deriving microservice code. However, the described UML profile covers tactical DDD patterns only.

Certainly, many other modeling tools and languages allowing to describe software architectures exist. However, no tools on the basis of the strategic DDD patterns to express Context Maps [24, 98] and/or support *context mapping* [10] as a technique existed before Context Mapper. Graphical representations of Context Maps were introduced by Brandolini [10] and Vernon [98]. Plöd [73] proposed another formal notation. Nonetheless, no tool support for these approaches already exists.

6.2 Discovery Library for Reverse Engineering

Reverse engineering components such as our discovery library have also been implemented in other software architecture modeling tools. Structurizr [91]

[1]`https://isis.apache.org/`

[2]`https://openxava.org/`

for example provides a similar library to derive their *C4 models* [11] from existing code. Other architecture modeling tools based on UML, such as Enterprise Architect [88], are often limited to reverse engineer class diagrams from code. O'Brien et al. [66] presented current approaches towards software architecture reconstruction in 2002. However, the presented approaches describe recovery techniques which are not specifically designed for (micro-) service-oriented architectures. They further proposed an architecture reconstruction tool called ARMIN (Architecture Reconstruction and MINing) [87]. Cuadrado et al. [17] presented a recovery process for Service-oriented Architectures (SOAs). Granchelli et al. [31] proposed an architecture recovery approach for microservice architectures called MicroART. In comparison to our static analysis approach they perfom dynamic analysis at runtime as well.

A disadvantage of many reverse engineering tools that generate architecture diagrams from source code is that they are too detailed. They often generate class diagrams but do not provide good overviews over the system's components. In this regard our discovery library is similar to and is actually inspired by the libraries[3] of Simon Brown's Structurizr [91]. His and our approach allow to recover parts of a system on a higher level than classes. Both can use different strategies to discover system parts based on their names, annotations, or other characteristics in the code. Our library concretely allows to discover Bounded Contexts, Aggregates, and business objects (Entities, Value Objects, etc.), while Structurizr [91] works with the terms of the C4 model [11] (context, container, components, and code). Both approaches enable a user to generate diagrams on different levels of abstraction as soon as the model has been created.

6.3 Architectural Refactorings (ARs)

With our Architectural Refactorings (ARs) [103] we provide single steps to decompose the modeled system based on Decomposition Criteria (DCs) compiled in our previous work [50] in an iterative manner. We collected these criteria from the Service Cutter coupling criteria catalog [37], our own professional experience, and mostly gray literature regarding how Bounded Contexts can be identified. In the following we will mention all the literature consulted to design our ARs.

One of the first research papers regarding the criteria to be used to decompose systems has been presented by D. L. Parnas [68]. The mentioned approach to separate parts which change often from others has led to one of our ARs. Use Cases and other domain heuristics such as language, domain expert boundaries, business process steps, data flow, or ownership are other criteria mentioned by Tune and Millet [96]. As many other DDD experts they mention the importance of coevolving organizational and technical boundaries which is also widely known as *Conway's law* [16].

Tigges [93] presents another list of criteria to be used when breaking a domain down to Bounded Contexts: domain objects and their relations, use cases, processes, workflows, quality goals, non-functional requirements, and organizational aspects. Plöd [74] emphasizes that the linguistic differences and model

[3]`https://github.com/structurizr`

differences are the primary drivers for Bounded Context identification. Microservice characteristics such as the organization around business capabilities [82], decentralized governance, and evolutionary design suit the idea behind Bounded Contexts according to him. The statement that it is important to decompose a system in several iterations by Steinegger et al. [90] supports our hypothesis that a Context Map should be evolved iteratively as well. Brandolini [10] shows how Context Maps can evolve in multiple steps. He further invented *event storming* [8, 9], a workshop technique to analyze a domain and discover Bounded Contexts.

All these mentioned authors provide criteria, practices, and heuristics that are important to decompose software systems. However, none of them offer concrete, systematic, or algorithmic procedures describing how the decomposition shall be done. The proposed ARs of our modeling framework use the researched criteria and heuristics mentioned above but additionally provide concrete procedures and steps which can be realized as code refactorings for a DDD-based modeling language such as the Context Mapper DSL (CML).

Comparable architecture modeling tools that are based on a DSL and also support refactorings for the language do not exist. However, Mens and Tourwé mention a research trend towards refactorings on design level artifacts in their survey of software refactoring [60]. For example, Boger et al. [7] discuss how the idea of refactorings can be extended to UML models. Although no similar approaches based on DSLs exist, the concept behind the implementation of our refactorings is not new. They are based on model transformations [49] applied to the Eclipse Modeling Framework (EMF) models [89] behind our Xtext-based DSL [23]. Ivkovic and Kontogiannis [42] introduce another approach to refactor software architecture artifacts using model transformations. Grunske [33] further presents an approach to formalize architectural refactorings as hypergraph transformations with the goal to apply them to architectural specifications automatically.

6.4 Systematic Service Decomposition

The approach mentioned above offers a stepwise decomposition by applying a series of the proposed ARs. With the integration of Service Cutter, a structured and systematic service decomposition approach by Gysel et al. [36], we go one step further and provide an approach which generates service decompositions automatically. The approach uses graph clustering algorithms and a scoring system using their coupling criteria catalog [37] to calculate decompositions.

Tyszberowicz et al. [97] propose another systematic approach to identify microservices using functional decomposition based on use cases. A similar algorithmic solution based on clustering has also been suggested by Mazlami, Cito, and Leitner [59]. In comparison to Service Cutter, their approach does not depend on a prepared input model but constructs the monolithic structure to be decomposed into microservices from a Git[4] repository. Another approach to decompose a monolithic system based on existing source code has been proposed by Kamimura et al. [45]. They use their own dependency-based software

[4]`https://git-scm.com/`

clustering algorithm «SArF» (Software Architecture Finder) [51] and visualize the microservice candidates with a city metaphor [52].

With the structured service decomposition component of our framework we do not contribute new algorithmic solutions. We use the clustering-based approach of Service Cutter [36] to generate the decompositions. However, with the integration we contribute a systematic decomposition tool with strategic DDD models (Context Maps) as input and output. This component is designed to integrate different algorithmic solutions. Thus, future projects could integrate other decomposition algorithms such as the above mentioned.

6.5 Architecture Diagram Generation

A user can generate Context Maps, UML component diagrams, and UML class diagrams with the two graphical generators provided by our framework. The UML diagrams are based on PlantUML [72]. In comparison with the C4 [11] approach we cover three of the four perspectives. The *context* diagram of C4 corresponds to the DDD Context Map. We can generate a *component* diagram with PlantUML [72] and the *code* diagram corresponds to our class diagram. The C4 model [11] supports a *container* diagram which we however do not support yet.

Of course many other tools to create architecture diagrams exist, amongst which are UML tools, although these are not listed herein. The only mentioned one, PlantUML [72], is based on a textual language as well. Thus, the generated diagrams by our framework can be adjusted and used within other editors supporting PlantUML.

There are no other tools specifically designed to create DDD Context Maps besides Context Mapper. Our generator is however based on Graphviz [30] which supports the Graph Description Language (DOT). Hence, users could create similar Context Maps by describing the graphs in Graphviz directly or by using another graph visualization tool. We have already mentioned such tools in the evalution in Chapter 3. In practice, architecture diagrams are often created with free diagramming tools such as Visio[5] or Gliffy[6] as well.

6.6 Microservice Contract and Code Generation

With the Microservices Domain-Specific Language (MDSL) service contract generator, we provide a tool that assists the architect regarding the question how the (micro-)service-oriented architecture can be implemented concretely. In future projects our users may be able to generate microservice stubs out of our Context Maps automatically. Similar approaches towards code generation from service contracts are provided by the OPEN API initiative [67] and Swagger [86]. From the service contracts described in their tools users can generate service stubs as well.

[5] `https://products.office.com/visio/`

[6] `https://www.gliffy.com/`

A DSL-approach comparable with our language but specifically designed to generate code is provided by JHipster [43] and their JHipster Domain Language (JDL). JDL allows to describe applications with their domain models (entities with their relationships) and deployments. Out of the JDL language they are able to generate complete microservice applications based on frameworks such as Spring Boot [71]. In comparison to the CML language, JDL is less powerful and specifically designed for the JHipster [43] application generator. The goal of our framework is to offer a technology-neutral modeling language from which we can generate arbitrary representations. However, just like our language, JDL allows to specify the individual microservices (Bounded Contexts in our case) and which entities belong to which individual service. It further generates UML diagrams out of the specified entities.

In this chapter we discussed related work with respect to the individual components and functionalities of the proposed modeling framework. The next and final chapter of this report summarizes the results and contributions of this thesis and outlines future work.

Chapter 7

Conclusion and Outlook

In the previous chapters we introduced our modeling framework, validated the results, and compared it with related work. This final chapter summarizes the thesis and its contributions. Finally, it outlines potential future work.

7.1 Thesis Summary and Results

With this thesis we conceptualized a modeling framework for strategic Domain-driven Design (DDD) and service decomposition. The Context Mapper DSL (CML) modeling language has been complemented with tools supporting to reverse engineer DDD Context Maps and decompose them iteratively. In addition, generator tools allow users to transform the architecture models into different representations. By applying empirical research methods [47, 100] such as *action research* and *case study* we validated our approach and evolved the Context Mapper tool [15] towards the suggested modeling framework.

The framework provides a formal way for *context mapping*, a technique for service decomposition practiced manually on paper by DDD adopters so far. Through the core component of our framework, the CML language, we established a strategic DDD meta-model and semantic rules [46] that define how the patterns can be combined on Context Maps.

A set of eleven Architectural Refactorings (ARs) implemented as code refactorings for CML, ease the evolution of Context Maps. Seven ARs that are based on researched Decomposition Criteria (DCs) were already developed in our previous term project [50]. Four new ARs conceptualized during this thesis support refactoring relationships between Bounded Contexts and therefore evolving Context Maps.

As part of this thesis we researched how microservice projects are implemented in open source projects and which technologies are popular. Based on the results we conceptualized a discovery library which allows framework users to reverse engineer the Context Map and the Bounded Context domain models of such microservice projects. In comparison to most reverse engineering tools, we not only support generating diagrams on class level but also support the discovery of higher-level system units such as Aggregates and Bounded Contexts. The strategy-based design of the discovery library allows us and/or the open source community to implement new discovery strategies on the basis of other programming languages and technologies in future projects.

The integration of a structured and systematic service decomposition approach allows users to derive suggestions on how the reverse engineered or manually created CML models could be further improved automatically. The tool decomposes the modeled systems with the goal to optimize coupling and cohesion based on a catalog of coupling criteria [37]. This framework component has to be improved and enhanced with new algorithms in future projects. The design of the Service Cutter library[1] realized as part of this thesis supports the inclusion of other algorithmic approaches.

The generator tools of our modeling framework allow to transform the CML Context Maps into graphical representations. With the Context Map, Unified Modeling Language (UML) component, and UML class diagrams, we offer the generation of architecture diagrams on different levels of abstraction. Besides the graphical transformations, the Microservices Domain-Specific Language (MDSL) [102] generator supports architects regarding the question how the (micro-)service-oriented systems can be implemented. In future projects, our generators may even be able to produce (micro-)service server and client stubs.

In summary, our modeling framework supports software architects and business analysts in creating and evolving strategic DDD Context Maps and therefore the architecture of software systems. With the proposed framework components we further simplify the creation of the models and the generation of other representations. Although parts of our framework have to be improved, the validation activities suggest that our original hypothesis is true. Software architects and especially DDD adopters can benefit from a tool which supports the creation of the DDD models in a rigorous and expressive way. The maintenance of models on varying levels of abstraction for different user roles, the «model-code» gap [26], and the Integrated Development Environment (IDE) support for the Domain-specific Language (DSL) are challenges that must be addressed in the future. The consistency of the modeling language with the DDD patterns and the increased productivity in *context mapping* are strengths of the approach. The transformation tools and ARs which allow evolving the models iteratively promote the application of the approach in agile projects as well. We consider the concept based on domain modeling and DDD promising and future-proof since it is independent of technologies and architectural styles.

7.2 Future Work

In future projects we plan to further improve and enhance the framework and its components. We identified potential future work regarding the language tooling, in the discovery library, the ARs, systematic decomposition algorithms, and the generation tools.

[1] `https://github.com/ContextMapper/service-cutter-library`

7.2.1 Language Tooling

A liability already identified during our previous projects [46, 50] is the IDE support. Many software engineers and architects use other IDEs than the currently supported Eclipse[2] IDE. In future projects we may implement editor support for CML within other IDEs like IntelliJ IDEA[3] or Visual Studio Code[4]. Implementing a web-based editor would be another solution. The approach presented by Bünder [12] using Theia[5] and the Language Server Protocol (LSP) would already provide support for Xtext-based [23] DSLs.

Another weakness with respect to the language tooling is that models currently have to be written in one single CML file. Future projects could tackle this issue by providing an import mechanism which allows users to distribute model parts into multiple files. Large CML files could be avoided, for example by modeling each Bounded Context in a separate file. In addition, it is currently not possible to increase the level of detail of a domain model within another file. If different levels of abstraction of a domain model are required to satisfy the demands of different user roles, such as business analysts and software architects, creating multiple models and therefore duplicating code is currently the only solution. An extension mechanism that supports increasing the level of detail in a separate file could solve this issue in a future project. This would allow us to keep the stages in our example models[6] as explained in Section 4.7 of Chapter 4 but reduce the duplicated code.

7.2.2 Discovery Library

Additional discovery strategies are required to reverse engineer other technologies apart from Spring Boot [71]. By supporting other programming languages and/or frameworks, the tool may become interesting for more users. Relationships between Bounded Contexts may also be discovered using other mechanisms besides Docker [20]. In addition, the existing prototype can be improved to discover supplementary CML model data.

The prototypic Bounded Context domain model discovery implemented during this thesis may be improved to identify domain object types correctly (i.e. Entity vs. Value Object). In addition, the creation of duplicate domain objects in different Aggregates must be avoided as we hide coupling between these Aggregates in this way. This would improve the effectiveness of systematic decomposition approaches applied to discovered CML models since they are based on the coupling between domain objects.

