UNIVERSITY OF APPLIED SCIENCES OF EASTERN SWITZERLAND (HSR FHO)

TERM PROJECT

Service Decomposition as a Series of Architectural Refactorings

---

Author:
Stefan KAPFERER

Supervisor:
Prof. Dr. Olaf ZIMMERMANN

A project 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

August 30, 2019

Declaration of Authorship

I, Stefan KAPFERER, declare that this thesis titled, "Service Decomposition as a Series of Architectural Refactorings" 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, August 30, 2019

Stefan Kapferer

UNIVERSITY OF APPLIED SCIENCES OF
EASTERN SWITZERLAND (HSR FHO)

Abstract

Master of Science FHO in Engineering focusing on Information and Communication Technologies

Service Decomposition as a Series of Architectural Refactorings

by Stefan KAPFERER

Decomposing a system into modules or services always has been a hard design problem. With the current trend towards microservices, this issue has become even more relevant and challenging. Domain-driven Design (DDD) with its Bounded Contexts provides one popular technique to decompose a domain into multiple parts. The open source tool Context Mapper, developed in our previous term project, offers a Domain-specific Language (DSL) for the strategic DDD patterns. DSL and supporting tools assist architects in the process of finding service decompositions. Context Mapper has already been used in practice projects, which led to suggestions how to improve the DSL to further increase its usability. Moreover, Context Mapper at present does not offer any transformations or refactoring tools to improve and evolve the DDD models. Finally, our previous work only gives very basic advice on how to implement systems that have been modeled in Context Mapper in a (micro-) service-oriented architectural style.

This work presents a series of Architectural Refactorings (ARs) for strategic DDD models based on corresponding Decoupling Criteria (DC) collected from literature and personal experience. These refactorings allow a software architect to (de-)compose a domain iteratively. Aiming for a broad DC coverage, a set of seven ARs has been implemented. These ARs are realized as code refactorings for the Context Mapper DSL (CML) language and support splitting, extracting and merging Bounded Contexts and/or Aggregates. Therefore, DSL users are able to refactor their CML models within the provided Eclipse plugin. A new service contract generator offers assistance how to implement the DDD models in an (micro-)service-oriented architecture. The resulting contracts are written in the Microservices Domain Specific Language (MDSL), another emerging DSL for specifying service contracts.

The provided DSL with its seven ARs, implemented as model transformations, support evolving DDD-based models in an iterative way. The conducted validation activities support our hypothesis that software architects can benefit from such an approach and tool. Action research has been applied to improve Context Mapper in each iteration of the prototypical implementation. Basic case studies conducted on real world projects in the industry indicated the usefulness and effectiveness of the modeling language. More advanced validation activities still have to be conducted to analyze and demonstrate the practicability of the ARs.

Acknowledgements

I want to express my gratitude to Prof. Dr. Olaf Zimmermann for sharing his knowledge and supporting this project as a supervisor. His insight and expertise greatly supported this work. As an early adopter of the Context Mapper tool [12] he further assisted the research and validation activities during the course of this work beyond his supervision duties.

I am also grateful to Moritz Habegger and Micha Schena for using our tool in their thesis [19] and providing the opportunity to apply our approach as a case study to a real-world project. Their feedback and the results of the case study have supported our validation activities notably.

Contents

Chapter 1

Introduction

1.1 Context and Problem

The Context Mapper DSL (CML) language [12] developed in our previous project [20] provides a modeling tool based on the strategic Domain-driven Design (DDD) patterns. With its bounded contexts[1] and context maps[1], DDD offers a popular approach to model service decompositions. The Domain-specific Language (DSL) implemented in the Context Mapper tool is meant to provide a foundation for service decomposition based on this DDD approach. However, by applying action research, conducting case studies, and using the tool as part of an application architecture exercise lesson at our university [20] we evaluated the tool and detected possible improvements within the grammar of the language. In addition, the tool currently only allows describing a certain state of a systems architecture. There is no tool support to evolve and improve the architecture iteratively. Changes to the models must be realized manually. Furthermore, the tool does not address the question how the DDD-based models can be realized in an (micro-)service-oriented architecture. To support software architects in evolving and implementing architectures modeled in CML the tool must therefore be improved.

1.2 Vision

Since we believe that DDD context maps and the architectures they describe are artifacts which evolve and change over time, we hypothesize that architects can benefit from a tool such as Context Mapper which supports the creation and evolution of DDD-based models in a formal and expressive way. However, the tool does currently only support the creation of such models but not evolving them with provided transformations. With this project we propose a series of *Architectural Refactorings (ARs)* [43] based on decomposition and coupling criteria collected from the literature and our own software engineering experience. The DSL concept offers the possibility to implement these ARs as code refactorings for the CML language. These refactorings support architects in evolving and improving the decomposition of a systems architecture in an iterative way. As part of this project we provide a set of AR implementations for the Context Mapper tool as a "proof of concept". Other refactorings may be implemented

[1] Please note that the DDD terms and patterns will be used throughout this paper without a re-introduction. For an introduction to the Context Mapper and DDD concepts we refer to our previous work [20] and the DDD literature [15, 16, 41].

in the future to extend the possibilities of the tool. In our own previous work [21] we have shown how such refactorings for DSLs can be implemented.

In addition to the ARs as the main objective of this project we present a "proof of concept" for a (micro-)service contract generator using the Microservices Domain-Specific Language (MDSL) [42]. The generator shall support architects in realizing the modeled systems and produces service contracts out of DDD context maps automatically. In this way we answer the question how DDD-based models with its bounded contexts can be implemented in a microservice architecture which has not been clearly answered yet.

In summary, the goals of this project are to improve existing grammar issues, analyze the criteria to be used to decompose a system, implement at least four ARs within the Context Mapper tool [12], realize a first "proof of concept" for a service contract generator and provide CML examples for the implemented ARs [22].

1.3 Results

Based on existing literature, the criteria catalog provided by Service Cutter [17], and our own professional experience [8, 23] we have selected a set of *Decomposition Criteria (DCs)*. Based on these DCs we derived seven ARs. Chapter 2 will explain our selection process and the criteria for the selection of the DCs and ARs in detail. As a prototype we implemented these seven ARs as code refactorings on the basis of CML in our Context Mapper tool [12]. Figure 1.1 provides an overview of the selected and implemented ARs.

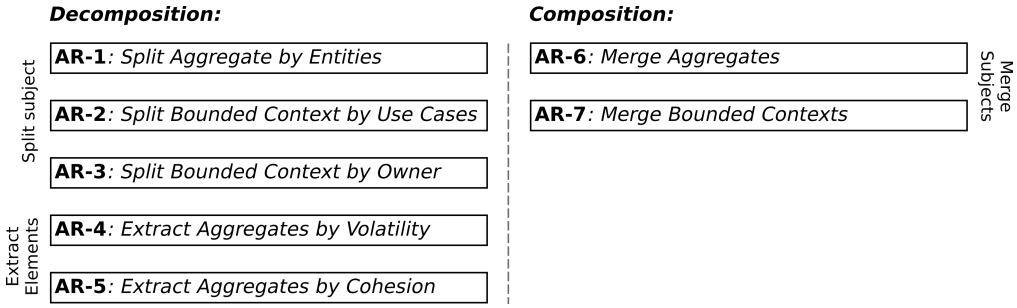

FIGURE 1.1: The implemented ARs organized by *decomposition vs. composition* and their operations *split*, *extract*, and *merge*.

With this set of ARs we aim for a broad coverage of coupling and decomposition criteria. The selection further allows to decompose (*split* and *extract* ARs) and compose (*merge* ARs) existing context maps modeled in CML.

Besides the ARs as the main result of this work, the CML language has been improved in order to achieve a better usability. In addition, the service contract generator implementation provides a suggestion how a (micro-)service-oriented architectural style can be derived from strategic DDD models. With this transformation a software architect is able to generate service contracts written in the MDSL language [42][2].

[2]MDSL leverages Microservice API Patterns (MAP) [44] as language constructs.

1.4 Related Work

Regarding the decomposition of software systems and the criteria to be used for this complex issue many research papers already exist. D. L. Parnas [31] presented one of the first papers regarding this topic many years ago. However, with this project we have not conducted a complete literature review, since the Service Cutter project [17] already presents a coupling criteria catalog [18] based on the available literature. In addition to this catalog and our own experience [8, 23] we consulted DDD-based, mostly gray, literature regarding the question how bounded contexts can be identified. We discuss these sources below. In Chapter 2 we will describe how we derived the ARs based on the criteria collected in the mentioned literature.

Brandolini [4] illustrates how context maps can evolve by splitting the initial problem domain iteratively. Based on his example, we created a "proof of concept" for an AR on our CML language [21]. With his example *"same term, different meaning"* he refers to the language as natural boundary between contexts. With *"same concept, different use"* he suggests to respect different use cases to find bounded contexts. He further introduced *event storming* [3, 9], a workshop-based technique to analyze a domain and discover bounded contexts.

Tune and Millett [39] describe how bounded contexts can be discovered using use cases and other domain heuristics such as language, domain expert boundaries, business process steps, dataflow, or ownership. They emphasize the importance of coevolving organizational and technical boundaries. The fact that the structure and architecture of a system reflects the organizational structures of a company is widely known as *Conway's law* [13]. Tigges [29] also presented a set of criteria to be considered to break down a domain into bounded contexts: domain objects and their relations, use cases, processes, workflows, quality goals, non-functional requirements and organizational aspects.

Plöd [33] calls the linguistic and model differences between parts of a system the primary drivers for the identification of bounded contexts. He further emphasizes the organization around business capabilities [25], decentralized governance and evolutionary design as microservice characteristics which suit the idea behind bounded contexts. Steinegger et al. [37] further mention the importance of decomposing a system in several iterations, which supports our hypothesis that a DDD context map should evolve iteratively.

All of the already mentioned authors aim to support DDD adopters and practitioners in decomposing systems and identifying bounded contexts. They provide criteria to be used for decomposing systems and are therefore complementary. However, besides the criteria to be used they mainly provide practical advice but do not describe concrete and structured procedures how the decomposition shall be done. Gysel et al. [17] and Tyszberowicz et al. [40] in comparison, propose structured approaches. Tyszberowicz et al. [40] illustrate an approach to identify microservices using functional decomposition which is based on use cases. Gysel et al. [17] implemented a structured approach, namely Service Cutter, based on graph clustering algorithms and a scoring system which uses their coupling criteria catalog.

In summary, all the mentioned literature and approaches tackle the problem how to decompose software systems. Therefore they contribute solutions

for the same main problem discussed in our work. However, the mentioned DDD-based approaches work with practices, especially *context mapping*, which are done manually by practitioners today. With our work we aim to answer the question whether these practitioners can benefit from tool-support for this practices or not. The existing practical advises and the criteria itself are not sufficient and not concrete enough to be implemented within such a tool. With our work we derive concrete transformation proposals in the form of ARs from the existing practices and criteria to be used for the decomposition of software systems. These transformations can then be implemented in a modeling tool such as Context Mapper [12].

Chapter 2

Decomposition Criteria and Architectural Refactorings (AR) Analysis

This chapter presents our analysis and selection of the criteria to be used to derive service decompositions. Based on these Decomposition Criteria (DCs) we derived a set of Architectural Refactorings (ARs). The chapter further explains how the DCs and ARs were selected for the implementation of our prototype in the Context Mapper [12] tool.

2.1 Service Decomposition Criteria Overview

Gysel et al. [17] present a catalog of coupling criteria for service decomposition distilled from the literature and industry experience. To select the decomposition criteria for our ARs, we first elaborated the criteria known from our own professional experience in a brainstorming.

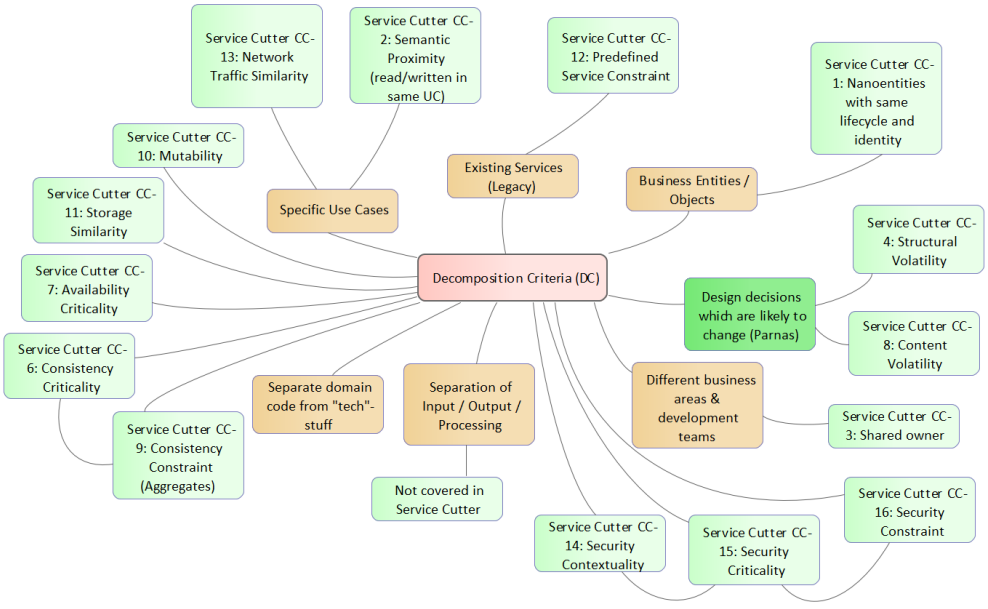

FIGURE 2.1: Decomposition Criteria (DC) «Brainstorming»

We additionally added the criterion from D. L. Parnas [31]. In a second step we compared our results with the catalog of Gysel et al. [17] which we used as our main source, since they already conducted an extensive literature review on this topic. Figure 2.1 illustrates the result of the brainstorming with the added criteria from Service Cutter [17] mapped to our own input. The complete criteria catalog of Service Cutter can also be found online[1]. The criteria we acquired from our own experience (brainstorming) were the following:

- Business entities and objects of the problem domain
- Use cases (data accessed by same use cases)
- Different business areas and development teams
- Existing services (decomposition is affected by legacy systems)
- Separation of input/output and processing (business) logic
- Separation of core domain and technical code

We further added the following criterion from D. L. Parnas original paper *On the criteria to be used in decomposing systems into modules* [31]:

- Decomposition by design decisions which are likely to change (isolate things which change often)

Together with the coupling criteria catalog of Gysel et al. [17] and our own input we derived the following list of DCs illustrated by Table 2.1. The table further maps our DCs to Service Cutters coupling criteria.

TABLE 2.1: Decomposition Criteria (DC) Overview

DC	**Name**	**Description**	**Source**	**Corresponding criteria from Gysel et al. [17]**
DC-1	Business entities	Entities/Objects and attributes which belong together according to problem domain.	Personal experience & Gysel et al. [17]	Identity and lifecycle commonality
DC-2	Use cases	Entities/Objects and attributes which are accessed by the same use cases.	Personal experience & Gysel et al. [17]	Semantic Proximity
DC-3	Business areas & development teams	Data belong to a same business area or are managed by the same development team.	Personal experience & Gysel et al. [17]	Shared Owner

[1] `https://github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria`

TABLE 2.1: Decomposition Criteria (DC) Overview (continued)

DC	Name	Description	Source	Corresponding criteria from Gysel et al. [17]
DC-4	Existing services	Data are coupled due to already existing legacy system which is hard to change.	Personal experience & Gysel et al. [17]	Predefined Service Constraint
DC-5	Input / Output vs. business logic	Processing or business logic is often separated from input/output code.	Personal experience	Not covered
DC-6	Business vs. technical code separation	Separation of core business logic and technical code.	Personal experience	Not covered
DC-7	Likelihood for change	Isolate things which are likely to change.	D. L. Parnas [31]	Structural Volatility, Content Volatility
DC-8	Mutability	Systems exchanging immutable objects are typically less coupled.	Gysel et al. [17]	Mutability
DC-9	Storage similarity	Data have to be stored in same storage.	Gysel et al. [17]	Storage Similarity
DC-10	Availability	Data which must provide similar availability characteristics.	Gysel et al. [17]	Availability Criticality
DC-11	Consistency	Data which must provide similar consistency characteristics.	Gysel et al. [17]	Consistenty Criticality, Consistency Constraint
DC-12	Security	Data which must provide similar security characteristics.	Gysel et al. [17]	Security Contextuality, Security Criticality, Security Constraint

2.2 Selection of Service Decomposition Criteria

Since the scope of the project and the amount of ARs implemented in the prototype had to be limited, we had to select a set of DCs from which we then derived ARs. The selection of the DCs has been done on the basis of the following selection criteria:

1. *Relevance in practice*: Choose criteria which are relevant for all software projects first and not only in specific contexts.
2. *Representativeness*: Prefer criteria which appeared in both, our own experience and in the criteria catalog of Gysel et al. [17] (multiple sources).
3. *Generality*: If the selected criteria are used for ARs, others can be used and implemented in a similar way.

Since our goal was to select five to seven ARs we decided to first select five DCs based on the criteria above. *DC-1: business entities* and *DC-2: use cases* are criteria which are very relevent in mostly every software project in practice (1). In addition both criteria are covered by our own considerations and Gysel et al. [17] (2). Other criteria such as *DC-4: existing services* are similar to the already mentioned DC-1 and DC-2 in terms of implementation. The criteria *DC-8: Mutability*, *DC-9: Storage similarity*, *DC-10: availability*, *DC-11: consistency* and *DC-12: security* are all concerning some specific Non-Functional Requirement (NFR) and we decided to derive a generalized DC and later a generalized AR for such cases. Based on this analysis and the selection criteria above we selected the following five DCs as foundation for the ARs:

1. DC-1: Business entities
2. DC-2: Use cases
3. DC-3: Business areas & development teams
4. DC-7: Likelihood for change
5. DC-{8-12}: *Generalized* non-functional requirement (NFR)
    - Generalized from DC-8, DC-9, DC-10, DC-11 and DC-12

Based these DCs the next section will explain how we derived and selected corresponding ARs.

2.3 Selection of Architectural Refactorings (ARs)

This section presents all candidates for ARs we determined by using the Decomposition Criteria (DC) from Section 2.2 above, the context mapping approaches of Brandolini [4], and ideas for structural transformations on the Context Mapper [12] DSL language itself.

2.3.1 AR Selection Criteria

To limit the amount of ARs for the prototype and the scope of this project we have defined the following selection criteria:

1. *Coupling Criteria Coverage*: The selected five coupling criteria in Section 2.2 must be covered.
2. *Implementation Generality*: If the selected ARs are implemented, others can be realized in a similar way.

3. *Operation Coverage*: The selection of the ARs shall include possibly all different operations, so that a user is able to evolve the model in different directions. Concretely, it should not only be possible to split the model into smaller bounded contexts and aggregates, but also to merge these objects to decrease the granularity.

2.3.2 Implementation Approaches

In order to select ARs based on these criteria, we categorize them into different implementation approaches. There are basically three different approaches from the technical point of view, listed in Table 2.2. The categories are based on different data input sources which are needed to implement the ARs. The design and implementation costs of ARs within the same category should be similar. By selecting at least one AR of each category we fulfill criterion (2) of the AR selection criteria in Section 2.3.1.

TABLE 2.2: Architectural Refactoring Design Categories

#	AR Category	Description
1	Simple «language-based» ARs	These ARs do not need any additional input and can be realized on the basis of the given DSL model.
2	ARs based on criteria added to the DSL	ARs of these category need additional information which have to be provided by the DSL. New language features are needed to realize these ARs.
3	ARs based on external user input	This category includes ARs which are based on the given DSL and external user input. These ARs might have higher implementation costs since corresponding User Interfaces (UIs) must be implemented in Eclipse.

Note that ARs of category 1 or 2 may require user input for parameters as well, but they are given by the selection of an element such as a bounded context or aggregate within the DSL editor. They do not require additional UIs to request user input.

2.3.3 Operations

In the following section presenting the AR candidates we will further distinguish between different operations as listed in Table 2.3.

TABLE 2.3: Architectural Refactoring Operations

Operation	Description
Split	A *split* AR splits the subject (aggregate or bounded context) into multiple objects of the same type. For example, splitting a bounded context into multiple bounded contexts based on one of the presented DCs. ARs of this type *decompose* a subject into multiples.

TABLE 2.3: Architectural Refactoring Operations (continued)

Operation	Description
Extract	An *extract* AR creates one new element of the type of the subject (bounded contexts) and moves a set of child-elements of the subject (aggregates) to the new object. For example, moving a set of aggregates which fulfill a specific DC from the existing bounded context into a new bounded context. ARs of this type *decompose* a subject into two objects.
Merge	A *merge* AR takes two subjects of the same type (aggregates or bounded contexts) and merges them together to one object. For example, merging two bounded context together into one bounded context. ARs of this type *compose* two subjects together into one.

The operation name is always part of the name of the AR and indicates the general purpose independent of the concrete DC it uses. By selecting at least one AR for each operation we fulfill criterion (3) of the AR selection criteria in Section 2.3.1.

2.3.4 AR Candidates

Based on the three implementation approaches we elaborated a set of AR candidates. These candidates are based on the presented DCs and the first experiences with the Context Mapper DSL (CML) language of the author and the supervisor of this project. Note that the list of candidates does not claim completeness and other ARs might be suggested in the future.

Simple «language-based» ARs

First, Table 2.4 lists all refactoring candidates which operate on the language constructs only. They do not require additional language features or external user input.

TABLE 2.4: Simple «language-based» ARs

#	AR Name	Parameters	Description
1	Split Aggregate by Entities (based on *DC-1: Business entities*)	Reference to aggregate	Splits an aggregate with multiple entities. The AR creates one aggregate per entity.
2	Extract Aggregate	Reference to aggregate	Removes the selected aggregate from the existing bounded context and moves it to a new bounded context.
3	Merge Bounded Contexts (BCs)	References to both BCs	Merges a bounded context with another one. The resulting context contains all aggregates of both bounded contexts.

