UNIVERSITY OF APPLIED SCIENCES OF EASTERN SWITZERLAND (HSR FHO)

TERM PROJECT

A Domain-specific Language for Service Decomposition

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

December 20, 2018

Declaration of Authorship

I, Stefan KAPFERER, declare that this thesis titled, "A Domain-specific Language for Service Decomposition" and the work presented in it are my own. I confirm that:

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

Rapperswil, December 20, 2018

Stefan Kapferer

UNIVERSITY OF APPLIED SCIENCES OF
EASTERN SWITZERLAND (HSR FHO)

Abstract

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

A Domain-specific Language for Service Decomposition

by Stefan KAPFERER

Microservices have gained a huge attention in the industry and in the academic field over the last years. Companies are adopting microservice architectures in order to increase agility, maintainability and scalability of their software. At the same time, decomposing an application into appropriately sized services is challenging. With its strategic patterns and especially the Bounded Contexts, Domain-driven Design (DDD) provides an approach for decomposing a domain into multiple independently deployable services. However, existing modeling tools supporting DDD mainly focus on the tactical patterns. Not many approaches to a formal definition of the strategic patterns exist and there are different interpretations and opinions regarding their applicability.

This project presents a Domain-specific Language (DSL) based on the strategic DDD patterns. The model behind the language and its semantic rules aim to provide one concise interpretation of the patterns and how they can be combined. The DSL concept offers a tool to model a system in an expressive way, using the DDD language. With the implemented Service Cutter integration we further provide a proof of concept showing how the DSL can be used as input for structured service decomposition approaches. The presented results and our evaluation of this approach illustrate the capabilities of DDD-based models towards service decomposition. To convert the DSL-based models into a graphical representation, the developed tool offers an additional transformation to create PlantUML diagrams.

The DSL is meant to provide a foundation for other service decomposition approaches. Future projects may propose architectural refactorings for the DSL based on model transformations. Other approaches based on algorithms and heuristics similar to Service Cutter could be applied as well. A code generator to create microservice project templates for the modeled Bounded Contexts might be another promising feature for the future.

Acknowledgements

I would like to thank my project advisor Prof. Dr. Olaf Zimmermann who provided insight and expertise which greatly supported this project. The weekly discussions were of great help and I always appreciated his guidance and assistance.

Contents

Chapter 1

Introduction

1.1 Context

Domain-driven Design (DDD) has been introduced by Eric Evans in his book *Domain-Driven Design: Tackling Complexity in the Heart of Software* [13] more than 10 years ago. With the microservices trend during the last few years, Evans work has gained even more attention again. The decomposition of an application into appropriate sized services is challenging. Achieving high cohesion and loose coupling between the service boundaries is crucial to keep the application scalable and maintainable. DDD plays a key role here. With its strategic bounded contexts and the tactic aggregates it provides an approach for decomposing a domain into multiple components. However, especially regarding the strategic DDD patterns a certain ambiguity and different interpretations of how they shall be applied exists. Existing modeling tools which help software architects and engineers expressing these concepts mainly focus on the tactical DDD patterns.

1.2 Vision

This work proposes a modeling tool based on strategic DDD patterns. Thereby, we strive for a meta-model providing a concise interpretation of these patterns and their applicability, avoiding ambiguity. The created models shall further be used as input for structured service decomposition approaches, such as Service Cutter [20]. The concept of Domain-specific Languages (DSLs) allow the creation of models in an expressive way, adapted to the common language of a domain. Further, models can be processed and transformed into other languages and representations.

With this project, we provide a DSL to model systems based on DDD patterns. The proposed meta-model behind the language and its semantics reflect our interpretation of the strategic DDD patterns and how they can be combined. With the implemented Service Cutter [20] integration we intend to present a proof of concept showing how such DDD-based models can be used to decompose services. The DSL shall further provide a foundation for other structured service decomposition approaches developed by future projects.

Another DSL processing application which may be useful in the context of a modeling tool, is the convertion of a model into a graphical representation. The proof of concept implemented by this project converts the DSL models into Unified Modelling Language (UML) diagrams using PlantUML [42].

1.3 Related Work

Decomposing monolithic systems into microservice architectures [56] is a topic with a huge attention within the last years not only in the industry but in the academic field as well [5, 17, 19, 21, 26, 34, 40]. Furthermore, DDD with its bounded contexts is said to be a promising approach facing this challenging task [17, 26, 30, 34, 41, 43]. However, there are not many tools which support modeling and specifying a system formally in terms of the strategic DDD patterns in order to decompose it in a structured manner.

Rademacher [44] presents a formal modeling language based on UML. The UML profile which extends meta-classes with stereotypes for DDD patterns shall be used for modeling microservice architectures. They further aim to derive microservices code from their UML models in future projects. However, the profile seems to focus on modeling bounded contexts with the tactical DDD patterns. The strategic patterns concerning the relationships between bounded contexts are not mentioned explicitly.

Le et al. [32] propose a DDD approach using meta-attributes to capture domain-specific requirements. The meta-attributes are implemented as Java annotations. Their aim is to overcome gaps between different domain models of different stakeholders such as domain experts, designers and programmers. This approach mainly aims to support the software designing process on a tactical level as well. Furthermore, it differs from our approach in the sense that it does not explicitely expresses DDD patterns.

A few projects implementing DSLs based on DDD patterns exist, such as Sculptor [46], fuin.org's DDD DSL [18] and DSL Platform [11]. Further approaches and projects based on annotations exist as well. However, they all have in common that they focus on the tactic DDD patterns and do not cover the strategic patterns concerning the relationships between bounded contexts.

Graphical representations of context maps and the strategic DDD patterns were introduced by Brandolini [3] and Vernon [52]. Plöd [35] further presented a proposal for a formal notation of context maps. The graphical examples within this project are inspired by these approaches.

Chapter 2

Domain-driven Design Analysis

This chapter will introduce the concepts and patterns on which the proposed Domain-specific Language (DSL) relies. It presents our understanding of how the strategic and tactic Domain-driven Design (DDD) patterns can be used and combined. The interpretations are derived from the DDD literature of Evans [13] and Vernon [52], personal professional experience [27], the inputs given by the supervisor of this project and other literature of experts, such as Brandolini [3] and Plöd [36]. The DSL shall offer the possibility to create models of applications in terms of DDD patterns and thereby strengthen the semantics and meanings of the patterns, which is foreseen to be one of the scientific contributions of this work. From the perspective of designing a DSL, DDD can be referred to as the «Meta Model» or «Semantic Model» [16] of the language.

Having a clear understanding of the semantics of the DSL to be developed and thus, a solid knowledge of the «Semantic Model» is of crucial importance. Therefore, the following sections aim to give a definition of this model as we understand it, as precise as possible.

2.1 Strategic Patterns

Since this projects goal is providing a foundation for structured service decomposition approaches, it focuses mainly on the strategic patterns of DDD. The DSL has to support the tactic patterns as well, but they will not be specifically introduced. The interpretations of a few strategic DDD patterns seem to be somehow ambiguous and used differently by different authors. Within this chapter decisions for particular interpretations were made, based on personal experience [27] and the literature [13, 14, 52]. These decisions and pattern descriptions within this section explain and justify the grammar of the designed DSL.

2.1.1 Pattern Overview & Implementation Decisions

The following sections list all strategic patterns and concepts mentioned in [13] and [14]. It further documents our decisions wether the patterns are implemented within the DSL or not. The list is ordered by importance for our DSL and thus, the patterns not covered are listed at the end. The decisions are justified briefly. This overview is followed by more detailed descriptions of our understanding of the patterns used for the DSL.

Context Map

A model describing bounded contexts and especially their relationships. Brandolini [3] provides a very good introduction into context mapping.

Decision: Yes, this pattern is covered by our DSL. Context mapping is the central feature which should be provided by the language.

Bounded Context

A context with an explicit boundary within which a particular domain model, implementing parts of subdomains, applies.

Decision: Covered. Bounded contexts are the essential concept needed towards service decomposition approaches.

Partnership

The partnership[1] pattern describes an intimate relationship between two bounded contexts. Their domain models somehow relate and have to be evolved together.

Decision: Covered. Modeling the relationships between bounded contexts is central within our DSL.

Shared Kernel

Desribes an intimate relationship between two bounded contexts which share a common part of the domain model and manage it as a common library.

Decision: Covered. Modeling the relationships between bounded contexts is central within our DSL.

Customer/Supplier

Describes a relationship where one bounded context is customer and the other supplier which work closely together. The supplier prioritizes the implementation with respect to the customers requirements.

Decision: Covered. Modeling the relationships between bounded contexts is central within our DSL.

Conformist

Describes a role of a bounded context in an Upstream/Downstream relationship. Since there is no influence on the upstream, the downstream team has to deal with what they get and «conform» to it.

Decision: Covered. Modeling the relationships between bounded contexts is central within our DSL.

[1] Not mentioned in the original DDD book of Evans [13], but in the later published DDD reference [14].

Open Host Service

Describes a role of a bounded context which is providing certain functionality needed by many other contexts. Because of the broad usage, a public API is provided.

Decision: Covered. Modeling the relationships between bounded contexts is central within our DSL.

Anticorruption Layer

Describes a mechanism used by downstreams in order to protect themselves from changes of the upstream.

Decision: Covered. Modeling the relationships between bounded contexts is central within our DSL.

Published Language

The published language describes the shared knowledge two bounded contexts need for their interaction. Typically defined by the upstream providing an Open Host Service.

Decision: Covered. Modeling the relationships between bounded contexts is central within our DSL.

Subdomain (Core, Supporting, Generic)

A subdomain is a part of the domain. Regarding subdomains we differentiate between Core Domains, Supporting Subdomains and Generic Subdomains. A bounded context implements parts of one or multiple subdomains.

Decision: Covered, since it describes which parts of a domain are implemented within a bounded context it might be of interest for users of our DSL.

Domain Vision Statement

A domain vision statement provides a short description of the core domain and its value (value proposition).

Decision: Covered. The DSL provides an optional possibility to add such a short description to bounded contexts and subdomains.

Responsibility Layers

Recommends assigning responsibilities to each domain object, aggregate and module. The responsibilities of all these objects should fit within the responsibility of one layer.

Decision: Covered. The DSL provides an optional possibility to assign bounded contexts, aggregates and modules its responsibilities.

Knowledge Level

A pattern advising the use of two groups of objects (different levels), one very concrete, the other reflecting rules and knowledge that a user is able to customize. Should be used to avoid that the same objects are used in different

ways in different situations. Duplication of such classes should be avoided as well.

Decision: Covered. The DSL offers a simple attribute (enum) to differ between *concrete* and *meta* knowledge level. This attribute is available for bounded contexts as well as aggregates.

Domain

The domain[2] describes the world within your organization is working.

Decision: Implicitly covered. There is no need to explicitely model the concept of a domain within the DSL, since a corresponding model instance as a whole represents the domain.

Segregated Core

Core concepts should be separated from supporting concepts so that designers and developers can clearly see the most important relationships within the core domain.

Decision: Not covered. Separating core domains from supporting domains is a practical pattern which should be reflected by the resulting model but does not need a special language concept.

Abstract Core

If there is a lot of communication between subdomains in separate modules, an abstract model (factor out abstract classes and interfaces) expressing the interactions should be implemtented in a separate module. The specialized, detailed implementations are left in their own module.

Decision: Not covered. From our understanding this is more a practice pattern and it further seems to affect the tactical and not strategical patterns.

Highlighted Core

This pattern advices to either write a brief document that describes the core domain or to flag the elements within the model which are part of the core domain.

Decision: Not covered. Since the DSL will provide the possibility to map a bounded context to the subdomains it implements and a subdomain can be a core domain, this information already exists. However, it can be added to the DSL later if needed.

Cohesive Mechanisms

If the design is getting complex and the implementation of the «what» is swamped by the technical, mechanistic «how», such technical mechanisms should be partitioned into separate generic subdomains in form of libraries or lightweight frameworks.

[2]This is not a pattern in the original DDD book of Evans, but still a very important term related to the patterns *Subdomain* and *Bounded Context* as described by Vernon [52].

Decision: Not covered. This is more a practice pattern than a structural one. Applying the pattern has an influence on how the subdomains and bounded contexts are separated but does not need a specific language concept. But it may be a candidate for a refactoring (splitting bounded contexts) in the next project.

Continuous Integration

Evans [13, 14] mentions this pattern to keep a bounded context sound within itself using a process which integrates changes frequenty with automated tests.

Decision: Not covered. This pattern can be seen as an agile practice which is more a process pattern rather then a structural pattern. Since our DSL expresses structure, there is no need to represent this pattern explicitely.

Separate Ways

Describes the situation where two bounded contexts have no relationship with each other.

Decision: Not covered. There is no need to explicitely model this pattern in the DSL since it is applied by simply do not add a relationship between two bounded contexts.

Big Ball of Mud

There might be situations where it does not make sense to create separate bounded contexts in systems where there simply are no boundaries. In such a case it might be better to draw a boundary arround the whole system and call it a «Big Ball of Mud»[3] [15].

Decision: Not covered. There is no language concept needed to represent this pattern, since it can be realized by simply model one single bounded context.

Evolving Order

This pattern advices to evolve large-scale structures with the application avoiding over-constraining the design and model decisions.

Decision: Not covered. This pattern manifests itself in the way the DSL is used but has no influence to the structure of the language.

System Metaphor

This is a practice originating from Extreme Programming (XP), recommending the use of analogies or metaphors capturing the teams imagination and thinking. The metaphor becomes part of the Ubiquitous Language, describing a part of the system.

Decision: Not covered. This pattern is realized by giving the bounded contexts corresponding names. It describes a «best practice» regarding the used names within the DSL.

[3] Not mentioned in the original DDD book of Evans [13], but in the later published DDD reference [14]. However, the term was introduced by Brian Foote and Joseph Yoder [15] originally.

To summarize the scope of our DSL, the following Table 2.1 lists all patterns which are decided to be part of the language, grouped by the categorization of Evans DDD Reference [14].

TABLE 2.1: Implemented patterns

Category	**Patterns**
Putting the **Model** to Work	Bounded Context
Context Mapping for Strategic Design	Context Map, Partnership, Shared Kernel, Customer/Supplier, Conformist, Anticorruption Layer, Open Host Service, Published Language
Distillation for Strategic Design	Subdomain, Domain Vision Statement
Large-scale Structure for Strategic Design	Responsibility Layers, Knowledge Level

Having decided which patterns are part of the DSL, the next section presents our strategic DDD domain model which illustrates our interpretation of the relationships between all these patterns.