By implementing an approach to integrate the discovered (generated) CML code with handwritten adjustments, such as *protected regions* [32], future projects should close the «model-code» gap [26]. This would allow the framework users to update the CML models with the discovery library and ensure that they are not outdated with respect to the codebase. In addition, users may want to update the CML models according to manual changes in the generated artifacts

[2]`https://www.eclipse.org/`

[3]`https://www.jetbrains.com/idea/`

[4]`https://code.visualstudio.com/`

[5]`https://theia-ide.org/`

[6]`https://github.com/ContextMapper/context-mapper-examples`

such as the MDSL contracts. Hence, we would close the cycle as illustrated in the framework overview in Chapter 1.

For example, in combination with a new generator that produces microservice server and/or client stubs (see Section 7.2.5) we could generate an application for our fictitious insurance example[7], change the source code manually, and then update the CML model with the discovery library as illustrated in Figure 7.1.

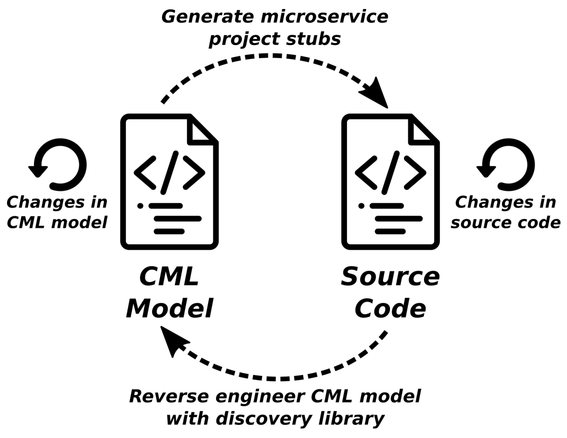

Generate microservice project stubs
Changes in CML model
Changes in source code
CML Model
Source Code
Reverse engineer CML model with discovery library

FIGURE 7.1: Closed «Model-Code» Gap [26] (Future Work)

7.2.3 Architectural Refactorings (ARs)

Future case studies in real-world projects have to validate whether the current collection of ARs provided by the framework is sufficient and feasible. A tutorial and chain of refactorings could help new users to understand how the ARs can be applied to decompose a Context Map. Based on the validation activities and user feedbacks, future projects might propose and implement new ARs.

7.2.4 Systematic Service Decomposition Algorithms

The integrated Service Cutter [35] library providing the structured and systematic decomposition functionality has to be enhanced with other algorithms in future projects. According to our validation activities the current approach that calculates completely new decompositions must be questioned. In future projects we may conceptualize alternative algorithmic solutions that are able to suggest the application of specific ARs. Thereby, the integration of the solution into the users decomposition workflow can be improved. Instead of computing new decompositions, the algorithm could for example suggest Aggregates that should be extracted into a new Bounded Context. However, the solution should still be based on (de-)coupling criteria with the goal to improve coupling and cohesion.

[7]`https://github.com/ContextMapper/context-mapper-examples/tree/master/src/main/cml/insurance-example`

7.2.5 Generation Tools

Generating code out of the CML Context Maps would be another interesting project for the future. Concretely, we could generate microservice project stubs. This would allow users to model a DDD Context Map and then generate a corresponding microservice project automatically. In combination with future adjustments in the discovery library (see Section 7.2.2) we could establish an update mechanism that allows keeping the CML models up-to-date as illustrated in Figure 7.1. Generating Swagger [86] or Open API [67] contracts out of MDSL would be one approach since the Open API project already provides code generators. Using JHipster [43] for the microservice stub generation is another interesting approach in this direction. JHipster Domain Language (JDL) models could be generated out of CML. Thereby our framework users would be able to generate microservices with the technologies already supported by JHipster [43].

In this final chapter we summarized the results of this thesis and outlined potential future work. Based on the positive validation results and the received user feedback we believe that the proposed modeling framework has the potential to benefit software architects and business analysts in applying DDD.

We envision the tool to increase the productivity of DDD adopters in *analyzing and describing domains* as well as *evolving and communicating software architectures*. The ARs and the systematic decomposition component ease the iterative evolution of the software architecture models conforming to agile principles [1]. Users in brownfield projects can reverse engineer the CML models easily. The generation of different architecture diagrams, service contracts, or code is another benefit of formalizing DDD Context Maps.

We consider the approach to be promising and will continue our work on the Context Mapper open source project[8] [15] after this thesis.

[8]`https://contextmapper.org/`

Appendix A

Revised CML Language Reference

This appendix contains a reference for the current version **v5.6.1**[1] of the Context Mapper DSL (CML) language. It is an updated version of the language references we provided in our previous work [46, 50]. This reference is adjusted with respect to all language changes implemented during this thesis.

Please note that this reference explains syntax and semantic rules of the CML language only. The purposes and goals of the individual language features are explained in our previous work [46] and Chapter 4 of this report in case its a new feature. All CML language concepts and the syntax explained in this language reference can further be found in our online documentation[2].

The tactic Domain-driven Design (DDD) parts within the Aggregates are realized with the Sculptor Domain-specific Language (DSL) [83] and not explained within this language reference. We refer to the Sculptor online documentation[3].

A.1 Language Design

The design of our DSL and its rules is based on the domain model presented in our previous project [46]. It can also be found online[4].

A.2 Terminals

The grammar snippets within the language reference use the terminals defined in Listing 37.

```
terminal OPEN: '{';
terminal CLOSE: '}';
```

LISTING 37: Xtext CML Terminals

[1] https://github.com/ContextMapper/context-mapper-dsl/tree/v5.6.1

[2] https://contextmapper.org/docs

[3] http://sculptorgenerator.org/documentation/

[4] https://contextmapper.org/docs/language-model/

A.3 Root Rule

The root elements allowed in a CML file are the *Context Map*, *Bounded Contexts*, *Domains* and *Use Cases*. A CML model can contain one Context Map only. All other root elements can occur multiple times. Listing 38 shows the root grammar rule of the language.

```
ContextMappingModel:
  (
    (map = ContextMap)? &
    (boundedContexts += BoundedContext)* &
    (domains += Domain)* &
    (useCases += UseCase)*
  )
;
```

LISTING 38: Xtext Root Grammar Rule

The order in which these root elements occur does not matter, but they have to occur in one block per type. All Bounded Contexts, domains and use cases have to occur in a block for each type. Whether the CML file lists Bounded Contexts or domains first does not matter. We recommend to use the order as given by the grammar rule, since the application of Architectural Refactorings (ARs) currently unparses the whole model in this order. We mentioned this as a known limitation in our previous project report [50].

A.4 Context Map

The Context Maps grammar rule is shown in Listing 39. A Context Map is declared with the *ContextMap* keyword followed by an optional name for the map. With the *state* keyword the *ContextMapState* is assigned, whereas the *type* keyword allows the assignment of the *ContextMapType*. With the *contains* keyword multiple Bounded Contexts can be assigned to the Context Map. It is possible to use *contains* multiple times, but also to list multiple Bounded Contexts with just a single usage of *contains* (comma-separated). At the end of the grammar rule body the Bounded Context relationships can be added.

```
ContextMap:
  {ContextMap}
  'ContextMap' (name=ID)?
  OPEN
    (('type' ('=')? type=ContextMapType)? &
    ('state' ('=')? state=ContextMapState)?)
    ('contains' boundedContexts += [BoundedContext]
          ("," boundedContexts += [BoundedContext])*)*
    relationships += Relationship*
  CLOSE
;
```

LISTING 39: Xtext Context Map Grammar Rule

Listing 40 illustrates an example for the Context Map rule. Note that the order of the *state* and *type* does not matter. The listing further illustrates both variants how to add Bounded Contexts with the *contains* keyword and a few examples for relationships.

```
ContextMap DDD_Sample_Map {
  type = SYSTEM_LANDSCAPE
  state = AS_IS

  /* add bounded contexts to map: */

  contains CargoBookingContext, VoyagePlanningContext
  contains LocationContext

  /* relationship examples: */

  CargoBookingContext [SK]<->[SK] VoyagePlanningContext

  CargoBookingContext [D]<-[U,OHS,PL] LocationContext

  LocationContext [U,OHS,PL]->[D] VoyagePlanningContext
}
```

LISTING 40: Syntax example for the ContextMap rule

Listing 41 shows the enums *ContextMapState* and *ContextMapType* which define the possible values for the Context Map attributes *type* and *state*.

```
enum ContextMapState:
  AS_IS | TO_BE
;

enum ContextMapType:
  SYSTEM_LANDSCAPE | ORGANIZATIONAL
;
```

LISTING 41: Xtext: ContextMapState & ContextMapType

The *Relationship* rule which can be used to add Bounded Context relationships to a Context Map, allows the application of the two rules *SymmetricRelationship* and *UpstreamDownstreamRelationship*, as shown in Listing 42.

```
Relationship:
  SymmetricRelationship | UpstreamDownstreamRelationship
;
```

LISTING 42: Xtext: Relationship Rule

The *SymmetricRelationship* rule further allows the application of the rules *Partnership* or *SharedKernel* (Listing 43).

```
SymmetricRelationship:
  Partnership | SharedKernel
;
```

LISTING 43: Xtext: SymmetricRelationship Rule

For the syntax of the *Partnership* rule we refer to Section A.9. The *SharedKernel* rule is explained in Section A.10.

The rule *UpstreamDownstreamRelationship* shown in Listing 44 allows either the application of the *CustomerSupplierRelationship* rule or directly writing a generic upstream-downstream relationship.

```
UpstreamDownstreamRelationship:
  CustomerSupplierRelationship |
  (
    (
      // variant 1: arrow from left to right or vice versa
      (upstream = [BoundedContext] (('[''U'']') | ('['('U'',')?
        (upstreamRoles+=UpstreamRole) ("," upstreamRoles+=UpstreamRole)*)']')?
        '->'
        (('[''D'']') | ('['('D'',')?(downstreamRoles+=DownstreamRole)
        ("," downstreamRoles+=DownstreamRole)*)']')? downstream = [BoundedContext]
      ) |
      (downstream = [BoundedContext] (('[''D'']') | ('['('D'',')?
        (downstreamRoles+=DownstreamRole) ("," downstreamRoles+=DownstreamRole)*)']')?
        '<-'
        (('[''U'']') | ('['('U'',')?(upstreamRoles+=UpstreamRole)
        ("," upstreamRoles+=UpstreamRole)*)']')? upstream = [BoundedContext]
      ) |

      // variant 2: long keywords
      (upstream = [BoundedContext] ('['((upstreamRoles+=UpstreamRole)
        ("," upstreamRoles+=UpstreamRole)*)?']')?
        'Upstream-Downstream'
        ('['((downstreamRoles+=DownstreamRole)
        ("," downstreamRoles+=DownstreamRole)*)?']')? downstream = [BoundedContext]
      ) |
      (downstream = [BoundedContext] ('['((downstreamRoles+=DownstreamRole)
        ("," downstreamRoles+=DownstreamRole)*)?']')?
        'Downstream-Upstream'
        ('['((upstreamRoles+=UpstreamRole)
        ("," upstreamRoles+=UpstreamRole)*)?']')? upstream = [BoundedContext]
      )
    )
    // name and body
    (':' name=ID)?
    (OPEN (
      ('implementationTechnology' ('=')? implementationTechnology=STRING)? &
      (('exposedAggregates' ('=')? upstreamExposedAggregates += [Aggregate])
        ("," upstreamExposedAggregates += [Aggregate])*
        (exposedAggregatesComment=SL_COMMENT)?
      )? &
      ('downstreamRights' ('=')? downstreamGovernanceRights=DownstreamGovernanceRights)?
    )
    CLOSE)?
  )
;
```

LISTING 44: Xtext: UpstreamDownstreamRelationship Rule

Please note that we are aware of the fact that the readability of the rule above is not very good in this report. The complete grammar in the original line length can be found in our repository on GitHub[5] and might be easier to read. The length and complexity of this rule is increased due to the different variants we offer our users to declare relationships.

As declared in the grammar rule, there are basically two alternative syntaxes which allow the specification of the same Upstream-Downstream relationship. Listing 45 and Listing 46 show a corresponding example in all possible ways. All these declarations are semantically equal and the *LocationContext* is always upstream whereas the *CargoBookingContext* is downstream.

```
// from left to right
LocationContext [U]->[D] CargoBookingContext
// or without the brackets:
LocationContext -> CargoBookingContext

// from right to left
CargoBookingContext [D]<-[U] LocationContext
// or without the brackets:
CargoBookingContext <- LocationContext
```

LISTING 45: CML: Upstream-Downstream Variant 1 (Arrow)

The first variant (short) uses the abbreviations *U* for upstream and *D* for downstream. Note that if the variant with the arrows (-> or <-) is used, the arrow always points from the upstream towards the downstream, reflecting the influence flow [73] (the downstream is influenced by and depends on the upstream).

The upstream and downstream roles Open Host Service (OHS), Published Language (PL), Anticorruption Layer (ACL) and Conformist (CF) are declared within the brackets. In the short syntax with the arrows, the *U* and the *D* are optional within the brackets. However, if the *U* and *D* are used, the relationship patterns must always be listed behind them.

```
LocationContext Upstream-Downstream CargoBookingContext
// or inverse:
CargoBookingContext Downstream-Upstream LocationContext
```

LISTING 46: CML: Upstream-Downstream Variant 2
(long keywords)

In the variant with the long keywords we use the same brackets for the roles but always without the abbreviations for upstream and downstream (*U*, *D*). Note that it is also possible to declare only one bracket, if there are no relationship patterns on one side of the relationship.

Listing 47 and Listing 48 illustrate the same relationships as before but with upstream and downstream roles. They show various options how a relationship can be declared, emphasizing that brackets, *U's*, and *D's* are optional. All listed options are syntactically correct in CML.