TABLE 2.4: Simple «language-based» ARs (continued)

#	AR Name	Parameters	Description
4	Merge Aggregates	References to both aggregates	Merges an aggregate with another one. The resulting aggregate contains domain objects (entities, value objects, etc.) of both aggregates.
5	Extract Shared Kernel to new BC	Reference to Shared Kernel relationship.	Given a Shared Kernel relationship, this refactoring creates a new bounded context for the Shared Kernel and creates new relationship declarations between the existing bounded contexts and the new one.
6	Split by Duplicate Entity (inspired by Brandolini [4] and used in our previous work as "proof of concept" for ARs [21])	(Duplicate) entity name	Splits a bounded context if it contains two aggregates which both contain an entity with the same name. Results in two bounded contexts each containing one of the two entities.

ARs based on Criteria added to the DSL

The following AR candidates need additional language structures to specify the needed input. For the ARs of this category which are part of the prototype we will introduce the changes to the CML language in Chapter 4.

TABLE 2.5: ARs based on Criteria added to the DSL

#	AR Name	Needed input (DSL)	Description
7	Split Bounded Context by Use Cases[2] (based on *DC-2: use cases*)	Aggregates have to be assigned to use cases.	Splits a bounded context by grouping those aggregates together into one bounded context which are used by the same use case.
8	Split Bounded Context by Owner[2] (based on *DC-3: business areas & development teams*)	Aggregates have to be assigned to the owner (owning team).	Splits a bounded context by grouping those aggregates together into one bounded context which are managed by the same owner. We expect distinct ownership for this AR, meaning that an aggregate has exactly one owner.

[2] *Split Bounded Contexts by Use Cases* vs. *Split Bounded Contexts by Owner*: Note that the difference in the naming of these two ARs (singular vs. plural) highlights that an aggregate can be used by *multiple use cases* but is owned by only *one owner*.

TABLE 2.5: ARs based on Criteria added to the DSL (continued)

#	AR Name	Needed input (DSL)	Description
9	Extract Aggregates by Volatility (based on *DC-7: Likelihood for change*)	Requires information which aggregates are likely to change. (volatility)	Moves all aggregates with a certain volatility (likelihood for change) into a new bounded context.

ARs based on external user input

The last part of the AR candidates list in Table 2.6 shows ARs which need additional user input not provided by the CML model.

TABLE 2.6: ARs based on External User Input

#	AR Name	User Input	Description
10	Extract Aggregates by Cohesion (based on *DC-{8-12}: Generalized NFR*)	Aggregates which shall be extracted.	The user selects the aggregates to form a new bounded context manually based on any NFR criterion. The goal of the AR is to improve the cohesion within the bounded contexts.
11	Change relationship type	Specification of the relationship.	With this AR the user shall be able to refactor the context map, especially the relationships between the bounded contexts. The AR allows to change the type of a specific relationship.

2.3.5 AR Selection

Based on our AR selection criteria listed in Section 2.3.1, *coupling criteria coverage*, *implementation generality* and *operation coverage*, we derived the following set of seven ARs to be implemented in the prototype[3]:

- **AR-1**: Split Aggregate by Entities (*#1*)
- **AR-2**: Split Bounded Context by Use Cases (*#7*)
- **AR-3**: Split Bounded Context by Owner (*#8*)
- **AR-4**: Extract Aggregates by Volatility (*#9*)
- **AR-5**: Extract Aggregates by Cohesion (*#10*)

[3]Note that we have renumbered the selected ARs. The original numbers according to the candidates list above are written in brackets.

- **AR-6**: Merge Aggregates (#*4*)
- **AR-7**: Merge Bounded Contexts (#*3*)

This selection of ARs fulfills our first criterion *coupling criteria coverage* as illustrated in Figure 2.2. The five DCs selected in Section 2.2 are fully covered by these ARs.

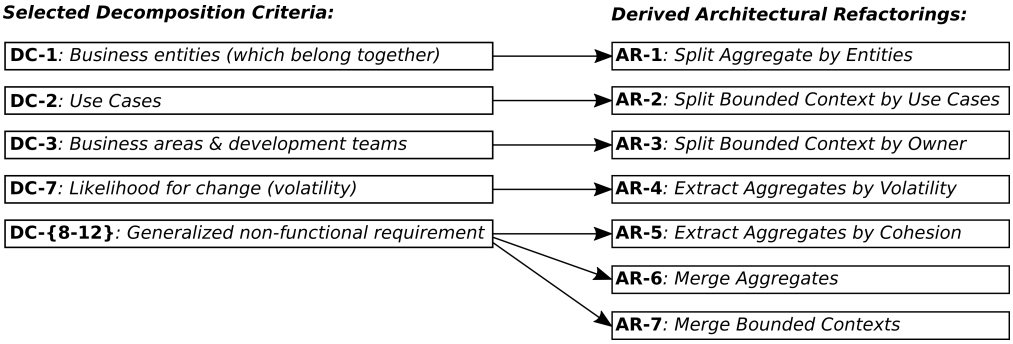

FIGURE 2.2: AR Selection by Decomposition Criteria (DC)

Note that AR-5, AR-6 and AR-7 are based on manual selections of the subjects by the user. Thereby these ARs support the application of refactorings by any NFRs. Whether an application of such an AR improves the architecture or not depends on the knowledge of the user.

The selection does also fulfill our second criterion *implementation generality*. As the reader may check, the selection contains ARs of all three implementation approaches presented in Section 2.3.2. Finally, the criterion *operation coverage* is fulfilled as well, since the selection contains all three operations *split*, *extract* and *merge* as presented in Section 2.3.3.

2.4 Architectural Refactoring (AR) Summaries

To complete this chapter concerning the analysis of the Decomposition Criteria (DCs) and Architectural Refactorings (ARs) this section summarizes the seven selected ARs for our prototype. The summaries describe the context, motivation, solution and effects for each AR. Examples illustrating how the ARs work will be presented in Chapter 4 by using CML.

2.4.1 AR-1: Split Aggregate by Entities

Context

On the level of attributes, or *nanoentities* in the terminology of Service Cutter [17], it is common to group those together which *belong to the same identity and share a common lifecycle*[4] to form entities. On the level of business entities we typically try to group those entities together which belong to the same part or

[4] `https://github.com/ServiceCutter/ServiceCutter/wiki/CC-1-Identity-and-Lifecycle-Commonality`

area of the business (domain). These different areas form *linguistic boundaries* and often also *domain expert boundaries* as explained by Tune and Millett [39].

Thereby we always aim to reduce the coupling between the entities and increase the cohesion within the entities. 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 to split your aggregate into multiple aggregates 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.

2.4.2 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 [29], Plöd [33], Tune and Millett [39] or Tyszberowicz et al. [40] 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 [10]. The approach further supports the *single responsibility principle* by R.C. Martin et al. [26]. In the Service Cutter [17] coupling criteria catalog this principle corresponds to the *semantic proximity*[5] 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

[5] `https://github.com/ServiceCutter/ServiceCutter/wiki/CC-2-Semantic-Proximity`

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.

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.

2.4.3 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 [4], Tune [28], Plöd [33] and Tune and Millett [39] is widely known as *Conway's law* [13]. Tune [28] emphasizes that «*bounded contexts decouple parts*» and «*parts are code and teams*». In our previous work [20] we already respected this aspect and implemented the possibility to model teams in CML. Service Cutter [17] handles this aspect with the *shared owner*[6] 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 [33] and enables team autonomy [28].

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.

[6] `https://github.com/ServiceCutter/ServiceCutter/wiki/CC-3-Shared-Owner`

2.4.4 AR-4: Extract Aggregates by Volatility

Context

With the paper *On the criteria to be used in decomposing systems into modules* D.L. Parnas [31] 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 are likely to change*. According to Service Cutters [17] criterion *structural volatility*[7] 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 [31].

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*.

2.4.5 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. As already explained in Section 2.3.4, we derived this AR to enable architects to decompose by any *generalized* NFR criterion. Thereby it is possible to manually select the aggregates to be extracted aiming for improved coupling and cohesion. Examples for such 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

[7] `https://github.com/ServiceCutter/ServiceCutter/wiki/CC-4-Structural-Volatility`

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 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*.

2.4.6 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 [39]. 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 [25], 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.

2.4.7 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 [25], 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 [25], 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*

Within this chapter we analyzed the criteria to be used for decomposing a domain into bounded contexts and proposed a set of Architectural Refactorings (ARs) for DDD-based architecture models. These ARs allow a software architect to evolve DDD context maps and improve the architecture iteratively. The next chapter will discuss the requirements for our prototype implementation within the Context Mapper tool [12].

Chapter 3

Domain-specific Language, AR and Generator Requirements

This chapter discusses the requirements this projects and especially the prototypic implementations in Context Mapper [12] have to fulfill. It presents the functional requirements for the Architectural Refactorings (ARs) and the service contract generator implemented during this project. In addition, it lists the Non-Functional Requirement (NFR) for all new Context Mapper features including changes to the Context Mapper DSL (CML).

First we will present the functional requirements for the ARs and the service contract generator in the form of User Stories (US). All user stories [2] are based on the "Role-Feature-Reason" template invented 2001 by a team at Connextra in the UK [1]:

> «As a <***who*** *wants to accomplish something*>,
> I want to <***what*** *they want to accomplish*>
> so that <***why*** *they want to accomplish that thing*>.» [1]

3.1 Architectural Refactoring User Stories

In our previous project [20] we have already considered to implement ARs for the CML language. The corresponding user story was defined as follows:

> «As a software architect or engineer, I want to apply manual or automatic transformations and *Architectural Refactorings* [7, 43] to my model so that I can port and modernize the architecture.» [20]

A Domain-driven Design (DDD) variant of the same user story was stated as follows:

> «As a software architect or engineer, I want to apply manual or automatic transformations and *Architectural Refactorings* [7, 43] to my DDD context map so that I can decompose services (bounded contexts) in order to decrease the coupling between them.» [20]

With the following user stories we concretize the idea behind this first AR story based on the ARs we have selected in Chapter 2. For each AR we present a user story illustrating the requirements from a users perspective. The first user story US-0 highlights the general purpose and goal behind this project, namely providing tools to support evolving a DDD context map and the corresponding software architecture iteratively.

3.1.1 US-0: Craft and evolve Context Maps iteratively

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.

3.1.2 US-1: Split Aggregate by Entities

As a software architect or engineer, I want to split an existing aggregate by its entities in case they do not belong together according to Decomposition Criterion (DC) *DC-1* and the aggregate exhibits an unsatisfying cohesion, so that the entities are distributed to different aggregates and coupling and cohesion of the aggregates improves.

3.1.3 US-2: Split Bounded Context by Use Cases

As a software architect or engineer, I want to split an existing bounded context by the use cases of the aggregates, so that aggregates which are involved in the execution of the same use cases form a bounded context and therefore the cohesion within these bounded contexts is high while the coupling between bounded contexts is reduced.

3.1.4 US-3: Split Bounded Context Owner

As a software architect or engineer, I want to split an existing bounded context by the owner (team) of the aggregates, so that aggregates which are managed by the same owner form a bounded context leading to team autonomy and clear responsibilities and therefore the cohesion within these bounded contexts is high while the coupling between bounded contexts is reduced.

3.1.5 US-4: Extract Aggregates by Volatility

As a software architect or engineer, I want to extract aggregates which are likely to change into a separate bounded context, so that these aggregates are isolated within their own component and changes are hidden from the others.

3.1.6 US-5: Extract Aggregates by Cohesion

As a software architect or engineer, I want to extract aggregates which share a specific characteristic regarding an arbitrary DC such as *DC-12: security*, so that these aggregates form a new bounded context and the cohesion within the existing and the new bounded context is improved.

3.1.7 US-6: Merge Aggregates

As a software architect or engineer, I want to merge aggregates which have become to fine-granular and contain entities which belong together (high coupling), so that the amount of aggregates is reduced, the coupling between the remaining aggregates is reduced, and the cohesion within the aggregates is increased.

3.1.8 US-7: Merge Bounded Contexts

As a software architect or engineer, I want to merge bounded contexts in situations where the decomposition is too fine-granular and the coupling between the corresponding bounded contexts is too high, so that the amount of bounded contexts is decreased, the coupling between contexts reduced, and the cohesion within the merged bounded context is high.

3.2 Service Contract Generator User Stories

Besides the main topic of this project, the ARs, we also implemented a "proof of concept" for a service contract generator as already mentioned in the introduction. The following user story shall illustrate what value for the users we expect from such a tool. With the concrete implementation explained in Chapter 4 we further propose one approach how (micro-)service architectures can be derived from DDD-based models (context maps).

3.2.1 US-8: Generate Service Contracts out of DDD Context Maps

As a software architect, I want to generate service contracts out of my CML context maps providing suggestions how the (micro-)services and message exchange can be implemented, so that I can analyze which services, endpoints and operations are needed for the concrete implementation quickly and am supported in planning and coordinating the development of the system.

3.3 Non-Functional Requirements (NFRs)

Note that all NFRs for the Context Mapper tool [12] which have already been defined in our previous project [20] still have to be fulfilled. Some of the NFRs below are therefore copied from the previous project report [20].

3.3.1 Refinement of Context Mapper DSL

Through the validation of the CML language in our previous work [20] we derived improvements regarding the syntax of the language in order to increase its usability. The concrete issues were managed in our Github repository[1]. However, all changes regarding the CML language implemented as part of this project will be explained in detail in Chapter 4. The changes to the Domain-specific Language (DSL) must conform to the following NFRs.

Quickly Writable without Redundancy

The definition of relationships on a CML context map must be 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.

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

Well Readable

The definitions on a context map shall be well readable. A user which is familiar with our DDD meta-model[2] [20] 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 [20] 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 [14].

«In line» with Common DDD Styles and the DDD Literature

The new syntax definitions should not deviate from common styles and understanding of the patterns in the DDD community and the literature. For example if pattern name abbreviations are used, the implementation should use the ones which are already established in the community.

3.3.2 Architectural Refactoring (AR) NFRs

The following NFRs concern the implementation of the ARs only.

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.

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[3].

3.3.3 Common NFRs

Besides the already mentioned NFRs for the CML language and the ARs, the following NFRs must be fulfilled for all implementations and prototypes of this project.

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

Future-oriented Use of Tools and Frameworks

The tools and libraries used for the development of the ARs and generator tools 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 (such as XML or ECore).

Reliability

The developed tools should work reliable 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).

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 «General Public License (GPL)» must not be used.

Supportability and Maintainability

The projects code quality should be kept at a good level. Setup appropriate tools and mechanisms to support this goal (updating *master* only by pull request, integrate static code analysis tools into the continuous integration pipeline). 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 [12] shall be documented on the project's documentation website[4].

[4] `https://contextmapper.github.io/docs`

Chapter 4

Context Mapper: Design and Implementation

The prototypic implementations of this project are based on the foundations we have already realized during our last project [20] and continues the work on the open source project Context Mapper [12]. The following sections will first explain the changes made to the Context Mapper DSL (CML) language followed by the design and implementation of the new features, namely the Architectural Refactorings (ARs) and the (micro-)service contract generator.

4.1 Revised Context Mapper DSL (CML) Syntax

During the evaluation of the previous project [20] we detected a few ambiguities and possible improvements within the grammar of the CML language. The detailed feedback of the evaluators has been documented in the project report [20]. This section documents the changes applied to the grammar in order to improve the language according to the users feedbacks. Note that Appendix B contains a complete and revised CML language reference explaining the current version of the syntax.

The semantic model of the language has not changed and still corresponds to the language version as described in [20]. The following Figure 4.1 illustrates the Context Mapper [12] language meta-model [20] to recall the language semantics based on the strategic Domain-driven Design (DDD) patterns.

However, the syntax of the bounded context relationships on the context maps has been revised and improved to increase consistency and reduce ambiguities. The new syntax works with abbreviations for the DDD patterns and is therefore more compact in comparison to the previous version.

4.1.1 DDD Pattern Abbreviations

The following Table 4.1 lists all strategic DDD patterns which are used in bounded context relationships with their abbreviation. The abbreviations **OHS**, **PL** and **ACL** are used according to Vernon [41]. The abbreviation **CF** is chosen according to the formal notation proposal by Plöd [27]. In line with all authors and DDD experts we use the **U** for upstream and the **D** for downstream.

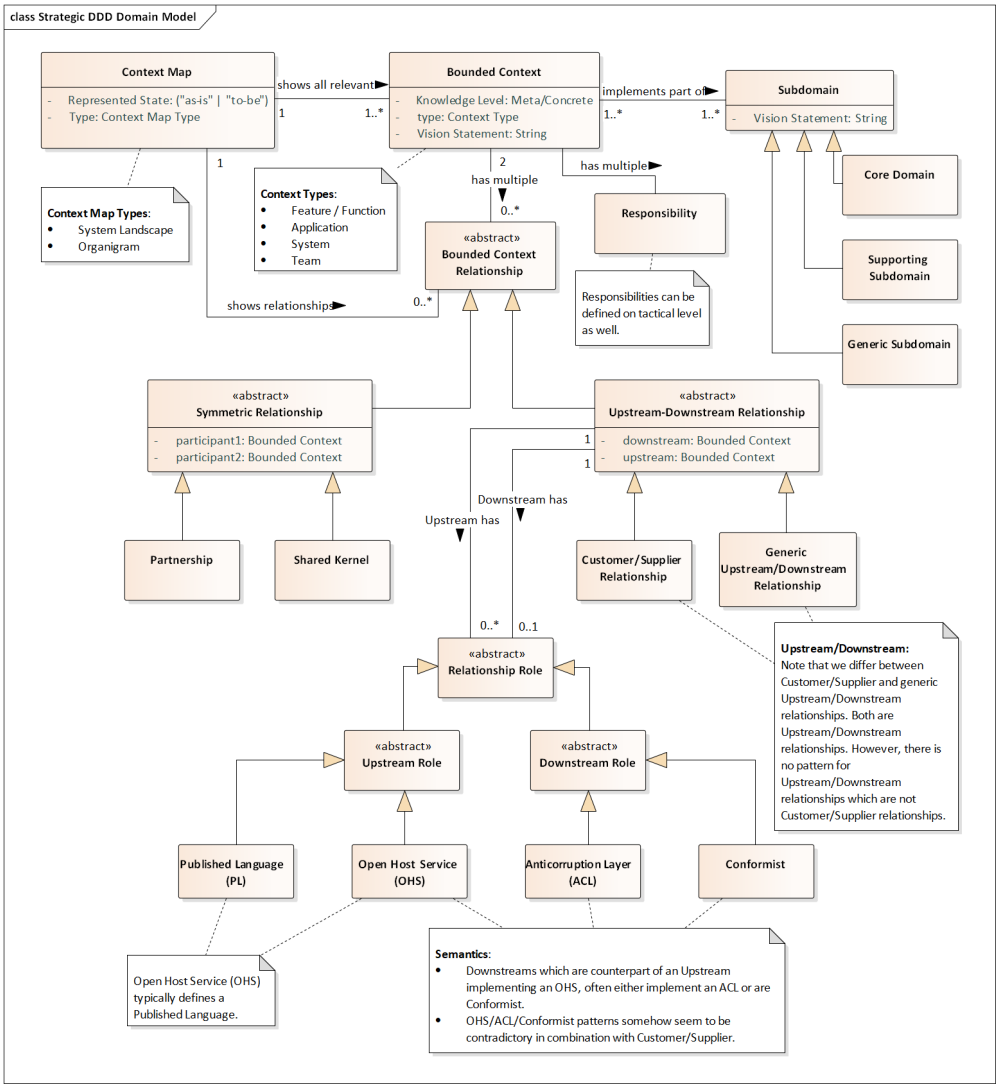

FIGURE 4.1: Context Mapper Semantic Model [12]

For the customer-supplier relationship, which is a special case of an upstream-downstream relationship according to our interpretation, we use **S** (supplier) and **C** (customer) respectively.

TABLE 4.1: DDD Pattern Abbreviations

Abbreviation	**Pattern**
OHS	Open Host Service
PL	Published Language
ACL	Anti-Corruption Layer
CF	Conformist
U	Upstream in upstream-downstream relationship
D	Downstream in upstream-downstream relationship
S	Supplier in customer-supplier relationship
C	Customer in customer-supplier relationship
P	Partner in a Partnership relationship

TABLE 4.1: DDD Pattern Abbreviations (continued)

Abbreviation	Pattern
SK	Shared Kernel

4.1.2 Symmetric Bounded Context Relationships

For all relationship declarations, CML provides a "short" syntax and a longer alternative. The "short" version for symmetric relationships uses an arrow directing to the left and to the right (<->), indicating symmetry.

Partnership

Listing 1 illustrates an example of a Partnership relationship. The full example of this and all following listings can be found in our examples repository[1].

```
ContextMap {
  contains PolicyManagementContext
  contains RiskManagementContext

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

LISTING 1: Partnership Relationship in "short" Syntax

The "short" syntax variant allows the brackets to be placed in front of the bounded context name or behind. This allows different variations (listings 2, 3, 4 and 5) depending on the users preferences.

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

LISTING 2: Brackets: Option 1 (Right/Left)

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

LISTING 3: Brackets: Option 2 (Left/Left)

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

LISTING 4: Brackets: Option 3 (Right/Right)

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

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

LISTING 5: Brackets: Option 4 (Left/Right)

As in the first CML version, a longer alternative version exists with the whole pattern name, as shown in Listing 6.

```
RiskManagementContext Partnership PolicyManagementContext
```

LISTING 6: Partnership Alternative Syntax

Shared Kernel

Similar to the Partnership relationship, the Shared Kernel describes a symmetric relationship. Therefore, the syntax is identical, but uses the **SK** abbreviation. Listing 7 shows an example in the "short" syntax version and Listing 8 in the alternative version.

```
PolicyManagementContext [SK]<->[SK] DebtCollection {
  implementationTechnology = "Shared Java Library, Communication over RESTful HTTP"
}
```

LISTING 7: Shared Kernel "short" Syntax

```
PolicyManagementContext Shared-Kernel DebtCollection
```

LISTING 8: Shared Kernel Alternative Syntax

Analogous to the Partnership pattern, the brackets of the "short" syntax version can be written to the left or to the right of the bounded context name (see the four options as described for the Partnership relationship). By using a colon it is optionally possible to give every relationship a name, as shown in Listing 9.

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

LISTING 9: Relationship Name

4.1.3 Upstream-Downstream Relationships

The declarations of upstream-downstream relationships use arrows either to the left (<-) or to the right (->). The arrow always points from the upstream towards the downstream and expresses the *influence flow* [27]. The upstream influences the downstream, whereas the downstream has no influence on the upstream.

Generic Upstream-Downstream Relationship

Listing 10 illustrates an upstream-downstream relationship in the "short" CML syntax.