2.1.2 Domain Model

Figure 2.1 shows the domain model on which our DSL is based on. It connects the strategic DDD patterns and proposes certain semantics regarding how they can be combined or not. Note that the model contains a few concepts such as *context map types* and *context types* which will be introduces later in this chapter.

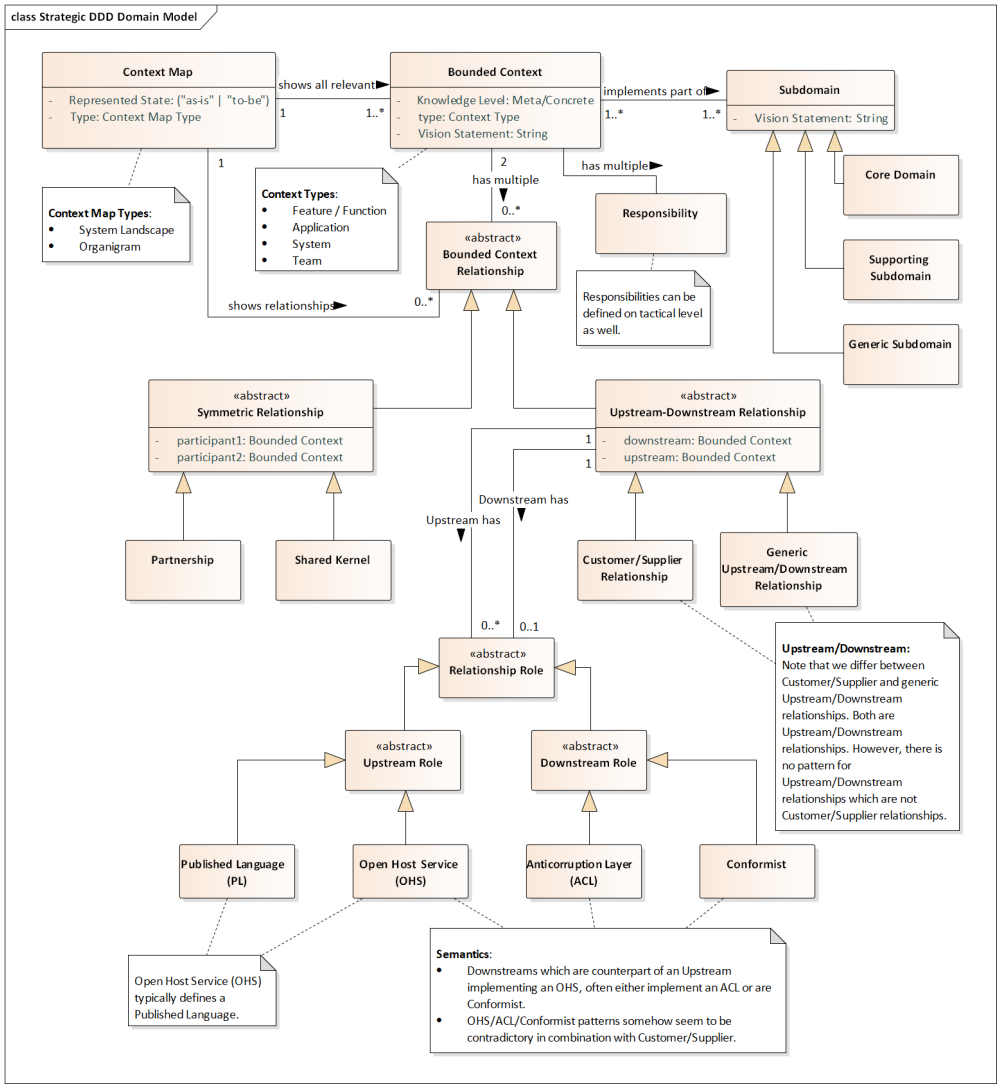

FIGURE 2.1: Strategic DDD Meta-Model (Domain Model)

The following definitions attempt to describe our understanding of the patterns and the domain model shown in Figure 2.1. They are based on the literature of Evans [13, 14] and Vernon [52]. These descriptions summarize our interpretation of the patterns and justify the given domain model and the semantics it implies. We further want to give an insight how context maps and bounded contexts may evolve [3], since this concerns future work towards service decomposition based on the provided model and the DSL.

2.1.3 Bounded Context and Context Map

A bounded[4] context defines an explicit boundary within which a particular domain model applies. It is a boundary in terms of team organization, physical manifestations such as code bases and database schemas as well. The team defines a domain model, expressing a «Ubiquitous Language», which has to stay strictly consistent within this bounds, but without caring if the domain model can be applied to other contexts. However, they still need to have an understanding how their context relates to other contexts, since necessary integrations with those other contexts will require translations.

> «Treat bounded contexts like the borders of a country. Nothing should pass into the bounded context unless it goes through the border control and is valid.», BOUNDED CONTEXTS = BORDER CONTROL, Millet [38]

Finding the right borders and thus the bounded contexts is not trivial. Context mapping may provide a powerful tool evolving those. This technique and some indicators which give you hints that different contexts might be in play are proposed by Brandolini [3].

The context map provides a global view over all bounded contexts which are related to the one you or your team is working on. It describes the points of interaction between bounded contexts and outlines how the translation and communication is done. Brandolini [3] explaines four possible reasons why you might be forced to create a new bounded context. These possible indicators which will lead us to proposals for architectural refactorings [55] are explained step by step in the following paragraphs. To explain the concepts a fictitious insurance company scenario is used. Note that the example is a combination of personal experience [27] in this sector and inspiration provided by the «Lakeside Mutual» project [37]. Besides, the example is used throughout this work to explain concepts and the DSL.

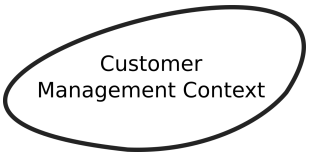

FIGURE 2.2: Context Mapping First Step: The whole system

We start with the most simple context map just containing one bounded context and we assume that we are working on the «Customer Management» part of our insurance company software (Figure 2.2).

The first example of an indication that you need a separate bounded context is having the «same term with different meanings» multiple times. Assume the development team is implementing a new «Customer Self-Service» frontend, allowing the insurance customers to change their address online. For this purpose you need «Accounts» for the customers, or lets say users. However you realize that the term «Account» is already used within your system for storing

[4] «Bounded» within this context means the adjective with similar meaning to «limited» or «confined» («begrenzt» in German) and not the past form of the verb «to bound».

data about the customers bank accounts in case of refunds. This is a typical case of two different domains and therefore two different bounded contexts. Figure 2.3 shows the introduced bounded context within our context map.

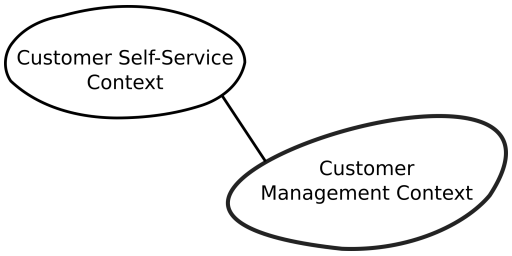

FIGURE 2.3: Context Mapping: «Self-Service» Context

Another reason for splitting bounded contexts described by [3] is «same concept, different use». Imagine your team is implementing the part of your insurance software dealing with contracts and policies. A contract or policy somehow has to relate to a customer, of course. But should you really reuse the already exising «Customer» entity? Of course it would be a possible solution but somehow you are not satisfied since you realize that you do not need the data and methods on the existing customer entity, even if its conceptually the same object.

In this case it might make sense to have two different bounded contexts with their own representation of a customer. Figure 2.4 shows a corresponding solution applied to our context map.

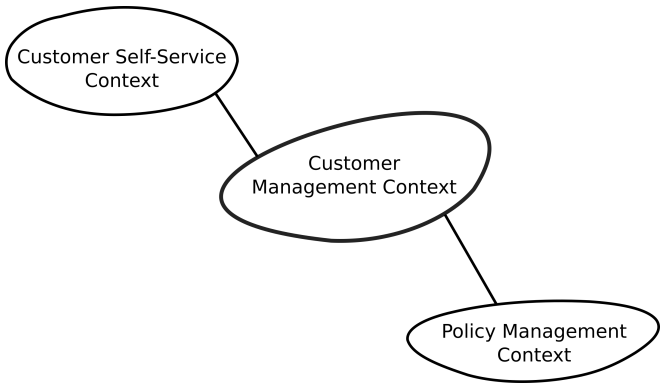

FIGURE 2.4: Context Mapping: «Policy Management» Context

External systems build bounded contexts as well. For example it might be reasonable to use an external printing solution for documents and bills which have to be sent to the customer. Such a context might be also used by multiple contexts, as you can see in our new context map in Figure 2.5.

Last but not least, scaling up your organization can be a major motivation to create new bounded contexts. As your company grows and development teams get bigger you are probably forced to split bounded contexts to build separate teams working on isolated domains.

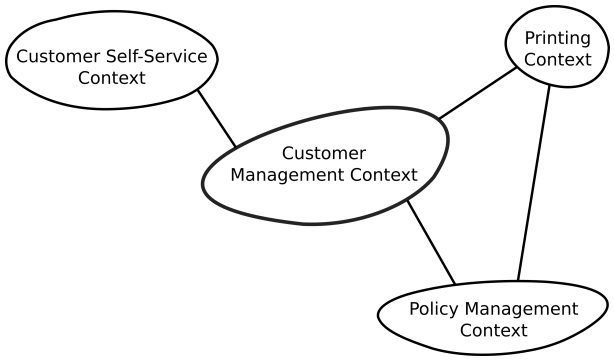

FIGURE 2.5: Context Mapping: «Printing» Context

Figure 2.6 shows an example for such a case. Policy management includes calculating risks regarding your customers. Assume this risk management part of your «Policy Management» context gets complex and you decide to build a new team dealing with it.

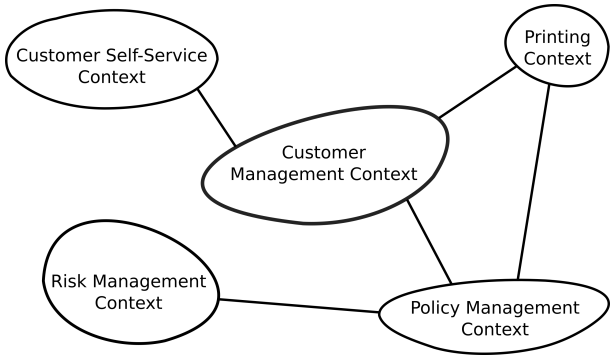

FIGURE 2.6: Context Mapping: «Risk Management» Context

Besides providing a definition of context maps and bounded contexts, this context mapping process is explained since it describes a potential service decomposition approach which may be implemented in a future project using our DSL. The single steps described above lead us to the following Architectural Refactorings (ARs) [55] in table 2.2.

TABLE 2.2: Splitting bounded contexts as Architectural Refactorings [55]

#	**Architectural Refactoring**	**Reason for Split**
1	Add another bounded context to resolve ambiguity.	Domain model has to be split since a term with different meanings is present.
2	Split context to reduce domain model complexity.	If domain objects are used in many different ways the complexity raises and it may be better to have different objects for the different usages. (Single Responsibility Principle)

TABLE 2.2: Splitting bounded contexts as ARs [55] (continued)

#	Architectural Refactoring	Reason for Split
3	Add additional bounded context for new external system.	A certain functionality within a system may get complex and if it is part of the core domain, it might be replaced with an external system. Such an external system represents a new bounded context.
4	Split contexts by domain concept for scaling reasons.	In order to scale your organization, you may split bounded contexts by domain concepts which were solved in one bounded context so far.

After this introduction into the patterns bounded context and context map, the next sections will discuss subdomains and the relationships between bounded contexts.

2.1.4 Subdomains

The domain describes the world within an organization is working or what the organization does. Banking, Insurance or IT in a broad sense, can be seen as examples for domains. Further, every domain has its subdomains. For example in insurance companies you will find subdomains like Customer Management, Contract / Policy Management or Claim Management.

Subdomains are differentiated by the three specializations Core Domain, Supporting Domain and Generic Subdomain. A Core Domain covers a subdomain which is very important for the success of your software or the company working with it. If a subdomain is still essential and somehow related to the business but not part of the core, it is a Supporting Domain. A Generic Subdomain is still needed for the whole system to be successful but it is not directly related to the core and the main business of the software.

A bounded context may implement one specific or parts of a subdomain. But it is also possible that a bounded context implements multiple parts of different subdomains, as we illustrated in our domain model in Figure 2.1.

2.1.5 Domain Vision Statement

Especially in the beginning of a project it is important to have a vision to focus the development. However, having a description of the most important values of a system can be useful during the whole lifetime of a project. A Domain Vision Statement is a short text describing the core domain and its value.

Within our domain model, the Domain Vision Statement is realized as an attribute of a bounded context or subdomain.

2.1.6 Relationships between Bounded Contexts

As already seen in the domain model in Figure 2.1, bounded contexts have different kinds of relationships with each other. The next sections will describe each of those relationships and illustrate them within our insurance example.

Symmetric vs. Asymmetric & Upstream vs. Downstream Terminology

Note that we differentiate between symmetric and asymmetric relationships. In asymmetric relationships we use the terms Upstream and Downstream, as Evans [13] and Vernon [52] do it in their books. To clarify this terminology, the upstream is always influencing the downstream. In other words, the downstream depends on functionality or features of the upstream, whereas the upstream is not influenced by the model of the downstream. Thus, the upstream is the bounded context which provides some functionality, for example by implementing the Open Host Service and/or Published Language pattern, whereas the downstream consumes this functionality and often implements the Anti-corruption Layer or Conformist pattern. These asymmetric relationships are called Upstream/Downstream relationships, as long it is not a Customer/ Supplier relationship. A Customer/Supplier relationship is a special case of an Upstream/Downstream relationship, where the Upstream is called Supplier and the Downstream is called Customer.

The Shared Kernel and Partnership patterns in comparison, describe very intimate relationships where no clear Upstream and Downstream can be defined, which is why we call them symmetric relationships.

Shared Kernel

A Shared Kernel provides a solution for teams which work on two bounded contexts which are very closely related. If the two domain models overlap very much, permanently implementing translations at the boundaries of the contexts might cause more work than simply, continuously integrate the same domain model. The Shared Kernel contains the common parts of the domain model which are maintained by both teams, working closely together. Keeping the Shared Kernel as small as possible and defining a continuous integration process is very important for success here.