[5]`https://github.com/ContextMapper/context-mapper-dsl/blob/master/org.contextmapper.dsl/src/org/contextmapper/dsl/ContextMappingDSL.xtext`

```
LocationContext -> CargoBookingContext
LocationContext [U]->[D] CargoBookingContext
LocationContext [U]-> CargoBookingContext
LocationContext [OHS,PL]-> CargoBookingContext
LocationContext [U,OHS,PL]-> CargoBookingContext
LocationContext [U,OHS,PL]->[ACL] CargoBookingContext
LocationContext [U,OHS,PL]->[D,ACL] CargoBookingContext
LocationContext [OHS,PL]->[ACL] CargoBookingContext
// or inverse:
CargoBookingContext <- LocationContext
CargoBookingContext [D]<-[U] LocationContext
CargoBookingContext [D]<- LocationContext
CargoBookingContext [ACL]<- LocationContext
CargoBookingContext [D,ACL]<- LocationContext
CargoBookingContext [D,ACL]<-[OHS,PL] LocationContext
CargoBookingContext [D,ACL]<-[U,OHS,PL] LocationContext
CargoBookingContext [ACL]<-[OHS,PL] LocationContext
```

LISTING 47: CML Upstream-Downstream Variant 1:
Examples with Roles

```
LocationContext Upstream-Downstream CargoBookingContext
LocationContext[OHS,PL] Upstream-Downstream CargoBookingContext
LocationContext[OHS,PL] Upstream-Downstream [ACL]CargoBookingContext
// or inverse:
CargoBookingContext Downstream-Upstream LocationContext
CargoBookingContext[ACL] Downstream-Upstream LocationContext
CargoBookingContext[ACL] Downstream-Upstream [OHS,PL]LocationContext
```

LISTING 48: CML Upstream-Downstream Variant 2:
Examples with Roles

Listing 49 shows the Xtext enumerations *UpstreamRole* and *DownstreamRole* which specify the allowed values for the roles inside the corresponding brackets.

```
enum UpstreamRole:
        PUBLISHED_LANGUAGE = 'PL' | OPEN_HOST_SERVICE = 'OHS'
;

enum DownstreamRole:
        ANTICORRUPTION_LAYER = 'ACL' | CONFORMIST = 'CF'
;
```

LISTING 49: Xtext: UpstreamRole and DownstreamRole

With a colon at the end of the specification followed by a string it is possible to give every relationship in CML a name. Listing 50 illustrates an example for a relationship declaration with name.

```
LocationContext [U]->[D] VoyagePlanningContext : ExampleName
```

LISTING 50: CML: Relationship with Name

Within the body of the rule (inside the terminals OPEN and CLOSE, which are optional), the implementation technology, the exposed aggregates and the downstream rights can be defined. The corresponding keywords are *implementationTechnology*, *exposedAggregates* and *downstreamRights*. Please note that this language reference does not state rationale or the goals for language features. Chapter 4 of this work together with the report of our previous projects [46, 50] explain all the language features and the reasons why they were added. Listing 51 shows an example for a relationship specification with body and corresponding attributes. All attributes here are optional and the order does not matter.

```
LocationContext [U]->[D] VoyagePlanningContext : ExampleName {
  implementationTechnology = "RESTful HTTP"
  exposedAggregates = Location, OtherAggregate
  downstreamRights = INFLUENCER
}
```

LISTING 51: CML: Upstream-Downstream Example with Attributes (Body)

In addition, the assigment sign (=) became optional with one of the versions released during this thesis. Therefore, the relationship from Listing 51 can also be written as in Listing 52.

```
LocationContext [U]->[D] VoyagePlanningContext : ExampleName {
  implementationTechnology "RESTful HTTP"
  exposedAggregates Location, OtherAggregate
  downstreamRights INFLUENCER
}
```

LISTING 52: CML: Upstream-Downstream Example with Attributes (Body; without assignment sign)

The *exposedAggregates* attribute must reference Aggregates (see syntax in Section A.18) which are part of the upstream Bounded Context of the relationship. The compiler will throw an error if a referenced Aggregate is specified within another Bounded Context. Listing 53 shows the rule specifying the allowed values for the *downstreamRights* attribute.

```
enum DownstreamGovernanceRights:
        INFLUENCER | OPINION_LEADER | VETO_RIGHT | DECISION_MAKER | MONOPOLIST
;
```

LISTING 53: Xtext: DownstreamGovernanceRights

The alternative in the *UpstreamDownstreamRelationship* rule, defined by the rule *CustomerSupplierRelationship*, is explained in Section A.11.

A.4.1 Context Map Semantic Rules

Note that semantic validators exist for the Context Map. This means that not everything is allowed, even if it is syntactically correct according to the rules explained above. The following rules apply to a Context Map:

- A Bounded Context which is not part of the Context Map (referenced with the *contains* keyword), can not be referenced from a relationship within that Context Map.
- A Bounded Context of the type TEAM (*BoundedContextType* rule) can not be contained in a Context Map if the Context Map type is SYSTEM_LANDSCAPE (*ContextMapType* rule).
- If the Context Map type is ORGANIZATIONAL (*ContextMapType* rule), every Bounded Context added to the Context Map (with the *contains* keyword) has to be of the type TEAM (*BoundedContextType* rule).
- Aggregates which are exposed by relationships must be part of the corresponding upstream Bounded Context.
- Context Map relationships must always be declared between two different Bounded Contexts. A relationship where both participants are the same Bounded Context is not allowed.

A.5 Bounded Context

A Bounded Context can be defined according to the *BoundedContext* grammar rule, shown in Listing 54 and Listing 55.

With the keyword *domainVisionStatement* a Domain Vision Statement is assigned to the Bounded Context. The keyword *type* allows the assigning of a *BoundedContextType*. With the *responsibilities* keyword, multiple Responsibility Layers (see Section A.16) can be assigned. The keyword *implementationTechnology* assigns an implementation technology and the keyword *knowledgeLevel* a Knowledge Level (see Section A.17).

```
BoundedContext:
  (comment=SL_COMMENT)?
  'BoundedContext' name=ID
    (('implements' (implementedDomainParts+=[DomainPart])
              ("," implementedDomainParts+=[DomainPart])*)? &
     ('realizes' (realizedBoundedContexts+=[BoundedContext])
             ("," realizedBoundedContexts+=[BoundedContext])*)? &
     ('refines' refinedBoundedContext=[BoundedContext])?
  )
  (
    OPEN
      (('domainVisionStatement' ('=')? domainVisionStatement=STRING)? &
      ('type' ('=')? type=BoundedContextType)? &
      (('responsibilities' ('=')? responsibilities+=STRING)
                          ("," responsibilities+=STRING)*)? &
```

LISTING 54: Xtext: BoundedContext rule (1)

```
    ('implementationTechnology' ('=')? implementationTechnology=STRING)? &
    ('knowledgeLevel' ('=')? knowledgeLevel=KnowledgeLevel)?)
    modules += Module*
    aggregates += Aggregate*
   CLOSE
  )?
;
```

LISTING 55: Xtext: BoundedContext rule (2)

The allowed values for the enum's *BoundedContextType* and *KnowledgeLevel* are given by the rules in Listing 56.

```
enum BoundedContextType:
  FEATURE | APPLICATION | SYSTEM | TEAM
;
enum KnowledgeLevel :
  META | CONCRETE
;
```

LISTING 56: Xtext: BoundedContextType & KnowledgeLevel

Responsibilities can further be added as a list of strings (also mentioned in Section A.16). The Bounded Context further allows to contain modules and Aggregates. Modules are not further explained within this language reference since it is a Sculptor [83] concept. However it is modified and can contain Aggregates in addition to the other Sculptor [83] elements. Aggregates are explained in Section A.18.

With the *implements* keyword it is possible to define which domain or subdomains is/are implemented by the Bounded Context. The referenced subdomains must be specified within a domain as explained in Section A.6. In addition, the *refines* keyword allows to reference another Bounded Context which is refined by the current context. Thereby we provide an inheritance mechanism that allows to specify whether one Bounded Context is a specialization of another one. Listing 57 and Listing 58 show an example for a Bounded Context specification.

```
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  domainVisionStatement = "The customer management context is responsible for ..."
  implementationTechnology = "Java, JEE Application"
  responsibilities = Customers, Addresses { "The addresses of a customer" }
  knowledgeLevel = CONCRETE

  Module addresses {
    Aggregate Addresses {
      Entity Address {
        String city
      }
    }
  }
  Aggregate Customers {
```

LISTING 57: CML: Bounded Context Example (1)

```
    Entity Customer {
      aggregateRoot

      - SocialInsuranceNumber sin
      String firstname
      String lastname
      - List<Address> addresses
    }
  }
}
```

LISTING 58: CML: Bounded Context Example (2)

Listing 59 illustrates an example how to use the *refines* keyword.

```
BoundedContext DDD_Modeling_Tool {
  type APPLICATION
  domainVisionStatement "A tool that allows to model software architecture by using DDD
                         patterns"
}

BoundedContext ContextMapper refines DDD_Modeling_Tool {
  implementationTechnology "Xtext DSL, Java"
}
```

LISTING 59: CML: Bounded Context *refines* Example

If the Bounded Context is of the type *TEAM*, it is allowed to use the *realizes* keyword and specify which Bounded Context is implemented by the team. Listing 60 shows an example for this use case.

```
BoundedContext CustomersBackofficeTeam implements CustomerManagementDomain realizes
                                                           CustomerManagementContext {
  type = TEAM
  domainVisionStatement = "This team is responsible for implementing ..."
}
```

LISTING 60: Xtext: *realizes* Keyword Example

A.5.1 Bounded Context Semantic Rules

Note that semantic validators exist for a Bounded Context. This means that not everything is allowed, even if it is syntactically correct according to the rules explained above. The following rules apply to a Bounded Context:

- The *realizes* keyword of the *BoundedContext* rule can only be used if the type of the Bounded Context is TEAM (*BoundedContextType* rule).

A.6 Domain and Subdomains

Domains can be defined as root elements of a CML file. A domain is defined by a name and the definitions of its subdomains. Listing 61 illustrates the corresponding *Domain* grammar rule.

```
Domain:
  'Domain' name=ID
  (
    OPEN
      (('domainVisionStatement' ('=')? domainVisionStatement=STRING)? &
      (subdomains += Subdomain)*)
    CLOSE
  )?
;
```

LISTING 61: Xtext: Domain Rule

The Subdomain pattern is defined by the grammar rule in Listing 62. As on a Bounded Context (A.5) the domain as well as the subdomain allow to specify a Domain Vision Statement. The *type* attribute on subdomains allows values defined by the *SubDomainType* enum, illustrated in Figure 63.

```
Subdomain:
  'Subdomain' name=ID
  (
    OPEN
      (('type' ('=')? type=SubDomainType)? &
      ('domainVisionStatement' ('=')? domainVisionStatement=STRING)?)
      entities += Entity*
    CLOSE
  )?
;
```

LISTING 62: Xtext: Subdomain Rule

The subdomain further offers the possibility to add Entities (Sculptor [83], Entity rule), which may be useful to describe the subdomain in more detail. However, note that they are currently not used within the generators. The Entities within Bounded Contexts and Aggregates are relevant there.

```
enum SubDomainType:
  CORE_DOMAIN | SUPPORTING_DOMAIN | GENERIC_SUBDOMAIN
;
```

LISTING 63: Xtext: SubDomainType enum

Listing 64 and Listing 65 illustrate an example how a domain with its subdomains can be specified in CML.

```
Domain InsuranceDomain {
  Subdomain CustomerManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing everything customer-related."
  }
```

LISTING 64: CML: Domain and Subdomains (1)

```
  Subdomain PolicyManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing contracts and policies."
  }
  Subdomain RiskManagementDomain {
    type = GENERIC_SUBDOMAIN
    domainVisionStatement = "Subdomain supporting risk management."
  }
}
```

LISTING 65: CML: Domain and Subdomains (2)

The *implements* keyword already seen in the Bounded Context grammar (see Section A.5) references so-called *DomainPart*'s. This rule is illustrated in Listing 66 and only used to enable references to domains as well as subdomains.

```
DomainPart:
  Domain | Subdomain
;
```

LISTING 66: Xtext: *DomainPart* Super-Type for Domain and Subdomain References

Note that a Bounded Context is supposed to implement subdomains of one and only one domain. The CML validator will create a warning if multiple domains are referenced from one Bounded Context.