```
ContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  contains PolicyManagementContext
  contains PrintingContext

  PrintingContext [U]->[D] PolicyManagementContext {
    implementationTechnology = "SOAP"
  }
}
```

LISTING 10: Upstream-Downstream "short" Syntax (1)

The arrow can be used in both directions. Thus, the relationship declaration in Listing 11 is semantically the same declaration as the one in Listing 10.

```
PolicyManagementContext [D]<-[U] PrintingContext {
  implementationTechnology = "SOAP"
}
```

LISTING 11: Upstream-Downstream "short" Syntax (2)

Similar as with symmetric relationships, the "short" version of the syntax allows the brackets to be on the left or on the right of a bounded context name, which leads to the four options illustrated by the listings 12, 13, 14, and 15.

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

LISTING 12: Brackets: Option 1 (Right/Left)

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

LISTING 13: Brackets: Option 2 (Left/Left)

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

LISTING 14: Brackets: Option 3 (Right/Right)

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

LISTING 15: Brackets: Option 4 (Left/Right)

The upstream and downstream roles, such as Open Host Service (OHS) or Anti-Corruption Layer (ACL), are declared within the brackets in the new CML version. Listing 16 shows an example of an upstream-downstream relationship with role patterns added. Note that the brackets always have to start with the **U** or **D**, followed by the roles.

```
1 PrintingContext [U,OHS,PL]->[D,ACL] PolicyManagementContext {
2   implementationTechnology = "SOAP"
3 }
```

LISTING 16: Upstream-Downstream "short" Syntax with Roles

As well as for symmetric relationships, CML provides a longer alternative syntax version for upstream-downstream relationships. In this case, the *Upstream-Downstream* or *Downstream-Upstream* keyword is used instead of the arrows. The listings 17 and 18, which are semantically equivalent, show a corresponding example.

```
1 PrintingContext Upstream-Downstream PolicyManagementContext
```

LISTING 17: Upstream-Downstream Alternative Syntax (1)

```
1 PolicyManagementContext Downstream-Upstream PrintingContext
```

LISTING 18: Upstream-Downstream Alternative Syntax (2)

This syntax variant works with brackets for the relationship role patterns as well. In contrast to the "short" version the brackets contain the roles only, since the **U** and **D** are given by the keyword and are thus not necessary. Listing 19 illustrates an example.

```
1 CustomerManagementContext[OHS,PL] Upstream-Downstream [CF]PolicyManagementContext {
2   implementationTechnology = "RESTful HTTP"
3 }
```

LISTING 19: Upstream-Downstream Alt. Syntax with Roles

Note: In this syntax version, the brackets must be written next to the relationship keyword in the middle (on the left and right of *Upstream-Downstream* and *Downstream-Upstream* respectively) and can not be placed as free as with the "short" syntax version.

Customer-Supplier Relationship

A customer-supplier relationship is a special case of an upstream-downstream relationship, according to our interpretation. Thus, the syntax corresponds to the upstream-downstream syntax as introduced above. If the user wants to

declare an upstream-downstream relationship to be a customer-supplier relationship, he can simply add the abbreviations **C** (customer) and **S** (supplier). Listing 20 illustrates an example.

```
1 CustomerSelfServiceContext [D,C]<-[U,S] CustomerManagementContext
```

LISTING 20: Customer-Supplier "short" Syntax (1)

The parser also allows to ommit the **U** and the **D** in a customer-supplier relationship, since the upstream is always the supplier and the downstream the customer. In this case the same example would be declared as in Listing 21.

```
1 CustomerSelfServiceContext [C]<-[S] CustomerManagementContext
```

LISTING 21: Customer-Supplier "short" Syntax without *U*/*D*

Remember that the arrow must still direct from the upstream towards the downstream (influence flow), or in a customer-supplier relationship from the supplier towards the customer. However, you can use the arrow in both directions, as already explained for the generic upstream-downstream case.

Note: We would recommend to use the syntax in Listing 20 instead of Listing 21, which shows that the relationship is an upstream-downstream as well as a customer-supplier relationship. This avoids any ambiguity and clearly states that a customer-supplier relationship is always an upstream-downstream relationship as well.

Similar to the generic upstream-downstream case, it is possible to add further relationship roles to the brackets, as illustrated by Listing 22.

```
1 CustomerSelfServiceContext [D,C,ACL]<-[U,S,PL] CustomerManagementContext
```

LISTING 22: Customer-Supplier "short" Syntax with Roles

The customer-supplier relationship can be declared with a longer alternative syntax as well. Instead of the keywords *Upstream-Downstream* or *Downstream-Upstream*, you simply use the keywords *Customer-Supplier* or *Supplier-Customer*. The listings 23 and 24, which are semantically equivalent, illustrate an example.

```
1 CustomerSelfServiceContext Customer-Supplier CustomerManagementContext
```

LISTING 23: Customer-Supplier Alternative Syntax (1)

```
1 CustomerManagementContext Supplier-Customer CustomerSelfServiceContext
```

LISTING 24: Customer-Supplier Alternative Syntax (2)

Additional pattern roles can again be added with the brackets near the relationship keyword in the middle (see Listing 25).

```
1 CustomerSelfServiceContext[ACL] Customer-Supplier [PL]CustomerManagementContext
```

LISTING 25: Customer-Supplier Alternative Syntax with Roles

Finally, as for all relationship declarations, the upstream-downstream and customer-supplier relationship declarations support to add a name at the end, by using a colon. Listing 26 illustrates an example.

```
1 CustomerSelfServiceContext [D,C]<-[U,S] CustomerManagementContext : ExampleName
```

LISTING 26: Customer-Supplier Relationship with Name

Arrow Defaults

Note that it is still possible to use the arrows (->, <- and <->) without the brackets, even if we do not recommend it due to decreased readability. However, if you use the arrows without brackets it is important to know that -> and <- declare upstream-downstream relationships and **not** customer-supplier relationships. Using the symmetric arrow <-> without brackets declares a **Shared Kernel**.

4.1.4 Other Small Syntax Changes & Syntactic Sugar

The following Table 4.2 lists all additional and small syntax changes which have been made. We will not explain these little changes in detail, since we provide an online documentation[2] which explains all features regarding the current version *v4.1.1*[3] of the language.

TABLE 4.2: Small Syntax Changes & Syntactic Sugar

Change	Description
Name for context map	It is now possible to give context maps names. The name can simply be added after the *ContextMap* keyword. For example: `ContextMap myInsuranceContextMap { ... }`
Adding multiple bounded contexts with one *contains* keyword	Earlier versions required to use one *contains* keyword for each bounded context added to a context map. With the latest release it is possible to add multiple bounded contexts by using only one *contains* keyword and separating the bounded contexts with commas. For example: `contains context1, context2`

[2] https://contextmapper.github.io/docs/

[3] `https://github.com/ContextMapper/context-mapper-dsl/releases/tag/v4.1.1`

TABLE 4.2: Small Syntax Changes & Syntactic Sugar (continued)

Change	Description
Introduced *Domain* keyword	*Subdomains* are no longer root elements of the model but part of a *Domain*. The *Domain* is declared on the root level of the CML file.
Brackets *{ ... }* optional for entities and aggregates	The brackets which surround the contents of entities and aggregates are optional now and do not have to be written if no content is specified.
Responsibilities are simple strings without ID.	Responsibilities are simple comma-separated string lists now. The specific grammar rule with an ID for the responsibility has not been used yet.

4.1.5 Additional Language Features

Besides changes in the syntax, this project added a few new language features to CML. As we will se later in the section about the ARs, these changes are needed to provide specific information needed by the refactorings. This section summarizes the additional language features added during this project quickly.

Downstream Rights

As Plöd [33] mentions, a team or bounded context might have varying power or influence on the other part of a relationship. Concretely, it can be of interest how many influence a downstream has regarding the activities of the upstream in an upstream-downstream relationship.

In CML we added the possibility to specify the downstream's power or influence on the upstream with the *downstreamRights* attribute. Listing 27 shows an example how this can be specified.

```
1 VoyagePlanningContext [D,ACL]<-[U,OHS,PL] LocationContext {
2   implementationTechnology = "RESTful HTTP"
3   downstreamRights = VETO_RIGHT
4 }
```

LISTING 27: *downstreamRights* Attribute Example

Table 4.3 lists the possible values which are accepted for the *downstreamRights* attribute.

TABLE 4.3: Downstream Right Characteristics

Value	Description
INFLUENCER	The downstream has no direct power but aims to influence the upstream activities.
OPINION_LEADER	In this case the downstream has influence in such a way that his opinions are respected by the upstream.

TABLE 4.3: Downstream Right Characteristics (continued)

Value	Description
VETO_RIGHT	The downstream has a veto right regarding all decisions of the upstream and can at least prevent undesired changes.
DECISION_MAKER	In this case the downstream has the power to make decisions about the upstream activities and changes.
MONOPOLIST	The downstream has the exclusive power to decide about changes in the upstream context. There are no other downstream contexts which interfere.

Exposed Aggregates

An upstream bounded context exposes parts of his model to the downstream bounded contexts with which he has relationships. For our service contract generator which will be introduced later, it will be important to know which aggregates an upstream bounded context exposes in his relationships.

In CML it is now possible to define the exposed aggregates with the *exposedAggregates* attribute for each upstream-downstream or customer-supplier relationship. Listing 28 illustrates an example of such a declaration.

```
1 VoyagePlanningContext [D,ACL]<-[U,OHS,PL] LocationContext {
2   implementationTechnology = "RESTful HTTP"
3   downstreamRights = VETO_RIGHT
4   exposedAggregates = Customers, Addresses
5 }
```

LISTING 28: *exposedAggregates* Attribute Example

Note that the two aggregates *Customers* and *Addresses* in the example above must be part of the upstream, which is the *LocationContext*.

Use Cases

As we have already seen in Chapter 2, use cases play an important role regarding the decomposition of a domain into bounded contexts. To be able to apply refactorings based on this information, we must know which parts of the system are accessed by which use cases. Therefore, we added the possibility to CML to assign aggregates to use cases.

Listing 29 illustrates how the user can define use cases on the root level of the model. These use cases can then be assigned to the corresponding aggregates, as shown in Figure 30.

```
1 /* Simple use case (only name given) */
2 UseCase UpdateContract
3 UseCase UpdateOffer
```

LISTING 29: Simple Use Case Declaration

```
Aggregate Contract {
  useCases = UpdateContract

  Entity Contract {
    aggregateRoot

    /* removed content to save space */
  }

  /* removed other domain objects to save space */
}
```

LISTING 30: Aggregate: Assigning Use Cases

The *useCases* attribute takes a list of references, thereby it is possible to assign multiple use cases to an aggregate (comma-separated). Note that it is also possible to specify use cases in more detail as illustrated in Figure 31.

```
/* Extended declaration with read and written attributes */
UseCase CreateOfferForCustomer {
  reads "Customer.id", "Customer.name"
  writes "Offer.offerId", "Offer.price", "Offer.products", "Offer.client"
}
```

LISTING 31: Advanced Use Case Declaration

This use case declaration with the attributes which are read and written corresponds to the use case declaration needed for the Service Cutter [17] integration explained in our previous work [20]. Thereby you can reuse the same way of declaration and future releases of the Service Cutter input generator may use these CML declarations directly.

Owners

Another important criterion for service decomposition are the teams and the structure of the organisation, as we have seen in Chapter 2. Aligning bounded contexts arround teams which own a certain part of a system and have the autonomy over it is again important for one of our ARs. Therefore, CML has been extended so that aggregates can be assigned to a team. Thereby the team becomes the *owner* of the aggregate. Listing 32 illustrates how an aggregate can be assigned to a team by using the *owner* attribute. A team is declared with a bounded context of the type *TEAM*, a feature which already existed in CML.

```
Aggregate CustomerFrontend {
  owner = CustomerFrontendTeam

  /* the rest of the aggregate has been removed to save space */
}
```

LISTING 32: Aggregate Owner Example

The semantic checkers of the CML language check that the referenced bounded context is of the type *TEAM* and throw an exception if this is not the case.

Structural Volatility

Another important Decomposition Criterion (DC) mentioned in Chapter 2 is the volatility, first introduced by D.L. Parnas [31]. In order to implement our *AR-4: Extract Aggregates by Volatility*, we added an attribute *likelihoodForChange* on aggregate level allowing to specify the volatility. Listing 33 illustrates how it can be used.

```
1 Aggregate CustomerFrontend {
2   likelihoodForChange = OFTEN
3
4   DomainEvent CustomerAddressChange {
5     - UserAccount issuer
6     - Address changedAddress
7   }
8 }
```

LISTING 33: Likelihood for Change (Volatility) on Aggregate

The possible values for the *likelihoodForChange* attribute are *RARELY*, *NORMAL* and *OFTEN*.

After we have summarized the CML language changes realized during this project the next sections will focus on the core topics and new features, namely the ARs and the (micro-)service contract generator.

4.2 Architectural Refactorings (ARs) Design and Concepts

With this project we implemented the seven ARs selected in Chapter 2 as a prototype in the Context Mapper tool [12]. Figure 1.1 in our introduction (Chapter 1) already provided an overview of the selected ARs organized by the decomposition operations *split* and *extract*, and the composition operation *merge*.

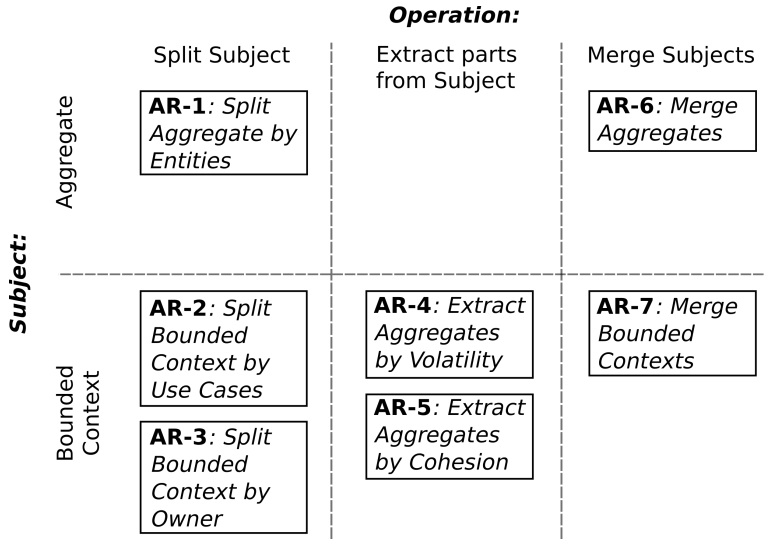

FIGURE 4.2: The implemented ARs organized by their operations *split*, *extract*, *merge* and by the subjects *aggregate* and *bounded context*.

The refactorings support to either *split* a subject, *extract* parts from the subject, or *merge* two subjects, as we have already explained in Chapter 2.

With Figure 4.2 we provide another overview of the implemented ARs, but additionally organized by subject. The two possible subjects are the *aggregate* and the *bounded context*. The subject of an AR is the type of object on which the refactoring is available in the Context Mapper Eclipse plugin [12]. The two ARs *AR-1* and *AR-6* can be applied to an aggregate whereas *AR-2*, *AR-3*, *AR-4*, *AR-5* and *AR-7* are applicable on bounded contexts.

In this section we will explain how the ARs illustrated in Figure 4.2 have been implemented for CML. We will further illustrate their behavior using corresponding examples.

4.2.1 Architectural Refactorings as CML Code Refactorings

On an abstract level the ARs presented in this project describe architectural changes of systems designed on the basis of strategic DDD patterns. Therefore they are basically applicable to every architectural model based on these patterns whether they are written in CML or not. Even if the model is written in CML such ARs can always be applied manually without tool support. However, with this project we provide the presented seven ARs as code refactorings on the CML Domain-specific Language (DSL).

The advantage of this approach is that the refactorings provide operations always leading to a syntactically correct and valid model. The refactorings ensure that all places in the code which are affected by a desired change are adjusted correspondingly. For example if an aggregate which was exposed by one bounded context is moved to another bounded context, relationships on the context map may become invalid. If such a change is done manually the user has to fix subsequent errors in the code manually as well. With the implementation of our DSL refactorings we provide transformations which execute such changes as a whole and in one single step. With this approach we aim for an improved productivity in evolving DDD context maps iteratively.

4.2.2 Refactoring Implementation

The CML DSL is realized with Xtext [14] which generates an Eclipse Modeling Framework (EMF) Ecore [36] model. This generated model corresponds to the DDD meta-model [20] in our case. The implementations of our ARs process this meta-model. The application of one of our ARs to a CML file is an endogenous in-place model transformation [21]. The ARs only change the structure of the modeled systems, without changing the level of abstraction of the models. They are therefore horizontal model transformations [21]. They always proceed in the following three steps:

1. CML Text $\xrightarrow{parsing}$ Abstract Syntax Tree (AST) $\rightarrow$ Model (EMF)
2. Model (EMF) $\xrightarrow{transformation}$ Model (EMF)
3. Model (EMF) $\rightarrow$ Abstract Syntax Tree (AST) $\xrightarrow{unparsing}$ CML Text

The text in the CML file is parsed by the Xtext [14] framework which creates an Abstract Syntax Tree (AST) and provides the model on the basis of the EMF Ecore meta-model [36]. We then apply transformations, in our case our ARs, to this model and it is unparsed back to the CML file. In our previous work [21] we have already described how such transformations can be implemented on the basis of Henshin [38]. However, the transformation in step 2 can also be implemented without additional frameworks by simply manipulating the model via the Application Programming Interface (API) provided by EMF.

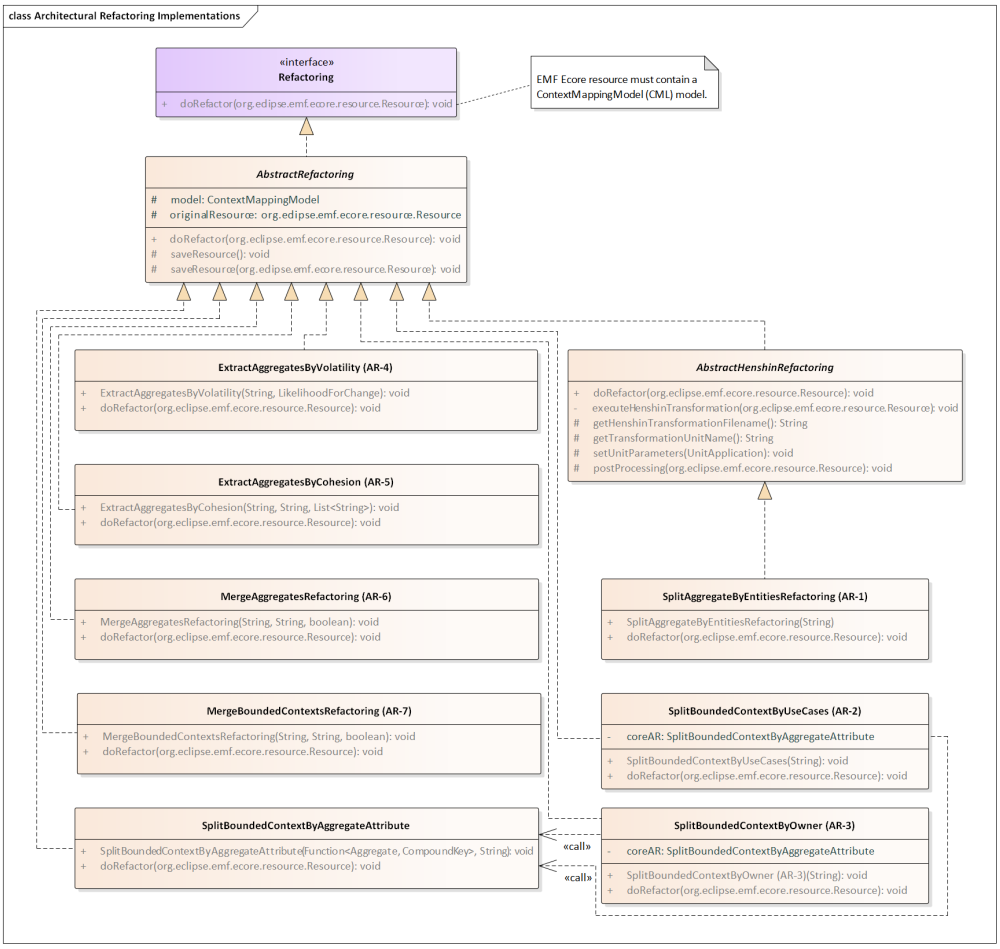

FIGURE 4.3: AR Implementations Logical View

During the implementation of the ARs we had to observe that the description of the transformations with the graph-based approach of Henshin [21, 38] can get complicated with increased complexity in the meta-model of the language and in the transformations. Describing the ARs in such a way is, at least in our case, often way more complex then simply solve the problem in an imperative way using the EMF API. For this reason we finally implemented only one refactoring with the presented approach in [21]. The other ARs of this prototype are simply implemented in Java by using the EMF API [36].

Figure 4.3 shows a logical view [24] of the current implementation of the ARs in Context Mapper. All ARs implement the same interface with the *doRefactor* method which takes an Eclipse EMF Ecore model resource as input. The

given resource must contain a valid CML model on which the refactoring will be applied. The refactorings are configured by constructor parameters. As Figure 4.3 further indicates, only the *AR-1: Split Aggregate by Entities* is implemented with the Henshin approach [21], since its class *SplitAggregateByEntitiesRefactoring* inherits from *AbstractHenshinRefactoring*. The two ARs *AR-2: Split Bounded Context by Use Cases* and *AR-3: Split Bounded Context by Owner* work identical with the aggregate attribute (use case or owner) as the only difference. For this reason, this AR is generalized in the class *SplitBoundedContextByAggregateAttribute*. The two AR classes *SplitBoundedContextByUseCases* and *SplitBoundedContextByOwner* both use this generalized implementation.

The source code of the AR implementations can be found in the package *org.contextmapper.dsl.refactoring* of the main DSL project in our Github repository *context-mapper-dsl*[4]. The following Table 4.4[5] lists the input parameters, preconditions and the corresponding result for all implemented ARs.

TABLE 4.4: Input, Results and Preconditions of the ARs

AR	Input	Preconditions	Result / Output
AR-1: Split Aggregate by Entities	• The name of the aggregate which shall be split.	• The input aggregate must at least contain two entities.	• Multiple aggregates, one for each entity. • All entities become *aggregate roots*.
AR-2: Split Bounded Context by Use Cases	• The name of the bounded context to be split.	• The bounded context must contain at least two aggregates. • The aggregates must be assigned to different use cases.	• This AR creates multiple bounded contexts. Each bounded context contains one or more aggregates which are used by the same use cases.
AR-3: Split Bounded Context by Owner	• The name of the bounded context to be split.	• The bounded context must contain at least two aggregates. • The aggregates must be assigned to different teams.	• This AR creates multiple bounded contexts. Each bounded context contains one or more aggregates which are owned by the same team.

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

[5] A table listing the input and output of all ARs can also be found in our online documentation: `https://contextmapper.github.io/docs/architectural-refactorings/`

TABLE 4.4: Input, Results and Preconditions of the ARs (continued)

AR	Input	Preconditions	Result / Output
AR-4: Extract Aggregates by Volatility	• The name of the bounded context from which Aggregates shall be extracted. • The *volatilty* or *likelihood for change* by which shall be extracted.	• The selected bounded context must contain at least two aggregates. • The aggregates must have different *likelihoods for change (volatility)*.	• This AR creates another bounded context containing all the aggregates with the chosen volatility.
AR-5: Extract Aggregates by Cohesion	• The name of the bounded context from which the aggregates shall be extracted. • The aggregate selection to extract.	• The selected bounded context must at least contain two aggregates.	• A new bounded context which contains all the selected aggregates.
AR-6: Merge Aggregates	• The names of two aggregates which belong to the same bounded context.	• The model must contain two aggregates within one bounded context which can be merged. • The two aggregates are not allowed to contain domain objects with the same name.	• One aggregate which contains all objects of the two input aggregates.
AR-7: Merge Bounded Contexts	• The names of two bounded contexts which shall be merged.	• The model must at least contain two bounded contexts.	• The result of this AR is one bounded context containing all aggregates of the two input bounded contexts.

4.2.3 User Interface: Refactorings within the CML Editor

The ARs are integrated within the Context Mapper [12] CML Eclipse editor. Thereby, the ARs can be used as any other refactorings for a language such as Java within the Integrated Development Environment (IDE). As Figure 4.4

illustrates, the refactorings are available within the CML editor by using the context menu on the two subjects *Bounded Context* or *Aggregate*. The context menu always checks for the preconditions of the ARs and offers only refactorings which can be applied on the selected object.

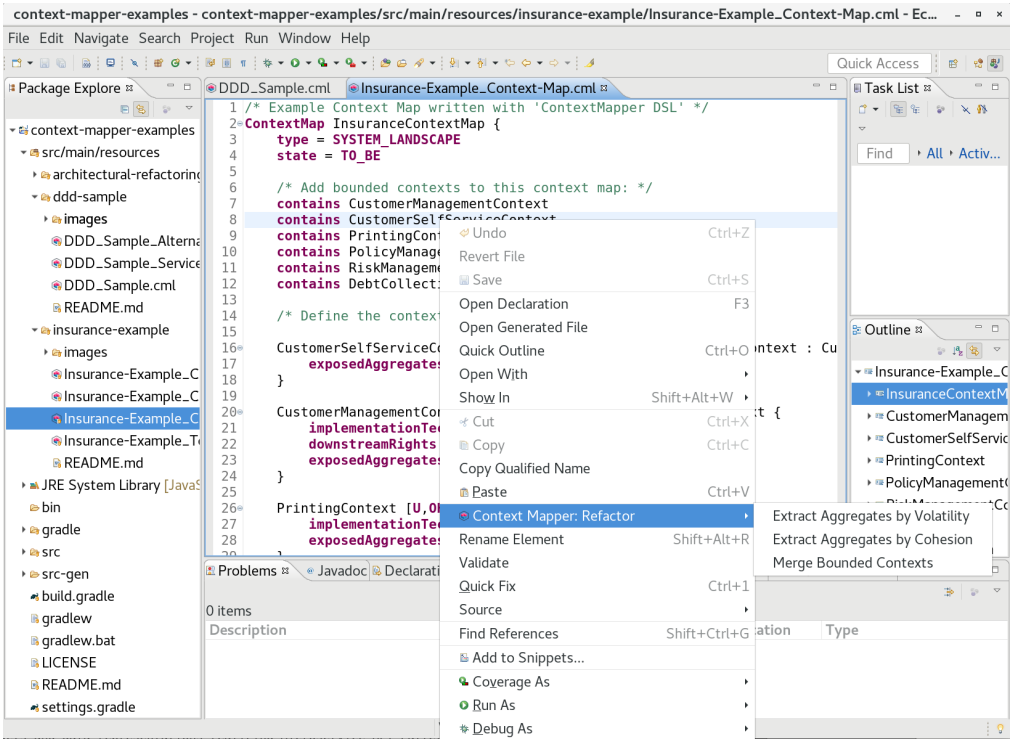

FIGURE 4.4: ARs in Context Mapper Eclipse Plugin

4.2.4 Examples

In the following we explain how the ARs work on the basis of simple CML examples. The listings only contain the necessary parts to illustrate the behavior of the ARs. The complete examples including all parts of the model can be found in Appendix A.

Please note that the following CML listings contain syntax and language concepts which are not explained within this paper. For a language reference and complete documentation we refer to the Context Mapper website[6], Appendix B, or the report of our previous project [20]. All tactic DDD concepts within the aggregates (entities, value objects, services, etc.) are based on the Sculptor DSL [34]. For an introduction to the Sculptor syntax we recommend to consult their online documentation[7] as well.

AR-1: Split Aggregate by Entities

Listing 34 illustrates a simplified CML example with an aggregate containing two entities.

[6] `https://contextmapper.github.io/docs`
[7] `http://sculptorgenerator.org/documentation/`