In order to have a representative example which contains all relationship patterns, we extend the insurance example with another bounded context. If a customer has a contract, he will also receive bills. We assume that the whole debt collection management was implemented within the policy management context, but due to complexity the insurance company decides to form a separate bounded context called «Debt Collection».

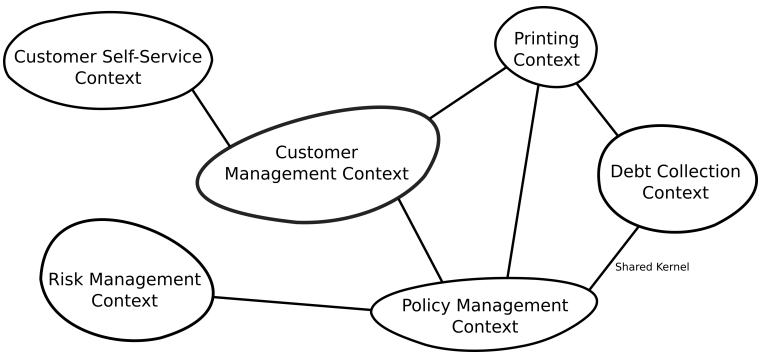

FIGURE 2.7: «Policy Management» Context & «Debt Collection» Context Shared Kernel Example

Since their domain model still overlaps substantially, they decide to implement a Shared Kernel. Thus, we get a new context map for our scenario, which is illustrated by Figure 2.7. Note that the debt collection context depends on the printing context as well, since bills have to be printed. Shared Kernels are typically implemented as libraries which are used by both bounded contexts.

Partnership

As the Shared Kernel did, a Partnership indicates a very intimate relationship between two bounded contexts. Nevertheless, a Partnership is technically not that close as a Shared Kernel. Two teams in a Partnership have to work very closely together on an organizational level. Their bounded contexts are somehow related and have to evolve together. The resulting product of the two bounded contexts can only fail or success as a whole. Thus, the two teams coordinate and plan their developments «in-sync» with each other.

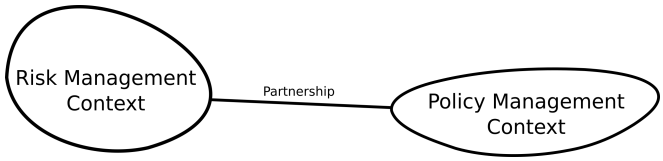

FIGURE 2.8: «Risk Management» Context & «Policy Management» Context Partnership Example

The «Risk Management» and «Policy Management» contexts in our example represent a valid scenario for such a relationship (Figure 2.8).

Customer/Supplier

As already mentioned, in a Customer/Supplier relationship the Customer is the downstream and the Supplier the upstream. However, if the upstream in a Upstream/Downstream relationship succeeds interdependently with the downstream, the needs of the downstream become relevant for the upstream and they may form a Customer/Supplier relationship.

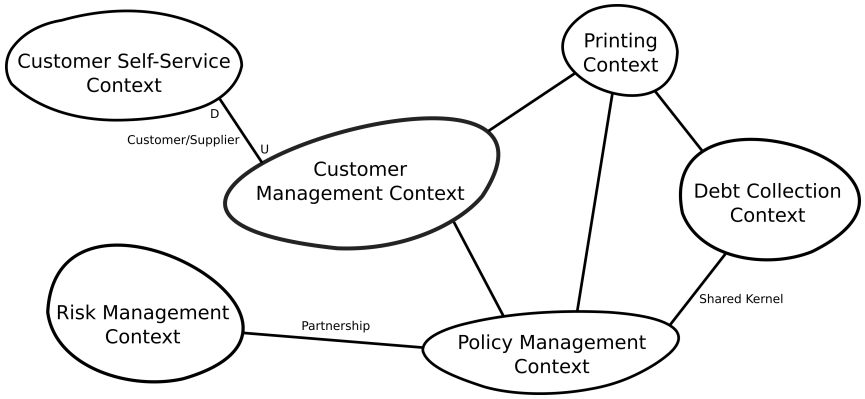

FIGURE 2.9: «Customer Management» Context & «Customer Self-Service» Context Customer/Supplier Example

In comparison to a generic Upstream/Downstream relationship, the supplier cares about the customer and his requirements. This means that the planning in the Supplier team is done with respect to the priorities of the Customer team.

Within our insurance example, the relationship between the customer self-service context and the customer management context seems to be a plausible scenario for such a relationship. The self-service context represents the customer which needs the functionalities of the customer management context. However, the customer management context may be also interested that the self-service application for the customers always succeeds. Therefore the two contexts form a Customer/Supplier relationship, as visualized in Figure 2.9.

Inspired by Brandolini [3] and Vernon [52], the illustrations of context maps within this work use a «**U**» and a «**D**» to indicate which bounded context is the **U**pstream and which one is the **D**ownstream within a relationship.

Open Host Service (OHS)

If a team is in a upstream role and has to provide its service to multiple downstreams, integrating with all downstreams separately may be get too expensive. Thus, the team may decide to implement an Open Host Service (OHS), which is a protocol that gives all the downstreams a unified access to the services. It is an public Application Programming Interface (API), all other contexts can use if they need it.

In our insurance example, the printing context clearly represents such a situation. It provides the same feature for multiple other bounded contexts and therefore it makes perfectly sense to provide one API/OHS for all downstreams. Maybe another example might be the customer management context. Even if our example is too small to illustrate it, but the customer management context is typically a very central context in an insurance scenario which is used by many other contexts. Applying the mentioned OHS cases to our context map, results in the map illustrated by Figure 2.10.

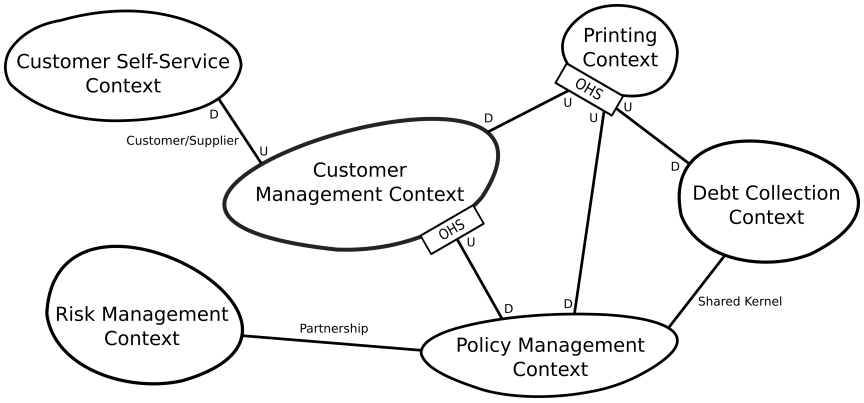

FIGURE 2.10: «Open Host Service» Examples

According to our understanding and our strategic DDD model from Figure 2.1, another remark concerning this pattern has to be mentioned, since this is not explicitly defined in the DDD literature and thus may be new to the reader. The OHS pattern is not applicable within a Customer/Supplier relationship,

since their pattern descriptions are contradictory. Whereas the supplier context in a Customer/Supplier relationship implements its services according to the customers specific needs, an upstream implementing the OHS pattern implements a «one for all» solution, without caring about the needs of a single downstream. Thus, according to our interpretation of these DDD patterns, the OHS pattern only fits to Upstream/Downstream relationships which are not Customer/Supplier relationships.

Published Language (PL)

Communication between two bounded contexts needs a common language, since both contexts have their own ubiquitous language which differ from each other. Thus, a so-called Published Language has to be defined which is always used for translation. This patterns is often used together with Open Host Service (OHS), since the context providing the OHS mostly defines the Published Language (PL) as well.

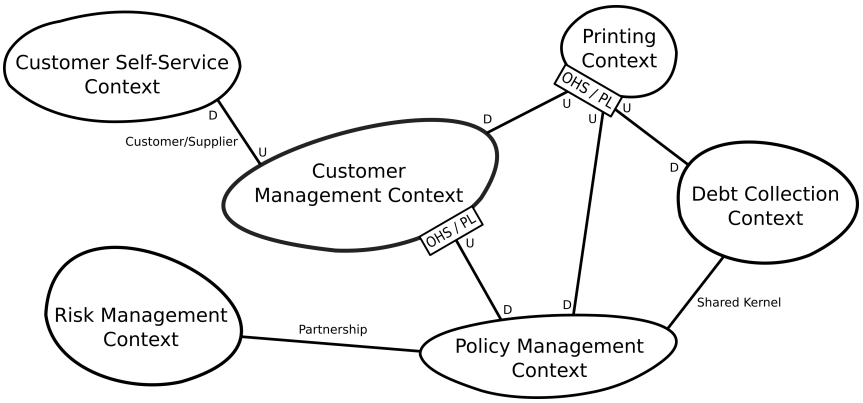

FIGURE 2.11: «Published Language» Examples

Conformist

Sometimes two bounded contexts are in an Upstream/Downstream relationship and the upstream does not care about the needs of the downstreams. In such a case the upstream context just defines which services it provides and how they are implemented. The downstream context simply has to deal with what it gets. If a downstream team needs those services and decides to use the services, they maybe decide to just conform with the language published by the upstream. Thus, their role in the relationship is called «Conformist». They always adapt according to the changes of the upstream context.

According to our interpretation, this pattern is not applicable in a Customer/ Supplier relationship, since the pattern mentions that the upstream does not care about the needs of the downstream. Further, we will see in the next section that a downstream either implements the Conformist pattern or the Anticorruption Layer but not both. The Anticorruption Layer can be seen as the inverse pattern of the Conformist for situations where a downstream has to deal with an upstream who does not care about the downstream needs.

To apply this pattern to our insurance example, we assume that the customer management context provides an OHS without caring about the needs of the downstreams.

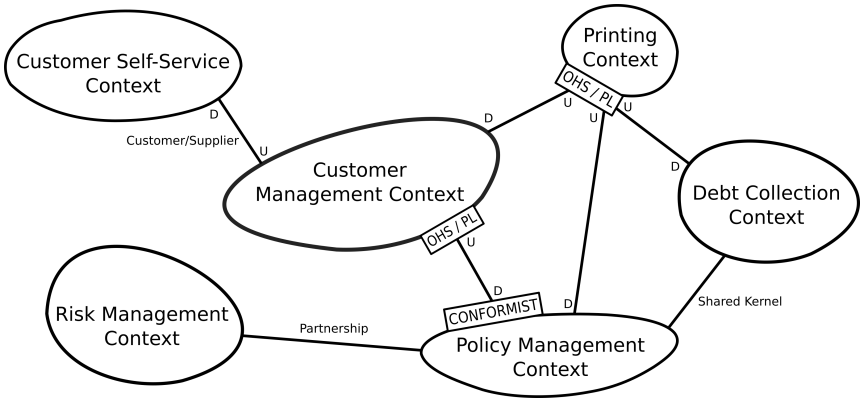

FIGURE 2.12: «Conformist» Examples

Thus, the policy management context team may decide to conform to this API which leads to the context map in Figure 2.12.

Anticorruption Layer (ACL)

Translation between bounded contexts can become difficult if the teams are not in a Partnership, Shared Kernel or Customer/Supplier relationship. The upstream is somehow dictating how the communication has to be done and the downstream team might want to protect itself from breaking changes of the upstream as good as possible. An Anticorruption Layer isolates the upstream model from the downstreams own model. Thus, the downstream team is able to use the other system in terms of their own domain model, since the Anticorruption Layer (ACL) makes the translation. Changes in the upstream do no longer affect the downstream domain model. As already mentioned in the last section, this is somehow the opposite or inverse pattern to the Conformist pattern. Thus, they can not be applied both in one relationship. Further, it is less likely that this pattern is used in combination with Customer/Supplier, at least according to our interpretation of the patterns.

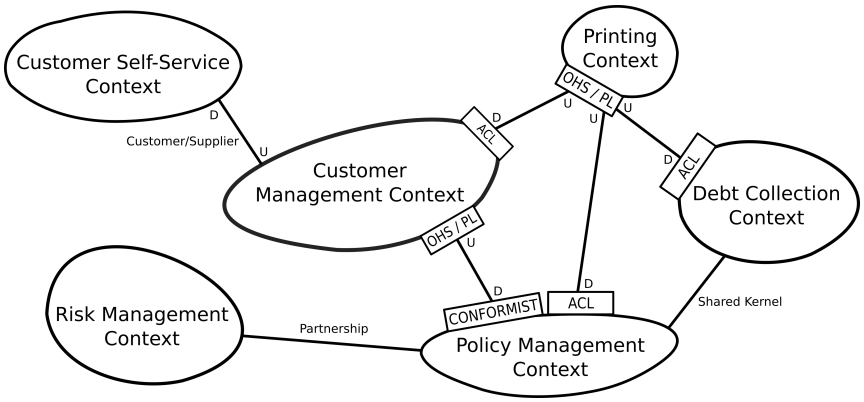

FIGURE 2.13: «Anticorruption Layer» Examples

In a Customer/Supplier relationship, the customer is not «dictated» by the supplier and therefore has no motivation to protect itself. However, the downstream context may implement a translation layer anyway, but since it has not a defensive purpose we would not call it Anticorruption Layer. Nevertheless, the DSL implementation shall allow this combination.

This pattern might be used by our customer management, policy management, and debt collection contexts to protect themselves from the changes of the printing context (Figure 2.13).

All relationship patterns have been presented now. Note that Figure 2.13 illustrates the final context map for the insurance example which will be used in Chapter 4 again to explain the implemented DSL.

2.1.7 Responsibility Layers

This pattern addresses Responsibility-driven Design which can be applied on different levels. As programmers we give classes and objects responsibilities and try to avoid implementing multiple responsibilities within the same classes according to the single responsibility principle. This idea of splitting things by responsibilities can be applied on higher levels such as aggregates, modules and bounded contexts as well.

2.1.8 Knowledge Level

When roles and relationships between entities vary in different situations, the complexity of the software can explode. Often neither full generality nor a highly customizable systems are the perfect solution. The concept of having concrete and abstract objects is familiar to a programmer and often provides a solution for keeping the complexity in such situations under control. Similar to the last pattern, this concept can be applied to higher levels as well.

This pattern suggests to create two levels of objects, one concrete, the other reflecting the rules and knowledge that a user or super-user is able to customize.