A.7 Use Cases

Use cases can be defined on the root level of a CML file and are then referenced from Aggregates (see Section A.18). They are used to specify which Aggregates are accessed by the same use cases. Listing 67 illustrates the corresponding grammar rule.

```
UseCase:
  'UseCase' name=ID
  (OPEN
    (('isLatencyCritical' ('=')? isLatencyCritical?='true')? &
    (('reads' nanoentitiesRead+=STRING*) ("," nanoentitiesRead+=STRING)*)? &
        (('writes' nanoentitiesWritten+=STRING*) ("," nanoentitiesWritten+=STRING)*)?)
  CLOSE)?
;
```

LISTING 67: Xtext: Use Cases

Use cases can either be defined in a simple way by just giving its name, or with more details regarding which attributes are read/written by the use case. The *isLatencyCritical* attribute further allows to specify whether the use case is latency critical or not. Listing 68 shows two examples how use cases can be defined in CML.

```
// simple:
UseCase UpdateContract

// with details:
UseCase CreateOffer {
  isLatencyCritical = true // if false, just remove this line
  reads "Customer.firstName", "Customer.familyName", "Contract.contractId"
  writes "Offer.offerId", "Offer.products", "Offer.price"
}
```

LISTING 68: CML: Use Cases

A.8 Domain Vision Statement

The Domain Vision Statement pattern is implemented as a description attribute (String) on Bounded Contexts and subdomains. For the corresponding grammar rules, we refer to Section A.5 and Section A.6. Listing 69 shows an example Bounded Context with a Domain Vision Statement, and Listing 70 a subdomain accordingly.

```
BoundedContext CustomerContext {
  domainVisionStatement = "This context is responsible for ..."
}
```

LISTING 69: Xtext: Domain Vision Statement on Bounded Context

```
Subdomain CustomerManagementDomain {
  type = CORE_DOMAIN
  domainVisionStatement = "Subdomain managing everything customer-related."
}
```

LISTING 70: Xtext: Domain Vision Statement on Subdomain

A.9 Partnership

The Partnership relationship pattern is defined by the grammar rule illustrated in Listing 71 and Listing 72. There are two syntax variants to declare a Partnership relationship.

```
Partnership:
  (
    // variant 1: arrow
    (participant1 = [BoundedContext] '[''P'']'
      '<->'
      '[''P'']' participant2 = [BoundedContext]) |
    ('[''P'']' participant1 = [BoundedContext]
      '<->'
      '[''P'']' participant2 = [BoundedContext]) |
```

LISTING 71: Xtext: Partnership Rule (1)

```
    (participant1 = [BoundedContext] '[''P'']'
      '<->'
      participant2 = [BoundedContext] '[''P'']') |
    ('[''P'']' participant1 = [BoundedContext]
      '<->'
      participant2 = [BoundedContext] '[''P'']') |

    // variant 2: (long) Partnership keyword
    (participant1 = [BoundedContext] 'Partnership' participant2 = [BoundedContext])
  )
  (':' name=ID)?
  (OPEN
    ('implementationTechnology' ('=')? implementationTechnology=STRING)?
  CLOSE)?
;
```

LISTING 72: Xtext: Partnership Rule (2)

Listing 73 and Listing 74 illustrate applications of both syntax options.

```
PolicyManagementContext [P]<->[P] DebtCollection
```

LISTING 73: CML: Partnership Syntax Variant 1

The second variant uses the *Partnership* keyword whereas the first uses an arrow pointing in both directions, indicating symmetry, and the abbreviation *P* within brackets for both *partners*.

```
PolicyManagementContext Partnership DebtCollection
```

LISTING 74: CML: Partnership Syntax Variant 2

The variant with the arrow allows to place the brackets in different positions. Listing 75 illustrates all possible variants. All four variants are semantically equal. Whitespaces around the brackets are ignored by the compiler, so that the user is free to add whitespaces between the brackets, arrows and Bounded Context names or not.

```
PolicyManagementContext [P]<->[P] DebtCollection // brackets centered

[P]PolicyManagementContext <-> DebtCollection[P] // brackets outside

[P]PolicyManagementContext <-> [P]DebtCollection // both on the left side

PolicyManagementContext[P] <-> DebtCollection[P] // both on the right side
```

LISTING 75: CML: Partnership Bracket Placements

With a colon at the end of the specification followed by a string it is possible to give every relationship in CML a name. Listing 76 illustrates an example for a Partnership relationship declaration with name.

```
PolicyManagementContext [P]<->[P] DebtCollection : exampleRelationship
```

LISTING 76: CML: Partnership Relationship with Name

As Listing 77 illustrates, both syntax variants allow to declare the implementation technology for a Partnership relationship inside the optional *OPEN* and *CLOSE* brackets.

```
// Variant 1:
PolicyManagementContext Partnership DebtCollection : exampleRelationship {
  implementationTechnology = "Java application"
}
// Variant 2:
PolicyManagementContext [P]<->[P] DebtCollection : exampleRelationship {
  implementationTechnology = "Java application"
}
```

LISTING 77: CML: Partnership Relationships with Implementation Technology

Note that the Shared Kernel relationship is the default with respect to the two symmetric relationships. A relationship declaration with arrow but without brackets as illustrated by Listing 78 is possible as well. However, it is important to note that this declares a Shared Kernel relationship and **not** a Partnership relationship.

```
PolicyManagementContext <-> DebtCollection // declares a Shared Kernel (not Partnership)
```

LISTING 78: CML: Shared Kernel as Default Asymmetric Relationship

A.10 Shared Kernel

The Shared Kernel relationship pattern is defined by the grammar rule illustrated in Listing 79 and Listing 80. There are two syntax variants to declare a Shared Kernel relationship, corresponding to the Partnership relationship.

```
SharedKernel:
  (
    // variant 1: arrow
    (participant1 = [BoundedContext] '[''SK'']'
      '<->'
      '[''SK'']' participant2 = [BoundedContext]) |
    ('[''SK'']' participant1 = [BoundedContext]
      '<->'
      '[''SK'']' participant2 = [BoundedContext]) |
    (participant1 = [BoundedContext] '[''SK'']'
      '<->'
      participant2 = [BoundedContext] '[''SK'']') |
```

LISTING 79: Xtext: Shared Kernel Rule (1)

```
    ('[''SK'']' participant1 = [BoundedContext]
      '<->'
      participant2 = [BoundedContext] '[''SK'']') |

    // variant 2: (long) Shared-Kernel keyword
    (participant1 = [BoundedContext] 'Shared-Kernel' participant2 = [BoundedContext]) |

    // default case for symmetric relationships
    (participant1 = [BoundedContext] '<->' participant2 = [BoundedContext])
  )
  (':' name=ID)?
  (OPEN
    ('implementationTechnology' ('=')? implementationTechnology=STRING)?
  CLOSE)?
;
```

LISTING 80: Xtext: Shared Kernel Rule (2)

The first uses the arrow (indicating symmetry) again, whereas the second uses the *Shared-Kernel* keyword. Listing 81 and Listing 82 illustrate examples for both variants.

```
PolicyManagementContext [SK]<->[SK] DebtCollection
```

LISTING 81: CML: Shared Kernel Syntax Variant 1

```
PolicyManagementContext Shared-Kernel DebtCollection
```

LISTING 82: CML: Shared Kernel Syntax Variant 2

The second variant with the arrow allows to place the brackets in different positions similar to the Partnership relationship presented in Section A.9. Listing 83 illustrates all possible variants. All four variants are semantically equal. Whitespaces around the brackets are ignored by the compiler, so that the user is free to add whitespaces between the brackets, arrows and Bounded Context names or not.

```
PolicyManagementContext [SK]<->[SK] DebtCollection // brackets centered

[SK]PolicyManagementContext <-> DebtCollection[SK] // brackets outside

[SK]PolicyManagementContext <-> [SK]DebtCollection // both on the left side

PolicyManagementContext[SK] <-> DebtCollection[SK] // both on the right side
```

LISTING 83: CML: Shared Kernel Bracket Placements

With a colon at the end of the specification followed by a string it is possible to give every relationship in CML a name. Listing 84 illustrates an example for a Shared Kernel relationship declaration with name.

```
PolicyManagementContext [SK]<->[SK] DebtCollection : exampleRelationship
```

LISTING 84: CML: Shared Kernel Relationship with Name

As Listing 85 illustrates, both syntax variants allow to declare the implementation technology for a Shared Kernel relationship inside the optional *OPEN* and *CLOSE* brackets.

```
// Variant 1:
PolicyManagementContext Shared-Kernel DebtCollection : exampleRelationship {
  implementationTechnology = "Java application"
}
// Variant 2:
PolicyManagementContext [SK]<->[SK] DebtCollection : exampleRelationship {
  implementationTechnology = "Java application"
}
```

LISTING 85: CML: Shared Kernel Relationships with Implementation Technology

Note that the Shared Kernel relationship is the default with respect to the two symmetric relationships. A relationship declaration with arrow but without brackets as illustrated by Listing 86 is possible as well. However, it is important to note that this declares a Shared Kernel relationship and **not** a Partnership relationship.

```
PolicyManagementContext <-> DebtCollection // declares a Shared Kernel (not Partnership)
```

LISTING 86: CML: Shared Kernel as Default Asymmetric Relationship

A.11 Customer-Supplier

The Customer-Supplier relationship pattern is defined by the grammar rule illustrated in Listing 87 and Listing 88. Note that Customer-Supplier is a special case of an Upstream-Downstream relationship. Thus, the syntax is principally the same besides the keywords. The *Upstream-Downstream* keyword is replaced with *Customer-Supplier* and the *Downstream-Upstream* keyword is replaced with *Supplier-Customer*. The short syntax with the *U* for upstream and *D* for downstream is extended in this case with a *S* for supplier and a *C* for customer.

```
CustomerSupplierRelationship:
  (
    (
      // variant 1: arrows
      (upstream = [BoundedContext] '['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
        ("," upstreamRoles+=UpstreamRole)*)?']'
        '->'
```

LISTING 87: Xtext: Customer-Supplier Rule (1)

```
        '['('D'',')?'C'(','(downstreamRoles+=DownstreamRole)
        ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext]) |
      (downstream = [BoundedContext] '['('D'',')?'C'(','
        (downstreamRoles+=DownstreamRole) ("," downstreamRoles+=DownstreamRole)*)?']'
        '<-'
        '['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
        ("," upstreamRoles+=UpstreamRole)*)?']' upstream = [BoundedContext]) |

      // variant 2: long keywords
      (downstream = [BoundedContext] ('['((downstreamRoles+=DownstreamRole)
        ("," downstreamRoles+=DownstreamRole)*)?']')?
        'Customer-Supplier'
        ('['((upstreamRoles+=UpstreamRole)
        ("," upstreamRoles+=UpstreamRole)*)?']')? upstream = [BoundedContext]) |
      (upstream = [BoundedContext] ('['((upstreamRoles+=UpstreamRole)
        ("," upstreamRoles+=UpstreamRole)*)?']')?
        'Supplier-Customer'
        ('['((downstreamRoles+=DownstreamRole)
        ("," downstreamRoles+=DownstreamRole)*)?']')? downstream = [BoundedContext])
    )
    (':' name=ID)?
    (OPEN (
      ('implementationTechnology' ('=')? implementationTechnology=STRING)? &
      (('exposedAggregates' ('=')? upstreamExposedAggregates += [Aggregate])
                              ("," upstreamExposedAggregates += [Aggregate])*
                              (exposedAggregatesComment=SL_COMMENT)?)? &
      ('downstreamRights' ('=')? downstreamGovernanceRights=DownstreamGovernanceRights)?
    )
    CLOSE)?
  )
;
```

LISTING 88: Xtext: Customer-Supplier Rule (2)

Please note that we are aware of the fact that the readability of the rule above is not very good in this report. The complete grammar in the original line length can be found in our repository on GitHub[6] and might be easier to read. The length and complexity of this rule is increased due to the different variants we offer our users to declare relationships.

As declared in the grammar rule, there are basically two alternative syntaxes which allow the specification of the same Customer-Supplier relationship. Listing 89 and Listing 90 show corresponding examples for both options. All these declarations are semantically equal and the LocationContext is always supplier/upstream whereas the CargoBookingContext is customer/downstream.

```
LocationContext [U,S]->[D,C] CargoBookingContext
CargoBookingContext [D,C]<-[U,S] LocationContext // inverse

// or: alternatively, the U and D can be omitted in customer-supplier relationships
LocationContext [S]->[C] CargoBookingContext
CargoBookingContext [C]<-[S] LocationContext // inverse
```

LISTING 89: Xtext: Customer-Supplier Variant 1

[6]https://github.com/ContextMapper/context-mapper-dsl/blob/master/org.contextmapper.dsl/src/org/contextmapper/dsl/ContextMappingDSL.xtext

```
LocationContext Supplier-Customer CargoBookingContext
// or inverse:
CargoBookingContext Customer-Supplier LocationContext
```

LISTING 90: Xtext: Customer-Supplier Variant 2

Note that if the variant with the arrows (-> or <-) is used, the arrow always points from the supplier (upstream) towards the customer (downstream), reflecting the influence flow [73] (the downstream is influenced by and depends on the upstream). The upstream and downstream roles OHS, PL, ACL and CF are declared within the brackets behind the *S* and the *C*. In the variant with the long keywords we use the same brackets for the rules but without the abbreviations for supplier and customer.

Listing 91 and Listing 92 illustrate the same relationships as before but with upstream and downstream roles.

```
LocationContext [U,S,PL]->[D,C,ACL] CargoBookingContext
CargoBookingContext [D,C,ACL]<-[U,S,PL] LocationContext // inverse
// or: alternatively, the U and D can be omitted in customer-supplier relationships
LocationContext [S,PL]->[C,ACL] CargoBookingContext
CargoBookingContext [C,ACL]<-[S,PL] LocationContext // inverse
```

LISTING 91: CML: Customer-Supplier Variant 1 with Roles

```
LocationContext[PL] Supplier-Customer [ACL]CargoBookingContext
// or inverse:
CargoBookingContext[ACL] Customer-Supplier [PL]LocationContext
```

LISTING 92: CML: Customer-Supplier Variant 2 with Roles

Listing 93 shows the Xtext enumerations *UpstreamRole* and *DownstreamRole* which specify the allowed values for the roles inside the corresponding brackets.

```
enum UpstreamRole:
        PUBLISHED_LANGUAGE = 'PL' | OPEN_HOST_SERVICE = 'OHS'
;

enum DownstreamRole:
        ANTICORRUPTION_LAYER = 'ACL' | CONFORMIST = 'CF'
;
```

LISTING 93: Xtext: UpstreamRole and DownstreamRole

With a colon at the end of the specification followed by a string it is possible to give every relationship in CML a name. Listing 94 illustrates an example for a Customer-Supplier relationship declaration with name.

```
LocationContext [U,S]->[D,C] VoyagePlanningContext : ExampleName
```

LISTING 94: CML: Customer-Supplier Relationship with Name

Within the body of the rule (inside the terminals OPEN and CLOSE, which are optional), the implementation technology, the exposed aggregates and the downstream rights can be defined. The corresponding keywords are *implementationTechnology*, *exposedAggregates* and *downstreamRights*. Listing 95 shows an example for a relationship specification with body and corresponding attributes. All attributes here are optional and the order does not matter.

```
LocationContext [U,S]->[D,C] VoyagePlanningContext : ExampleName {
  implementationTechnology = "RESTful HTTP"
  exposedAggregates = Location, OtherAggregate
  downstreamRights = DECISION_MAKER
}
```

LISTING 95: CML: Customer-Supplier Example with Attributes (Body)

The *exposedAggregates* attribute must reference Aggregates (see syntax in Section A.18) which are part of the supplier (upstream) Bounded Context of the relationship. The compiler will throw an error if a referenced aggregate is specified within another Bounded Context. Listing 96 shows the rule specifying the allowed values for the *downstreamRights* attribute.

```
enum DownstreamGovernanceRights:
        INFLUENCER | OPINION_LEADER | VETO_RIGHT | DECISION_MAKER | MONOPOLIST
;
```

LISTING 96: Xtext: DownstreamGovernanceRights

A.11.1 Customer-Supplier vs. Upstream-Downstream

Note that according to our understanding of the patterns and our semantic model[7] [46] the Customer-Supplier relationship is a special case of an Upstream-Downstream relationship. With the *Customer-Supplier* keyword you always declare Customer-Supplier relationships. For *generic* Upstream-Downstream relationships which are not Customer-Supplier relationships, use the *Upstream-Downstream* keyword explained in Section A.4.