```
BoundedContext CustomerManagementContext {
  Aggregate Customers {
    Entity Customer {
      aggregateRoot

      String firstname
      String lastname
    }
    Entity Address {
      String street
      String city
    }
  }
}
```

LISTING 34: AR-1: Example Input

If an aggregate contains more then one entities, *Split Aggregate by Entities* can be applied. The AR will generate one aggregate per entity. Applied to the aggregate *Customers* shown in Listing 34, *Split Aggregate by Entities* produces the following result illustrated by Listing 35.

```
BoundedContext CustomerManagementContext {
  Aggregate Customers {
    Entity Address {
      aggregateRoot

      String street
      String city
    }
  }
  Aggregate NewAggregate1 {
    Entity Customer {
      aggregateRoot

      String firstname
      String lastname
    }
  }
}
```

LISTING 35: AR-1: Example Output

The new aggregates which are created by the AR are named *NewAggregateX* where *X* is an incremented number starting at 1. Note that all entities become *aggregate roots* within their aggregate.

AR-2: Split Bounded Context by Use Cases

As already explained earlier in this chapter, the current CML version allows to assign use cases to aggregates. Listing 36 shows how this can be done and is further an example on which *Split Bounded Context by Use Cases* can be applied.

```
BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  Aggregate Offers {
    useCases = CreateOffer4Customer

    Entity Offer {
      aggregateRoot

      int offerId
      /* ... */
    }
  }
  Aggregate Products {
    useCases = CreateOffer4Customer

    Entity Product {
      aggregateRoot

      - ProductId identifier
      String productName
    }
    /* ... */
  }
  Aggregate Contract {
    useCases = UpdateContract

    Entity Contract {
      aggregateRoot

      - ContractId identifier
      /* ... */
    }
    /* ... */
  }
}

UseCase UpdateContract
UseCase CreateOffer4Customer
```

LISTING 36: AR-2: Example Input

Note that it would also be possible to assign multiple use cases to one aggregate (comma-separated). In the example above we have one bounded context with three aggregates. The first two aggregates *Offers* and *Products* are used by the same use case *CreateOffer4Customer*. The third aggregate is assigned another use case *UpdateContract*. The bounded context contains aggregates used by two use cases in total. Therefore, applying *Split Bounded Context by Use Cases* will create two bounded contexts. One bounded context for every use case. The listings 37 and 38 illustrate the result after the application of this AR to the example shown in Listing 36 above.

```
BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  Aggregate Contract {
    useCases = UpdateContract

    Entity Contract {
      aggregateRoot

      - ContractId identifier
```

LISTING 37: AR-2: Example Output (1)

```
      /* ... */
    }
    /* ... */
  }
}

BoundedContext NewBoundedContext1 {
  Aggregate Offers {
    useCases = CreateOffer4Customer

    Entity Offer {
      aggregateRoot

      int offerId
      /* ... */
    }
  }
  Aggregate Products {
    useCases = CreateOffer4Customer

    Entity Product {
      aggregateRoot

      - ProductId identifier
      String productName
    }
    /* ... */
  }
}
```

LISTING 38: AR-2: Example Output (2)

The AR leaves one use case within the existing bounded context and creates new bounded contexts for all other use cases. In the result above, the *Contract* aggregate which is used by the *UpdateContract* use case has been left in the existing *PolicyManagementContext*. The other use case *CreateOffer4Customer* with the two aggregates *Offers* and *Products* has been moved to a new bounded context *NewBoundedContext1*.

The goal of this refactoring is to create one bounded context for every use case. However, if aggregates are used by multiple use cases, we use the complete set of use cases and create one bounded context for each unique set. For example, if there are two aggregates which are both used by *Use Case X* and *Use Case Y*, both aggregates will be within the same bounded context. If there is a third aggregate which is only used by *Use Case X* but not by *Use Case Y*, we create a separate bounded context for this case. In other words, only if aggregates are assigned to the very same set of use cases, they will end in the same bounded context. Different sets of use cases lead to different bounded contexts.

AR-3: Split Bounded Context by Owner

Split Bounded Context by Owner works very similar to *Split Bounded Context by Use Cases* but uses the *owner* attribute instead of the *useCases* attribute. This CML language feature has already been introduced in Section 4.1.5. The following Listing 39 shows how aggregates can be assigned to their owner and represent an example input for the *Split Bounded Context by Owner* AR.

```
BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  Aggregate CustomerFrontend {
    owner = CustomerFrontendTeam

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    owner = CustomerBackendTeam

    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}
```

LISTING 39: AR-3: Example Input

This AR is slightly simpler than *Split Bounded Context by Use Cases* since an aggregate can only have one owner and not multiple. The owners referenced in the example above must be specified as bounded contexts of the type *TEAM* as shown in the following Listings 40.

```
/* team definitions */
BoundedContext CustomerBackendTeam {
  type = TEAM
}
BoundedContext CustomerFrontendTeam {
  type = TEAM
}
```

LISTING 40: AR-3: Owner Specifications

Applying *Split Bounded Context by Owner* will create one bounded context for each owner. In the example input above we have one bounded context *CustomerSelfServiceContext* which contains two aggregates owned by two different teams. Applying this AR will create two bounded contexts as shown in the following listings 41 and 42.

```
BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  Aggregate CustomerFrontend {
    owner = CustomerFrontendTeam

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
```

LISTING 41: AR-3: Example Output (1)

```
    }
  }
}

BoundedContext NewBoundedContext1 {
  Aggregate Acounts {
    owner = CustomerBackendTeam

    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}
```

LISTING 42: AR-3: Example Output (2)

As in the previous AR, the existing bounded context is reused for one of the teams. For all the other teams new bounded contexts are created. In the example above, a new bounded context *NewBoundedContext1* has been created for all aggregates of the team *CustomerBackendTeam* while all aggregates of the team *CustomerFrontendTeam* remained in the existing bounded context.

AR-4: Extract Aggregates by Volatility

We have already introduced the *volatility* attribute *likelihoodForChange* added to CML in Section 4.1.5. The AR *Extract Aggregates by Volatility* uses this attribute to extract aggregates with a specific *likelihood for change*. The following Listing 43 shows an example how this language feature can be used and illustrates a situation where *Extract Aggregates by Volatility* can be applied.

```
BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  Aggregate CustomerFrontend {
    likelihoodForChange = OFTEN

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}
```

LISTING 43: AR-4: Example Input

Note that a bounded context on which this AR shall be applied must contain at least two aggregates with different *likelihoodForChange* values. In the example

above only the *CustomerFrontend* has the value set explicitely to *OFTEN*. The default value for the *likelihoodForChange* attribute, and therefore the value for the *Accounts* aggregate above, is *NORMAL*. The preconditions to apply this AR are therefore fulfilled. The Listing 44 illustrates the result after applying the refactoring with the value *OFTEN* as input.

```
BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}

BoundedContext CustomerSelfServiceContext_Volatility_OFTEN {
  Aggregate CustomerFrontend {
    likelihoodForChange = OFTEN

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
}
```

LISTING 44: AR-4: Example Output

In this case all aggregates with the *likelihoodForChange* value *OFTEN* are extracted into a new bounded context. The existing bounded context in this example has the name *CustomerSelfServiceContext* from which the name for the new bounded context, *CustomerSelfServiceContext_Volatility_OFTEN*, is derived.

AR-5: Extract Aggregates by Cohesion

The goal of the AR *Extract Aggregates by Cohesion* is to extract specific aggregates into new bounded contexts in order to improve the cohesion inside the contexts. Since we currently can not calculate or score the cohesion automatically, the implementation of this AR allows the architect to choose the aggregates manually. Thereby the user can extract aggregates on the basis of any Non-Functional Requirement (NFR) or criteria which influences the cohesion. The listings 45 and 46 illustrate an example of a bounded context with multiple aggregates on which this AR can be applied.

```
BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  Aggregate Offers {
    Entity Offer {
      aggregateRoot
```

LISTING 45: AR-5: Example Input (1)

```
      int offerId
      /* ... */
    }
  }
  Aggregate Products {
    Entity Product {
      aggregateRoot
      - ProductId identifier
      String productName
    }
    /* ... */
  }
  Aggregate Contract {
    Entity Contract {
      aggregateRoot

      - ContractId identifier
      /* ... */
    }
    /* ... */
  }
}
```

LISTING 46: AR-5: Example Input (2)

The bounded context in the example above contains three aggregates. By applying the AR *Extract Aggregates by Cohesion* we can extract a specific set of these aggregates and move them into a new bounded context. The following Listing 47 shows the result of the application using the aggregate name *Offers* as input for the AR.

```
BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  Aggregate Products {
    Entity Product {
      aggregateRoot
      - ProductId identifier
      String productName
    }
    /* ... */
  }
  Aggregate Contract {
    Entity Contract {
      aggregateRoot

      - ContractId identifier
      /* ... */
    }
    /* ... */
  }
}
BoundedContext SalesBoundedContext {
  Aggregate Offers {
    Entity Offer {
      aggregateRoot

      int offerId
      /* ... */
    }
  }
}
```

LISTING 47: AR-5: Example Output

The AR creates a new bounded context with the *Offers* aggregate. The other aggregates remain in the already existing bounded context. Since this AR requires external user input, a corresponding dialog appears before the AR is applied. The dialog for this case is shown in Figure 4.5.

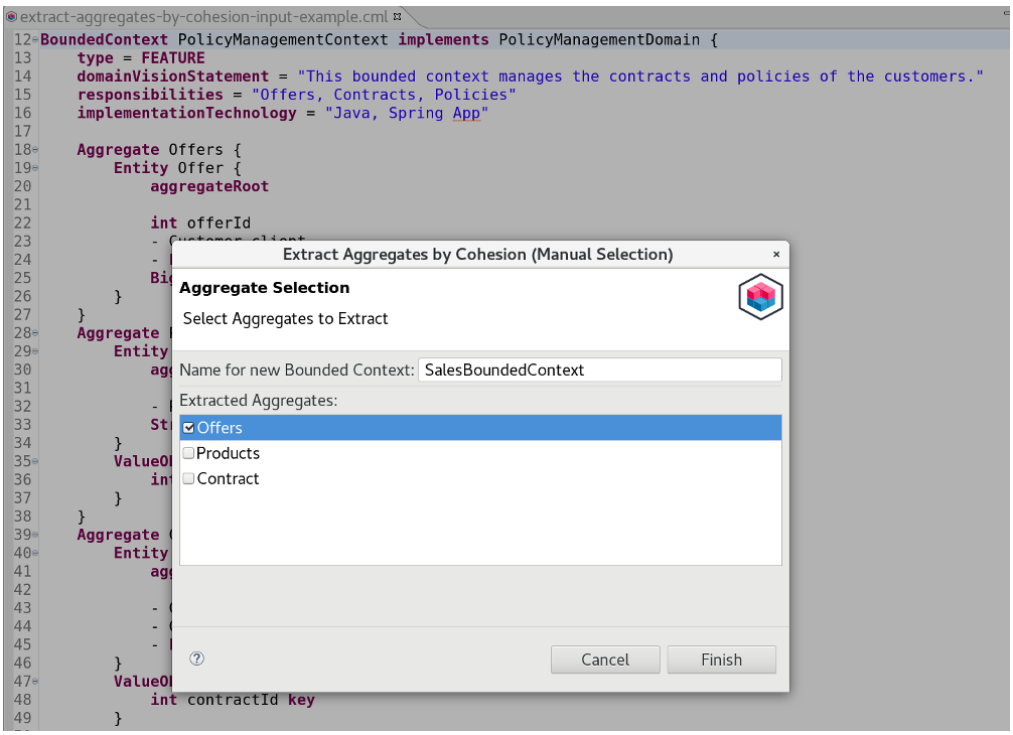

FIGURE 4.5: AR-5: Dialog for External User Input

The dialog further allows the user to specify how the new bounded context shall be named.

AR-6: Merge Aggregates

The *Merge Aggregates* AR allows to merge two aggregates within a bounded context. The listings 48 and 49 show a bounded context in CML which contains more than one aggregates.

```
1  BoundedContext CustomerManagementContext implements CustomerManagementDomain {
2    Aggregate Customers {
3      Entity Customer {
4        aggregateRoot
5
6        String firstname
7        String lastname
8        /* ... */
9      }
10     /* ... */
11   }
```

LISTING 48: AR-6: Example Input (1)

```
12    Aggregate Addresses {
13      Entity Address {
14        String street
15        int postalCode
16        /* ... */
17      }
18    }
19  }
```

LISTING 49: AR-6: Example Input (2)

Applying *Merge Aggregates* with the two aggregate names *Customers* and *Addresses* as input parameters on the example above results in the CML model shown in Listing 50.

```
1   BoundedContext CustomerManagementContext implements CustomerManagementDomain {
2     Aggregate Customers {
3       Entity Customer {
4         aggregateRoot
5
6         String firstname
7         String lastname
8         /* ... */
9       }
10
11      Entity Address {
12        String street
13        int postalCode
14        /* ... */
15      }
16
17      /* ... */
18    }
19  }
```

LISTING 50: AR-6: Example Output

Note that this AR merges the elements of the second aggregate (second parameter) into the first aggregate given. In the example above we passed *Customers* as the first parameter and *Addresses* as the second. Therefore all objects within the *Addresses* aggregate are merged into *Customers*. In the Context Mapper tool the user is able to switch this behavior on the User Interface (UI) in order to use the second aggregate as the target if needed. However, all aggregate attributes such as the name are always taken from this target aggregate (first parameter by default).

AR-7: Merge Bounded Contexts

Merge Bounded Contexts works similar as *Merge Aggregates* but merges bounded contexts instead of aggregates. To apply this AR at least two bounded contexts are needed. Listing 51 illustrates an example on which we can apply *Merge Bounded Contexts*.

```
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  Aggregate Customers {
    Entity Customer {
      aggregateRoot

      String firstname
      String lastname
      /* ... */
    }
    /* ... */
  }
}

BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  Aggregate CustomerFrontend {
    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}
```

LISTING 51: AR-7: Example Input

As with the *Merge Aggregates* AR the *Merge Bounded Contexts* has two name parameters and the first bounded context (first parameter) will be the target bounded context. All aggregates of the second bounded context will be merged into the first bounded context. The listings 52 and 53 show the result of this AR applied to the example above with the parameters *CustomerManagementContext* and *CustomerSelfServiceContext*.

```
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  Aggregate Customers {
    Entity Customer {
      aggregateRoot

      String firstname
      String lastname
      /* ... */
    }
    /* ... */
  }
  Aggregate CustomerFrontend {
    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
```

LISTING 52: AR-7: Example Output (1)

```
20    Aggregate Acounts {
21      Entity UserAccount {
22        aggregateRoot
23
24        String username
25        - Customer accountCustomer
26      }
27    }
28  }
```

LISTING 53: AR-7: Example Output (2)

As the listings above illustrate, all three aggregates of the originally two bounded contexts are part of the remaining merged bounded context after the application of *Merge Bounded Contexts*.

4.2.5 Context Map Consistency

As already mentioned previously, the main benefit of implementing the ARs as code refactorings on the basis of our DSL is that the model always stays in a valid state. This means that the implementations of the refactorings must adjust places which are somehow affected by a desired change. For our refactorings this mainly affects the context map.

If a refactoring for example deletes a bounded context (e.g. in *Merge Bounded Contexts*) the implementation has to adjust all relationships on the context map in which the deleted bounded context has been a participant. The deletion or movement of an aggregate has similar implications since relationships on context maps in CML specify which upstream aggregates are exposed in a relationship (see Section 4.1.5). In the following we summarize the applied heuristics to resolve these situations.

Extracted Aggregates

The following four ARs may move one or multiple aggregates into a new bounded context:

- 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

The two ARs above which *split* a bounded context always leave some aggregates in the existing bounded context, but also move one or more into a new bounded context. The two *extract* ARs extract specific aggregates and create new bounded contexts for them as already explained in the examples above.

The context map of a model is getting invalid if an aggregate, which is exposed in a relationship, is moved into a new bounded context. To explain this situation with an example we use the simple context map in Listing 54 with only one relationship between two contexts.