2.2 Additional Concepts & Features

In addition to the DDD patterns a few other features were added to the DSL. These additional features are our own input and proposals based on experience.

2.2.1 «FAST» Context Types

Bounded contexts are created for different reasons as already explained, using the reasons presented by Brandolini [3]. Our DSL provides an optional attribute on bounded contexts which reflects this kind of type or reason why the context has been evolved. Table 2.3 lists the supported bounded context types, **F**eature, **A**pplication, **S**ystem and **T**eam (FAST). We further understand these types as different viewpoints corresponding to the «4+1» view model of software architecture [29].

TABLE 2.3: «FAST» Context Types

Type	Description
Feature / **F**unction	A bounded context reflecting a certain feature or requirement which has been identified by the Object-oriented Analysis (OOA) [31]. In terms of the «4+1» model [29], it represents a context from the «Scenario» viewpoint.
Application	A bounded context which represents a certain application, such as the «Self-Service» application in our insurance example. It is evolved by Object-oriented Design (OOD) [31] and from our understanding reflects the «Logical» and «Development» viewpoint in terms of «4+1» [29].
System	A bounded context representing an external system. The printing context in our insurance example might be such an external system which has to be integrated but is implemented by another software vendor. By using this type a bounded context illustrates a system from the physical and/or process viewpoint («4+1» [29]). This perspective concerns about how systems communicate and integrate, for example by implementing Enterprise Integration Patterns (EIP) [22].
Team	A bounded context of this type represents a team. A new context of this type might be created when a team has to be split to scale the company. However, this perspective is inspired by Conway's Law [10], stating that a systems design copies the communications structures of an organization.

2.2.2 Context Map Types

Inspired by Brandolini [3] once again, it might be interesting to consider a context map in the perspective of a team organization. Instead of thinking about systems, features or applications, one might think about bounded contexts in terms of teams. For this reason we added a *type* attribute to the context map on our domain model. This type differentiates between a «System Landscape» which shows typical bounded contexts representing features, applications and systems, and «Organization» maps which reflect the teams and their relationships. Thus, the DSL provides the two different context map types as shown in table 2.4.

TABLE 2.4: Context Map Types

Type	Description
System Landscape	A context map showing the involved systems, features, applications and their relationships. Such a map should contain bounded contexts of the types «Feature», «Application» or «System» only.

TABLE 2.4: Context Map Types (continued)

Type	Description
Organizational Map	Such a context map shows the bounded contexts in perspective of team organization. The bounded contexts must be of the type «Team».

To give an example of a map of the type «Organizational», the insurance example can by analyzed again from this perspective. We assume that the customer self-service context is implemented by the team «Customers Frontend» and the customer management context by the team «Customers Backend». Further the team «Contracts» implements the policy management context while the risk management context is implemented by the team «Claims». This would lead to a organization context map as seen in Figure 2.14.

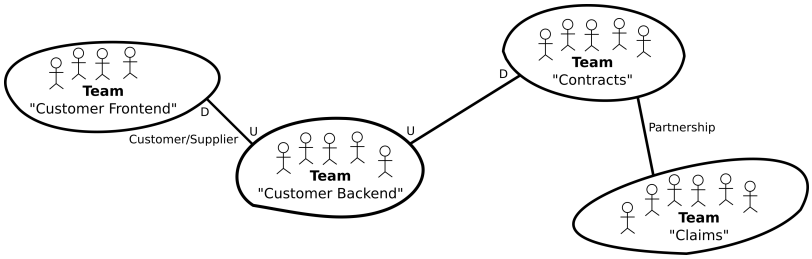

FIGURE 2.14: Example: Map of type «Organization»

The Printing context is not part of this map since it is an external system and thus not implemented by a team of the organization. Of course one could also imagine that a team implements multiple contexts which might lead to another organizational map.

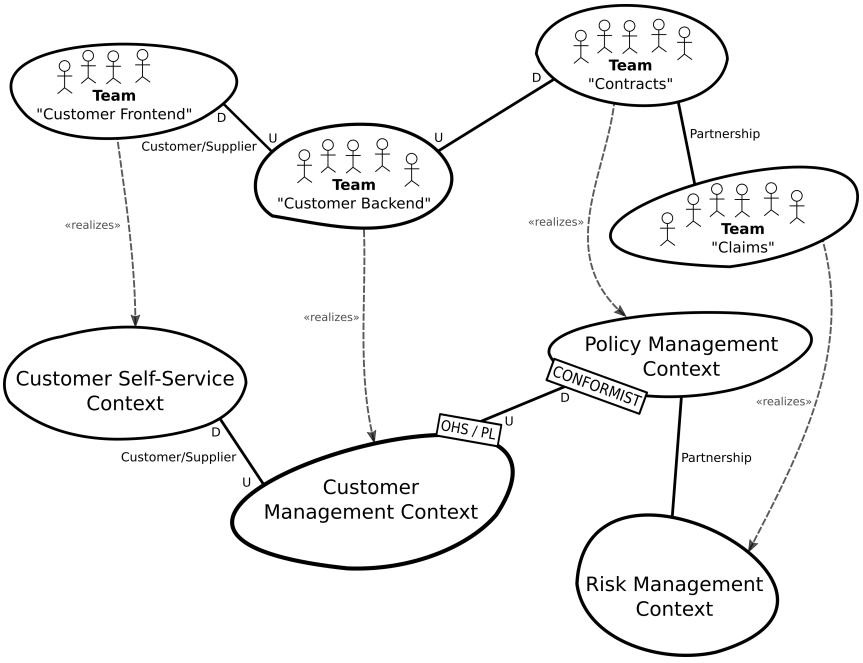

FIGURE 2.15: Example map: teams «realize» bounded contexts

Additionally, the DSL shall provide the possibility to combine the two approaches and visualize which bounded contexts are realized by which team, as in Figure 2.15.

2.2.3 Represented State: «As-Is» vs. «To-Be»

A context map can either represents the current situation of a system («as-is») or the desired state to be developed («to-be»). The DSL provides an attribute to classify a context map by those two states.

2.3 Tactic Patterns

Specifying the bounded contexts by using tactic DDD patterns shall be supported by the DSL as well. This is crucial in order to be able to apply structured service decomposition approaches. The coupling and cohesion between bounded contexts is defined by their domain models. As already mentioned in Chapter 1, DSL approaches based on tactic DDD patterns already exist. Thus, we do not want to realize our own language concerning the tactic patterns, but use and integrate one of the existing approaches.

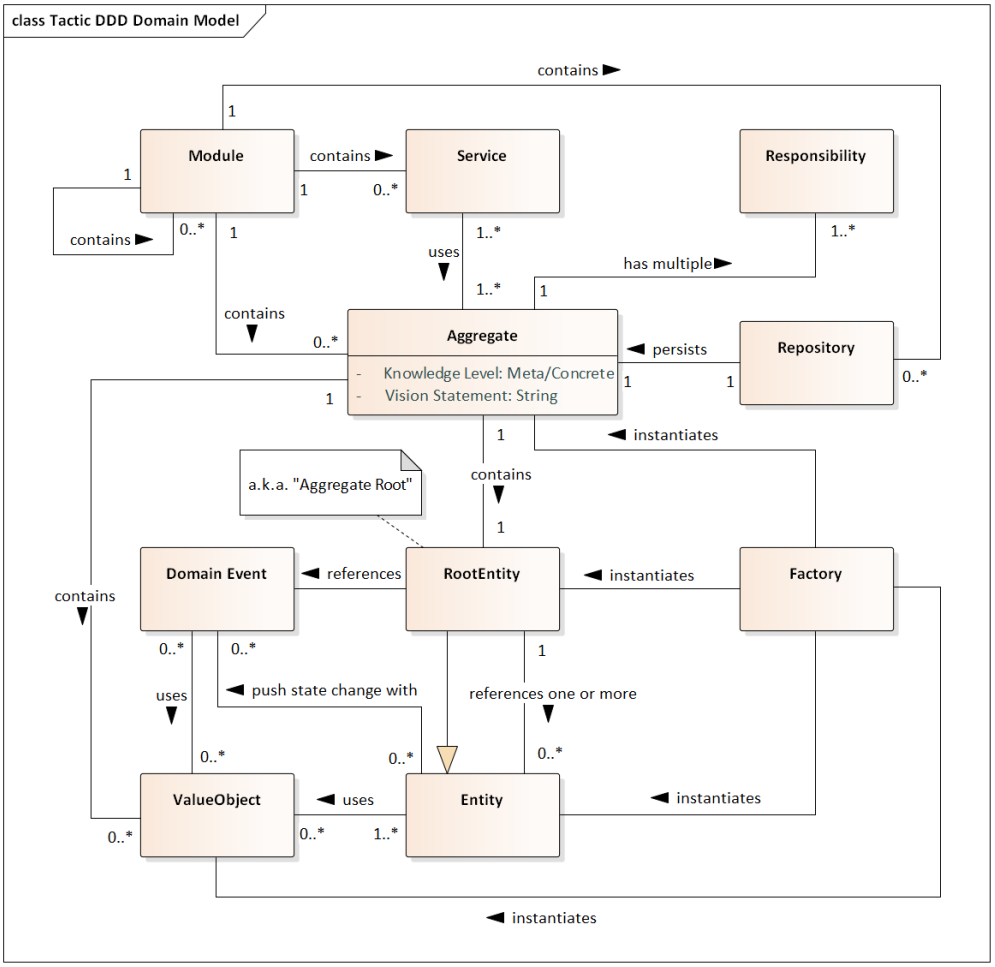

FIGURE 2.16: Tactic DDD Meta-Model (Domain Model)

However, we still want to quickly mention the patterns we expect from such an approach to be supported. Additionally, the already presented patterns Responsibility Layers, Knowledge Level and Domain Vision Statement affect the tactic DDD part of the model and have to be integrated. Figure 2.16 illustrates the tactic DDD patterns within a domain model. Note that the domain model contains all tactical DDD patterns according to [13, 14], even if we do not require all of them to be supported by the DSL.

In order to be able to describe a bounded contexts structure, the DSL shall at least support the patterns Aggregate (and Aggregate Root), Entity and Value Object. The aggregate additionally has an attribute *knowledge level* representing the implementation of the Knowledge Level pattern. Further, it has an attribute to realize the Domain Vision Statement pattern on tactical level as well as on strategical level. Last but not least, multiple responsibilities can be assigned to an aggregate which implements the Responsibility Layers pattern.

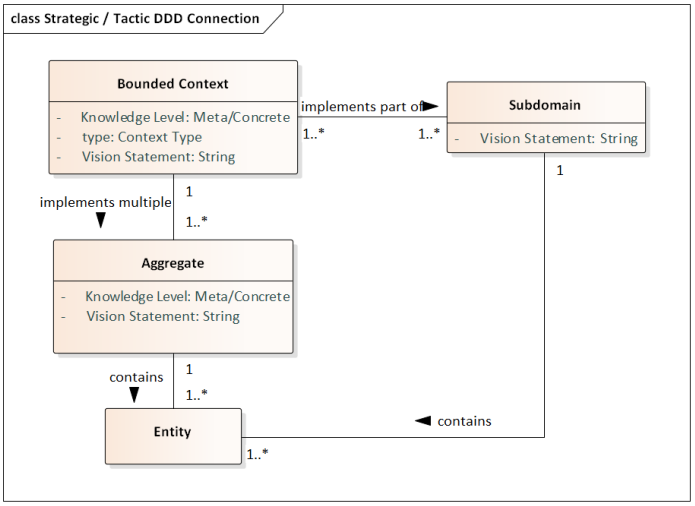

FIGURE 2.17: Connection between Strategic and Tactical DDD

Figure 2.17 illustrates how to strategic DDD domain model presented in Figure 2.1 is connected with the tactical model from Figure 2.16. A bounded context contains one or multiple aggregates. Additionally, the DSL provides the possibility to specify entities within a subdomain. With this feature it is possible to describe a subdomain in more detail and specify which domain objects are part of the subdomain. However, these entities are currently not used in any transformations and provide just a modeling feature allowing to increase the meaning of a subdomain. The aggregates and the structure inside the aggregates are relevant for the transformations, concretely the Service Cutter [20] input and the PlantUML [42] diagram generation.

This chapter introduced all concepts, especially the strategic DDD patterns, and the model on which the proposed DSL is based on. By evolving an example DDD context map step by step, it further suggested ideas for architectural refactorings [55] which may be implemented in future projects using the DSL presented by this work. The next chapter will present the requirements potential users of the implemented DSL may expect to be fulfilled. The presented user stories will explain for which use cases we expect the DSL to be a useful tool. Additionally, the Non-Functional Requirements (NFRs) we expect the language to fulfill will be introduced.

Chapter 3

DSL Requirements

This chapter discusses the requirements the Domain-specific Language (DSL) presented by this project should cover. Besides the functional requirements, which are described as User Stories, the chapter also presents the Non-Functional Requirements (NFRs) the DSL has to fulfill.

3.1 User Stories

With the User Stories (US) presented in this section it is shown which stakeholders in a software project might be interested in using our DSL to create models and what goals they would be able to achieve with it. As table 3.1 shows, the following User Stories cover all common disciplines of a software development process.

TABLE 3.1: Modeling in software development disciplines

Discipline	**User Stories**
Business Modeling & Requirements	US-1
Analysis & Design	US-1, US-2, US-4, US-5
Implementation & Test	US-2, US-5, US-6
Operating & Maintaining	US-2, US-3, US-4, US-5

The User Stories [2] are based on the «Role-Feature-Reason» template [1] invented 2001 by a team at Connextra in the UK.

«As a <*type of user*>, I want <*some goal*> so that <*some reason*>.»

3.1.1 US-1: Understanding and Analyzing the Domain

As a project member[1], I want to develop a domain model reflecting the aquired knowledge about the problem domain and the project scope so that all stakeholders have the same understanding of the domain and a common vocabulary is established.

[1]Project members include all roles involved in the software engineering process (domain experts, business analysts, software architects, software engineers, etc.).

DDD Variant: Evolving a Ubiquitous Language

As a team member[2], I want to develop a domain model, using tactic Domain-driven Design (DDD) patterns, reflecting the aquired knowledge about the problem domain and the scope so that the team speaks a Ubiquitous Language and all members have the same understanding of the domain.

3.1.2 US-2: Describing and Communicating the Architecture

As a software architect responsible for big and/or complex systems, I want to model and communicate architecture which decomposes the whole system into smaller components so that the implementation of separated components can be assigned to specialized teams and the software engineers know the boundaries of those components.