A Customer-Supplier relationship is an Upstream-Downstream relationship where the downstream priorities factor into upstream planning. The upstream team may succeed interdependently of the fate of the downstream team and therefore the needs of the downstream have to be addressed by the upstream. They interact as customer and supplier. A generic Upstream-Downstream relationship is not necessarily a Customer-Supplier relationship. You have to specify this explicitely in CML .

[7] `https://contextmapper.org/docs/language-model/`

A.11.2 Customer-Supplier Semantic Rules

Note that semantic validators exist for the Customer-Supplier relationship. This means that not everything is allowed, even if it is syntactically correct according to the rules explained above. The following rules apply to Customer-Supplier:

- The Conformist pattern (*DownstreamRole*) is not applicable in a Customer-Supplier relationship.
- The Open Host Service pattern (*UpstreamRole*) is not applicable in a Customer-Supplier relationship.
- The Anticorruption Layer pattern (*DownstreamRole*) shall not be used in a Customer-Supplier relationship.
  - Note that this rule produces a **Warning** only.

A.12 Conformist

The Conformist (CF) pattern is implemented as a value of the *DownstreamRole* enum, as shown in Listing 97.

```
enum DownstreamRole:
  ANTICORRUPTION_LAYER = 'ACL' | CONFORMIST = 'CF'
;
```

LISTING 97: Xtext: DownstreamRole enum

The CONFORMIST (CF) role can be used as a role for the downstream context in any Upstream-Downstream relationship. Listing 98 illustrates an example.

```
PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerManagementContext {
  implementationTechnology = "RESTful HTTP"
  exposedAggregates = Customers
}
```

LISTING 98: Xtext: Conformist Example

A.12.1 Conformist Semantic Rules

Note that semantic validators exist for the Conformist pattern. This means that not everything is allowed, even if it is syntactically correct according to the rules explained above. The following rules apply to Conformist:

- The Conformist pattern (*DownstreamRole*) is not applicable in a Customer-Supplier relationship.

A.13 Open Host Service

The Open Host Service (OHS) pattern is implemented as a value of the *UpstreamRole* enum, as shown in Listing 99.

```
enum UpstreamRole:
  PUBLISHED_LANGUAGE = 'PL' | OPEN_HOST_SERVICE = 'OHS'
;
```

LISTING 99: Xtext: UpstreamRole enum

The OPEN_HOST_SERVICE (OHS) role can be used as a role for the upstream context in any Upstream-Downstream relationship. Listing 100 illustrates an example.

```
CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext {
  implementationTechnology = "SOAP"
  downstreamRights = INFLUENCER
  exposedAggregates = Printing
}
```

LISTING 100: CML: Open Host Service Example

A.13.1 Open Host Service Semantic Rules

Note that semantic validators exist for the Open Host Service pattern. This means that not everything is allowed, even if it is syntactically correct according to the rules explained above. The following rules apply to Open Host Service:

- The Open Host Service pattern (*UpstreamRole*) is not applicable in a Customer-Supplier relationship.

A.14 Anticorruption Layer

The Anticorruption Layer (ACL) pattern is implemented as a value of the *DownstreamRole* enum, as shown in Listing 101.

```
enum DownstreamRole:
  ANTICORRUPTION_LAYER = 'ACL' | CONFORMIST = 'CF'
;
```

LISTING 101: Xtext: DownstreamRole enum

The ANTICORRUPTION_LAYER (ACL) role can be used as a role for the downstream context in any Upstream-Downstream relationship. Listing 102 illustrates an example.

```
CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext {
  implementationTechnology = "SOAP"
  downstreamRights = INFLUENCER
  exposedAggregates = Printing
}
```

LISTING 102: CML: Anticorruption Layer Example

A.14.1 Anticorruption Layer Semantic Rules

Note that semantic validators exist for the Anticorruption Layer pattern. This means that not everything is allowed, even if it is syntactically correct according to the rules explained above. The following rules apply to Anticorruption Layer:

- The Anticorruption Layer pattern (*DownstreamRole*) shall not be used in a Customer-Supplier relationship.
  - Note that this rule produces a **Warning** only.

A.15 Published Language

The Published Language (PL) pattern is implemented as a value of the *UpstreamRole* enum, as shown in Listing 103.

```
enum UpstreamRole:
  PUBLISHED_LANGUAGE = 'PL' | OPEN_HOST_SERVICE = 'OHS'
;
```

LISTING 103: Xtext: UpstreamRole Enum

The PUBLISHED_LANGUAGE (PL) role can be used as a role for the upstream context in any Upstream-Downstream relationship. Listing 104 illustrates an example.

```
PrintingContext [U,OHS,PL]->[D,ACL] PolicyManagementContext {
  implementationTechnology = "SOAP"
  exposedAggregates = Printing
}
```

LISTING 104: CML: Published Language Example

A.16 Responsibility Layers

The implementation of the Responsibility Layers pattern has changed lately. Responsibilities no longer have ID's, since we do not reference them at the moment. The responsibilities can now be added as simple list of strings to Bounded Contexts and Aggregates. Listing 105 illustrates the two corresponding grammar rules.

```
BoundedContext:
  (comment=SL_COMMENT)?
  'BoundedContext' name=ID
    (('implements' (implementedDomainParts+=[DomainPart])
               ("," implementedDomainParts+=[DomainPart])*)? &
     ('realizes' (realizedBoundedContexts+=[BoundedContext])
             ("," realizedBoundedContexts+=[BoundedContext])*)? &
     ('refines' refinedBoundedContext=[BoundedContext])?
  )
  (
    OPEN
      (('domainVisionStatement' ('=')? domainVisionStatement=STRING)? &
      ('type' ('=')? type=BoundedContextType)? &
      (('responsibilities' ('=')? responsibilities+=STRING)
                             ("," responsibilities+=STRING)*)? &
      ('implementationTechnology' ('=')? implementationTechnology=STRING)? &
      ('knowledgeLevel' ('=')? knowledgeLevel=KnowledgeLevel)?)
      modules += Module*
      aggregates += Aggregate*
    CLOSE
  )?
;
Aggregate :
  (comment=SL_COMMENT)?
  (doc=STRING)?
  "Aggregate" name=ID (OPEN
    (
      (('responsibilities' ('=')? responsibilities+=STRING)
                             ("," responsibilities+=STRING)*)? &
      (('useCases' ('=')? useCases += [UseCase]) ("," useCases += [UseCase])*)? &
      ('owner' ('=')? owner=[BoundedContext])? &
      ('knowledgeLevel' ('=')? knowledgeLevel=KnowledgeLevel)? &
      ('likelihoodForChange' ('=')? likelihoodForChange=LikelihoodForChange)?
    )
    ((services+=Service) |
     (resources+=Resource) |
     (consumers+=Consumer) |
     (domainObjects+=SimpleDomainObject))*
  CLOSE)?;
```

LISTING 105: Xtext: Responsibility Layers on Bounded Contexts and Aggregates

The following CML Listing 106 illustrates how responsibilities can be added to Bounded Contexts and Aggregates.

```
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  domainVisionStatement = "The customer management context is responsible for ..."
  implementationTechnology = "Java, JEE Application"
  responsibilities = "Customers", "Addresses"

  Aggregate Customers {
    responsibilities = "Customers"

    Entity Customer
  }
}
```

LISTING 106: CML: Responsibility Layers on Bounded Contexts and Aggregates

A.17 Knowledge Level

The Knowledge Level pattern is implemented with an Xtext enum which can be used on Bounded Contexts and Aggregates. The allowed values are defined by the *KnowledgeLevel* enum, illustrated in Listing 107. Listing 108 shows the two grammar rules for Bounded Contexts and Aggregates, highlighting the corresponding Knowledge Level attributes.

```
enum KnowledgeLevel :
  META="META" | CONCRETE="CONCRETE"
;
```

LISTING 107: Xtext: KnowledgeLevel enum

```
BoundedContext:
  (comment=SL_COMMENT)?
  'BoundedContext' name=ID
    (('implements' (implementedDomainParts+=[DomainPart])
              ("," implementedDomainParts+=[DomainPart])*)? &
     ('realizes' (realizedBoundedContexts+=[BoundedContext])
            ("," realizedBoundedContexts+=[BoundedContext])*)? &
     ('refines' refinedBoundedContext=[BoundedContext])?
  )
  (
    OPEN
      (('domainVisionStatement' ('=')? domainVisionStatement=STRING)? &
      ('type' ('=')? type=BoundedContextType)? &
      (('responsibilities' ('=')? responsibilities+=STRING)
                            ("," responsibilities+=STRING)*)? &
      ('implementationTechnology' ('=')? implementationTechnology=STRING)? &
      ('knowledgeLevel' ('=')? knowledgeLevel=KnowledgeLevel)?)
      modules += Module*
      aggregates += Aggregate*
    CLOSE
  )?
;
Aggregate :
  (comment=SL_COMMENT)?
  (doc=STRING)?
  "Aggregate" name=ID (OPEN
    (
      (('responsibilities' ('=')? responsibilities+=STRING)
                            ("," responsibilities+=STRING)*)? &
      (('useCases' ('=')? useCases += [UseCase]) ("," useCases += [UseCase])*)? &
      ('owner' ('=')? owner=[BoundedContext])? &
      ('knowledgeLevel' ('=')? knowledgeLevel=KnowledgeLevel)? &
      ('likelihoodForChange' ('=')? likelihoodForChange=LikelihoodForChange)?
    )
    ((services+=Service) |
     (resources+=Resource) |
     (consumers+=Consumer) |
     (domainObjects+=SimpleDomainObject))*
  CLOSE)?;
```

LISTING 108: Xtext: Knowledge Level on Bounded Contexts and Aggregates

Listing 109 shows an example on a Bounded Context and Listing 110 on an Aggregate.

```
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  knowledgeLevel = CONCRETE
}
```

LISTING 109: CML: Knowledge Level on Bounded Context

```
Aggregate Customers {
  knowledgeLevel = CONCRETE

  Entity Customer {
    aggregateRoot

    /* ... attributes ... */
  }
}
```

LISTING 110: CML: Knowledge Level on Aggregate

A.18 Aggregate

The Aggregate rule shown in Listing 111 has been added to CML to also support tactic DDD patterns within Bounded Contexts. All elements within the Aggregates are realized with the Sculptor [83] grammar. Therefore, all other tactic DDD patterns are not documented here. We refer to the Sculptor project [83] and their documentation[8].

```
Aggregate :
  (comment=SL_COMMENT)?
  (doc=STRING)?
  "Aggregate" name=ID (OPEN
    (
      (('responsibilities' ('=')? responsibilities+=STRING)
                             ("," responsibilities+=STRING)*)? &
      (('useCases' ('=')? useCases += [UseCase]) ("," useCases += [UseCase])*)? &
      ('owner' ('=')? owner=[BoundedContext])? &
      ('knowledgeLevel' ('=')? knowledgeLevel=KnowledgeLevel)? &
      ('likelihoodForChange' ('=')? likelihoodForChange=LikelihoodForChange)?
    )
    ((services+=Service) |
     (resources+=Resource) |
     (consumers+=Consumer) |
     (domainObjects+=SimpleDomainObject))*
  CLOSE)?
;
```

LISTING 111: Xtext: Aggregate rule

The Aggregate supports the Responsibility Layers pattern and the Knowledge Level pattern explained in Section A.16 and Section A.17 respectively. As shown in Listing 111 they are specified with the keywords *responsibilities* and *knowledgeLevel*.

[8] http://sculptorgenerator.org/documentation/

An Aggregate can further specify which use cases access it by using the *useCases* keyword. The attribute takes a list of references to use cases. How the corresponding use cases can be specified is explained in Section A.7. The *owner* attribute allows to specify by which *TEAM* an Aggregate is owned. It takes a reference to a Bounded Context of the type *TEAM*. The compiler ensures that the referenced Bounded Context has this type. Section A.5 explains how the type of a Bounded Context can be declared.

With the *likelihoodForChange* attribute a user can define how volatile (likely for change) an Aggregate is (used for the corresponding AR). Listing 112 illustrates the enum specifying the allowed values for the *likelihoodForChange* attribute.

```
enum LikelihoodForChange :
  NORMAL | RARELY | OFTEN
;
```

LISTING 112: Xtext: LikelihoodForChange enum

An Aggregate can further contain Services, Resources, Consumers and Simple-DomainObjects (Entities, Value Objects, Domain Events, etc.) which are not further introduced here. The according rules are defined by the Sculptor [83] DSL, as already mentioned. However, Listing 113 illustrates an example of an Aggregate with the explained attributes above and tactic DDD elements in the Sculptor [83] syntax.

```
Aggregate Contract {
  responsibilities = "Contracts", "Policies"
  knowledgeLevel = CONCRETE
  useCases = UpdateContract, CreateOffer
  owner = ContractsTeam
  likelihoodForChange = NORMAL

  Entity Contract {
    aggregateRoot

    - ContractId identifier
    - Customer client
    - List<Product> products
  }

  ValueObject ContractId {
    int contractId key
  }

  Entity Policy {
    int policyNr
    - Contract contract
    BigDecimal price
  }
}
```

LISTING 113: CML: Aggregate Example

A.19 Complete CML Grammar

The previous sections in this language reference illustrated their corresponding parts of the CML grammar. The complete CML grammar file in the version *v5.6.1* documented in this report can be found in our GitHub repository[9].

[9] https://github.com/ContextMapper/context-mapper-dsl/blob/v5.6.1/org.contextmapper.dsl/src/org/contextmapper/dsl/ContextMappingDSL.xtext

Appendix B

Architectural Refactoring (AR) Catalog

This appendix contains the summaries and descriptions of all Architectural Refactorings (ARs) implemented in Context Mapper. They can also be found online[1]. Note that only the ARs *AR-8*, *AR-9*, *AR-10* and *AR-11* were realized as part of this thesis. The ARs *AR-{1-7}* including their summaries have been developed in our previous term project [50].