```
ContextMap InsuranceContextMap {
  contains CustomerManagementContext, PolicyManagementContext

  PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerManagementContext {
    implementationTechnology = "RESTful HTTP"
    exposedAggregates = Customers, CustomerFrontend
  }

}
```

LISTING 54: Simple context map with upstream-downstream relationship exposing aggregates

From the listing above we know that the two aggregates *Customers* and *CustomerFrontend* are part of and exposed by the *CustomerManagementContext*. The *PolicyManagementContext* uses the Open Host Service (OHS) which exposes the two aggregates. We assume now that the user splits the *CustomerManagementContext* with one of our ARs and thereby moves the *CustomerFrontend* aggregate to a new bounded context. He could also extract the same aggregate to a new bounded context, which leads to the same situation.

In this case, the context map in Listing 54 will no longer compile since *CustomerFrontend* is not found anymore within the *CustomerManagementContext*. Our ARs always apply the same heuristic in this case to keep the context map valid.

First, we **remove the moved aggregate from the existing relationship (1)**. This would make the context map already compile again. However, semantically this would imply that the *PolicyManagementContext* no longer uses the *CustomerFrontend* aggregate. Since we can not presume that this is the case, we **create a new relationship between the downstreams and the new bounded context (2)** in a second step. Thereby we ensure that all downstream bounded contexts which had access to the aggregate, still have this access.

Asuming the *CustomerFrontend* aggregate is moved to a new bounded context called *CustomerSelfServiceContext*, our ARs adjust the context map as explained above, which leads to the result illustrated in Listing 55.

```
ContextMap InsuranceContextMap {
  contains CustomerManagementContext, PolicyManagementContext
  contains CustomerSelfServiceContext

  PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerManagementContext {
    implementationTechnology = "RESTful HTTP"
    exposedAggregates = Customers
  }

  CustomerSelfServiceContext [U,OHS,PL]->[D,CF] PolicyManagementContext {
    implementationTechnology = "RESTful HTTP"
    exposedAggregates = CustomerFrontend
  }
}
```

LISTING 55: New context map after moving *CustomerFrontent* by applying an AR

The result is a new relationship on the context map with the *PolicyManagementContext* as downstream and the new *CustomerSelfServiceContext* as upstream, exposing *CustomerFrontend*. The relationship attributes such as the roles (OHS, PL, etc.) or the implementation technology are taken from the previously existing relationship.

With this strategy our ARs always produce a valid context map in case an aggregate which has been exposed is moved to a new bounded context. The user can adjust the new relationship after applying the AR in case the generated one does not express the desired state.

Splitted Aggregates

Another issue to be respected in context maps are splitted aggregates which are exposed to other bounded contexts. Even though the splitting of an aggregate does not produce an invalid context map, it seams unreasonable in many cases to leave the original aggregate the only exposed aggregate. Downstream contexts which used the exposed aggregate may need both resulting aggregates after the application of such a refactoring. This issue affects the following AR:

- AR-1: Split Aggregate by Entities

The refactoring solves this issue by adding the new aggregate which is created due to the splitting operation to the *exposed aggregates* of all relationships which previously exposed the original aggregate.

```
1 ContextMap InsuranceContextMap {
2   contains CustomerManagementContext, PolicyManagementContext
3
4   PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerManagementContext {
5     implementationTechnology = "RESTful HTTP"
6     exposedAggregates = Customers
7   }
8
9 }
```

LISTING 56: Simple context map with upstream-downstream relationship exposing one aggregate

Assuming we have a context map such as the one in Listing 56 and we split the *Customers* aggregate into two aggregates, the result would be as illustrated in Listing 57.

```
1 ContextMap InsuranceContextMap {
2   contains CustomerManagementContext, PolicyManagementContext
3
4   PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerManagementContext {
5     implementationTechnology = "RESTful HTTP"
6     exposedAggregates = Customers, NewAggregate1
7   }
8
9 }
```

LISTING 57: Example context map after splitting aggregate

Merged Aggregates

Merging two aggregates implies that one of the two original aggregates will no longer exist. This can cause invalid context maps in case the corresponding aggregate was exposed in a relationship. To solve this issue the *Merge Aggregates* AR must ensure that if one of the two affected aggregates is part of a relationship, the resulting exposed aggregate corresponds to the final aggregate of the merge operation.

For example if a relationship exposes the aggregate *Addresses*, it can be the case that this aggregate is merged into the aggregate *Customers*. In this case the final and resulting aggregate is *Customers*. The aggregate *Addresses* will no longer exist. Therefore our AR *Merge Aggregates* will change the context map and replace the exposed aggregate *Addresses* with *Customers*.

Merged Bounded Contexts

The last AR which needs adjustments on the context map is *Merge Bounded Contexts*. Merging two contexts basically deletes one of the two corresponding contexts. The context which is deleted can naturally be part of relationships on the context map. In this case our refactoring adjusts the context map with two steps. First, we **delete direct relationships between the two contexts (1)**. Since the two contexts are merged into one, the relationships between the two become needless. In a second step, we **replace the deleted bounded context in all relationships with the remaining bounded context (2)**.

Listing 58 illustrates an example to explain this behavior. The context map contains four bounded contexts and three relationships. Note that the example is constructed to illustrate the AR behavior at this point and that the relationships between the contexts might be questionable.

```
1  ContextMap InsuranceContextMap {
2    contains CustomerManagementContext, CustomerSelfServiceContext
3    contains PolicyManagementContext, PrintingContext
4
5    CustomerSelfServiceContext [D,C]<-[U,S] CustomerManagementContext {
6      exposedAggregates = Customers
7    }
8
9    CustomerSelfServiceContext [D,ACL]<-[U,OHS,PL] PrintingContext {
10     implementationTechnology = "SOAP"
11     downstreamRights = INFLUENCER
12     exposedAggregates = Printing
13   }
14
15   PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerSelfServiceContext {
16     implementationTechnology = "RESTful HTTP"
17     exposedAggregates = CustomerFrontend
18   }
19 }
```

LISTING 58: Context map merging contexts example

If we now apply *Merge Bounded Contexts* to the two contexts *CustomerManagementContext* and *CustomerSelfServiceContext* of the give example, the first direct relationship between the two contexts will disappear. We further assume that

we merge *CustomerSelfServiceContext* into *CustomerManagementContext* and not vice versa. Thus, we basically delete the *CustomerSelfServiceContext* and the remaining context will be the *CustomerManagementContext*. Since the second and third relationship in the example incorporate *CustomerSelfServiceContext* as a member, they have to be adjusted to preserve the valid context map. Our AR will solve this issue by replacing the occurrences of *CustomerSelfServiceContext* with *CustomerManagementContext* in all relationships. Therefore, we finally get the valid context map illustrated by Listing 59.

```
4  ContextMap InsuranceContextMap {
5    contains CustomerManagementContext
6    contains PolicyManagementContext, PrintingContext
7
8    CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext {
9      implementationTechnology = "SOAP"
10     exposedAggregates = Printing
11   }
12
13   PolicyManagementContext [D,CF]<-[U,OHS,PL] CustomerManagementContext {
14     implementationTechnology = "RESTful HTTP"
15     exposedAggregates = CustomerFrontend
16   }
17
18 }
```

LISTING 59: Context map after merging the bounded contexts

We have now presented all implementation details of the ARs developed in this project. In the next section we will finish the part about CML and the ARs by list a number of known limitations and issues which could not be solved during the project.

4.2.6 Known Limitations

During the implementation of the ARs we came accross a few technical issues, especially regarding the *unparsing* process. The unparsing converts the transformed Ecore model, after the application of the refactoring, back into the textual DSL form. Concerning this matter the tool currently has some limitations regarding formatting and the order in which the root elements are unparsed.

Formatting

In general, the Xtext framework unparses only the changed parts of the model. This means that the convertion from the runtime model into the DSL text is only done for the parts which have changed. The advantage of this approach is that no text changes if there were no changes in that part of the model. However, due to an Xtext framework issue[8] we run into cases where new elements (such as newly added bounded contexts) are added at the end of the file which may lead to a result which cannot be parsed because it does not correspond to the grammar.

[8] https://bugs.eclipse.org/bugs/show_bug.cgi?id=369175

This has to do with the fact that the root elements of our grammar must occur block-wise per type. Root elements in a CML file are the following:

- Context Map (only one per file/model)
- Bounded Contexts
- Domains
- Use Cases

The order in which these types are written in a CML file does not matter, but each type must be written as one block. For example, all bounded contexts must be written in one block. It is not allowed to write a bounded context followed by a domain followed by a bounded context. Whether the file contains a block with bounded contexts first, followed by a block of domains or vice versa, does on the other hand not matter as long as all elements of one type are written in one block.

Since we add new bounded contexts in some of our ARs, the framework issue mentioned above led to situations where bounded contexts were added at the end of file, even though the bounded context block was not the last block in the file. Therefore, the resulting file was no longer parsable. To solve this issue we implemented the workaround[9] proposed by a member of the Eclipse community. This workaround ensures the correct order of the root elements.

However, the workaround comes with a big disadvantage. The CML file is unparsed completely every time, even if only small parts of the model have changed. Unparsing also means that the CML code is brought into the formatting as implemented by our code formatters. If the user used another formatting, this differences get lost once one applies an AR. This leads to the limitation that user-specific formatting cannot be preserved when working with the refactorings.

Root Element Order

Besides the formatting which is always aligned according to the predefined formatters, the order of the root elements is another issue. Due to the workaround mentioned in the *Formatting* section above, the unparsing mechanism always uses the default order according to the grammar, which is *Context Map*, *Bounded Contexts*, *Domains*, *Use Cases*. If the user uses another order, which is generally possible, the application of an AR reorders all the elements according to this default order.

To reduce the inconvenience caused by this limitation we recommend to always use the default order. If this order is used upfront, the application of an AR does not reorder the elements in the CML file.

Scoping & Multiple CML Files

In the current version of Context Mapper [12] we have not yet handled the scoping issues which Xtext brings by default if the model is spread over multiple files. If you create multiple CML files within your project the DSL editor

[9] `https://www.eclipse.org/forums/index.php/t/1080047/`

allows you to reference objects in other files and lists them in the autocompletion mechanism. However, if you use a file as input resource for generators or if you apply ARs the Xtext API does not resolve references to objects in other files. This is a limitation to be solved in the future since the editor allows to create references which cannot be processed by generators later (at least not with the default Xtext setup). To solve this properly, an import mechanism is needed which allows the user to import other files explicitely. References to objects which are not part of the imported files should no longer be possbible then. For the moment we recommend to strictly model one context map with all its bounded contexts within one single CML file.

To finish the implementation chapter the next section will present the "proof of concept" of our service contract generator, another topic covered by this project. However, it is somehow a side project included in this work but not part of the main topic which is service decomposition with ARs.

4.3 Service Contract Generation

With the Microservices Domain-Specific Language (MDSL)[10] [42] generator we provide a tool to produce service contracts which can be used as assistance towards implementing the system modeled in CML in an (micro-)service-oriented architecture. The generator produces contracts for all upstream-downstream relationships modeled in your context map. The produced contracts describe the APIs which should be provided by the corresponding upstream bounded contexts. MDSL is a DSL to specify (micro-)service contracts and their data representations. The language concepts of MDSL use and support the Microservice API Patterns[11] [44]. The current implementation of the generator is compatible with MDSL in the version *v1.0*. Note that we will not introduce details about MDSL and its syntax here. For details about the language we refer to the online documentation[10].

4.3.1 Preconditions

The current implementation of the MDSL generator provides a first "proof of concept" and therefore not all MDSL features are used and generated. In addition, some preconditions have to be fullfilled by the model so that a service contract is generated.

- The CML relationship must be an upstream-downstream relationship.
- The relationship declaration must declare the exposed aggregates of the upstream (see *exposedAggregates* attribute explained in Section 4.1.5).
- The exposed aggregates must declare an *aggregate root* entity.
- The aggregate must declare at least one method/operation either on the root entity or within a service.

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

[11] `https://microservice-api-patterns.org/`

4.3.2 Data Mapping

The generator produces (micro-)service contracts according to the following mapping definitions, which reflect our proposal how we would derive services from models based on strategic DDD.

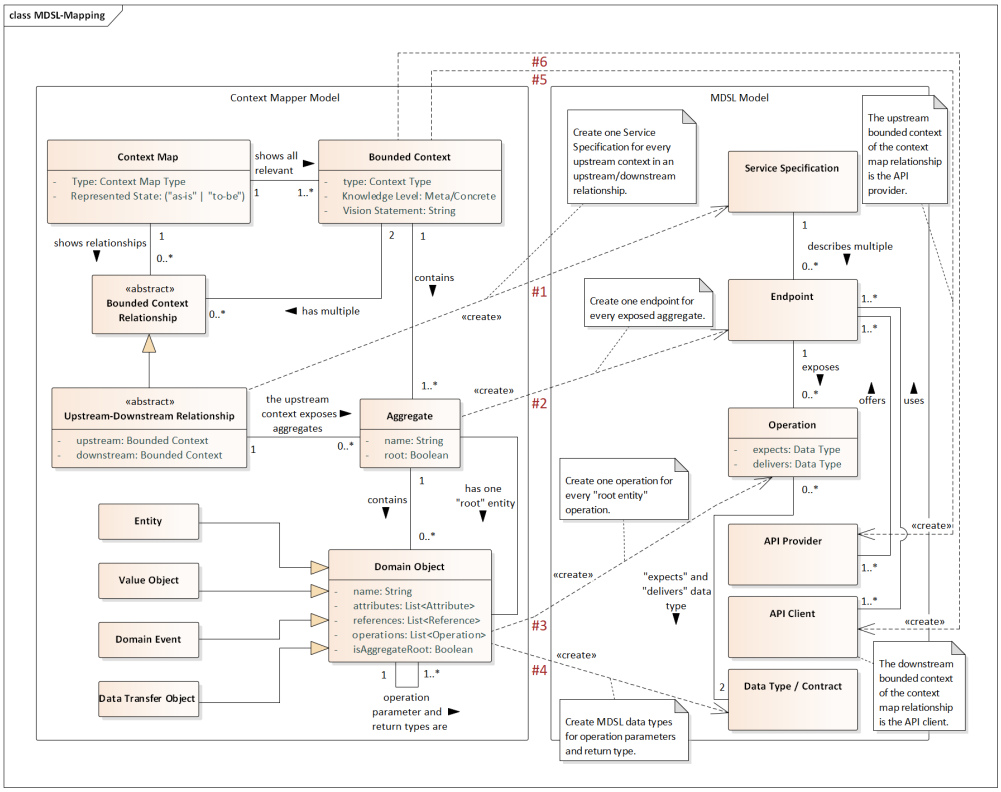

FIGURE 4.6: CML to MDSL Mapping

Figure 4.6 illustrates how we generate these contracts and how the Context Mapper model is mapped to the MDSL language. The mappings between the two models are illustrated with the dashed lines («create»). In addition, each mapping in Figure 4.6 is numbered and explained in Table 4.5. We generate one API description for every upstream bounded context (#1). The API contains an endpoint for every exposed aggregate (#2). The endpoint itself contains operations which are derived by the aggregate root entity operations and services in CML (#3). The payload data types are derived by the parameters and return types of these methods in CML (#4). We further generate an API provider for the upstream context (#5) and an API client for the downstream context (#6).

TABLE 4.5: CML to MDSL Mapping Table

#	CML input	MDSL output	Description
1	Upstream bounded contexts from upstream-downstream relationships	Service specification (API description)	We create one service for each upstream bounded context from our context map which exposes parts of its model to others.

TABLE 4.5: CML to MDSL Mapping Table (continued)

#	CML input	MDSL output	Description
2	Exposed aggregates	Service endpoint	Every exposed aggregate of the upstream bounded context results in an endpoint of the corresponding service.
3	Public methods/operations of the aggregate root entity or services	Operation	The exposed aggregates should contain methods/operations either on the aggregate root entity or in corresponding services. For every method/operation in those CML objects an operation in MDSL is generated.
4	Parameters & return types of methods	Base types or data type specifications if possible (using parameter trees in MDSL).	If primitive types in CML are used, they are mapped to corresponding primitive types of MDSL. References to other objects (such as entities) in CML lead to the generation of a corresponding parameter tree. Types which are not further declared are mapped to abstract, unspecified elements ("P" keyword; see MDSL documentation).
5	Upstream bounded contexts from upstream-downstream relationships	API provider	For the upstream bounded context we further generate an API provider.
6	Downstream bounded contexts from upstream-downstream relationships	API client	Downstream bounded contexts are mapped to corresponding API clients.

For further details regarding the MDSL generator, such as the mapping of the primitive data types, we refer to our Context Mapper [12] online documentation[12].

4.3.3 MDSL Example

Let us illustrate the concepts and the mapping explained above with an example. We use the insurance example scenario introduced in our previous work [20] which can also be found in our examples repository[13] online. Figure 4.7

[12] `https://contextmapper.github.io/docs/mdsl/`

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

illustrates the context map of the example in a graphical form inspired by Brandolini [4] and Vernon [41]. The CML code of the whole example can be found online.

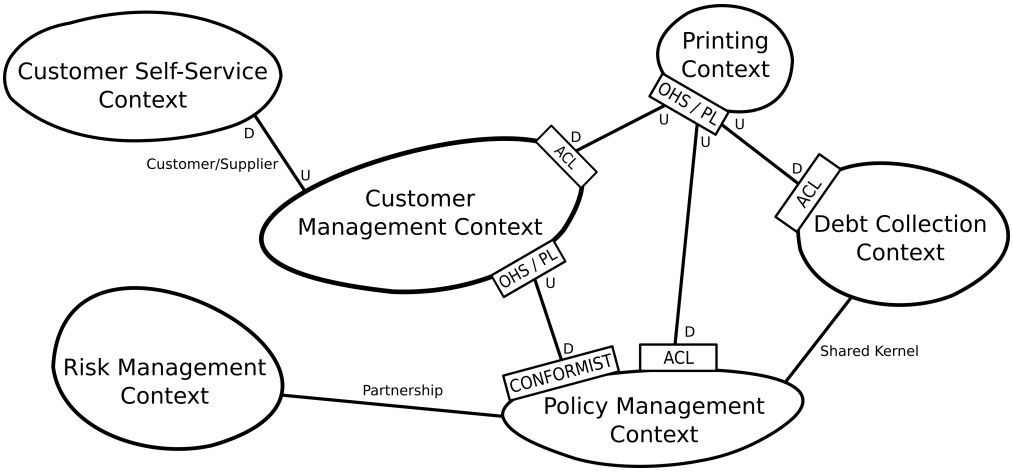

FIGURE 4.7: Context Mapper [12] "Insurance" Example

Since the generator only considers upstream-downstream relationships, the partnerships and shared kernels can be ignored. In the remaining relationships in Figure 4.7 we have two bounded contexts which are upstream, namely *Customer Management Context* and *Printing Context*. The former is used by the downstreams *Policy Management Context* and *Customer Self-Service Context* while the *Printing Context* is used by *Customer Management Context*, *Dept Collection Context* and *Policy Management Context*.

The MDSL generator will produce two API descriptions and thus, two MDSL files. The listings 60 and 61 show the MDSL file for the *Customer Management Context*.

```
API description CustomerManagementContextAPI

data type Address { "street":V<string>, "postalCode":V<int>, "city":V<string> }
data type AddressId P
data type changeCustomerParameter { "firstname":V<string>, "lastname":V<string> }

endpoint type CustomersAggregate
  exposes
    operation createAddress
      expecting
        payload Address
      delivering
        payload AddressId
    operation changeCustomer
      expecting
        payload changeCustomerParameter

API provider CustomerManagementContextProvider
  offers CustomersAggregate
  at endpoint location "http://localhost:8001"
    via protocol "RESTful HTTP"
```

LISTING 60: MDSL example: "Customer Management" API (1)

```
API client PolicyManagementContextClient
  consumes CustomersAggregate
API client CustomerSelfServiceContextClient
  consumes CustomersAggregate

IPA
```

LISTING 61: MDSL example: "Customer Management" API (2)

The *Customer Management Context* contains one aggregate *Customers* for which the generator created the endpoint *CustomersAggregate*. The aggregate contains methods and services which were mapped to operations and corresponding data types. As illustrated in the listings 60 and 61 we generate one API provider for the upstream and two API clients for the corresponding downstreams. Listing 62 shows another MDSL example file for the *Printing Context*.