DDD Variant: Splitting a Domain into multiple Bounded Contexts

As a software architect responsible for big and/or complex systems, I want to evolve a context map, using strategic DDD patterns, reflecting the bounded contexts and their relationships so that I can arrange specialized teams working on a single context and I am able to clearly communicate the boundaries between the contexts/teams.

3.1.3 US-3: Generating other Representations of the Model

As a software architect or engineer, I want to generate alternate representations of my model so that I can generate code, represent the model in different ways according to the target audience or process the model within other tools.

Variant 1: Generate Service Cutter Output for Service Decomposition

As a software architect or engineer, I want to generate a representation of my model which can be used as input for Service Cutter [20] so that I can generate proposals for new bounded contexts.

Variant 2: Generate Graphical Representation of the Model

As a software architect or engineer, I want to generate a graphical representation out of a model using tools such as PlantUML [42] so that it is easier to understand and I can use it for discussions and/or presentations with stakeholders.

3.1.4 US-4: Modeling the Design of Components

As a software engineer, I want to model the design of my component (maybe using tactic DDD patterns) before I start coding so that I develop a better understanding of what I am building, being able to manage the complexity and understand the design and its risks.

[2] A team member works in a development team working on one bounded context.

3.1.5 US-5: Analysing Existing Architecture and Finding Problems

As a software architect or engineer, I want to create models of existing systems and review them manually or (semi-) automatically with provided tools, such as Service Cutter [20], so that technical debts and Architectural Smells [55], such as circular dependencies, can be detected and corrected using DDD patterns.

3.1.6 US-6: Compare Alternative Design Specifications

As a software architect or engineer, I want to specify and compare different design solutions for particular problems so that I can discuss pros and cons with peers and form decisions with respect to realization and maintenance requirements.

3.1.7 US-7: Transforming Models

As a software architect or engineer, I want to apply manual or automatic transformations and «Architectural Refactorings» [55, 6] to my model so that I can port and modernize the architecture.

DDD Variant: Service Decomposition as Transformation on Context Map

As a software architect or engineer, I want to apply manual or automatic transformations and «Architectural Refactorings» [55, 6] to my DDD Context Map so that I can decompose services (Bounded Contexts) in order to decrease the coupling between them.

3.2 Personas

In order to sharpen our understanding of potential users, we briefly introduce the roles Domain Expert / Business Analyst, Software Architect and Software Engineer used in the user stories above. People in all of these roles are typically faced with constraints given by the projects setup. Such constraints may also influence the requirements of the users regarding tools.

3.2.1 Martin Analyst

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

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

Constraints

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

3.2.2 Lisa Developer

Lisa works as a senior software engineer for the team. She loves programming and is always interested in new technology and programming languages. However, since the projects time budget is tight as always, she is focused on producing software efficiently while still keeping the quality high. Thus, she does not like to do things multiple times. She appreciates models which clearly communicate the domain knowledge and reduce the complexity of the system, but she only wants to use modeling tools supporting transformations into other representations such as code. Maintaining and synchronizing equal models in different representations is no option for her.

Contraints

Software engineers want to focus on their development tasks. Models should ideally be written in a way which can be processed automatically with a program. Graphical models which have to be synchronized with code manually are difficult to establish among developers.

3.2.3 Bob Architect

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

Contraints

A software architect may has to supervise many teams implementing different applications or bounded contexts. Thus, he maybe does not have the time and budget to familiarize himself with very complex tools at the beginning of a project. The creation of models shall be efficient and ideally the tool allows the architect to adapt the level of abstraction to the target audience.

3.3 Non-Functional Requirements (NFRs)

Besides the functional aspect, several non-functional requirements have to be satisfied in order to achieve a high-quality design of the language and the tools arround it.

3.3.1 Simplicity of the DSL

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

3.3.2 Size of Specification

The design of the DSL grammar should not exceed a certain complexity. Our goal is to provide a tool not only for research purposes but also for being used in «real world» applications. The complexity of the language design is measured by the amount of its grammar rules and should not exceed 10 to 100 rules.

3.3.3 Representativeness and Expressiveness

The grammer of the DSL should cover all the «strategic» DDD [13, 52] concepts needed to model an application with its «Bounded Contexts» according to the Application Architecture DDD lectures by Olaf Zimmermann (2017)[3]. Further, it shoud represent these concepts in an expressive manner.

3.3.4 Future-oriented Use of Tools and Frameworks

The tools and libraries used for the development of the DSL and its 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).

3.3.5 Reliability and Performance

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). Additionally, the model transformations (Service Cutter Output and Graphical Representation) must be performed in 7 - 10 seconds.

[3]This scope and all covered DDD patterns are documented in chapter 2.

3.3.6 Licences

Since the project is planed to be 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.

3.3.7 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.

Chapter 4

Context Mapper Implementation

This chapter explains how the Domain-specific Language (DSL) has been implemented. Besides the language, it presents the design and architecture of the surrounding tools, states important decisions, and illustrates the results with examples.

4.1 Language Workbench Evaluation

There exist several frameworks or language workbenches for creating DSLs, since implementing such a language with a parser and all needed tools from scratch would be too complex and time-consuming. Fowler uses the well-known parser generator ANTLR [4] within his DSL book [16]. However, Xtext [12], MPS [25] and Spoofax [48] are more powerful and provide way more convenience in comparison. Further, these three tools are called «the current state of the art» in Voelters book *DSL Engineering* [53], which was published a few years later (2013).

Xtext actually uses ANTLR for generating the parser, but provides other tools and features such as a typed Abstract Syntax Tree (AST), scoping, unparsing (AST back to text) and validation. Whereas Xtext and Spoofax are parser-based approaches using conventional text files, MPS is an projection-based approach. The storage format of MPS is not plain text but a tool-specific format. This enables MPS to provide features such as embedding arbitrary languages into your own language, which might be an advantage for more complex scenarios. However, in our case a text-based approach is perfectly satisfying. Furthermore, the complexity of implementing a language in MPS is much higher and it has a steep learning curve in comparison with the other candidates. Since this project can not substantially benefit of these advanced features provided by MPS, we decided against it.

Spoofax is an academic project which is the least used approach of the three. Since Xtext is widespread and very mature we decided to use it for implementing our DSL. Additionally, while Spoofax implements its own parser, Xtext uses ANTLR, which can be a fall-back in case Xtext runs out of support in the future.

4.1.1 Integrated Development Environments (IDEs)

Unfortunately, none of the presented language workbenches provides a wide IDE support. Both, MPS as well as Xtext, somehow try to lock you into their own world. This is a disadvantage if the provided DSL is expected to be used

by many users. A software architect or software engineer probably will not use the language if there is no support for his or her preferred IDE. Since we decided for Xtext and implementing additional IDE plugins would have been too time-consuming during the project, the DSL has to be used within Eclipse [50] at the moment. However, providing support for other IDEs would be desirable, as the Eclipse user base tends to decrease these days. This issue is also mentioned by Chapter 5 as potential future work.

The eclipse foundation tends to be more open regarding integration with other IDEs, since an integration with IntelliJ IDEA [24] existed [51]. Unfortunately it is no longer supported since the Xtext release 2.11 due to development capacity. According to the project team, IDEA's Application Programming Interface (API) is subject to frequent change which makes it hard for them to keep pace. A revival of the native IntelliJ support seems unlikely since the integration with the Language Server Protocol (LSP) is the pursued solution. The LSP protocol might help integrating the DSL into other IDEs but currently IntelliJ does not officially support it. Using the single available open source plugin providing LSP support for Intellij [23] did not work with IDEA 2018.2 (plugin together with Xtext LSP lead to crashes), admitting that not many efforts have been put into solving this issue so far. Note that we have not found any evidence that IntelliJ is putting efforts into supporting other IDEs for MPS-based languages, which would mean that using MPS would not be an improvement regarding the vendor «lock-in» problem.

In summary, the IDE support of the DSL aside from Eclipse is an open issue and will hopefully be addressed in a future project.

4.2 Context Mapping DSL (CML)

This section presents the implemented DSL named Context Mapper DSL (CML) [8], the major result of this work. First, the language structure and its syntax is explained using the AST and example DSL snippets implementing the insurance example introduced in Chapter 2. Afterwards, the implemented semantics are introduced.

Note that this paper mainly describes the concepts and the design of the implementation. For current examples we refer to the Context Mapper examples repository [9]. The code of the examples by December 2018 can be found in Appendix B. For guidance how to download and use the Eclipse Plugin we refer to the Context Mapper website[1] [8]. The DSL implementation is open source and can be found in our Github repository[2]. All examples, design and semantic descriptions within this paper are compatible with Context Mapper in the version *v1.0.2*[3].

[1] `https://contextmapper.github.io/`

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

[3] `https://github.com/ContextMapper/context-mapper-dsl/tree/v1.0.2`

4.2.1 Syntax (AST)

As already mentioned in the last section, the Context Mapper DSL is implemented with Xtext [12]. Xtext uses EMF [49] models for representing the generated AST. This section uses this EMF model of our DSL to explain the structure of the language, since it is easier to illustrate and understand than the grammar rules. If the reader is interested in the grammar we refer to the language reference in Appendix A.

AST Model

Figure 4.1 illustrates the AST of the Context Mapper DSL. The *ContextMappingModel* object is the root node of the tree, aggregating the context map, a list of subdomains and a list of bounded contexts. Thus, these three objects are the top-level objects within a DSL file, which has the file extension «cml». A bounded context can reference multiple subdomains from which it is implementing parts. A context map references all bounded contexts which are part of the map and defines the relationships between these contexts. As already defined during the analysis in Chapter 2, a relationship can either be symmetric or asymmetric. The asymmetric relationships are named *UpstreamDownstreamRelationships* in the grammar and the AST respectively.

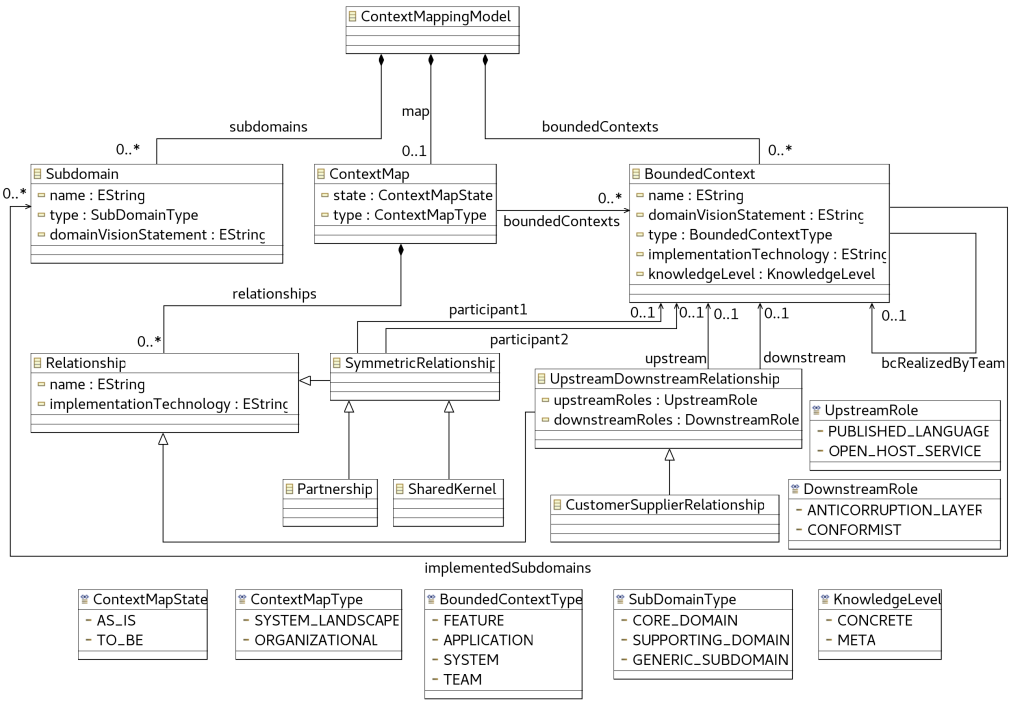

FIGURE 4.1: CML EMF Model (Abstract Syntax Tree) Diagram generated with Eclipse (Ecore Diagram)

A symmetric relationship either implements the Partnership or the Shared-Kernel Domain-driven Design (DDD) pattern. Upstream-Downstream relationships are enhanced with the roles or relationship patterns implemented by the upstream or downstream. While the upstream can implement Published

Language and Open Host Service, the possible downstream patterns are Anti-corruption Layer and Conformist. A special form of an Upstream-Downstream relationship is the Customer-Supplier relationship.

Bounded Contexts

Listing 1 illustrates the syntax of a bounded context, which is defined independently and later referenced on a context map.

```
/* Syntax example: Bounded Context */
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  domainVisionStatement = "The customer management context is responsible for managing
                           all the data of the insurance companies customers."
  implementationTechnology = "Java, JEE Application"
  responsibilities = Customers, Addresses
}
```

LISTING 1: Syntax example: «Customer Management» bounded context of the insurance scenario

With the *implements* keyword, the subdomain which is implemented by this bounded context can be referenced (see Listing 2). Within the example in Listing 1, this expresses that the CustomerManagementContext implements the CustomerManagementDomain or a part of it. It is further possible to characterize a bounded context with the following optional attributes:

TABLE 4.1: Bounded Context Attributes

Attribute	Possible values
type	A value of the enumeration *BoundedContextType*. FEATURE, APPLICATION, SYSTEM or TEAM.
domainVisionStatement	A text describing the domain vision statement (pattern mentioned in Chapter 2).
implementationTechnology	A text allowing to describe the technology used to implement this bounded context.
responsibilities	List of responsibilities which can be defined freely.
knowledgeLevel	Allows to define the knowledge level, META or CONCRETE, according to the knowledge level pattern.