B.1 Structural Refactorings

The following summaries describing the *structural* refactorings of our modeling framework provide the context, motivation, solution and effects for each individual AR.

B.1.1 AR-1: Split Aggregate by Entities

Context

On the level of attributes, or *nanoentities* in the terminology of Service Cutter [36], it is common to group those together which *belong to the same identity and share a common lifecycle*[2] to form Entities. On the level of business objects we typically try to group those Entities together which belong to the same part or area of the business (domain). These different areas form *linguistic boundaries* and often also *domain expert boundaries* as explained by Tune and Millett [96].

Thereby we always aim to reduce the coupling between the Entities and increase the cohesion within them. The same approach can be applied on the level of Aggregates. The Aggregates within one Bounded Context shall be structured in a way which reduces coupling between the Aggregates and increases the cohesion within them.

Motivation

This AR can be applied if a Bounded Context contains an Aggregate with Entities which exhibit an unsatisfying cohesiveness. In such a case you may want

[1]`https://contextmapper.org/docs/architectural-refactorings/`

[2]`https://github.com/ServiceCutter/ServiceCutter/wiki/CC-1-IdEntity-and-Lifecycle-Commonality`

to split your Aggregate into multiples for each Entity aiming for improved coupling and cohesion.

Solution and Effect

Given an Aggregate with an unsatisfying cohesiveness this AR is applied to *decompose* the Aggregate by its Entities. It splits an Aggregate into multiple Aggregates. Each resulting Aggregate will contain one of the Entities and each Entity becomes an *aggregate root*.

Inverse ARs

AR-6: Merge Aggregates can be seen as the inverse AR as it allows to *compose* Aggregates together. With *AR-6: Merge Aggregates* a user is able to invert the result produced by this AR.

B.1.2 AR-2: Split Bounded Context by Use Cases

Context

By decomposing a system into multiple Bounded Contexts we aim for loose coupling between the Bounded Contexts and a high cohesion within them. One approach to decompose a system into Bounded Contexts is splitting it by use cases. This approach is mentioned by many Domain-driven Design (DDD) experts such as Tigges [93], Plöd [74], Tune and Millett [96] or Tyszberowicz et al. [97] regarding the question how to break down a domain into Bounded Contexts. Chris Richardson further mentions use cases regarding the question *How to decompose the application into services?* in his *Microservice Architecture* pattern [79]. The approach further supports the *single responsibility principle* by R.C. Martin et al. [58]. In the Service Cutter [36] coupling criteria catalog this principle corresponds to the *semantic proximity*[3] criterion.

Motivation

This AR can be used on Bounded Contexts containing Aggregates which are involved in the execution of different use cases. Splitting such a Bounded Context by use cases can improve coupling and cohesion between and within Bounded Contexts.

Solution and Effect

Split the Bounded Context into multiple Bounded Contexts by creating one for each use case. The resulting Bounded Contexts will only contain Aggregates which are accessed by the same use case. The AR creates as many Bounded Contexts as use cases exist.

[3]https://github.com/ServiceCutter/ServiceCutter/wiki/CC-2-Semantic-Proximity

Inverse ARs

AR-7: Merge Bounded Contexts can be seen as the inverse AR as it allows to *compose* multiple Bounded Contexts together. With *AR-7: Merge Bounded Contexts* a user is able to invert the result produced by this AR.

B.1.3 AR-3: Split Bounded Context by Owner

Context

Another approach to decompose a domain into Bounded Contexts besides use cases is to build the contexts arround teams (owners). This organizational aspect also pointed out by many DDD experts such as Brandolini [10], Tune [64], Plöd [74] and Tune and Millett [96] is widely known as *Conway's law* [16]. Tune [64] emphasizes that «*Bounded Contexts decouple parts*» and «*parts are code and teams*». In our previous work [46] we already respected this aspect and implemented the possibility to model teams in Context Mapper DSL (CML). Service Cutter [36] handles this aspect with the *shared owner*[4] criterion.

Motivation

This AR shall be applied if a Bounded Context contains Aggregates which are owned by different teams. Splitting a Bounded Context by owners can not only improve the coupling and cohesion on a technical level, but further lead to improvements on the organizational level. Striving for one team per Bounded Context leads to clear responsibilities [74] and enables team autonomy [64].

Solution and Effect

Split the Bounded Context into multiple Bounded Contexts by creating one for each owner/team. The application of this AR results in a context map with only one team per Bounded Context. There will be as many Bounded Contexts as teams exist.

Inverse ARs

As already mentioned for *AR-2: Split Bounded Context by Use Cases*, the inverse AR of this one is *AR-7: Merge Bounded Contexts* which allows a user to invert the changes of this refactoring.

B.1.4 AR-4: Extract Aggregates by Volatility

Context

With the paper *On the criteria to be used in decomposing systems into modules* D.L. Parnas [69] presented one of the first approaches to decompose a system. This AR is based on Parnas approach which states that we should *isolate parts which*

[4]`https://github.com/ServiceCutter/ServiceCutter/wiki/CC-3-Shared-Owner`

are likely to change. According to Service Cutters [36] criterion *structural volatility*[5] we used the term *volatility* to name this AR.

Motivation

This AR allows to separate Aggregates according to their volatility. By isolating Aggregates which are very likely to change in separate Bounded Contexts it is possible to protect parts of a system from frequent changes. Hiding things which are likely to change from others reduces the impact of revised design decisions [69].

Solution and Effect

The AR extracts all Aggregates with a specific volatility and moves them into a new Bounded Context. It presumes that all Aggregates have a volatility value, *rarely*, *normal* or *often*, assigned in order to separate them. The result is a new Bounded Context containing all Aggregates with the volatility value given as input parameter to the AR.

Inverse ARs

The changes of this AR can be reverted by using *AR-7: Merge Bounded Contexts*.

B.1.5 AR-5: Extract Aggregates by Cohesion

Context

Besides the already presented approaches for decomposing Bounded Contexts many others based on different Decomposition Criteria (DCs) exist. We derived this AR to enable architects to decompose by any *generalized* Non-Functional Requirement (NFR) criterion. Thereby it is possible to manually select the Aggregates to be extracted aiming for improved coupling and cohesion. Examples for such Decomposition Criteria (DCs) have already been presented within this chapter, namely *DC-8: mutability*, *DC-9: storage similarity*, *DC-10: availability*, *DC-11: consistency* or *DC-12: security*.

Motivation

This AR can be applied in cases where the user knows Aggregates which shall be extracted from a Bounded Context based on any NFR affecting cohesion negatively. By extracting Aggregates which share a certain characteristic regarding the concerned NFR, for example regarding *security*, it is possible to achieve improved cohesion within the Bounded Context.

Solution and Effect

If a subset of Aggregates within a Bounded Context has other requirements regarding a specific NFR criterion as the others, extract them into a separate

[5] `https://github.com/ServiceCutter/ServiceCutter/wiki/CC-4-Structural-Volatility`

Bounded Context. For example, if a few Aggregates have other requirements regarding *security* in comparison to all other Aggregates, extract them from the Bounded Context. The AR moves a selection of Aggregates from an existing Bounded Context into a new Bounded Context. It allows to isolate a set of Aggregates based on a manual selection.

Inverse ARs

Similar to the previous AR the changes of this AR can be reverted by using *AR-7: Merge Bounded Contexts*.

B.1.6 AR-6: Merge Aggregates

Context

As explained for *AR-1: Split Aggregate by Entities* it is a common approach to group business Entities which belong to the same part or area of the business (domain). These groups may form *linguistic* or *domain expert* boundaries [96]. This approach is typically not only applied on the higher level of Bounded Contexts but as well on Aggregates which group Entities. In the process of designing a Bounded Context in terms of Aggregates the granularity may get to high and different Aggregates contain Entities which should belong together in order to increase cohesion and reduce coupling between these Aggregates.

Motivation

If the decomposition of Aggregates within a Bounded Context is too fine-granular and different Aggregates contain Entities which should belong together according to domain experts, business capabilities [82], linguistic boundaries, or other criteria, merging these Aggregates together may improve coupling and cohesion.

Solution and Effect

If two Aggregates have a high coupling because their Entities belong together with respect to any criteria (for example business capabilities), merge the Aggregates together to reduce coupling and increase the cohesion. This AR merges two Aggregates within a Bounded Context together into one Aggregate. Therefore, the resulting Aggregate contains all business objects (Entities, Value Objects, etc.) of both original Aggregates.

Inverse ARs

The AR *AR-1: Split Aggregate by Entities* can be seen as an inverse AR to this one as it *decomposes* (splits) Aggregates whereas this one *composes* them.

B.1.7 AR-7: Merge Bounded Contexts

Context

With the decomposition of a domain into Bounded Contexts we aim for loose coupling between the contexts and high cohesion within them. However, there may be situations where the decomposition is too fine-granular and decreasing the granularity would improve the coupling and cohesion.

Motivation

If two Bounded Contexts contain Aggregates which belong together according to domain experts, business capabilities [82], linguistic boundaries, or other criteria, the coupling between them may be high. This AR can be applied in situations where merging Bounded Contexts with a high coupling improves the cohesion within the resulting Bounded Context and reduces the coupling between contexts as the decomposition has become too fine-granular.

Solution and Effect

If two Bounded Contexts exhibit a high coupling because their Aggregates belong together according to criteria such as business capablities [82], merge them together to reduce coupling. This AR merges two Bounded Contexts together into one Bounded Context. Therefore, the resulting Bounded Context will contain all Aggregates of both original Bounded Contexts.

Inverse ARs

The following ARs can all be seen as inverses of this AR since they *decompose* Bounded Contexts whereas this AR *composes* them together:

- *AR-2: Split Bounded Context by Use Cases*
- *AR-3: Split Bounded Context by Owner*
- *AR-4: Extract Aggregates by Volatility*
- *AR-5: Extract Aggregates by Cohesion*

B.1.8 AR-8: Extract Shared Kernel

Context

A Shared Kernel describes an intimate relationship where two Bounded Contexts share a part of their domain model. Typically this shared model part is implemented in a shared library used by both Bounded Contexts. This kind of relationship leads to a higher coupling in comparison with other relationship types such as Upstream-Downstream.

Motivation

A Shared Kernel leads to interdependencies between two teams and may come with undesired coupling. Changes within the shared model parts may influence both development teams in the relationship. If we strive for clear responsibilities [74] and team autonomy [64], we may want to reduce the coupling between teams. This AR can be applied in such a case if the Shared Kernel reached a size where the common model part can also be seen as a separate Bounded Context.

Solution and Effect

If the Shared Kernel model part between two teams is getting big and costly to maintain, it might be a solution to build a separate Bounded Context and team for this domain model. This AR creates a new Bounded Context for the Shared Kernel and establishes Upstream-Downstream relationships between the new context and the existing ones. Figure B.1 illustrates the transformation.

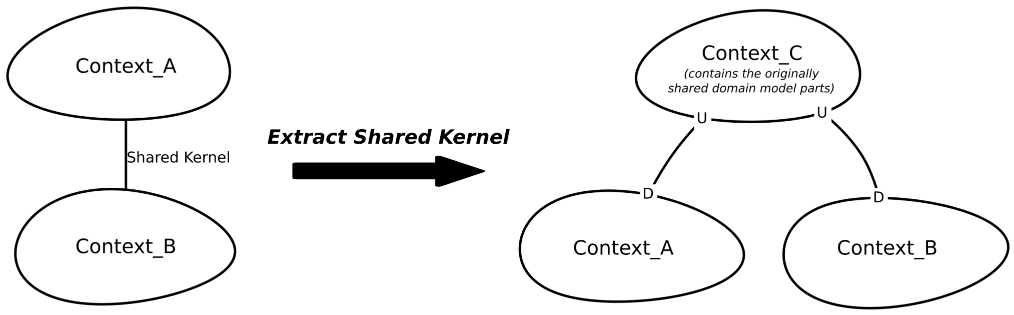


FIGURE B.1: AR-8: Extract Shared Kernel

The resulting situation with Upstream-Downstream relationships instead of a Shared Kernel may improve the coupling between the contexts and the cohesion within them.

Inverse ARs

The inverse AR to *AR-8: Extract Shared Kernel* has not been implemented yet. In Chapter 3 we have described how the inverse refactoring would work briefly.

B.1.9 AR-9: Suspend Partnership

Context

A Partnership relationship describes another intimate relationship between two Bounded Contexts and/or development teams. In comparison to the Shared Kernel however, it does not have to be the case that the teams share parts of their domain models. The intimacy in a Partnership is defined by interdependent features and a joint management of planning and integration. The organizational and feature-related interdependencies between the two teams leads to a situation where both teams can either only fail or succeed together. New developments and releases must always be coordinated between the teams.

Motivation

The interdependencies between the two teams in a Partnership may lead to an undesired coupling and increased inter-team coordination and communication to keep the product stable. To develop clear responsibilities [74] and team autonomy [64] it might be necessary to decouple the teams and suspend the Partnership. This AR can be applied if the coupling between two teams is getting painful and the Partnership must be suspended.

Solution and Effect

If the coupling between two teams in a Partnership relationship must be reduced this AR can be applied to suspend the Partnership.