```
API description PrintingContextAPI

data type PrintStatus P
data type PrintindId P

endpoint type PrintingAggregate
  exposes
    operation printJob
      expecting
        payload PrintindId
      delivering
        payload PrintStatus

API provider PrintingContextProvider
  offers PrintingAggregate
  at endpoint location "http://localhost:8000"
    via protocol "SOAP"

API client CustomerManagementContextClient
  consumes PrintingAggregate
API client DebtCollectionClient
  consumes PrintingAggregate
API client PolicyManagementContextClient
  consumes PrintingAggregate

IPA
```

LISTING 62: MDSL example: "Printing" API

With the MDSL generator all implementation details of this project have been introduced now. Within this chapter we presented the new or changed CML language features, the implementation of the ARs which are considered the major result of this project, and the MDSL generator. The next chapter will discuss the validation of our results, summarize the project and outline potential future work.

Chapter 5

Summary, Evaluation and Conclusion

This last chapter summarizes the results of the project and discusses our evaluation via prototyping, action research and case studies. It further presents the validation regarding the fulfillment of the requirements presented in Chapter 3 and gives an outlook to future work.

5.1 Results and Contributions

With this work we presented a set of seven Architectural Refactorings (ARs) for service decomposition based on strategic Domain-driven Design (DDD) patterns. We analyzed the criteria to be used to decompose a system based on existing literature and the literature review provided by Service Cutter [17]. In addition we studied opinions of DDD experts [3, 9, 28, 29, 33, 37, 39] regarding the question how a domain can be decomposed into bounded contexts. The proposed ARs are derived from these criteria, the expert input and our own professional experience [8, 23]. The ARs aim to support software architects using context maps and the DDD patterns in evolving a service decomposition in an iterative way.

Alongside this conceptual contribution, we implemented prototypes of the ARs as code refactorings for the Context Mapper DSL (CML) language. Therefore the Context Mapper tool [12] has been enhanced to not only create but also evolve DDD context maps on the basis of a formal Domain-specific Language (DSL). As part of this project the CML language has been improved in terms of usability on the basis of user feedback and our evaluation activities during the previous project [20]. In addition we added new features to the language to enable the prototypic implementation of the ARs in Context Mapper.

Besides these main results of this project we have implemented another generator. With our previous work [20] we already provided two generators producing PlantUML [32] diagrams and Service Cutter [17] input files out of CML models. The new service contract generator illustrates how DDD-based context maps can be implemented with a (micro)-service-oriented architecture. The generated contracts are written in the Microservices Domain-Specific Language (MDSL) [42]. With this prototype we propose one approach how to implement strategic DDD models as a conceptual contribution, but also support architects using Context Mapper towards implementing systems modeled in CML.

5.2 Evaluation via Prototyping, Action Research and Case Studies

Our validation approach corresponds to Shaw's recommendations and the validation type *experience* [35] with the goal to demonstrate correctness, usefulness and effectiveness of our concepts and tools. To validate the presented ARs, we implemented a prototype in the Context Mapper tool. During the implementation we applied action research [6] to improve the tool within short feedback cycles. The implemented prototype has then been validated by using the ARs on our fictitious "insurance" case study [11]. Thereby we especially validated the correctness of the AR implementations. However, regarding the ARs more validation has to be done to show their usefulness and effectiveness.

Regarding the CML language itself additional validation activities have been conducted after the language has been improved. With the project of Habegger and Schena [19] the Context Mapper tool has been applied to a real-world project and productive system in the health-care sector. This case study supported us in validating the usability and usefulness of the CML language as a modeling tool. The authors Habegger and Schena provided positive feedback regarding the language and mentioned it as a promising approach within their work [19] which supports our hypothesis that a modeling tool such as Context Mapper can be useful for software architects working with DDD.

However, they also mentioned that they currently would not recommend the tool for the identification of microservice decompositions for big monolithic systems. According to them it is too time-consuming to completely model such a system in CML and Context Mapper does currently not support to generate models out of existing source code as it other comparable modeling tools do. We have noted this as a potential weakness of the tool, at least for users working with existing monoliths. Future projects may provide tools generating CML out of existing source code to overcome this issue.

In addition, an industry contact of the advisor of this project could successfully model with CML, even though support regarding the DDD patterns was needed. Being familiar with the strategic DDD patterns clearly seems to be a prerequisite to be able to start modeling with our CML language quickly.

Besides the mentioned case studies we have further applied the new language version to our existing example models [11]. In summary, the results of our validation activities still suggest that our hypothesis regarding the usefulness of such a tool for software architects holds true.

5.3 Validation of Requirements

At the beginning of this project a set of concrete deliverables and critical success factors which are outlined in Chapter 1 have been defined. They required solving the existing CML grammar issues identified in our previous project [20], analyzing the criteria to be used to decompose a system, implementing at least four ARs, implementing a "proof of concept" for a service contract generator and finally providing meaningful examples of CML specifications and corresponding ARs. In addition the project definition [22] required that the tools are easy to use, robust and validated with respect to the project requirements.

The existing CML grammar issues have been solved and the language has been improved as explained in Chapter 4. We have analyzed the Decomposition Criteria (DCs) in Chapter 2 and implemented more then four (seven) ARs. We further implemented a generator for (micro-)service contracts as explained in Chapter 4. Examples of how the ARs work with corresponding CML models have been provided in Chapter 4, online[1], in our examples repository[2] [11] and in Appendix A. The user feedback we received from our evaluation activities suggests that the tool is easy to use and robust. Within this chapter we will further discuss all requirements listed in Chapter 3 and if they are fulfilled or not. Therefore, it can be stated that the critical success factors have been fulfilled. The next section will discuss and validate the functional requirements the project had to accomplish.

5.3.1 User Stories

In Chapter 3 we have derived user stories for all selected ARs. As we have implemented all of them within the Context Mapper tool [12] these single user stories are fulfilled. However, these single AR stories are derived from an abstract user story asking for transformation tools allowing the user to port, evolve and modernize the architecture in an iterative way. If this higher-level requirement is fulfilled basically depends on whether the selected ARs are sufficient for the case or not. We consider the functional requirements regarding the ARs as fulfilled for our prototype. As a productive tool it may require the implementation of further ARs to fulfill the users requirements.

Regarding the service contract generator we consider the implementation of the MDSL generator presented in Chapter 4 as sufficient to fulfill the corresponding user story US-8 in Chapter 3. It generates services with endpoints and operations and therefore provides a suggestion how a context map can be implemented in a (micro-)service architecture. Note that our goal concerning this matter was to provide a "proof of concept". The advisor of this project, who is the creator of the MDSL language [42], has already used our generator as well and his feedback confirms the fulfillment of this user story.

5.3.2 Non-Functional Requirements

Besides the functional requirements Chapter 3 also listed Non-Functional Requirements (NFRs). Within this section we want to discuss each of them briefly and validate if they are fulfilled. The NFRs are discussed in the order as presented in Chapter 3, starting with the CML refinements, continuing with the ARs, and ending with the general NFRs.

Quickly Writable without Redundancy

The new CML syntax uses shorter notations in comparison with the version presented in our previous project [20]. Redundancies and ambiguities with respect to our meta-model [20] were removed explicitely as part of this project. With the case studies of Habegger and Schena [19] and the already mentioned

[1] `https://contextmapper.github.io/docs/architectural-refactorings/`

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

industry contact of the advisor of this work we had new users who used the revised syntax. Habegger and Schena confirmed that they were able to model bounded contexts and a context map within the mentioned time limit. They stated that other CML examples are necessary to start modeling quickly. The industry contact was able to model his system as well. However, the feedback revealed that it is important to be familiar with the DDD patterns before start using our CML language. The results of our modeling and validation activities suggest that this NFR is fulfilled. However, we have not conducted a representative user test to measure and finally evaluate it yet.

Well Readable

Similar to the previous NFR we consider this NFR as fulfilled on the basis of our validation activities including the case studies with Habegger and Schena [19] and the mentioned industry contact. Whether the user is already familiar with the DDD patterns or not seems to be a critical factor for this NFR as well. To evaluate this NFR quantitatively we have to conduct a user test.

Consistent

This NFR requested our DSL syntax to be consistent with our meta-model [20], which we consider to be fulfilled. The syntax has been improved accordingly and existing deviations and ambiguities especially regarding the customer-supplier relationship have been solved. Additional semantic validators ensure consistency with our model and throw compiler errors in case of invalid pattern combinations.

Parsable by the Tool (Xtext)

This NFR is fullfilled since we were able to implement the defined syntax with the Xtext[3] framework.

«In line» with Common DDD Styles and the DDD Literature

The new CML syntax works with pattern abbreviations and we used the corresponding ones from literature [15, 16, 27, 41]. Therefore we consider this NFR as satisfied.

The following NFRs concern the implementation of the ARs.

Transformations must result in valid Models

The implemented transformations operate on bounded contexts and aggregates of the CML language. Potential conflicts occur whenever other objects refer the ones affected by an AR, which is the case for the context map in CML. As we have explained in Chapter 4, the context map is always adjusted accordingly so that the ARs result in valid CML models. Corresponding unit tests have been written for all ARs and all known cases. Therefore we acknowledge this NFR as fully addressed.

[3] `https://www.eclipse.org/Xtext/`

Performance

We have validated the performance of all ARs with our CML example models[4] as requested by this NFR and the transformations never took longer than two to three seconds. This NFR may has to be checked again in the future when bigger real-world projects are modeled with the tool. For this project we consider the NFR as fulfilled.

The remaining NFRs concern all Context Mapper implementations in general.

Future-oriented Use of Tools and Frameworks

For tools which were already used by Context Mapper before this project we refer to our previous work [20]. The only new tool or framework introduced by this project is Henshin[5] [38] which was used for the implementation of one AR. In our paper *Model Transformations for DSL Processing* [21] we have compared all available tools for such model transformations and concluded that Henshin is the most modern and most established. Note that the MDSL [42] generator currently not requires any dependencies to external tools or frameworks since we simply generate text. As already mentioned in our previous work [20], Xtext [14] uses ANTLR [5] to generate the parser which would simplify a potential future technology switch. In summary, we consider this NFR as fulfilled.

Reliability

The tool is tested with unit and integration tests and currently (v4.1.1) a test coverage of 93 percent is measured. During our validation and modeling activities we have further applied manual tests to ensure stability.

Licences

This NFR is fulfilled since the only new library introduced to the tool, namely Henshin, is licenced under an «Eclipse Public License». No GPL licences have been introduced.

Supportability and Maintainability

All tools and mechanisms promoting good code quality have already been set up for Context Mapper in our previous work [20]. With this project we were able to further increase the test coverage. We further avoided the usage of special and/or unkown language features of the used programming languages (Java, Xtend and Xtext). We therefore consider this NFR fulfilled.

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

[5] `https://www.eclipse.org/henshin/`

5.4 Conclusion and Future Work

In summary, the project requirements and critical goals have been fulfilled. Regarding the ARs there are still validation activities to be done to demonstrate their practicability. Future case studies may show that other ARs are needed in practice. The missing possibility to generate CML from existing source code is a potential liability of the Context Mapper tool in the current state. Users with big existing monoliths aiming to find a service decomposition might choose other tools. Another liability already mentioned in our previous project [20] is the fact that we currently only provide the tool for Eclipse, while many software engineers work with other Integrated Development Environments (IDEs). Overall the results and validation activities still suggest that the tool has the potential to be helpful for software architects which encourages us to continue our work in the future.

This project further provided the opportunity to strengthen the personal knowledge regarding the criteria to be used to decompose a software system. The elaboration of the ARs allowed us to study how these criteria can be used in the context of DDD with the goal to decompose domains into bounded contexts. Additional knowledge regarding how DDD-based architecture models can be implemented has been acquired through the development of the MDSL (micro-)service contract generator.

The decision which ARs to apply is currently left to the user. Using heuristics and algorithms similar to Service Cutter [17] to offer the architect AR suggestions based on our DCs automatically might be a promising approach for our next research project. In this way the Context Mapper tool could propose ARs which improve the architecture by analysing the current CML model. The collection of implemented ARs may be extended according to future user feedback and results from validation activities.

Future projects shall further decouple the CML language from the Eclipse IDE with the goal to provide support for other IDEs or a Command Line Interface (CLI). Context Mapper users have already asked for a CLI to use our generator tools.

Other future projects may increase the target user group by providing tools which generate CML models from existing source code. This would support users with already existing monolithic systems in identifying service decompositions. Generating (micro-)service project stubs out of the context maps might be an interesting feature as well. By generating specifications in a standard format such as OpenAPI [30] our users would be able to generate server stubs, client libraries, or Application Programming Interface (API) documentation by using other open source tools as well.

Appendix A

Complete AR Examples in Context Mapper DSL (CML)

In Chapter 4 we presented examples for all seven implemented Architectural Refactorings (ARs). Those examples are reduced to the necessary parts which are needed to explain the behavior of the ARs. This appendix contains the complete Context Mapper DSL (CML) example models which can be compiled by the CML tool in the current version[1] (v4.1.1). These examples can also be found in our examples repository[2] [11].

A.1 AR-1: Split Aggregate by Entities

A.1.1 Example Input Model

```
1  /* This is an example CML model for the 'Split Aggregate by Entities' refactoring and
2   * shows a small part of the insurance example.
3   */
4  ContextMap {
5    contains CustomerManagementContext
6
7  }
8
9  BoundedContext CustomerManagementContext {
10   type = FEATURE
11   domainVisionStatement = "The customer management context is responsible for ..."
12   implementationTechnology = "Java, JEE Application"
13   responsibilities = "Customers, Addresses"
14
15   /* With a right-click on the 'Customers' aggregate in our Eclipse IDE you can
16    * call the 'Split Aggregate by Entities' refactoring in the 'Context Mapper:
17    * Refactor' context menu. The refactoring will create a new aggregate and
18    * move one of the two entities to the new aggregate.
19    */
20   Aggregate Customers {
21     Entity Customer {
22       aggregateRoot
```

LISTING 63: Input AR-1: Split Aggregate by Entities (1)

[1] https://github.com/ContextMapper/context-mapper-dsl/releases/tag/v4.1.1

[2] https://github.com/ContextMapper/context-mapper-examples/tree/master/src/main/resources/architectural-refactorings

```
      - SocialInsuranceNumber sin
      String firstname
      String lastname
      - List<Address> addresses
    }
    Entity Address {
      String street
      int postalCode
      String city
    }
  }
}
```

LISTING 64: Input AR-1: Split Aggregate by Entities (2)

A.1.2 AR Result

```
/* This is an example CML model for the 'Split Aggregate by Entities' refactoring and
 * shows a small part of the insurance example.
 */
ContextMap {
  contains CustomerManagementContext

}

BoundedContext CustomerManagementContext {
  domainVisionStatement = "The customer management context is responsible for ..."
  responsibilities = "Customers, Addresses" implementationTechnology = "Java, JEE"
  Aggregate Customers {
    Entity Address {
      aggregateRoot

      String street
      int postalCode
      String city
    }
  }

  /* The newly created aggregate after applying 'Split Aggregate by Entities'
   * on the aggregate in the input listing above.
   *
   * Note that the refactoring does not produce meaningful aggregate names.
   * You can use the 'Rename Element' refactoring (SHIFT-ALT-R) to rename
   * the new aggregate.
   */
  Aggregate NewAggregate1 {
    Entity Customer {
      aggregateRoot

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

LISTING 65: Result AR-1: Split Aggregate by Entities

A.2 AR-2: Split Bounded Context by Use Cases

A.2.1 Example Input Model

```
/* This is an example CML model for the 'Split Bounded Context by Use Cases'
 * refactoring and shows a small part of the insurance example.
 */
ContextMap {
  contains PolicyManagementContext

}

/* With a right-click on the 'PolicyManagementContext' bounded context in our Eclipse
 * IDE you can execute the 'Split Bounded Context by Use Cases' refactoring. It will
 * split the existing bounded context and group the two aggregates of the
 * 'CreateOffer4Customer' use case together. The 'Contract' aggregate used by the
 * 'UpdateContract' use case will be separated.
 */
BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  type = FEATURE
  domainVisionStatement = "This bounded context manages the contracts and policies."
  responsibilities = "Offers, Contracts, Policies"
  implementationTechnology = "Java, Spring App"

  Aggregate Offers {
    useCases = CreateOffer4Customer

    Entity Offer {
      aggregateRoot

      int offerId
      - Customer client
      - List<Product> products
      BigDecimal price
    }
  }
  Aggregate Products {
    useCases = CreateOffer4Customer

    Entity Product {
      aggregateRoot

      - ProductId identifier
      String productName
    }
    ValueObject ProductId {
      int productId key
    }
  }
  Aggregate Contract {
    useCases = UpdateContract

    Entity Contract {
      aggregateRoot

      - ContractId identifier
      - Customer client
      - List<Product> products
    }
    ValueObject ContractId {
      int contractId key
    }
```

LISTING 66: Input AR-2: Split Bounded Context by Use Cases (1)

```
    Entity Policy {
      int policyNr
      - Contract contract
      BigDecimal price
    }
  }
}

/* Domain & Subdomain Definitions */
Domain InsuranceDomain {
  Subdomain PolicyManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing contracts and policies."
  }
}

UseCase UpdateContract
UseCase CreateOffer4Customer
```

LISTING 67: Input AR-2: Split Bounded Context by Use Cases (2)

A.2.2 AR Result

```
/* This is an example CML model for the 'Split Bounded Context by Use Cases'
 * refactoring and shows a small part of the insurance example.
 */
ContextMap {
  contains PolicyManagementContext

}

BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  domainVisionStatement = "This bounded context manages the contracts and policies."
  responsibilities = "Offers, Contracts, Policies"
  implementationTechnology = "Java, Spring App"

  Aggregate Contract {
    useCases = UpdateContract

    Entity Contract {
      aggregateRoot

      - ContractId identifier
      - Customer client
      - List<Product> products
    }
    ValueObject ContractId {
      int contractId key
    }

    Entity Policy {
      int policyNr
      - Contract contract
      BigDecimal price
    }
  }
}
```

LISTING 68: Result AR-2: Split Bounded Context by Use Cases (1)

```
/**
 * A new bounded context created by the 'Split Bounded Context by Use Cases'
 * refactoring applied to the CML in the input listing above.
 *
 * Note that the refactoring does not produce meaningful bounded context names.
 * You can use the 'Rename Element' refactoring (SHIFT-ALT-R) to rename the
 * new aggregate.
 */
BoundedContext NewBoundedContext1 {
  Aggregate Offers {
    useCases = CreateOffer4Customer

    Entity Offer {
      aggregateRoot

      int offerId
      - Customer client
      - List<Product> products
      BigDecimal price
    }
  }
  Aggregate Products {
    useCases = CreateOffer4Customer

    Entity Product {
      aggregateRoot

      - ProductId identifier
      String productName
    }
    ValueObject ProductId {
      int productId key
    }
  }
}

/* Domain & Subdomain Definitions */
Domain InsuranceDomain {
  Subdomain PolicyManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing contracts and policies."
  }
}

UseCase UpdateContract
UseCase CreateOffer4Customer
```

LISTING 69: Result AR-2: Split Bounded Context by Use Cases (1)

A.3 AR-3: Split Bounded Context by Owner

A.3.1 Example Input Model

```
/* This is an example CML model for the 'Split Bounded Context by Owner' refactoring and
 * shows a small part of the insurance example. */
ContextMap {
  contains CustomerSelfServiceContext

}

/* With a right-click on the 'CustomerSelfServiceContext' bounded context in our Eclipse
 * IDE you can execute the 'Split Bounded Context by Owners' refactoring. It will split
 * the existing bounded context according to the two owning teams 'CustomerBackendTeam'
 * and 'CustomerFrontendTeam'.
 */
BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  type = APPLICATION
  domainVisionStatement = "This context represents a web application which allows ..."
  responsibilities = "AddressChange"
  implementationTechnology = "PHP Web Application"

  Aggregate CustomerFrontend {
    owner = CustomerFrontendTeam

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    owner = CustomerBackendTeam

    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}

/* Team Definitions */
BoundedContext CustomerBackendTeam {
  type = TEAM
}

BoundedContext CustomerFrontendTeam {
  type = TEAM
}

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

LISTING 70: Input AR-3: Split Bounded Context by Owner

A.3.2 AR Result

```
/* This is an example CML model for the 'Split Bounded Context by Owner' refactoring
 * and shows a small part of the insurance example. */
ContextMap {
  contains CustomerSelfServiceContext

}

BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  domainVisionStatement = "This context represents a web application which allows ..."
  type = APPLICATION
  responsibilities = "AddressChange"
  implementationTechnology = "PHP Web Application"

  Aggregate CustomerFrontend {
    owner = CustomerFrontendTeam

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
}

/* Team Definitions */
BoundedContext CustomerBackendTeam {
  type = TEAM
}

BoundedContext CustomerFrontendTeam {
  type = TEAM
}

/**
 * The new bounded context created by the 'Split Bounded Context by Owners' refactoring
 * applied to the input CML model in the listing above.
 *
 * Note that the refactoring does not produce meaningful bounded context names.
 * You can use the 'Rename Element' refactoring (SHIFT-ALT-R) to rename the new
 * aggregate.
 *
 * The automated refactorings add newly created bounded contexts at the end of the
 * 'bounded context' block, which might not always be the desired order.
 * You may change the order after the refactoring manually.
 */
BoundedContext NewBoundedContext1 {
  Aggregate Acounts {
    owner = CustomerBackendTeam

    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}
```

LISTING 71: Result AR-3: Split Bounded Context by Owner (1)

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

LISTING 72: Result AR-3: Split Bounded Context by Owner (2)

A.4 AR-4: Extract Aggregates by Volatility

A.4.1 Example Input Model

```
/* This is an example CML model for the 'Extract Aggregates by Volatility' refactoring
 * and shows a small part of the insurance example. */
ContextMap {
  contains CustomerSelfServiceContext

}

/* With a right-click on the 'CustomerSelfServiceContext' bounded context in our
 * Eclipse IDE you can execute the 'Extract Aggregates by Volatility' refactoring.
 * If you choose the volatility 'OFTEN', it will extract the volatile
 * 'CustomerFrontend' aggregate and create a new bounded context for it.
 */
BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  type = APPLICATION
  domainVisionStatement = "This context represents a web application which allows ..."
  responsibilities = "AddressChange"
  implementationTechnology = "PHP Web Application"

  Aggregate CustomerFrontend {
    likelihoodForChange = OFTEN

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}

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

LISTING 73: Input AR-4: Extract Aggregates by Volatility

A.4.2 AR Result

```
/* This is an example CML model for the 'Extract Aggregates by Volatility' refactoring
 * and shows a small part of the insurance example. */
ContextMap {
  contains CustomerSelfServiceContext

}

BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  domainVisionStatement = "This context represents a web application which allows ..."
  type = APPLICATION
  responsibilities = "AddressChange"
  implementationTechnology = "PHP Web Application"

  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}

/**
 * The extracted bounded context after applying 'Extract Aggregates by Volatility'
 * to the input CML model in the listing above. The chosen volatility was 'OFTEN'.
 *
 * You may want to change the name of newly created bounded contexts after applying
 * refactorings.
 */
BoundedContext CustomerSelfServiceContext_Volatility_OFTEN {
  Aggregate CustomerFrontend {
    likelihoodForChange = OFTEN

    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
}

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

LISTING 74: Result AR-4: Extract Aggregates by Volatility

A.5 AR-5: Extract Aggregates by Cohesion

A.5.1 Example Input Model

```
/* This is an example CML model for the 'Extract Aggregates by Cohesion' refactoring
 * and shows a small part of the insurance example. */
ContextMap {
  contains PolicyManagementContext

}

/* With a right-click on the 'PolicyManagementContext' bounded context in our Eclipse
 * IDE you can execute the 'Extract Aggregates by Cohesion' refactoring. A dialog
 * will pop up which allows you to select the aggregates to be extracted. You can
 * further specify the name of the new bounded context. For example: As architect
 * you may want to extract the 'Offers' aggregate due to other differing
 * availability requirements.
 */
BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  type = FEATURE
  domainVisionStatement = "This bounded context manages the contracts and policies."
  responsibilities = "Offers, Contracts, Policies"
  implementationTechnology = "Java, Spring App"

  Aggregate Offers {
    Entity Offer {
      aggregateRoot

      int offerId
      - Customer client
      - List<Product> products
      BigDecimal price
    }
  }
  Aggregate Products {
    Entity Product {
      aggregateRoot