Subdomain

Listing 2 illustrates how a subdomain is specified. As on a bounded context, the domain vision statement pattern can be applied here as well. Further, the type of the subdomain can be defined by a value of the *SubDomainType* enumeration depicted in Figure 4.1. A subdomain is further allowed to contain entities. With this feature it is possible to add detailed information regarding which domain objects belong to a certain subdomain. Note that these entities are currently

not used in the generators presented later. They use the entities within the bounded contexts and aggregates. The syntax of the entities is defined by the Sculptor [46] DSL.

```
/* Syntax example: Subdomain */
Subdomain CustomerManagementDomain {
  type = CORE_DOMAIN
  domainVisionStatement = "Subdomain managing everything customer-related."

  Entity Customer {
    String firstname
    String familyname
  }

  /* Add more entities ... */
}
```

LISTING 2: Syntax example: «Customer Management» subdomain of the insurance scenario

Context Maps

The core element of the model is the context map which connects the bounded contexts and subdomains with each other. A syntax example of a simple context map with only one bounded context is visualized in Listing 3. A context map is either of the type *SYSTEM_LANDSCAPE* which specifies the bounded contexts and their relationship, or of the type *ORGANIZATIONAL*. An organizational map or «team» map illustrates teams and their relationships.

```
/* Syntax example: Context Map */
ContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  /* Add bounded context to this context map: */
  contains CustomerManagementContext
}
```

LISTING 3: Syntax example: Context Map

With the keyword *contains*, a bounded context is added to the corresponding context map. Our insurance example introduced in Chapter 2 contains the following bounded contexts:

- Customer Management Context
- Customer Self-Service Context
- Policy Management Context
- Risk Management Context
- Debt Collection Context
- Printing Context

Since this would take too much space, not every context is illustrated in CML at this point. We refer to Appendix B which contains the complete CML file of the insurance example. The following listings will introduce the syntax of bounded context relationships using the bounded contexts you can find there. However, to remind you of the example, Figure 4.2 shows the graphical representation again.

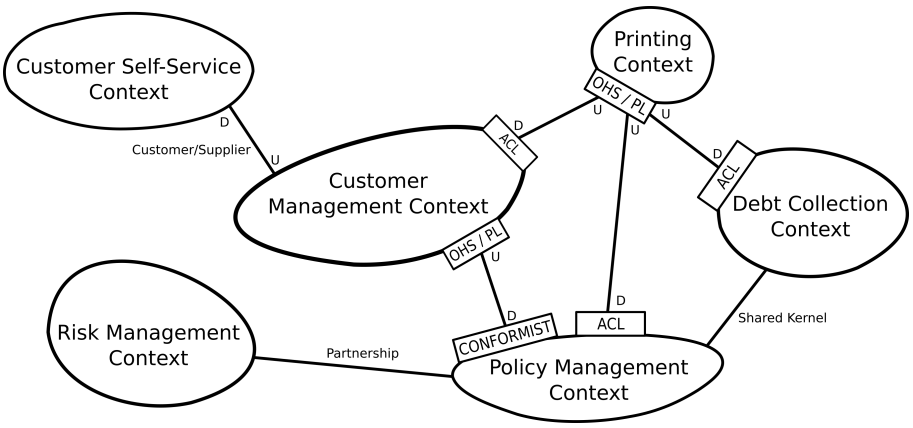

FIGURE 4.2: Context Map: Insurance Example

Relationships

Symmetric relationships are given between the risk management context and the policy management context, and between the policy management context and the debt collection context. Listing 4 illustrates the CML syntax of these two relationship patterns Partnership and Shared Kernel.

```
ContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  contains PolicyManagementContext
  contains RiskManagementContext
  contains DebtCollection

  @Risk_Policy_Partnership // optional relationship name
  RiskManagementContext <-> PolicyManagementContext : Partnership {
    implementationTechnology = "RabbitMQ"
  }

  @Policy_Debt_SharedKernel // optional relationship name
  PolicyManagementContext <-> DebtCollection : Shared-Kernel {
    implementationTechnology = "Shared Java Library, Communication over RESTful HTTP"
  }
}
```

LISTING 4: Syntax for Partnership & Shared Kernel

The <-> sign between the bounded contexts expresses the symmetry of the relationship. Therefore, if a context is written on the left side or on the right side has no semantical impact. The *implementationTechnology* attribute allows the user to specify some details regarding how the communication between

the two bounded contexts is implemented. Additionally, every relationship can optionally be annotated with a name as illustrated in Listing 4.

Listing 5 shows the asymmetric relationship between the customer management context and the policy management context.

```
ContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  contains PolicyManagementContext
  contains CustomerManagementContext

  PolicyManagementContext -> CustomerManagementContext : Upstream-Downstream {
    implementationTechnology = "RESTful HTTP"
    upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
    downstream implements CONFORMIST
  }
}
```

LISTING 5: Syntax for Upstream-Downstream Relationship (1)

Note that in this case the semantic changes if the bounded contexts are switched between the left and right side, since the arrow -> always points from the downstream to the upstream. However, since the arrow is allowed to be used in both directions, you can still switch the contexts without changing the semantics as long as you change the arrow as well:

- *Downstream -> Upstream*
- *Upstream <- Downstream*

Thus, the relationship written as in Listing 6 has the same semantical meaning as the one in Listing 5.

```
CustomerManagementContext <- PolicyManagementContext : Upstream-Downstream {
  implementationTechnology = "RESTful HTTP"
  upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
  downstream implements CONFORMIST
}
```

LISTING 6: Syntax for Upstream-Downstream Relationship (2)

Note that an alternative syntax is supported, which only uses the relationship keywords *Upstream-Downstream*, *Customer-Supplier*, *Partnership* and *Shared Kernel* without the arrows. Listing 7 illustrates examples for this alternative. For detailed descriptions of the possible syntax variants for each pattern, we refer to the language reference in Appendix A. Both variants evolved during the developments of this project. However, since we do not have enough experience and user feedback regarding the syntax in order to already decide for one variant, they shall be further tested in terms of an A/B testing [54] during the upcoming projects.

```
PolicyManagementContext Shared-Kernel DebtCollection {}

RiskManagementContext Partnership PolicyManagementContext {}

CustomerManagementContext Upstream-Downstream PrintingContext {}

CustomerSelfServiceContext Customer-Supplier CustomerManagementContext {}
```

LISTING 7: Alternative Syntax for Relationships

With the keywords *upstream implements* and *downstream implements* the relationship patterns implemented by the two contexts can be specified. The syntax for a customer/supplier relationship barely differs from the upstream/downstream relationship. Listing 8 shows an example for the relationship between the customer self-service and the customer management context. Note the main difference at the keywords *supplier implements* and *customer implements* introducing the relationship patterns. Note that this examples purpose is the illustration of the syntax and implementing an anticorruption layer in a customer/supplier relationship leads to a semantic warning, as we will explain later in the section about semantics.

```
ContextMap {
  type = SYSTEM_LANDSCAPE
  state = TO_BE

  contains CustomerManagementContext
  contains CustomerSelfServiceContext

  CustomerSelfServiceContext -> CustomerManagementContext : Customer-Supplier {
    implementationTechnology = "RESTful HTTP"
    supplier implements PUBLISHED_LANGUAGE
    customer implements ANTICORRUPTION_LAYER
  }
}
```

LISTING 8: Syntax for Customer-Supplier Relationship

Team Maps

The last not yet introduced syntactical element concerns team maps. A team map is created as a context map with the type *ORGANIZATIONAL*.

```
ContextMap {
  type = ORGANIZATIONAL
  state = TO_BE

  contains CustomersFrontofficeTeam // Add teams to this organizational map
  contains CustomersBackofficeTeam

  @CustomerTeamsRelationship // name of relationship (optional)
  CustomersFrontofficeTeam -> CustomersBackofficeTeam : Customer-Supplier
}
```

LISTING 9: Team Map Example (1)

Such a team map is only allowed to contain bounded contexts of the type *TEAM*. CML further allows to define which bounded contexts of the types *FEATURE*, *APPLICATION* or *SYSTEM* a team realizes.

```
11 BoundedContext CustomersBackofficeTeam realizes CustomerManagementContext {
12   type = TEAM
13   domainVisionStatement = "This team is responsible for implementing the customers
14                            module in the back-office system."
15 }
16 BoundedContext CustomersFrontofficeTeam realizes CustomerSelfServiceContext {
17   type = TEAM
18   domainVisionStatement = "This team is responsible for implementing the front-office
19                            application for the insurance customers."
20 }
21 BoundedContext CustomerManagementContext implements CustomerManagementDomain {
22   type = FEATURE
23 }
24 BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
25   type = APPLICATION
26 }
```

LISTING 10: Team Map Example (2)

Listings 9 and 10 illustrate such an example team map according to the example for the insurance scenario introduced in Chapter 2. The *realizes* keyword allows to define the bounded context a team is implementing.

4.2.2 Semantics

After we have seen the syntax of the CML language, this section is going to give you an overview over the implemented semantics. Note that this not only provides a feature every language needs, but it also describes our interpretations how the strategic DDD patterns interact and how they can be combined.

The implementation of the language accomplishes semantical rules concerning the strategical DDD patterns in two different ways. On the one hand, the AST, and thus the grammar of the language, already provides semantic rules by its design. Of course, this is no coincidence since the design of the language is driven by the semantic model developed and presented in Chapter 2. On the other hand, specific Xtext validators have been implemented to ensure certain semantic rules not covered by the structure of the language.

The following Table 4.2 lists important semantic rules given by the AST:

TABLE 4.2: Semantic rules defined by the AST

#	Rule	Reason / Motivation
1	The patterns Open Host Service (OHS) and Published Language can only be implemented by the upstream in an Upstream/Downstream relationship.	Trivially given by the definition of these patterns. Applying them to the downstream does not make sense, since the downstream context is calling the upstream context.

TABLE 4.2: Semantic rules defined by the AST (continued)

#	Rule	Reason / Motivation
2	The relationship patterns ACL and Conformist can only be implemented by the downstream context in an Upstream/Downstream relationship.	Trivially given by the definition of these patterns. Applying them to the upstream does not make sense, since the upstream context is the one who is called by the downstream. The upstream itself does not depend on the downstream and therefore does not have to conform or protect itself from changes of the downstream.
3	The relationship patterns Open Host Service (OHS), Published Language, Anticorruption Layer (ACL) and Conformist are not applicable for Partnership and Shared Kernel relationships.	A violation of this rule would lead to contradictions regarding the definitions of the patterns and how we understand them. In a Shared Kernel relationship, two bounded contexts share a subset of its domain model and thus, technically, share code. The interaction between the two bounded contexts happens via this shared code. The usage of the four mentioned patterns contradicts with this approach. The same applies to the very tightly coupled Partnership pattern. Even if the contexts do not share code, both can only succeed or fail together.

In addition, semantic checkers have been implemented for rules which are not implicitely ensured by the AST. The following Table 4.3 lists these currently implemented checkers:

TABLE 4.3: Implemented semantic checkers

#	Rule	Reason / Motivation
1	A bounded context which is not contained by the context map can not be part of a relationship either.	This checker provides consistency within the generated model.
2	The Conformist pattern is not applicable within a Customer/Supplier relationship.	In a Customer/Supplier relationship, the customer has an influence on the supplier and can at least negotiate regarding priorities of the requirements and the implementation. A conformist in contrast has no possibilities to influence the upstream and has to conform to what he gets.

TABLE 4.3: Implemented semantic checkers (continued)

#	Rule	Reason / Motivation
3	The Open Host Service (OHS) pattern is not applicable within a Customer/Supplier relationship.	Whereas the Customer/Supplier pattern implies that the two involved teams work closely together, meaning that the downstream team delivers the input in the upstreams planning sessions, the OHS pattern is meant to be applied if an upstream is used by many downstreams and the upstream team decides to implement one API in an «one for all» approach. This is somehow contradictory since it is unlikely that such an upstream implementing an OHS is able to have a close Customer/Supplier relationship with all its downstreams and fulfill all their expectations at the same time.
4	The Anticorruption Layer (ACL) pattern should not be needed within a Customer/Supplier relationship. **Note:** This checker only produces a compiler warning, not an error.	Similarly as in rule #3 the application of the ACL pattern is contradictory with the close Customer/Supplier relationship, where it should not be the case that the supplier implements changes from which the downstream has to protect itself. However, we only produce a warning questioning this situation since one might argue that a translation layer can be needed anyway and the difference between a translation layer and an anticorruption layer is not clearly defined or depends on how defensive it is implemented.
5	A context map of the type *ORGANIZATIONAL* (team map), can only contain bounded contexts of the type *TEAM*.	This checker provides consistency within team maps. On such a map a bounded context represents a team and not a classical bounded context such as a system, feature or application.
6	A bounded context of the type *TEAM* can not be contained by a context map of the type *SYSTEM_LANDSCAPE*.	This checker provides consistency within context maps. Can be seen as the inverse case of rule #5.
7	Only teams can realize bounded contexts.	This checker ensures that the *realize* keyword introduced in Listing 10 can only be used for bounded contexts of the type *TEAM*. The keyword is added to the language definition in order to reference the bounded contexts a team is realizing. It would not make sense for a classical bounded context (system, feature or application).

All CML concepts concerning strategic DDD have been introduced now. Note that the language reference in Appendix A provides an additional documentation of all supported patterns and corresponding sample code snippets. The following section will quickly introduce the language features regarding tactic DDD.

4.3 Tactic DDD Language Integration

Since the models written in CML shall be used by service decomposition tools such as Service Cutter [20], they have to provide the inner structure of the given bounded contexts. Only with this details it is possible to propose service cuts or how the bounded contexts should be split. Thus, a tactic DDD language is needed which allows to model the insides of the bounded contexts.

4.3.1 Tactic DSL Evaluation

DSLs for tactic DDD already exist and therefore we decided to use an already existing language and integrate it in CML rather then implement it from scratch. The research resulted in the following three existing tactic DDD DSLs:

TABLE 4.4: Existing tactic DDD DSLs

#	Name	Description
1	Sculptor [46]	«Sculptor is an open source productivity tool that applies the concepts from Domain-Driven Design and Domain Specific Languages for generating high quality Java code and configuration from a textual specification.» (citation from sculptors website)
2	fuin.org's DDD DSL [18]	Xtext based DSL supporting Domain-driven design (DDD).
3	DSL Platform [11]	«DSL Platform is a service which helps you design, build and maintain business applications.» (citation from DSL Platform's website)

We decided to integrate option #1, Sculptor, for the following reasons. Sculptor coverages all tactic DDD patterns which we at least require: Module, Aggregate (and Aggregate Root), Entity, Value Object, Domain Event, Repository and Service. These objects in Sculptor are all named after the original patterns in Evans book [13], whereas fuin.org's DDD DSL (#2) renamed certain patterns as for example the Module pattern. Further, Sculptor is licenced under the Apache 2.0 licence, while option #2 is licenced by a General Public License (GPL). Besides, using a library with a GPL licence violates our Non-Functional Requirements (NFRs) as specified in Chapter 3. Option #3, DSL Platform, is not a real option since it does not seem to be open source.