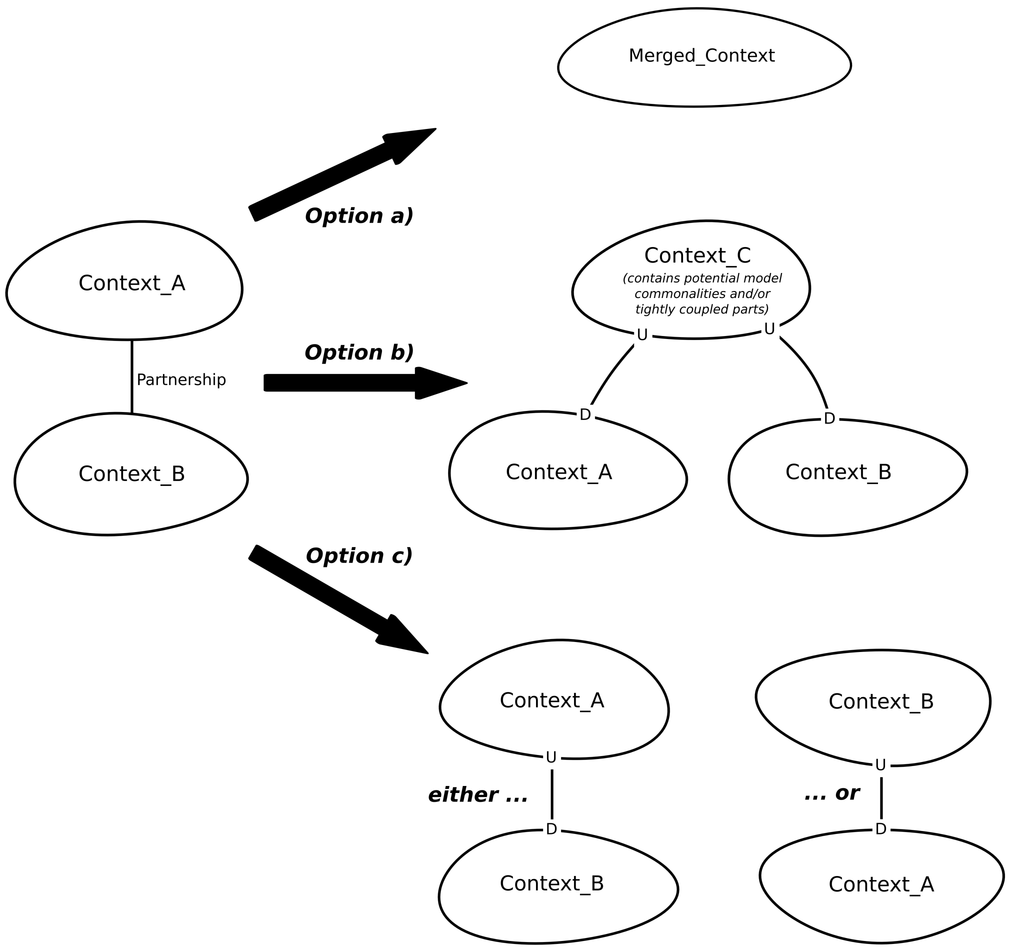


FIGURE B.2: AR-9: Suspend Partnership

The AR offers three options to get rid of the Partnership:

- a) Merge the two Bounded Contexts: if the teams are small enough and the coupling very high, merging the Bounded Context might be the right solution. This option corresponds to *AR-7: Merge Bounded Contexts*.
- b) Extract a new Bounded Context for tightly coupled model parts and establish Upstream-Downstream relationships: if the Partnership is mainly

defined by a common part of the domain models, the Partnership can be suspended in the same way as *AR-8: Extract Shared Kernel* works.

- c) Simply replace the Partnership with an upstream-downstream relationship: another solution might be that one of the two teams takes over the responsibilities regarding the common or highly coupled parts. This way one can establish an Upstream-Downstream relationship. Note that this option may move responsibilities and reduce the power of the new downstream team.

Figure B.2 illustrates the three possible solutions this AR offers to suspend a Partnership.

Inverse ARs

The inverse AR to *AR-9: Suspend Partnership* has not been implemented yet. In Chapter 3 we have described how the inverse refactoring could work briefly.

B.2 Relationship Refactorings

The *relationship* refactorings are very simple in comparison to the structural ones. We just provide short summaries for these ARs in the following.

B.2.1 AR-10: Change Shared Kernel to Partnership

Our relationship refactorings allow the user/modeller to change the type of a relationship on a Context Map easily without manual work. The symmetric relationships according to our semantic model [46], Shared Kernel and Partnership, are interchangeable without impacts to the structure of the decomposition. This refactoring changes a Shared Kernel relationship to a Partnership relationship.

Inverse ARs

The result of *AR-10: Change Shared Kernel to Partnership* can be inverted by applying *AR-11: Change Partnership to Shared Kernel*.

B.2.2 AR-11: Change Partnership to Shared Kernel

The symmetric relationships according to our semantic model [46], Shared Kernel and Partnership, are interchangeable without impacts to the structure of the decomposition. This refactoring changes a Partnership relationship to a Shared Kernel relationship.

Inverse ARs

The inverse AR of *AR-11: Change Partnership to Shared Kernel* is *AR-10: Change Shared Kernel to Partnership*.

List of Figures

List of Tables

List of Abbreviations

ACL Anticorruption Layer. 29, 53, 62, 75, 77, 78, 88, 92, 113, 127

AM Agile Modeling. 93

API Application Programming Interface. 34, 48, 57, 58, 60, 61, 72, 92

AR Architectural Refactoring. 1–4, 7, 11, 12, 14, 15, 17–19, 22, 25–33, 37, 40, 57, 64, 65, 69, 70, 82–93, 98, 99, 103, 104, 106, 107, 110, 135, 137–145

BSD license Berkeley Source Distribution (BSD) License. 34

CDDL-1.0 Common Development and Distribution License 1.0. 34

CF Conformist. 18, 29, 53, 75, 77, 78, 88, 113, 127

CI Continuous Integration. 13, 91

CLI Command Line Interface. 40

CML Context Mapper DSL. 1–3, 5, 7, 8, 10, 13, 14, 17–19, 22, 23, 29, 33, 37, 38, 40–42, 44–47, 50, 52, 55–58, 62–72, 74–79, 82, 83, 85, 87–94, 99, 101, 103–107, 109, 110, 113, 114, 118–120, 122, 124, 127, 128, 132, 134, 136, 139

CPL-1.0 Common Public License Version 1.0. 34

DC Decomposition Criterion. 1, 18, 22, 23, 57, 98, 103, 140

DDD Domain-driven Design. 1, 2, 7, 9–11, 14, 18, 33, 38, 46, 48, 57, 62, 65, 74, 76, 81, 82, 84, 85, 87, 88, 93, 95, 97–100, 103, 104, 107, 109, 134, 135, 138, 139

DOT Graph Description Language. 73, 74, 100

DSL Domain-specific Language. 1, 7, 9, 13, 14, 37, 40, 63, 82, 87, 92, 94, 97, 99, 101, 104, 105, 109, 135

DTO Data Transfer Object. 3, 19, 22, 50, 56, 88, 89

EAI Enterprise Application Integration. 7, 88

EMF Eclipse Modeling Framework. 65, 72, 99

GNU GPL GNU General Public License. 13, 23, 91

GNU GPLv3 GNU General Public License Version 3. 23, 24, 34

IDE Integrated Development Environment. 3, 40, 66, 94, 95, 104, 105

JDL JHipster Domain Language. 101, 107

JSON JavaScript Object Notation. 57, 62, 71

JVM Java Virtual Machine. 92

LSP Language Server Protocol. 105

MAP Microservice API Patterns. 54, 77

MDSL Microservices Domain-Specific Language. 1, 3, 11, 12, 17, 37, 58–61, 71, 76, 77, 83, 84, 100, 104, 106, 107

NFR Non-Functional Requirement. 7, 9, 12, 15, 35, 84, 90–92, 95, 140

OHS Open Host Service. 18, 28, 29, 53, 74, 75, 77, 78, 88, 113, 127

PL Published Language. 17, 18, 29, 53, 74, 75, 77, 78, 88, 93, 113, 127

PNG Portable Network Graphics. 73

SCL Service Cutter DSL. 63

SOA Service-oriented Architecture. 98

SVG Scalable Vector Graphics. 73, 74

UI User Interface. 63, 74

UML Unified Modeling Language. 9, 34, 37, 44, 48, 58, 60, 70, 71, 76, 88, 94, 97–101, 104

URL Uniform Resource Locator. 20

XML Extensible Markup Language. 20

YAML YAML Ain't Markup Language. 20, 51

Bibliography

[1] Agile Alliance. *Agile manifesto*. `https://www.agilealliance.org/agile101/the-agile-manifesto/`. [Online; Accessed: 2019-12-07].

[2] Agile Alliance. *Role-Feature-Reason User Story Template*. `https://www.agilealliance.org/glossary/role-feature/`. [Online; Accessed: 2019-09-24].

[3] Scott Ambler. *Agile modeling: effective practices for extreme programming and the unified process*. John Wiley & Sons, 2002.

[4] David E. Avison et al. "Action Research". In: *Commun. ACM* 42.1 (Jan. 1999), pp. 94–97. ISSN: 0001-0782. DOI: `10.1145/291469.291479`. URL: `http://doi.acm.org/10.1145/291469.291479`.

[5] Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. *Gephi: An Open Source Software for Exploring and Manipulating Networks*. 2009. URL: `http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154`.

[6] Chris Biemann. "Chinese Whispers: An Efficient Graph Clustering Algorithm and Its Application to Natural Language Processing Problems". In: *Proceedings of the First Workshop on Graph Based Methods for Natural Language Processing*. TextGraphs-1. Stroudsburg, PA, USA: Association for Computational Linguistics, 2006, pp. 73–80. URL: `http://dl.acm.org/citation.cfm?id=1654758.1654774`.

[7] Marko Boger, Thorsten Sturm, and Per Fragemann. "Refactoring Browser for UML". In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. Ed. by Mehmet Aksit, Mira Mezini, and Rainer Unland. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 366–377. ISBN: 978-3-540-36557-0.

[8] Alberto Brandolini. *Introducing Event Storming*. `http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html`. [Online; Accessed: 2019-12-06].

[9] Alberto Brandolini. *Introducing EventStorming: An act of Deliberate Collective Learning*. Leanpub, 2018.

[10] Alberto Brandolini. *Strategic Domain Driven Design with Context Mapping*. `https://www.infoq.com/articles/ddd-contextmapping`. [Online; Accessed: 2019-04-02].

[11] Simon Brown. *The C4 model for visualising software architecture*. `https://c4model.com/`. [Online; Accessed: 2019-12-07].

[12] Hendrik Bünder. "Decoupling Language and Editor - The Impact of the Language Server Protocol on Textual Domain-Specific Languages". In: *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,* INSTICC. SciTePress, 2019, pp. 131–142. ISBN: 978-989-758-358-2. DOI: `10.5220/0007556301310142`.

[13] CFinder. *CFinder - Clusters and Communities: Overlapping dense groups in networks*. `http://www.cfinder.org/`. [Online; Accessed: 2019-12-19].

[14] Context Mapper. *Context Mapper: CML examples repository*. `https://github.com/ContextMapper/context-mapper-examples`. [Online; Accessed: 2019-09-24].

[15] Context Mapper. *Context Mapper is an open source project providing a Domain-specific Language (DSL) based on Domain-Driven Design (DDD) patterns for context mapping and service decomposition.* `https://contextmapper.org/`. [Online; Accessed: 2020-01-24].

[16] Melvin Conway. *Conway's law*. 1968.

[17] F. Cuadrado et al. "A Case Study on Software Evolution towards Service-Oriented Architecture". In: *22nd International Conference on Advanced Information Networking and Applications - Workshops (aina workshops 2008)*. 2008, pp. 1399–1404. DOI: `10.1109/WAINA.2008.296`.

[18] P. Di Francesco, P. Lago, and I. Malavolta. "Migrating Towards Microservice Architectures: An Industrial Survey". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 29–2909. DOI: `10.1109/ICSA.2018.00012`.

[19] Docker Inc. *Docker Compose*. `https://docs.docker.com/compose/`. [Online; Accessed: 2019-12-08].

[20] Docker Inc. *Docker: Enterprise Container Platform*. `https://www.docker.com/`. [Online; Accessed: 2019-10-08].

[21] Stijn Dongen. "Graph Clustering by Flow Simulation". In: *PhD thesis, Center for Math and Computer Science (CWI)* (May 2000).

[22] dsl-platform.com. *DSL Platform: Domain-Driven Design*. `https://docs.dsl-platform.com/ddd-foundations`. [Online; Accessed: 2019-12-05].

[23] Eclipse Xtext. *Xtext - Language Engineering Made Easy!* `https://www.eclipse.org/Xtext/`. [Online; Accessed: 2019-10-05].

[24] Eric Evans. *Domain-driven design : tackling complexity in the heart of software*. eng. 18th prin. Upper Saddle River, NJ: Addison-Wesley, 2012. ISBN: 978-0-321-12521-7.

[25] Eric Evans. *Domain-Driven Design Reference: Definitions and Pattern Summaries*. [Online; Accessed: 2018-10-22]. `https://domainlanguage.com`, 2015. URL: `http://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf`.

[26] George Fairbanks. *Just enough software architecture: a risk-driven approach*. Marshall & Brainerd, 2010.

[27] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.

[28] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003. ISBN: 9780321127426.

[29] fuin.org. *DDD DSL: Xtext based DSL supporting Domain-driven design (DDD)*. `https://github.com/fuinorg/org.fuin.dsl.ddd`. [Online; Accessed: 2019-12-05].

[30] Emden R. Gansner and Stephen C. North. "An open graph visualization system and its applications to software engineering". In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11 (2000), pp. 1203–1233.

[31] G. Granchelli et al. "Towards Recovering the Software Architecture of Microservice-Based Systems". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 46–53. DOI: `10.1109/ICSAW.2017.48`.

[32] Timo Greifenberg et al. "Integration of Handwritten and Generated Object-Oriented Code". In: *Model-Driven Engineering and Software Development*. Ed. by Philippe Desfray et al. Cham: Springer International Publishing, 2015, pp. 112–132. ISBN: 978-3-319-27869-8.

[33] L. Grunske. "Formalizing architectural refactorings as graph transformation systems". In: *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*. 2005, pp. 324–329. DOI: `10.1109/SNPD-SAWN.2005.37`.

[34] Michael Gysel and Lukas Kölbener. "Service Cutter - A Structured Way to Service Decomposition". `https://eprints.hsr.ch/476/`. Bachelor Thesis. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2015.

[35] Michael Gysel and Lukas Kölbener. *Service Cutter - A Structured Way to Service Decomposition*. `http://servicecutter.github.io/`. [Online; Accessed: 2019-12-04].

[36] Michael Gysel et al. "Service Cutter: A Systematic Approach to Service Decomposition". In: *Service-Oriented and Cloud Computing*. Ed. by Marco Aiello et al. Cham: Springer International Publishing, 2016, pp. 185–200. ISBN: 978-3-319-44482-6.