      - ProductId identifier
      String productName
    }
    ValueObject ProductId {
      int productId key
    }
  }
  Aggregate Contract {
    Entity Contract {
      aggregateRoot

      - ContractId identifier
      - Customer client
      - List<Product> products
    }
    ValueObject ContractId {
      int contractId key
    }
```

LISTING 75: Input AR-5: Extract Aggregates by Cohesion (1)

```
    Entity Policy {
      int policyNr
      - Contract contract
      BigDecimal price
    }
  }
}

/* Domain & Subdomain Definitions */
Domain InsuranceDomain {
  Subdomain PolicyManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing contracts and policies."
  }
}
```

LISTING 76: Input AR-5: Extract Aggregates by Cohesion (2)

A.5.2 AR Result

```
/* This is an example CML model for the 'Extract Aggregates by Cohesion' refactoring
 * and shows a small part of the insurance example. */
ContextMap {
  contains PolicyManagementContext

}

BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  domainVisionStatement = "This bounded context manages the contracts and policies."
  responsibilities = "Offers, Contracts, Policies"
  implementationTechnology = "Java, Spring App"

  Aggregate Products {
    Entity Product {
      aggregateRoot

      - ProductId identifier
      String productName
    }
    ValueObject ProductId {
      int productId key
    }
  }
  Aggregate Contract {
    Entity Contract {
      aggregateRoot

      - ContractId identifier
      - Customer client
      - List<Product> products
    }
    ValueObject ContractId {
      int contractId key
    }
```

LISTING 77: Result AR-5: Extract Aggregates by Cohesion (1)

```
    Entity Policy {
      int policyNr
      - Contract contract
      BigDecimal price
    }
  }
}

/**
 * New bounded context after applying 'Extract Aggregates by Cohesion' to
 * the input CML model in listing above, with the following parameters:
 *   - New bounded context name: 'SalesBoundedContext'
 *   - Selected aggregates: 'Offers'
 */
BoundedContext SalesBoundedContext {
  Aggregate Offers {
    Entity Offer {
      aggregateRoot

      int offerId
      - Customer client
      - List<Product> products
      BigDecimal price
    }
  }
}

/* Domain & Subdomain Definitions */
Domain InsuranceDomain {
  Subdomain PolicyManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing contracts and policies."
  }
}
```

LISTING 78: Result AR-5: Extract Aggregates by Cohesion (2)

A.6 AR-6: Merge Aggregates

A.6.1 Example Input Model

```
/* This is an example CML model for the 'Merge Aggregates' refactoring and
 * shows a small part of the insurance example. */
ContextMap {
  contains CustomerManagementContext
  contains CustomerSelfServiceContext

  CustomerSelfServiceContext [D,C]<-[U,S] CustomerManagementContext {
    exposedAggregates = Customers, Addresses
  }
}
```

LISTING 79: Input AR-6: Merge Aggregates (1)

```
/* With a right-click on the 'Customers' aggregate (or on the 'Addresses' aggregate,
 * as you wish) in our Eclipse IDE you can execute the 'Merge Aggregates' refactoring.
 * A dialog will show up and ask you with which other aggregate you want to merge.
 * Choose the other aggregate and the refactoring will merge them.
 */
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  domainVisionStatement = "The customer management context is responsible for ..."
  implementationTechnology = "Java, JEE Application"
  responsibilities = "Customers, Addresses"

  Aggregate Customers {
    Entity Customer {
      aggregateRoot
      - SocialInsuranceNumber sin
      String firstname
      String lastname
      - List<Address> addresses
    }
    ValueObject SocialInsuranceNumber {
      String sin key
    }
  }
  Aggregate Addresses {
    Entity Address {
      String street
      int postalCode
      String city
    }
  }
}

BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  type = APPLICATION
  domainVisionStatement = "This context represents a web application which allows ..."
  responsibilities = "AddressChange"
  implementationTechnology = "PHP Web Application"

  Aggregate CustomerFrontend {
    Entity CustomerAddressChange {
      aggregateRoot
      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot
      String username
      - Customer accountCustomer
    }
  }
}

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

LISTING 80: Input AR-6: Merge Aggregates (2)

A.6.2 AR Result

```
/* This is an example CML model for the 'Merge Aggregates' refactoring and
 * shows a small part of the insurance example. */
ContextMap {
  contains CustomerManagementContext
  contains CustomerSelfServiceContext

  CustomerSelfServiceContext [D,C]<-[U,S] CustomerManagementContext {
    exposedAggregates = Customers
  }
}

/*
 * The resulting bounded context after applying 'Merge Aggregates' to the input CML
 * model in the listing above. The 'Addresses' aggregate has been merged into the
 * 'Customers' aggregate.
 */
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  domainVisionStatement = "The customer management context is responsible for ..."
  responsibilities = "Customers, Addresses"
  implementationTechnology = "Java, JEE Application"
  Aggregate Customers {
    Entity Customer {
      aggregateRoot
      - SocialInsuranceNumber sin
      String firstname
      String lastname
      - List<Address> addresses
    }
    ValueObject SocialInsuranceNumber {
      String sin key
    }
    Entity Address {
      String street
      int postalCode
      String city
    }
  }
}

BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  domainVisionStatement = "This context represents a web application which allows ..."
  type = APPLICATION
  responsibilities = "AddressChange"
  implementationTechnology = "PHP Web Application"

  Aggregate CustomerFrontend {
    Entity CustomerAddressChange {
      aggregateRoot
      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot
      String username
      - Customer accountCustomer
    }
  }
}
```

LISTING 81: Result AR-6: Merge Aggregates (1)

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

LISTING 82: Result AR-6: Merge Aggregates (2)

A.7 AR-7: Merge Bounded Contexts

A.7.1 Example Input Model

```
/* This is an example CML model for the 'Merge Bounded Contexts' refactoring
 * and shows a small part of the insurance example. */
ContextMap {
  contains CustomerManagementContext
  contains CustomerSelfServiceContext
  contains PrintingContext

  CustomerSelfServiceContext [D,C]<-[U,S] CustomerManagementContext {
    exposedAggregates = Customers
  }

  CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext {
    implementationTechnology = "SOAP"
    downstreamRights = INFLUENCER
    exposedAggregates = Printing
  }

}

/* With a right-click on the 'CustomerManagementContext' (or one of the other contexts,
 * as you wish) bounded context in our Eclipse IDE you can execute the
 * 'Merge Bounded Contexts' refactoring. A dialog will show up and ask you with which
 * other bounded context you want to merge. Choose a second bounded context and the
 * refactoring will merge them.
 */
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  domainVisionStatement = "The customer management context is responsible for ..."
  implementationTechnology = "Java, JEE Application"
  responsibilities = "Customers, Addresses"

  Aggregate Customers {
    Entity Customer {
      aggregateRoot

      - SocialInsuranceNumber sin
      String firstname
      String lastname
      - List<Address> addresses
    }
    Entity Address {
      String street
      int postalCode
      String city
    }
```

LISTING 83: Input AR-7: Merge Bounded Contexts (1)

```
    ValueObject SocialInsuranceNumber {
      String sin key
    }
  }
}

BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  type = APPLICATION
  domainVisionStatement = "This context represents a web application which allows ..."
  responsibilities = "AddressChange"
  implementationTechnology = "PHP Web Application"

  Aggregate CustomerFrontend {
    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}

BoundedContext PrintingContext implements PrintingDomain {
  type = SYSTEM
  responsibilities = "Document Printing"
  domainVisionStatement = "An external system which provides printing services."

  Aggregate Printing {
    Entity PrintingJob {
      aggregateRoot

      int printingId
      - Document document
      - Template template
    }

    Entity Document {
      DomainObject source
      String template
    }
  }
  Aggregate Templating {
    Entity Template {
      aggregateRoot

      int templateId
      String templateName
    }
  }
}

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

LISTING 84: Input AR-7: Merge Bounded Contexts (2)

```
  Subdomain PrintingDomain {
    type = SUPPORTING_DOMAIN
    domainVisionStatement = "Service (external system) to solve printing for ..."
  }
}
```

LISTING 85: Input AR-7: Merge Bounded Contexts (3)

A.7.2 AR Result

```
/* This is an example CML model for the 'Merge Bounded Contexts' refactoring
 * and shows a small part of the insurance example. */
ContextMap {
  contains CustomerManagementContext
  contains PrintingContext

  CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext {
    implementationTechnology = "SOAP"
    exposedAggregates = Printing
  }

}

/**
 * The merged bounded context after applying 'Merge Bounded Contexts' to the input CML
 * model in the listing above. We selected the 'CustomerSelfServiceContext' context
 * as the second bounded context.
 */
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  domainVisionStatement = "The customer management context is responsible for ..."
  responsibilities = "Customers, Addresses" , "AddressChange"
  implementationTechnology = "Java, JEE Application"
  Aggregate Customers {
    Entity Customer {
      aggregateRoot

      - SocialInsuranceNumber sin
      String firstname
      String lastname
      - List<Address> addresses
    }
    Entity Address {
      String street
      int postalCode
      String city
    }
    ValueObject SocialInsuranceNumber {
      String sin key
    }
  }
  Aggregate CustomerFrontend {
    Entity CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }
```

LISTING 86: Result AR-7: Merge Bounded Contexts (1)

```
  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}

BoundedContext PrintingContext implements PrintingDomain {
  domainVisionStatement = "An external system which provides printing services."
  type = SYSTEM
  responsibilities = "Document Printing"
  Aggregate Printing {
    Entity PrintingJob {
      aggregateRoot

      int printingId
      - Document document
      - Template template
    }

    Entity Document {
      DomainObject source
      String template
    }
  }
  Aggregate Templating {
    Entity Template {
      aggregateRoot

      int templateId
      String templateName
    }
  }
}

/* Domain & Subdomain Definitions */
Domain InsuranceDomain {
  Subdomain CustomerManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing everything customer-related."
  }
  Subdomain PrintingDomain {
    type = SUPPORTING_DOMAIN
    domainVisionStatement = "Service (external system) to solve printing for ..."
  }
}
```

LISTING 87: Result AR-7: Merge Bounded Contexts (2)

Appendix B

Revised CML Language Reference

This appendix contains a reference for the current version **v4.1.1**[1] of the Context Mapper DSL (CML) language. It is an updated version of the language reference we provided in our previous work [20]. This reference is adjusted with respect to all the language changes we have implemented during this project.

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 [20] 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) [34] and not explained within this language reference. We refer to the Sculptor online documentation[3].

B.1 Language Design

The design of our DSL and its rules is based on the domain model presented in our previous project [20]. It can also be found online[4].

B.2 Terminals

The grammar snippets within the language reference use the terminals defined in Listing 88.

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

LISTING 88: Xtext CML Terminals

[1] https://github.com/ContextMapper/context-mapper-dsl/tree/v4.1.1
[2] https://contextmapper.github.io/docs
[3] http://sculptorgenerator.org/documentation/
[4] https://contextmapper.github.io/docs/language-model/

B.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 have one context map only. All other root elements can occur multiple times. Listing 89 shows the root grammar rule of the language.

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

LISTING 89: 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 have mentioned this as a known limitation in Chapter 4.

B.4 Context Map

The context maps grammar rule is shown in Listing 90. 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* keywords 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 one 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
    (('state' '=' state=ContextMapState)? &
    ('type' '=' type=ContextMapType)?)
    ('contains' boundedContexts += [BoundedContext]
           ("," boundedContexts += [BoundedContext])*)*
    relationships += Relationship*
  CLOSE
;
```

LISTING 90: Xtext Context Map Grammar Rule

Listing 91 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 91: Syntax example for the ContextMap rule

Listing 92 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 92: 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 93.

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

LISTING 93: Xtext: Relationship Rule

The SymmetricRelationship rule further allows the application of the rules Partnership or SharedKernel (Listing 94).

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

LISTING 94: Xtext: SymmetricRelationship Rule

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

The rule UpstreamDownstreamRelationship shown in the listings 95 and 96 allows either the application of the CustomerSupplierRelationship rule or directly writing a generic upstream-downstream relationship.

```
UpstreamDownstreamRelationship:
  CustomerSupplierRelationship |
  (
    (
      // variant 1: 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]) |

      // variant 2: arrow from left to right
      (
        (upstream = [BoundedContext] '->' downstream = [BoundedContext]) |
        (upstream = [BoundedContext] '[''U'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' '->'
          '[''D'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext]) |
        ('[''U'(','(upstreamRoles+=UpstreamRole) ("," upstreamRoles+=UpstreamRole)*)?']'
          upstream = [BoundedContext] '->''[''D'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext]) |
        (upstream = [BoundedContext] '[''U'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']''->' downstream = [BoundedContext]
          '[''D'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']') |
        ('[''U'(','(upstreamRoles+=UpstreamRole) ("," upstreamRoles+=UpstreamRole)*)?']'
          upstream = [BoundedContext] '->' downstream = [BoundedContext]
          '[''D'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']')
      ) |

      // variant 3: arrow from right to left
      (
        (downstream = [BoundedContext] '<-' upstream = [BoundedContext]) |
        (downstream = [BoundedContext] '['']D'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' '<-'
          '[''U'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' upstream = [BoundedContext]) |
        ('[''D'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext]
          '<-''['']U'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' upstream = [BoundedContext]) |
```

LISTING 95: Xtext: UpstreamDownstreamRelationship Rule (1)

```
46          (downstream = [BoundedContext] '[''D'(','(downstreamRoles+=DownstreamRole)
47            ("," downstreamRoles+=DownstreamRole)*)?']''<-' upstream = [BoundedContext]
48            '[''U'(','(upstreamRoles+=UpstreamRole)
49            ("," upstreamRoles+=UpstreamRole)*)?']') |
50          ('[''D'(','(downstreamRoles+=DownstreamRole)
51            ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext]
52            '<-' upstream = [BoundedContext] '[''U'(','(upstreamRoles+=UpstreamRole)
53            ("," upstreamRoles+=UpstreamRole)*)?']')
54        )
55      )
56      // name and body
57      (':' name=ID)?
58      (OPEN (
59        ('implementationTechnology' '=' implementationTechnology=STRING)? &
60        (('exposedAggregates' '=' upstreamExposedAggregates += [Aggregate])
61          ("," upstreamExposedAggregates += [Aggregate])*)? &
62        ('downstreamRights' '=' downstreamGovernanceRights=DownstreamGovernanceRights)?
63      )
64      CLOSE)?
65    )
66  ;
```

LISTING 96: Xtext: UpstreamDownstreamRelationship 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[5] and might be easier to read. The length and complexity of this rule is increased due to the many different variants we offer our users to declare relationships.

As declared in the grammar rule, there are basically three alternative syntaxes which allow the specification of the same upstream-downstream relationship. The listings 97, 98 and 99 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.

```
1  LocationContext Upstream-Downstream CargoBookingContext
2  // or inverse:
3  CargoBookingContext Downstream-Upstream LocationContext
```

LISTING 97: CML: Upstream-Downstream Variant 1

The variants 2 and 3 use the abbreviations *U* for upstream and *D* for downstream.

```
1  LocationContext [U]->[D] CargoBookingContext
2  // or: (without the brackets and the 'U' and 'D' one declares an upstream-downstream
3  // relationship as well)
4  LocationContext -> CargoBookingContext
```

LISTING 98: CML: Upstream-Downstream Variant 2

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

```
CargoBookingContext [D]<-[U] LocationContext
// or: (without the brackets and the 'U' and 'D' one declares an upstream-downstream
// relationship as well)
CargoBookingContext <- LocationContext
```

LISTING 99: CML: Upstream-Downstream Variant 3

Note that if one of the variants with the arrows (-> or <-) is used, the arrow always points from the upstream towards the downstream, reflecting the influence flow [27] (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 behind the *U* and the *D*. In the variant with the long keywords we use the same brackets for the rules but without the abbreviations for upstream and downstream, since this would be redundant.

The following listings 100, 101 and 102 illustrate the same relationships as before but with upstream and downstream roles.

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

LISTING 100: CML: Upstream-Downstream Variant 1 with Roles

```
LocationContext [U,OHS,PL]->[D,ACL] CargoBookingContext
```

LISTING 101: CML: Upstream-Downstream Variant 2 with Roles

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

LISTING 102: CML: Upstream-Downstream Variant 3 with Roles

Listing 103 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 103: Xtext: UpstreamRole and DownstreamRole

The syntax with the arrows and the abbreviations further allows to place the brackets with the upstream (U), downstream (D) and relationship roles flexible in front or after the bounded context name. 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. Listing 104 shows all possible placements. However, all four variants are again semantically equal. Note that this flexible bracket placement is not possible for the variant with the long keywords *Upstream-Downstream* and *Downstream-Upstream*.

```
1 LocationContext [U]->[D] VoyagePlanningContext // brackets centered
2
3 [U]LocationContext -> VoyagePlanningContext[D] // brackets outside
4
5 [U]LocationContext -> [D]VoyagePlanningContext // both on the left side
6
7 LocationContext[U] -> VoyagePlanningContext[D] // both on the right side
```

LISTING 104: CML: Upstream-Downstream 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 105 illustrates an example for a relationship declaration with name.

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

LISTING 105: 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 project [20] explain all the language features and the reasons why they were added. Listing 106 shows an example for a relationship specification with body and corresponding attributes. All attributes here are optional and the order does not matter.

```
1 LocationContext [U]->[D] VoyagePlanningContext : ExampleName {
2   implementationTechnology = "RESTful HTTP"
3   exposedAggregates = Location, OtherAggregate
4   downstreamRights = INFLUENCER
5 }
```

LISTING 106: CML: Upstream-Downstream Example with Attributes (Body)

The *exposedAggregates* attribute must reference aggregates (see syntax in Section B.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 107 shows the rule specifying the allowed values for the *downstreamRights* attribute.

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

LISTING 107: Xtext: DownstreamGovernanceRights

The alternative UpstreamDownstreamRelationship defined by the role CustomerSupplierRelationship is explained in Section B.11.

B.4.1 Context Map Semantic Rules

Note that semantic validators exist for a 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 rule 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 of a context map 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.

B.5 Bounded Context

A bounded context can be defined according to the BoundedContext grammar rule, shown in Listing 108.

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 can be assigned. The keyword *implementationTechnology* assigns an implementation technology and the keyword *knowledgeLevel* allows the assigning of a KnowledgeLevel.

```
BoundedContext:
  'BoundedContext' name=ID
    (('implements' (implementedSubdomains+=[Subdomain])
                   ("," implementedSubdomains+=[Subdomain])*)?
     & ('realizes' (realizedBoundedContexts+=[BoundedContext])
                   ("," realizedBoundedContexts+=[BoundedContext])*)?)
  (
    OPEN
      (('domainVisionStatement' '=' domainVisionStatement=STRING)? &
      ('type' '=' type=BoundedContextType)? &
      (('responsibilities' '=' responsibilities+=STRING)
                          ("," responsibilities+=STRING)*)? &
      ('implementationTechnology' '=' implementationTechnology=STRING)? &
      ('knowledgeLevel' '=' knowledgeLevel=KnowledgeLevel)?)
      modules += Module*
      aggregates += Aggregate*
    CLOSE
  )?
;
```

LISTING 108: Xtext: BoundedContext rule

The allowed values for the enum's BoundedContextType and KnowledgeLevel are given by the rules in Listing 109.

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

LISTING 109: Xtext: BoundedContextType & KnowledgeLevel

Responsibilities can further be added as a list of strings (also mentioned in Section B.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 [34] concept. However it is modified and can contain aggregates in addition to the other Sculptor [34] elements. Aggregates are explained in Section B.18.

With the *implements* keyword it is further possible to define which subdomains the bounded context implements. The referenced subdomains must be specified within a domain as explained in Section B.6. The listings 110 and 111 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
```

LISTING 110: CML: Bounded Context Example (1)

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

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

LISTING 111: CML: Bounded Context Example (2)

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 112 shows an example for this use case.

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

LISTING 112: Xtext: *realizes* Keyword Example

B.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).

B.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 113 illustrates the corresponding Domain grammar rule.

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

LISTING 113: Xtext: Domain Rule

The Subdomain pattern is defined by the grammar rule in Listing 114. As on a bounded context (B.5) the subdomain allows to specify a domain vision statement string. The *type* attribute on subdomains allows values defined by the SubDomainType enum, illustrated in Figure 115.

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

LISTING 114: Xtext: Subdomain Rule

The subdomain further offers the possibility to add entities (Sculptor [34], 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 115: Xtext: SubDomainType enum

The listings 116 and 117 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."
  }
  Subdomain PolicyManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Subdomain managing contracts and policies."
  }
```

LISTING 116: CML: Domain and Subdomains (1)

```
  Subdomain RiskManagementDomain {
    type = GENERIC_SUBDOMAIN
    domainVisionStatement = "Subdomain supporting risk management."
  }
}
```

LISTING 117: CML: Domain and Subdomains (2)

B.7 Use Cases

Uses cases can be defined on the root level of a CML file and are then referenced from aggregates (see Section B.18). They are used to specify which aggregates are accessed by the same use cases. Listing 118 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 118: 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 119 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 119: CML: Use Cases

B.8 Domain Vision Statement

The Domain Vision Statement pattern is implemented as a description attribute (String) on bounded contexts (B.5) and subdomains (B.6). For the corresponding grammar rules, we refer to Section B.5 and Section B.6. Listing 120 shows an example bounded context with a domain vision statement, and Listing 121 a subdomain accordingly.