However, Sculptor perfectly fits our requirements and the fact that it is written in Xtext as well simplifies the integration. As described in the next section, a few little changes where applied to the Sculptor DSL used in CML.

4.3.2 Syntax

The following examples illustrate how the integrated tactic DDD DSL, Sculptor, is used within CML. Note that we not explain the whole language and all its concepts since it is based on another project. The Sculptor website already provides a complete documentation [45] about their language.

Nevertheless, an introdcution to the most important concepts and an example is given below. Table 4.5 lists the changes applied to the original Sculptor DSL.

TABLE 4.5: Changes applied to Sculptor DSL for CML

#	Change	Description
1	Changed aggregate concept to make it more explicit.	In Sculptor no explicit grammar rule for the aggregate pattern exists. Each entity is an aggregate root by default, if it is not declared otherwise with *!aggregateRoot* or *belongsTo*. With the *belongsTo* keyword, an entity, value object or domain event can be assigned to an aggregate root. In CML we changed this behavior and introduced an explicit aggregate grammar rule which includes all its entities, value object, etc. in a hierarchical way. An entity is not an aggregate root by default, but has to be marked as such with the keyword *aggregateRoot*.
2	Responsibility Layers pattern on aggregate.	The new aggregate rule further supports the Responsibility Layer pattern. This means that the user is allowd to assign certain responsibilities to an aggregate.
3	Knowledge Level pattern on aggregate	The Knowledge Level pattern is supported on an aggregate as well. For every aggregate, you can specify if the knowledge level is *CONCRETE* or *META*, as it is possible on bounded contexts.

To illustrate the syntax the policy management context of our insurance example is used. The root elements of the tactic DSL which can be used as the first-level child objects inside a bounded context are *Module* and *Aggregate*. Using modules is therefore optional.

```
1 BoundedContext PolicyManagementContext implements PolicyManagementDomain {
2   Module contracts {
3     Aggregate Offers {}
4     Aggregate Contracts {}
5   }
6 }
```

LISTING 11: Tactic DSL: Modules & Aggregates

You can either use modules and include the aggregates inside them, as shown in Listing 11. Or you can use the aggregates directly without modules, as shown in Listing 12.

```
BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  Aggregate Offers {}
  Aggregate Contracts {}
  Aggregate Products {}
}
```

LISTING 12: Tactic DSL: Aggregates

Within aggregates and modules it is now possible to use all the language features of Sculptor [45] which are allowed to be used inside their modules, such as domain objects, repositories, service, etc. Sculptor's root elements *Application* and *ApplicationPart* are not used, since those concepts are represented by bounded contexts in our language.

Listing 13 illustrates an example of a bounded context specification with the tactic DSL concepts inside. Every aggregate contains one Entity with the keyword *aggregateRoot*, establishing it as the root entity within the corresponding aggregate.

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

  Aggregate Products {
    Entity Product {
      aggregateRoot // keyword to define aggregate root

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

  Aggregate Contract {
    Entity Contract {
      aggregateRoot // keyword to define aggregate root

      - ContractId identifier
      - Customer client
      - List<Product> products
    }
    ValueObject ContractId {
      int contractId key
    }
    Entity Policy {
      int policyNr
      - Contract contract
      BigDecimal price
    }
  }
}
```

LISTING 13: Bounded Context Example in CML

Attributes of all domain objects, such as entities, value objects and domain events are allowed to use the primivite datatypes defined by Sculptor and the

types defined by your own domain objects. Note that references start with a hyphen (-). You can reference objects throughout the whole model, even if the referenced object belongs to another bounded context. For this reason it is not possible to use the name of a domain object twice within one model.

In our examples repository [9] you can find the whole insurance example with all bounded contexts. Further, we specified the DDD Sample [7] from Evans original DDD book [13] in CML, which can be found there as well.

All the CML language features have been introduced now and the next sections will explain the implemented transformations towards service decomposition and a graphical representation of the models.

4.4 Service Decomposition with Service Cutter

With the integration of Service Cutter [20] a proof of concept towards service decomposition is presented. Other approaches with the same goal using the CML language may be part of future projects. The following quotation introduces the Service Cutter concept briefly. For further defails, we refer to the corresponding paper [20].

> «We propose a structured approach to service decomposition by providing a comprehensive catalog of 16 coupling criteria. We abstracted them from existing literature, the experience of our industry partner and our thesis advisor.
>
> These coupling criteria are the basis of the Service Cutter tool, a prototype that extracts coupling information out of well-established software engineering artifacts such as domain models and use cases. Using this information, the Service Cutter suggests service cuts to assist an architect's decomposition decisions.
>
> We developed a scoring system that transforms the coupling data into an undirected, weighted graph. On this graph, we employ two graph clustering algorithms from the literature to find densely connected clusters as service candidates. This approach ensures that the Service Cutter produces service cuts that minimize coupling between services while promoting high cohesion within a service.»
> – Gysel et al. [20]

In order to integrate Service Cutter and calculate proposals for bounded context boundaries, a CML model has to be transformed into the Service Cutter input files. Figure 4.3 shows a Unified Modelling Language (UML) class diagram representing the Service Cutter input.

The input is splitted into two files. One file contains the *EntityRelationshipDiagram* part, whereas the other file contains the *UserRepresentationContainer*. The Entity Relationship Diagram (ERD) input file describes the structure of the given application in terms of entities and so-called nanoentities, while the user representations represent use cases, related groups and characteristics which influence the coupling criteria and thus, the Service Cutter result.

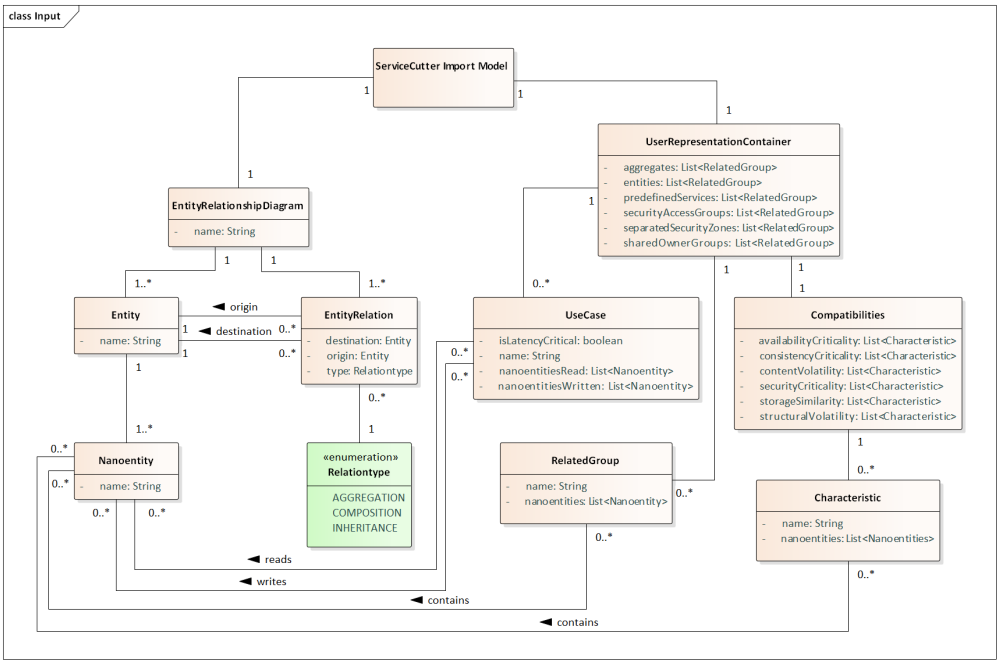

FIGURE 4.3: Service Cutter Input Model

A result of this project are two generators which produce these Service Cutter input files with a CML model as input. The processes of how the two inputs are generated out of a CML model are now described separately.

4.4.1 ERD Input Generation

The Service Cutter ERD input file is required in JavaScript Object Notation (JSON) format. The generator converts the given CML model (AST) into the Service Cutter input model (Figure 4.3) and serializes it to JSON.

The current implementation of the model transformation uses all bounded contexts of the given context map, the aggregates, entites, value objects and domain events. Note that entities, value objects and domain events have the common super type domain object in the Sculptor model. Each domain object has attributes and references, which are used in the transformation as well. All domain objects are simply mapped to entities and the attributes of a domain object to nanoentities. References to other domain objects of the context map are mapped to EntityRelation's. We further create an entity for every aggregate and bounded context. The according relations between bounded context and aggregates are mapped to EntityRelations, similarly as the relations between aggregates and domain objects.

Figure 4.4 expresses the transformation from the CML model to the Service Cutter model as a class diagram, which might be easier to understand then a textual description. The dashed lines indicate the mapping of every class and every relation.

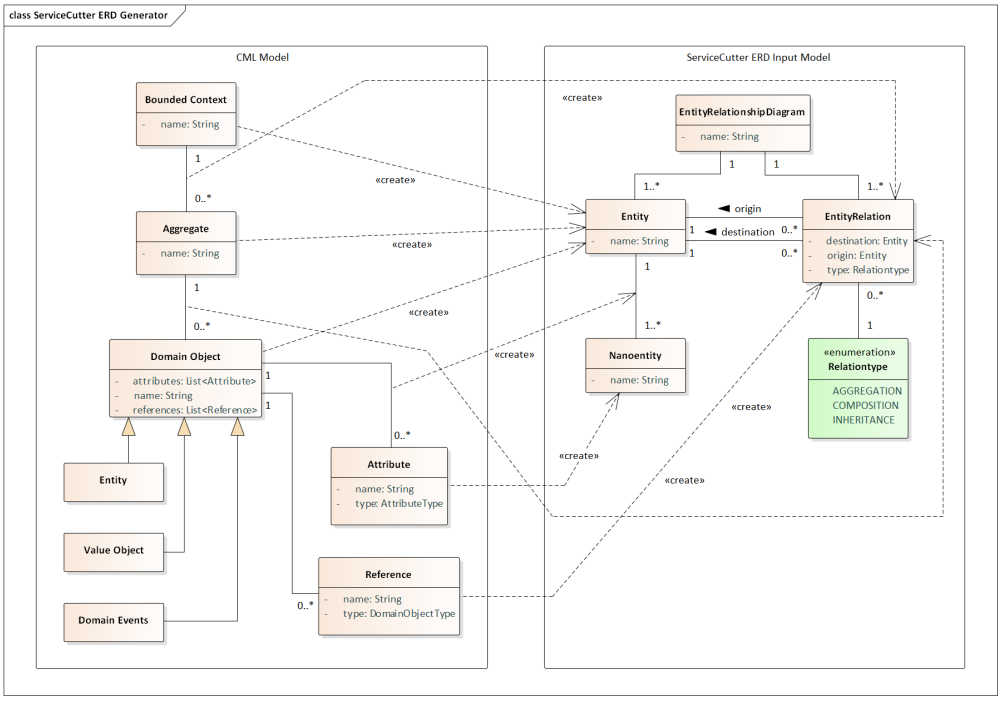

FIGURE 4.4: Mapping: CML to Service Cutter ERD Model

Additionally, Table 4.6 summarizes the mapping. The left column is mapped to the right. The Relationtype of the Service Cutter model is currently set to *AGGREGATION* in all cases, since it is not possible to derive such a distinction from our CML model.

TABLE 4.6: Mapping table: CML → Service Cutter

CML model	**Service Cutter model**
Bounded Context	Entity
Aggregate	Entity
Reference: Bounded Context / Aggregate	EntityRelation
Domain Object (Entity, Value Object, Domain Event)	Entity
Reference: Aggregate / Domain Object	EntityRelation
Attribute	Nanoentity
Reference	EntityRelation
Reference	EntityRelation

How the transformations are triggered within the Context Mapper Eclipse plugin is explained on our documentation website[4] [8].

4.4.2 User Representations Generation

The second file used by Service Cutter is describing the user representations part of the model shown in Figure 4.3. However, this input can not be fully derived from a CML model but has to be manually worked out. Since it has to

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

be provided in the JSON format as well and a manual preparation of the data without tool-support is very time-consuming, we decided to provide another DSL in order to simplify this process.

The DSL files end with the file extension «scl» for Service Cutter DSL (SCL) and its AST exactly corresponds to the Service Cutter user representations model (Figure 4.3, right part). Further, a generator which produces an SCL example file from an CML model has been implemented.

Use Cases

Use cases are specified by declaring the nanoentities which are read and written in the specific case, as illustrated in Listing 14.

```
1 UseCase UpdateCustomer {
2   reads "SocialInsuranceNumber.sin"
3   writes "Customer.firstname", "Customer.lastname", "Address.addresses"
4 }
```

LISTING 14: Service Cutter Use Case in SCL

The generator produces example use cases as templates to simplify the specification for the user.

Compatibilities

The compatibilities block contains the following Service Cutter concepts:

- Availability Criticality
- Consistency Criticality
- Content Volatility
- Security Criticality
- Storage Similarity
- Structural Volatility

Once again, we generate a sample block derived by the CML model by picking attributes randomly, but it can be used as a template. Listing 15 illustrates the syntax. If the user does not want to specify certain criteria, the corresponding blocks can be deleted. For more details regarding the presented criteria and how they influence the resulting service cuts we refer to the Service Cutter documentation [20, 47].

Related Groups

The last type of the user representations part are the Related Groups. Specifically, Service Cutter supports the following related groups of nanoentities:

- Entities

- Aggregates
- Predefined Services
- Security Access Groups
- Separated Security Zones
- Shared Owner Groups

The current implementation of the SCL generator derives Aggregates and Predefined Services from the CML model. For the other related groups templates and examples are generated.

```
Compatibilities {
  AvailabilityCriticality {
   characteristic Normal // Allowed characteristics: Critical, Normal, Low
   "ProductId.productId", "ContractId.contractId"
  }
  ConsistencyCriticality {
   characteristic Weak // Allowed characteristics: High, Eventually, Weak
   "SocialInsuranceNumber.sin", "CustomerRiskFactor.totalRiskFactor"
  }
  ContentVolatility {
   characteristic Often // Allowed characteristics: Often, Regularly, Rarely
   "Customer.firstname", "SocialInsuranceNumber.sin"
  }
  SecurityCriticality {
   characteristic Public // Allowed characteristics: Critical, Internal, Public
   "Address.postalCode", "Risk.risk"
  }
  StorageSimilarity {
   characteristic Normal // Allowed characteristics: Tiny, Normal, Huge
   "ProductId.productId", "UserAccount.username"
  }
  StructuralVolatility {
   characteristic Normal // Allowed characteristics: Often, Normal, Rarely
   "Address.city", "Product.productName"
  }
}
```

LISTING 15: Service Cutter Compatibilities in SCL (Examples)

Listing 16 shows two examples for the syntax of the Related Groups. The syntax for all other groups is identical.