[37] Michael Gysel et al. *Service Cutter Coupling Criteria Catalog*. `https://github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria`. [Online; Accessed: 2019-09-24].

[38] J. A. Hartigan and M. A. Wong. "Algorithm AS 136: A K-Means Clustering Algorithm". In: *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 28.1 (1979), pp. 100–108. ISSN: 00359254, 14679876.

[39] Erez Hartuv and Ron Shamir. "A clustering algorithm based on graph connectivity". In: *Information Processing Letters* 76.4 (2000), pp. 175 –181. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/S0020-0190(00)00142-3`.

[40] Jeffrey Heer, Stuart K. Card, and James A. Landay. “Prefuse: A Toolkit for Interactive Information Visualization”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’05. Portland, Oregon, USA: ACM, 2005, pp. 421–430. ISBN: 1-58113-998-5. DOI: 10.1145/1054972.1055031. URL: http://doi.acm.org/10.1145/1054972.1055031.

[41] Benjamin Hippchen et al. “Designing Microservice-Based Applications by Using a Domain-Driven Design Approach”. In: *International Journal on Advances in Software (1942-2628)* 10 (Dec. 2017), pp. 432 –445.

[42] I. Ivkovic and K. Kontogiannis. “A framework for software architecture refactoring using model transformations and semantic annotations”. In: *Conference on Software Maintenance and Reengineering (CSMR’06)*. 2006, 10 pp.–144. DOI: 10.1109/CSMR.2006.3.

[43] JHipster Team. *JHipster - Generate your Spring Boot + Angular/React applications!* https://www.jhipster.tech/. [Online; Accessed: 2019-12-13].

[44] JUNG Framework Development Team. *JUNG: Java Universal Network / Graph Framework*. http://jung.sourceforge.net. [Online; Accessed: 2019-10-09].

[45] M. Kamimura et al. “Extracting Candidates of Microservices from Monolithic Application Code”. In: *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 2018, pp. 571–580. DOI: 10.1109/APSEC.2018.00072.

[46] Stefan Kapferer. “A Domain-specific Language for Service Decomposition”. https://eprints.hsr.ch/722/. Term Project. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2018.

[47] Stefan Kapferer. “Empirical Research in Software Engineering”. https://eprints.hsr.ch/820/. Seminar Paper. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2019.

[48] Stefan Kapferer. *Master Thesis Project Definition: A Modeling Framework for Strategic Domain-driven Design and Service Decomposition*. 2019.

[49] Stefan Kapferer. “Model Transformations for DSL Processing”. https://eprints.hsr.ch/819/. Seminar Paper. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2018.

[50] Stefan Kapferer. “Service Decomposition as a Series of Architectural Refactorings”. https://eprints.hsr.ch/784/. Term Project. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2019.

[51] Kenichi Kobayashi et al. “Feature-gathering dependency-based software clustering using Dedication and Modularity”. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)* (2012). DOI: 10.1109/icsm.2012.6405308. URL: http://dx.doi.org/10.1109/ICSM.2012.6405308.

[52] Kenichi Kobayashi et al. *SArF Map: Visualizing Software Architecture from Feature and Layer Viewpoints*. 2013. arXiv: 1306.0958 [cs.SE].

[53] Philippe Kruchten. “The 4+1 View Model of Architecture”. In: *IEEE Software* 12.6 (1995), pp. 42–50. DOI: 10.1109/52.469759. URL: https://doi.org/10.1109/52.469759.

[54] Einar Landre, Harald Wesenberg, and Harald Rønneberg. “Architectural Improvement by Use of Strategic Level Domain-driven Design”. In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA ’06. Portland, Oregon, USA: ACM, 2006, pp. 809–814. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176728. URL: http://doi.acm.org/10.1145/1176617.1176728.

[55] Duc Minh Le, Duc-Hanh Dang, and Viet-Ha Nguyen. “On domain driven design using annotation-based domain specific language”. In: *Computer Languages, Systems & Structures* 54 (2018), pp. 199 –235. ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2018.05.001.

[56] Ian X. Y. Leung et al. “Towards real-time community detection in large networks”. In: *Physical Review E* 79.6 (2009). ISSN: 1550-2376. DOI: 10.1103/physreve.79.066107. URL: http://dx.doi.org/10.1103/PhysRevE.79.066107.

[57] Frank Lin and William W. Cohen. “Power Iteration Clustering”. In: *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel*. 2010, pp. 655–662. URL: https://icml.cc/Conferences/2010/papers/387.pdf.

[58] R.C. Martin et al. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN: 9780135974445.

[59] G. Mazlami, J. Cito, and P. Leitner. “Extraction of Microservices from Monolithic Software Architectures”. In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 524–531. DOI: 10.1109/ICWS.2017.61.

[60] T. Mens and T. Tourwe. “A survey of software refactoring”. In: *IEEE Transactions on Software Engineering* 30.2 (2004), pp. 126–139. ISSN: 2326-3881. DOI: 10.1109/TSE.2004.1265817.

[61] Microservice API Patterns (MAP) Project. *Lakeside Mutual*. https://github.com/Microservice-API-Patterns/LakesideMutual. [Online; Accessed: 2019-12-12].

[62] I. J. Munezero et al. “Partitioning Microservices: A Domain Engineering Approach”. In: *2018 IEEE/ACM Symposium on Software Engineering in Africa (SEiA)*. 2018, pp. 43–49.

[63] M. E. J. Newman and M. Girvan. “Finding and evaluating community structure in networks”. In: *Phys. Rev. E* 69 (2 2004), p. 026113. DOI: 10.1103/PhysRevE.69.026113. URL: https://link.aps.org/doi/10.1103/PhysRevE.69.026113.

[64] Nick Tune. *Domain-Driven Design: Hidden Lessons from the Big Blue Book*. Talk at Craft Conf Budapest, May 2019, http://ntcoding.co.uk/speaking/talks/domain-driven-design-hidden-lessons-from-the-big-blue-book/craft-conf-budapest-may-2019. [Online; Accessed: 2019-12-21].

[65] R. L. Nord and J. E. Tomayko. “Software architecture-centric methods and agile development”. In: *IEEE Software* 23.2 (2006), pp. 47–53. ISSN: 1937-4194. DOI: `10.1109/MS.2006.54`.

[66] Liam O’Brien, Christoph Stoermer, and Chris Verhoef. “Software Architecture Reconstruction: Practice Needs and Current Approaches”. In: (Jan. 2002).

[67] OpenAPI Initiative. *The OpenAPI Specification: a broadly adopted industry standard for describing modern APIs.* `https://www.openapis.org/`. [Online; Accessed: 2019-12-29].

[68] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules”. In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: `10.1145/361598.361623`. URL: `http://doi.acm.org/10.1145/361598.361623`.

[69] D. L. Parnas. “On the Criteria to Be Used in Decomposing Systems into Modules”. In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: `10.1145/361598.361623`. URL: `http://doi.acm.org/10.1145/361598.361623`.

[70] C. Pautasso et al. “Microservices in Practice, Part 1: Reality Check and Service Design”. In: *IEEE Software* 34.1 (2017), pp. 91–98. ISSN: 1937-4194. DOI: `10.1109/MS.2017.24`.

[71] Pivotal Software. *Spring Boot*. `https://spring.io/projects/spring-boot`. [Online; Accessed: 2019-10-05].

[72] plantuml.com. *Open-source tool that uses simple textual descriptions to draw UML diagrams.* `http://plantuml.com/`. [Online; Accessed: 2019-10-05].

[73] Michael Plöd. *DDD Context Maps - an enhanced view.* `https://speakerdeck.com/mploed/context-maps-an-enhanced-view`. [Online; Accessed: 2018-12-16].

[74] Michael Plöd. *Hands-on Domain-driven Design - by example*. Leanpub, 2019.

[75] F. Rademacher, J. Sorgalla, and S. Sachweh. “Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective”. In: *IEEE Software* 35.3 (2018), pp. 36–43. ISSN: 1937-4194. DOI: `10.1109/MS.2018.2141028`.

[76] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. “Towards a UML Profile for Domain-Driven Design of Microservice Architectures”. In: *Software Engineering and Formal Methods*. Ed. by Antonio Cerone and Marco Roveri. Cham: Springer International Publishing, 2018, pp. 230–245. ISBN: 978-3-319-74781-1.

[77] Chris Richardson. *Microservices patterns*. Manning Publications Shelter Island, 2018.

[78] Chris Richardson. *Pattern: Microservice Architecture*. `https://microservices.io/patterns/microservices.html`. [Online; Accessed: 2019-10-08].

[79] Chris Richardson. *Pattern: Microservice Architecture - How to decompose the application into services?* `https://microservices.io/patterns/microservices.html#how-to-decompose-the-application-into-services`. [Online; Accessed: 2019-12-21].

[80] Chris Richardson. *Pattern: Service instance per container*. `https://microservices.io/patterns/deployment/service-per-container.html`. [Online; Accessed: 2019-10-08].

[81] Fabrice Rossi and Nathalie Villa-Vialaneix. "Représentation d'un grand réseau à partir d'une classification hiérarchique de ses sommets". In: (Jan. 2011).

[82] Margaret Rouse. *Business Capability Definition*. `https://searchapparchitecture.techtarget.com/definition/business-capability`. [Online; Accessed: 2019-12-06].

[83] Sculptor Project. *Sculptor - Generating Java code from DDD-inspired textual DSL*. `https://github.com/sculptor/sculptor`. [Online; Accessed: 2019-12-05].

[84] Roded Sharan and Ron Shamir. "CLICK: A clustering algorithm with applications to gene expression analysis". In: *In Proc. 8th Int. Conf. Intelligent Systems for Molecular Biology*. 2000, pp. 307–316.

[85] Mary Shaw. "Writing Good Software Engineering Research Papers: Minitutorial". In: *Proceedings of the 25th International Conference on Software Engineering*. ICSE '03. Portland, Oregon: IEEE Computer Society, 2003, pp. 726–736. ISBN: 0-7695-1877-X. URL: `http://dl.acm.org/citation.cfm?id=776816.776925`.

[86] SmartBear. *Swagger*. `https://swagger.io/`. [Online; Accessed: 2019-12-29].

[87] Software Engineering Institute. *Architecture Reconstruction Case Study*. Tech. rep. CMU/SEI-2003-TN-008. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2003. URL: `http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=6431`.

[88] Sparx Systems. *Enterprise Architect*. `https://sparxsystems.com/`. [Online; Accessed: 2019-12-07].

[89] D. Steinberg et al. *EMF: Eclipse Modeling Framework*. Eclipse Series. Pearson Education, 2008. ISBN: 9780132702218.

[90] Roland H Steinegger et al. "Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications". In: *The Thrid Int. Conf. on Advances and Trends in Software Engineering*. 2017.

[91] Structurizr. *Structurizr - visualise, document and explore your software architecture*. `https://structurizr.com/`. [Online; Accessed: 2019-12-07].

[92] Davide Taibi. *A curated list of Open Source projects developed with a microservices architectural style*. `https://github.com/davidetaibi/Microservices_Project_List`. [Online; Accessed: 2019-10-05].

[93] Oliver Tigges. *How to break down a domain to bounded contexts?* `https://speakerdeck.com/otigges/how-to-break-down-a-domain-to-bounded-contexts`. [Online; Accessed: 2019-12-06].

[94] TIOBE - The Software Quality Company. *TIOBE Index*. `https://www.tiobe.com/tiobe-index/`. [Online; Accessed: 2019-10-08].

[95] V. A. Traag, L. Waltman, and N. J. van Eck. "From Louvain to Leiden: guaranteeing well-connected communities". In: *Scientific Reports* 9.1 (2019). ISSN: 2045-2322. DOI: `10.1038/s41598-019-41695-z`. URL: `http://dx.doi.org/10.1038/s41598-019-41695-z`.

[96] N. Tune and S. Millett. *Designing Autonomous Teams and Services: Deliver Continuous Business Value Through Organizational Alignment*. O'Reilly Media, 2017.

[97] Shmuel Tyszberowicz et al. "Identifying Microservices Using Functional Decomposition". In: *Dependable Software Engineering. Theories, Tools, and Applications*. Ed. by Xinyu Feng, Markus Müller-Olm, and Zijiang Yang. Cham: Springer International Publishing, 2018, pp. 50–65. ISBN: 978-3-319-99933-3.

[98] Vaughn Vernon. *Implementing Domain-Driven Design*. 1st. Addison-Wesley Professional, 2013. ISBN: 0321834577, 9780321834577.

[99] Harald Wesenberg, Einar Landre, and Harald Rønneberg. "Using domain-driven design to evaluate commercial off-the-shelf software". In: *Comp. to 21th Annual ACM SIGPLAN OOPSLA*. 2006, pp. 824–829. DOI: `10.1145/1176617.1176730`. URL: `https://doi.org/10.1145/1176617.1176730`.

[100] Claes Wohlin et al. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642290434, 9783642290435.

[101] yWorks. *yFiles Product Family*. `https://www.yworks.com/products/yfiles`. [Online; Accessed: 2019-10-09].

[102] Olaf Zimmermann. *A Domain-specific Language to specify (micro-)service contracts and data representations (realizing API Description pattern from MAP)*. `https://socadk.github.io/MDSL/`. [Online; Accessed: 2019-09-24].

[103] Olaf Zimmermann. "Architectural refactoring for the cloud: a decision-centric view on cloud migration". In: *Computing* 99.2 (2017), pp. 129–145. ISSN: 1436-5057. DOI: `10.1007/s00607-016-0520-y`. URL: `https://link.springer.com/article/10.1007/s00607-016-0520-y`.

[104] Olaf Zimmermann. "Microservices tenets". In: *Computer Science - Research and Development* 32.3 (2017), pp. 301–310. ISSN: 1865-2042. DOI: `10.1007/s00450-016-0337-0`. URL: `https://doi.org/10.1007/s00450-016-0337-0`.

[105] Olaf Zimmermann et al. *Microservice API Patterns*. `https://microservice-api-patterns.org`. [Online; Accessed: 2019-05-27].