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

LISTING 120: Xtext: Domain Vision Statement on Bounded Context

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

LISTING 121: Xtext: Domain Vision Statement on Subdomain

B.9 Partnership

The Partnership relationship pattern is defined by the grammar rule illustrated in Listing 122. There are two syntax variants to declare a partnership relationship.

```
Partnership:
  (
    (participant1 = [BoundedContext] 'Partnership' participant2 = [BoundedContext]) |
    (participant1 = [BoundedContext] '[''P'']' '<->'
      '[''P'']' participant2 = [BoundedContext]) |
    ('[''P'']' participant1 = [BoundedContext] '<->'
      '[''P'']' participant2 = [BoundedContext]) |
    (participant1 = [BoundedContext] '[''P'']' '<->'
      participant2 = [BoundedContext] '[''P'']') |
    ('[''P'']' participant1 = [BoundedContext] '<->'
      participant2 = [BoundedContext] '[''P'']')
  )
  (':' name=ID)?
  (OPEN
    ('implementationTechnology' '=' implementationTechnology=STRING)?
  CLOSE)?
;
```

LISTING 122: Xtext: Partnership Rule

The first uses the *Partnership* keyword whereas the second uses an arrow pointing in both directions, indicating symmetry, and the abbreviation *P* within brackets for both *partners*. The listings 123 and 124 illustrate examples for both variants.

```
PolicyManagementContext Partnership DebtCollection
```

LISTING 123: CML: Partnership Syntax Variant 1

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

LISTING 124: CML: Partnership Syntax Variant 2

The second variant with the arrow allows to place the brackets in different positions similar to upstream-downstream relationships presented in Section B.4. Listing 125 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 125: 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 126 illustrates an example for a partnership relationship declaration with name.

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

LISTING 126: CML: Partnership Relationship with Name

As Listing 127 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 127: CML: Partnership Relationships with Implementation Technology

Note that the shared kernel relationship is the default relationship regarding the two asymmetric relationships. A relationship declaration with arrow but without brackets as illustrated by Listing 128 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 128: CML: Shared Kernel as Default Asymmetric Relationship

B.10 Shared Kernel

The Shared Kernel relationship pattern is defined by the grammar rule illustrated in Listing 129. There are two syntax variants to declare a shared kernel relationship.

```
SharedKernel:
  (
    (participant1 = [BoundedContext] 'Shared-Kernel' participant2 = [BoundedContext]) |
    (participant1 = [BoundedContext] '<->' participant2 = [BoundedContext]) |
    (participant1 = [BoundedContext] '[''SK'']' '<->'
      '[''SK'']' participant2 = [BoundedContext]) |
    ('[''SK'']' participant1 = [BoundedContext] '<->'
      '[''SK'']' participant2 = [BoundedContext]) |
    (participant1 = [BoundedContext] '[''SK'']' '<->'
      participant2 = [BoundedContext] '[''SK'']') |
    ('[''SK'']' participant1 = [BoundedContext] '<->'
      participant2 = [BoundedContext] '[''SK'']')
  )
  (':' name=ID)?
  (OPEN
    ('implementationTechnology' '=' implementationTechnology=STRING)?
  CLOSE)?
;
```

LISTING 129: Xtext: Partnership Rule

The first uses the *Shared-Kernel* keyword whereas the second uses an arrow pointing in both directions, indicating symmetry, and the abbreviation *SK* within brackets. The listings 130 and 131 illustrate examples for both variants.

```
PolicyManagementContext Shared-Kernel DebtCollection
```

LISTING 130: CML: Shared Kernel Syntax Variant 1

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

LISTING 131: CML: Shared Kernel Syntax Variant 2

The second variant with the arrow allows to place the brackets in different positions similar to upstream-downstream relationships presented in Section B.4. Listing 132 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 132: 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 133 illustrates an example for a shared kernel relationship declaration with name.

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

LISTING 133: CML: Shared Kernel Relationship with Name

As Listing 134 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 134: CML: Shared Kernel Relationships with Implementation Technology

Note that the shared kernel relationship is the default relationship regarding the two asymmetric relationships. A relationship declaration with arrow but without brackets as illustrated by Listing 135 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 135: CML: Shared Kernel as Default Asymmetric Relationship

B.11 Customer-Supplier

The customer-supplier relationship pattern is defined by the grammar rule illustrated in the listings 136 and 137. 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: 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]) |
      // variant 2: arrow from left to right
      (
        (upstream = [BoundedContext] '['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' '->' '['('D'',')?'C'
          (','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext]) |
        ('['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' upstream = [BoundedContext] '->'
          '['('D'',')?'C'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext]) |
        (upstream = [BoundedContext] '['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' '->' downstream = [BoundedContext]
          '['('D'',')?'C'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']') |
        ('['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' upstream = [BoundedContext] '->'
          downstream = [BoundedContext] '['('D'',')?'C'
          (','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']')
      ) |
      // variant 3: arrow from right to left
      (
        (downstream = [BoundedContext] '['('D'',')?'C'
          (','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' '<-' '['('U'',')?'S'
          (','(upstreamRoles+=UpstreamRole) ("," upstreamRoles+=UpstreamRole)*)?']'
          upstream = [BoundedContext]) |
        ('['('D'',')?'C'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext] '<-'
          '['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']' upstream = [BoundedContext]) |
        (downstream = [BoundedContext] '['('D'',')?'C'
          (','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' '<-' upstream = [BoundedContext]
          '['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']') |
        ('['('D'',')?'C'(','(downstreamRoles+=DownstreamRole)
          ("," downstreamRoles+=DownstreamRole)*)?']' downstream = [BoundedContext] '<-'
          upstream = [BoundedContext] '['('U'',')?'S'(','(upstreamRoles+=UpstreamRole)
          ("," upstreamRoles+=UpstreamRole)*)?']')
      )
    )
```

LISTING 136: Xtext: Customer-Supplier Rule (1)

```
55      // name and body
56      (':' name=ID)?
57      (OPEN (
58        ('implementationTechnology' '=' implementationTechnology=STRING)? &
59        (('exposedAggregates' '=' upstreamExposedAggregates += [Aggregate])
60          ("," upstreamExposedAggregates += [Aggregate])*)? &
61        ('downstreamRights' '=' downstreamGovernanceRights=DownstreamGovernanceRights)?
62      )
63      CLOSE)?
64    )
65  ;
```

LISTING 137: 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 many different variants we offer our users to declare relationships.

As declared in the grammar rule, there are basically three alternative syntaxes which allow the specification of the same customer-supplier relationship. The listings 138, 139 and 140 show a corresponding example in all possible ways. All these declarations are semantically equal and the LocationContext is always supplier/upstream whereas the CargoBookingContext is customer/downstream.

```
1  LocationContext Supplier-Customer CargoBookingContext
2  // or inverse:
3  CargoBookingContext Customer-Supplier LocationContext
```

LISTING 138: Xtext: Customer-Supplier Variant 1

```
1  LocationContext [U,S]->[D,C] CargoBookingContext
2  // or: alternatively, the U and D can be omitted in customer-supplier relationships
3  LocationContext [S]->[C] CargoBookingContext
```

LISTING 139: Xtext: Customer-Supplier Variant 2

```
1  CargoBookingContext [D,C]<-[U,S] LocationContext
2  // or: alternatively, the U and D can be omitted in customer-supplier relationships
3  CargoBookingContext [C]<-[S] LocationContext
```

LISTING 140: Xtext: Customer-Supplier Variant 3

Note that if one of the variants with the arrows (-> or <-) is used, the arrow always points from the supplier (upstream) towards the customer (downstream),

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

reflecting the influence flow [27] (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, since this would be redundant.

The following listings 141, 142 and 143 illustrate the same relationships as before but with upstream and downstream roles.

```
LocationContext[PL] Supplier-Customer [ACL]CargoBookingContext
// or inverse:
CargoBookingContext[ACL] Customer-Supplier [PL]LocationContext
```

LISTING 141: CML: Customer-Supplier Variant 1 with Roles

```
LocationContext [U,S,PL]->[D,C,ACL] CargoBookingContext
// or: alternatively, the U and D can be omitted in customer-supplier relationships
LocationContext [S,PL]->[C,ACL] CargoBookingContext
```

LISTING 142: CML: Customer-Supplier Variant 2 with Roles

```
CargoBookingContext [D,C,ACL]<-[U,S,PL] LocationContext
// or: alternatively, the U and D can be omitted in customer-supplier relationships
CargoBookingContext [C,ACL]<-[S,PL] LocationContext
```

LISTING 143: CML: Customer-Supplier Variant 3 with Roles

Listing 144 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 144: Xtext: UpstreamRole and DownstreamRole

The syntax with the arrows and the abbreviations further allows to place the brackets with the supplier (S)/upstream (U), customer (C)/downstream (D) and relationship roles flexible in front or after the bounded context name. 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. Listing 145 shows all possible placements. However, all four variants are again semantically equal. Note that this flexible bracket placement is not possible for the variant with the long keywords *Customer-Supplier* and *Supplier-Customer*.

```
LocationContext [U,S]->[D,C] VoyagePlanningContext // brackets centered (1)
LocationContext [S]->[C] VoyagePlanningContext // brackets centered (2)

[U,S]LocationContext -> VoyagePlanningContext[D,C] // brackets outside (1)
[S]LocationContext -> VoyagePlanningContext[C] // brackets outside (2)

[U,S]LocationContext -> [D,C]VoyagePlanningContext // both on the left side (1)
[S]LocationContext -> [C]VoyagePlanningContext // both on the left side (2)

LocationContext[U,S] -> VoyagePlanningContext[D,C] // both on the right side (1)
LocationContext[S] -> VoyagePlanningContext[C] // both on the right side (2)
```

LISTING 145: CML: Customer-Supplier 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 146 illustrates an example for a customer-supplier relationship declaration with name.

```
LocationContext [U,S]->[D,C] VoyagePlanningContext : ExampleName
```

LISTING 146: 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 147 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 147: CML: Customer-Supplier Example with Attributes (Body)

The *exposedAggregates* attribute must reference aggregates (see syntax in Section B.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 148 shows the rule specifying the allowed values for the *downstreamRights* attribute.

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

LISTING 148: Xtext: DownstreamGovernanceRights

B.11.1 Customer-Supplier vs. Upstream-Downstream

Note that according to our understanding of the patterns and our semantic model[7] [20] the customer-supplier relationship is a special case of a 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 B.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. In CML you have to specify this explicitely.

B.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.

B.12 Conformist

The Conformist pattern is implemented as a value of the DownstreamRole enum, as shown in Listing 149.

```
1 enum DownstreamRole:
2   ANTICORRUPTION_LAYER = 'ACL' | CONFORMIST = 'CF'
3 ;
```

LISTING 149: Xtext: Conformist implementation

The CONFORMIST (CF) role can be used as a role for the downstream context in any upstream-downstream relationship. Listing 150 illustrates an example.

[7] https://contextmapper.github.io/docs/language-model/

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

LISTING 150: Xtext: Conformist Example

B.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.

B.13 Open Host Service

The Open Host Service pattern is implemented as a value of the UpstreamRole enum, as shown in Listing 151.

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

LISTING 151: CML: Open Host Service implementation

The OPEN_HOST_SERVICE (OHS) role can be used as a role for the upstream context in any upstream-downstream relationship. Listing 152 illustrates an example.

```
CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext {
  implementationTechnology = "SOAP"
  downstreamRights = INFLUENCER
  exposedAggregates = Printing
}
```

LISTING 152: CML: Open Host Service Example

B.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 (DownstreamRole) is not applicable in a customer-supplier relationship.

B.14 Anticorruption Layer

The Anticorruption Layer pattern is implemented as a value of the DownstreamRole enum, as shown in Listing 153.

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

LISTING 153: Xtext: Anticorruption Layer implementation

The ANTICORRUPTION_LAYER (ACL) role can be used as a role for the downstream context in any upstream-dDownstream relationship. Listing 154 illustrates an example.

```
CustomerManagementContext [D,ACL]<-[U,OHS,PL] PrintingContext {
  implementationTechnology = "SOAP"
  downstreamRights = INFLUENCER
  exposedAggregates = Printing
}
```

LISTING 154: CML: Anticorruption Layer Example

B.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.

B.15 Published Language

The Published Language pattern is implemented as a value of the UpstreamRole enum, as shown in Listing 155.

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

LISTING 155: Xtext: Published Language implementation

The PUBLISHED_LANGUAGE (PL) role can be used as a role for the upstream context in any upstream-downstream relationship. Listing 156 illustrates an example.

```
PrintingContext [U,OHS,PL]->[D,ACL] PolicyManagementContext {
  implementationTechnology = "SOAP"
  exposedAggregates = Printing
}
```

LISTING 156: CML: Published Language Example

B.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 157 illustrates the two corresponding grammar rules.

```
BoundedContext:
  'BoundedContext' name=ID (('implements' (implementedSubdomains+=[Subdomain])
    ("," implementedSubdomains+=[Subdomain])*)? & ('realizes'
    (realizedBoundedContexts+=[BoundedContext])
    ("," realizedBoundedContexts+=[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 :
  (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 157: Xtext: Responsibility Layers on Bounded Contexts and Aggregates

The following CML listings 158 and 159 illustrate 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"
}
```

LISTING 158: CML: Responsibility Layers on Bounded Contexts

```
Aggregate Customers {
  responsibilities = "Customers", "Addresses"

  Entity Customer {
    aggregateRoot

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

LISTING 159: CML: Responsibility Layers on Aggregates

B.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 160. The listings 161 and 162 show the two grammar rules for bounded contexts and aggregates, highlighting the corresponding knowledge level attribute.

```
enum KnowledgeLevel :
  META="META" | CONCRETE="CONCRETE"
;
```

LISTING 160: Xtext: KnowledgeLevel enum

```
BoundedContext:
  'BoundedContext' name=ID (('implements' (implementedSubdomains+=[Subdomain])
    ("," implementedSubdomains+=[Subdomain])*)? & ('realizes'
    (realizedBoundedContexts+=[BoundedContext])
    ("," realizedBoundedContexts+=[BoundedContext])*)?)
  (
    OPEN
      (('domainVisionStatement' '=' domainVisionStatement=STRING)? &
      ('type' '=' type=BoundedContextType)? &
```

LISTING 161: Xtext: Knowledge Level on Bounded Contexts and Aggregates (1)

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

Aggregate :
  (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 162: Xtext: Knowledge Level on Bounded Contexts and Aggregates (2)

Listing 163 shows an example on a bounded context and Listing 164 on an aggregate.

```
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  knowledgeLevel = CONCRETE
}
```

LISTING 163: CML: Knowledge Level on Bounded Context

```
Aggregate Customers {
  knowledgeLevel = CONCRETE

  Entity Customer {
    aggregateRoot

    /* ... attributes ... */
  }
}
```

LISTING 164: CML: Knowledge Level on Aggregate

B.18 Aggregate

The Aggregate rule shown in Listing 165 has been added to CML to also support tactic DDD patterns within bounded context. All elements within the aggregates are realized with the Sculptor [34] grammar. Therefore, all other tactic

DDD patterns are not documented here. We refer to the Sculptor project [34] and their documentation[8].

```
Aggregate :
  (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 165: Xtext: Aggregate rule

The aggregate supports the Responsibility Layers pattern (B.16) and the Knowledge Level pattern (B.17) explained in Section B.16 and Section B.17 respectively. As shown in Listing 165 they are specified with the keywords *responsibilities* and *knowledgeLevel*.

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 B.7. The *owner* attribute allows to specify which *TEAM* owns an aggregate. It takes a reference to a bounded context of the type *TEAM*. The compiler ensures that the referenced bounded context has this type. Section B.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 166 illustrates the enum specifying the allowed values for the *likelihoodForChange* attribute.

```
enum LikelihoodForChange :
  NORMAL | RARELY | OFTEN
;
```

LISTING 166: Xtext: LikelihoodForChange enum

An aggregate can further contain Services, Resources, Consumers and SimpleDomainObjects (Entities, Value Objects, Domain Events, etc.) which are not further introduced here. The according rules are defined by the Sculptor [34] DSL, as already mentioned. However, Listing 167 illustrates an example of an aggregate with the explained attributes above and tactic DDD elements in the Sculptor [34] syntax.

[8] `http://sculptorgenerator.org/documentation/`

```
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 167: Xtext: Aggregate Example

B.19 Complete CML Grammar

All previous sections in this language reference have illustrated their corresponding parts of the grammar. The complete CML grammar file in the version *v4.1.1* documented in this report can be found in our source code repository[9].

[9] https://github.com/ContextMapper/context-mapper-dsl/blob/v4.1.1/org.contextmapper.dsl/src/org/contextmapper/dsl/ContextMappingDSL.xtext

List of Figures

List of Tables

List of Abbreviations

ACL Anticorruption Layer. 92, 105

API Application Programming Interface. 38, 58–62, 68

AR Architectural Refactoring. 1–5, 7–19, 21–23, 25, 33, 35–58, 62–69, 88, 113

AST Abstract Syntax Tree. 38

CF Conformist. 92, 105

CLI Command Line Interface. 68

CML Context Mapper DSL. 1–3, 10–13, 15, 19, 21, 22, 25, 29, 33–42, 44, 46, 49, 50, 52, 56–58, 60–69, 87, 88, 93, 96–98, 100, 102, 106, 107, 110, 112, 114

DC Decomposition Criterion. 2, 5–10, 13, 16, 20, 36, 65, 68

DDD Domain-driven Design. 1–4, 14, 15, 18–22, 25, 37, 41, 59, 63, 64, 66, 68, 87, 112, 113

DSL Domain-specific Language. 1, 2, 9, 11, 12, 21, 22, 37, 39, 41, 52, 56–58, 63, 66, 87, 113

EMF Eclipse Modeling Framework. 37, 38

GPL General Public License. 23

IDE Integrated Development Environment. 40, 68

MAP Microservice API Patterns. 2

MDSL Microservices Domain-Specific Language. 2, 58, 60–63, 65, 67, 68

NFR Non-Functional Requirement. 8, 12, 13, 16, 17, 19, 21, 22, 47, 65–67

OHS Open Host Service. 53, 92, 105

PL Published Language. 92, 105

UI User Interface. 9, 50

US User Story. 19

Bibliography

[1] Agile Alliance. *Role-Feature-Reason User Story Template*. https://www.agilealliance.org/glossary/role-feature/. [Online; Accessed: 2019-03-13].

[2] Agile Alliance. *User Stories*. https://www.agilealliance.org/glossary/user-stories. [Online; Accessed: 2019-03-13].

[3] Alberto Brandolini. *Introducing Event Storming*. http://ziobrando.blogspot.com/2013/11/introducing-event-storming.html. [Online; Accessed: 2019-07-17].

[4] Alberto Brandolini. *Strategic Domain Driven Design with Context Mapping*. https://www.infoq.com/articles/ddd-contextmapping. [Online; Accessed: 2019-04-02].

[5] ANTLR. *ANTLR (ANother Tool for Language Recognition)*. https://www.antlr.org/. [Online; Accessed: 2019-08-26].

[6] 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.

[7] Christian Bisig. "Ein werkzeugunterstütztes Knowledge Repository für Architectural Refactoring". MA thesis. Rapperswil: University of Applied Sciences HSR, 2016.

[8] Michael Brandner et al. "Web services-oriented architecture in production in the finance industry". In: *Informatik Spektrum* 27.2 (2004), pp. 136–145. DOI: 10.1007/s00287-004-0380-2. URL: https://doi.org/10.1007/s00287-004-0380-2.

[9] A. Brandolini. *Introducing EventStorming: An act of Deliberate Collective Learning*. Leanpub, 2018.

[10] 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: 2019-08-13].

[11] Context Mapper. *Context Mapper: CML examples repository*. https://github.com/ContextMapper/context-mapper-examples. [Online; Accessed: 2019-08-15].

[12] 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.github.io/. [Online; Accessed: 2019-03-12].

[13] Melvin Conway. *Conway's law*. 1968.

[14] Eclipse Xtext. *Xtext - Language Engineering Made Easy!* https://www.eclipse.org/Xtext/. [Online; Accessed: 2019-08-20].

[15] 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.

[16] 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.

[17] 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.

[18] Michael Gysel et al. *Service Cutter Coupling Criteria Catalog*. https://github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria. [Online; Accessed: 2019-05-28].

[19] Moritz Habegger and Micha Schena. "Cloud-Native Refactoring in a mHealth Scenario". Bachelor Thesis. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2019.

[20] 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.

[21] Stefan Kapferer. "Model Transformations for DSL Processing". https://stefan.kapferer.ch/model-transformations-for-dsl-processing. Seminar Paper. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2018.

[22] Stefan Kapferer. *Project Definition: Service Decomposition as a Series of Architectural Refactorings*. 2019.

[23] Stefan Kapferer and Samuel Jost. "Attributbasierte Autorisierung in einer Branchenlösung für das Versicherungswesen". https://eprints.hsr.ch/602/. Bachelor Thesis. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2017.

[24] 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.

[25] Margaret Rouse. *Business Capability Definition*. https://searchapparchitecture.techtarget.com/definition/business-capability. [Online; Accessed: 2019-08-25].

[26] R.C. Martin et al. *Agile Software Development: Principles, Patterns, and Practices*. Alan Apt series. Pearson Education, 2003. ISBN: 9780135974445.

[27] Michael Plöd. *DDD Context Maps - an enhanced view*. https://speakerdeck.com/mploed/context-maps-an-enhanced-view. [Online; Accessed: 2018-12-16].

[28] 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: 2019-08-13].

[29] 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: 2019-08-09].

[30] OpenAPI Initiative. *OpenAPI*. https://www.openapis.org/. [Online; Accessed: 2019-08-26].

[31] 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.

[32] plantuml.com. *Open-source tool that uses simple textual descriptions to draw UML diagrams*. http://plantuml.com/. [Online; Accessed: 2019-08-26].

[33] M. Plöd. *Hands-on Domain-driven Design - by example*. Leanpub, 2019.

[34] Sculptor Project. *Sculptor - Generating Java code from DDD-inspired textual DSL*. http://sculptorgenerator.org/. [Online; Accessed: 2019-08-22].

[35] 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.

[36] D. Steinberg et al. *EMF: Eclipse Modeling Framework*. Eclipse Series. Pearson Education, 2008. ISBN: 9780132702218.

[37] 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.

[38] The Eclipse Foundation. *Henshin - A state-of-the-art Model Transformation Language for the Eclipse Modeling Framework*. https://www.eclipse.org/henshin/. [Online; Accessed: 2019-08-20].

[39] N. Tune and S. Millett. *Designing Autonomous Teams and Services: Deliver Continuous Business Value Through Organizational Alignment*. O'Reilly Media, 2017.

[40] 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.

[41] Vaughn Vernon. *Implementing Domain-Driven Design*. 1st. Addison-Wesley Professional, 2013. ISBN: 0321834577, 9780321834577.

[42] 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: 2019-05-27].

[43] 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.

[44] Olaf Zimmermann et al. *Microservice API Patterns*. https://microservice-api-patterns.org. [Online; Accessed: 2019-05-27].