```
/* This aggregate was generated by your CML model. You do not have to change it. */
Aggregate Risks {
  "CustomerRiskFactor.totalRiskFactor", "Risk.likelihood", "Risk.risk"
}

/* Shared Owner Groups cannot be derived from ContextMap.
 * This is a template/example how you can define them. If you do not want to specify
   any, remove this block. */
SharedOwnerGroup SharedOwnerGroupTemplate {
  "Product.productName", "Risk.risk"
}
```

LISTING 16: Service Cutter Related Groups in SCL (Examples)

The keywords to start the specifications are *Entity*, *Aggregate*, *PredefinedService*, *SecurityAccessGroup*, *SeparatedSecurityZone* and *SharedOwnerGroup*.

CML to SCL Transformation Mapping

Figure 4.5 provides a graphical summary of the transformation from CML to SCL as it was given for the ERD part of the Service Cutter input. However, as already mentioned and illustrated in the figure, not many objects are derived automatically. The aggregates in CML can be mapped directly to the aggregates in the service cutter input (SCL), since it represents the same concept in both worlds. Predefined services in the Service Cutter input describe already separated services which fit to our already existing bounded contexts before the service cutting process. Thus, the transformation creates a *predefined service* related group for every bounded context.

The nanoentities within the *aggregate* related groups of the SCL model are derived from all attributes within the corresponding aggregate defined in the CML model. The nanoentities in a *predefined service* related group are derived by all attributes within the corresponding bounded context accordingly.

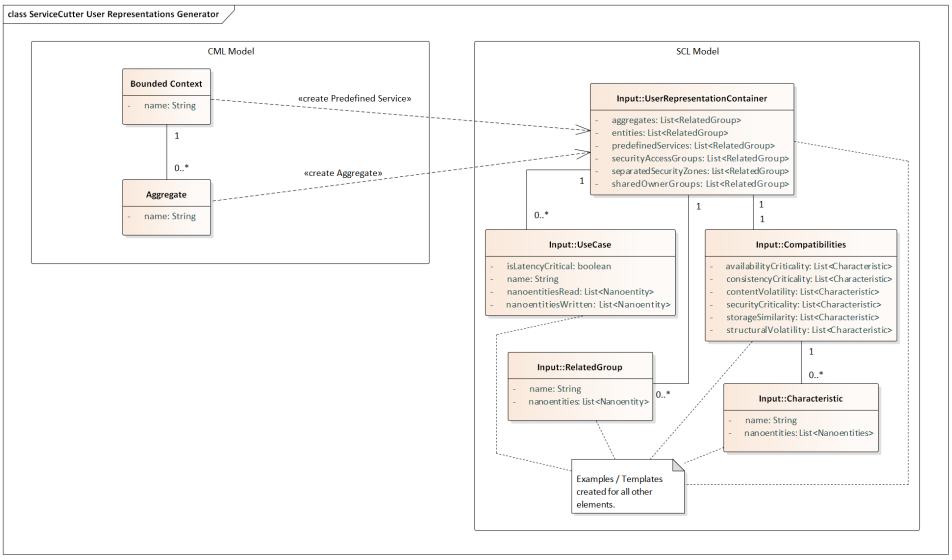

FIGURE 4.5: Mapping: CML to SCL (User Representations)

Another transformation provided by the implemented Eclipse plugin converts the SCL file into the JSON representation required by Service Cutter.

After having explained how the two input files for Service Cutter are generated, the next section presents the evaluation results using our insurance example.

4.4.3 Evaluation

The implemented service decomposition approach using Service Cutter has been evaluated with the insurance example and the DDD sample application. The bounded contexts of the insurance scenario are inspired by Lakeside Mutual [37], a fictitious insurance company. However, the inner structures of the

bounded contexts using the tactic DDD patterns are freely modeled based on our moderate knowledge about the insurance business. Regarding the tactic DDD concepts, the DDD sample application is modeled according to its original [7] as it is presented in Evans book [13]. Although, to make the example more interesting with respect to the strategic design, we split the application into three bounded contexts. Both examples with the complete CML source code can be found in our examples repository [9].

Expected Results

In order to evaluate our approach towards service decomposition, the implemented model transformation to the Service Cutter input and the modeled examples, we expect that the Service Cutter output reflects the given bounded contexts of the input CML model. If the modeled bounded contexts achieve high cohesion and loose coupling between each other, the Service Cutter output should result in the same bounded contexts.

Insurance example

The context map of the insurance example has already been shown in Figure 4.2. Note that we evaluated the results with a user representations file (SCL) containing the related groups which can be derived from the CML model, namely aggregates and predefined services. We have not yet specified any use cases for the insurance example.

The result produced by the Girvan-Newman algorithm [39] provided by Service Cutter is illustrated in Figure 4.6. The printing context, customer management context and customer self service context are reflected precisely. In Figure 4.6 *Service C* represents the customer management context, *Service F* the customer self service context and *Service E* the printing context.

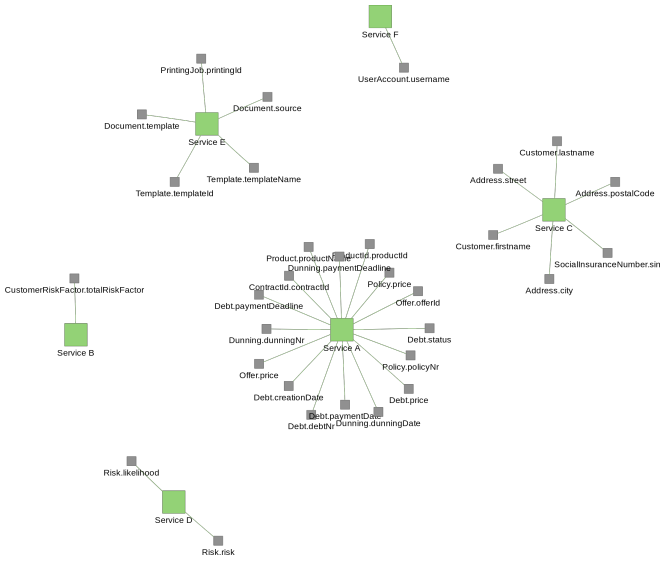

FIGURE 4.6: Result: Insurance example (Girvan-Newman)

However, the other services produced by Service Cutter slightly derive from our bounded contexts. The algorithm merges the policy management context

and the debt collection context together to *Service A*. The risk management context is split into the two services *Service B* and *Service D*. The latter might be an effect of the *number of services* parameter, which we set to six since this is the number of bounded contexts in our input. However, this result already fulfills our expectations to a certain degree since we are able to identify the structure of our model, concretely the bounded contexts, within the Service Cutter output. Note that the *Predefined Service Constraint* parameter actually had to be decreased to the value XS in order to produce this result. For some reason, the predefined services scoring does not seem to have the impact we expected with the Girvan-Newman algorithm.

Increasing the weight of this scoring parameter leads to a fully decomposed graph which is somehow contradictory. Increasing the parameter should actually increase the coupling between the entities and nanoentities of an existing bounded context. However, this phenomenon has not been further studied since the result is already good and the scoring based on the predefined services has a positive influence if the Leung [33] algorithm is used.

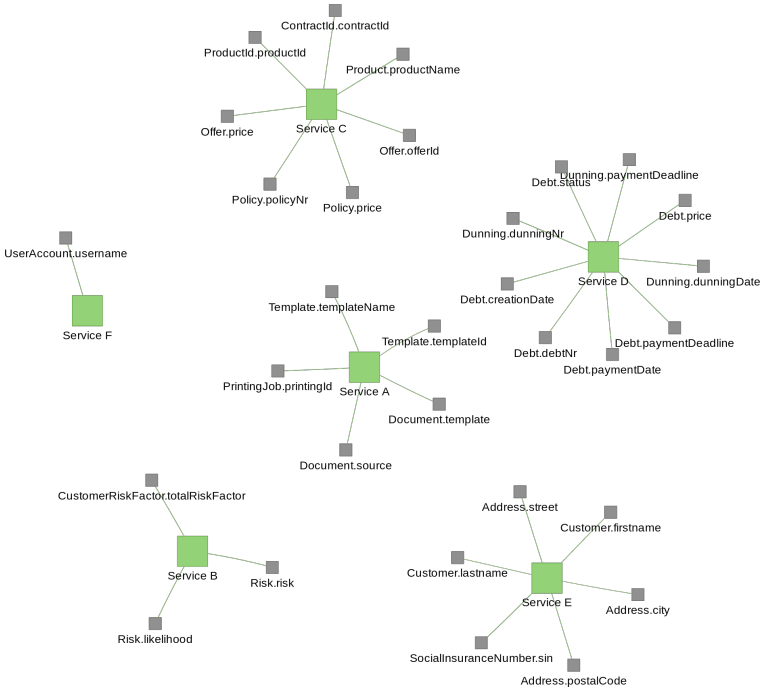

FIGURE 4.7: Result: Insurance example (Leung)

The result produced with the Leung [33] algorithm is shown in Figure 4.7. As already mentiond, the aggregates and predefined services user representations had a positive influence on that algorithm and lead to a result perfectly reflecting our bounded contexts. Further, the result is surprisingly stable. Even if the Leung algorithm is known to be non-deterministic, it produces always the same result in this case.

Table 4.7 shows how the services in Figure 4.7 map to the modeled bounded contexts.

TABLE 4.7: Leung Service Cutter Result Mapping

Resulted Service	Bounded Context
Service A	Printing Context
Service B	Risk Management Context
Service C	Policy Management Context
Service D	Debt Collection Context
Service E	Customer Management Context
Service F	Customer Self-Service Context

Sample DDD application

To evaluate the approach with a second example, the DDD sample app was split into three bounded contexts. Evans [13] already mentions the idea of creating a separate bounded context for the Voyage Planning part of the application. With this split we get two bounded contexts. The voyage planning context and the original cargo booking context. Evans further suggests a Shared Kernel between those contexts. Additionally, inspired by Rademacher [44], we created another bounded context managing the locations. With both changes the example results in the context map as shown in Figure 4.8.

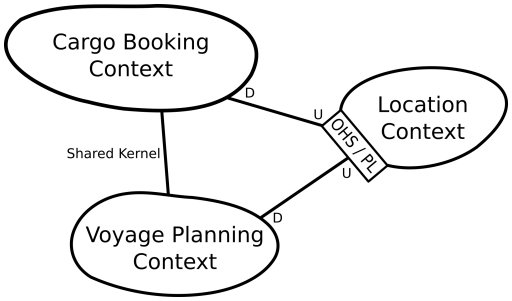

FIGURE 4.8: DDD Sample split into three Bounded Contexts

The result with the Girvan-Newman algorithm reflects the bounded contexts very well, if the priority of the predefined service constraint is adapted again.

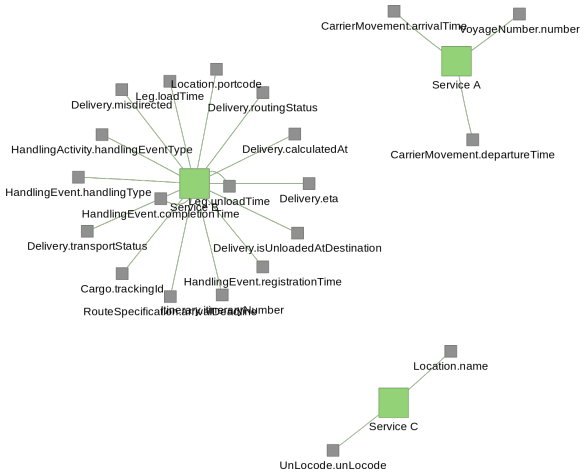

FIGURE 4.9: Result: DDD Sample (Girvan-Newman)

Figure 4.9 illustrates the result. Besides a few small derivations the resulting services can be mapped to the bounded contexts. For example, the nanonentity *Location.portcode* seems to be very tightly coupled to the cargo booking context and is not part of the location context. However, *Service B* clearly represents the cargo booking context, *Service A* the voyage planning context and *Service C* the location context.

The Leung algorithm behaves non-deterministic for the DDD sample. Thus, we cannot present one stable result. However, the results still have strong similarities and a few parts which are stable. Figure 4.10 shows just one arbitrary result. What all results have in common is the clearly identifiable cargo booking context, which is *Service A* in the example in Figure 4.10.

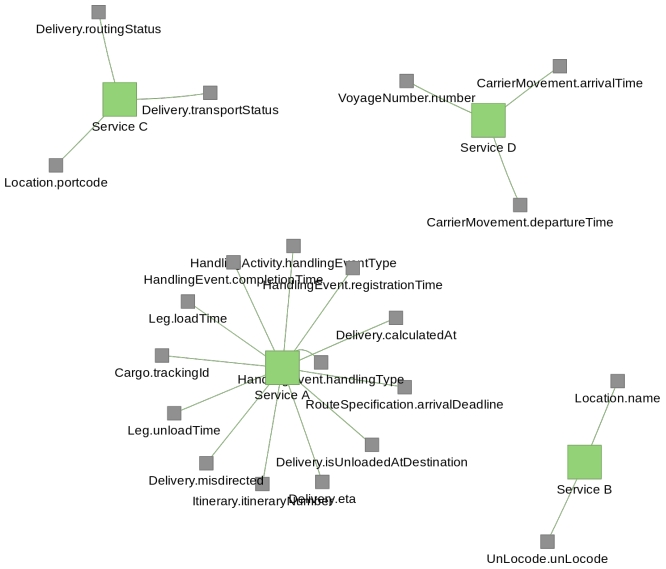

FIGURE 4.10: Result: DDD Sample (Leung)

Further, we determined that the Leung algorithm always splits the nanoentities *Delivery.routingStatus* and *Delivery.transportStatus* into a separate service (*Service C in Figure 4.10*). This may be an indicator that their coupling towards the cargo tracking context is not very strong. The location context and voyage planning context are reflected in this concrete example, but overall not stable with this algorithm.

In summary, the results for the implemented proof of concept fulfilled our expectations. The CML models and the structure which is generated with the Service Cutter input transformation reflects the services which are created by the graph clustering algorithms for both examples. The bounded contexts were always identifiable in the results and with changing the priorities of the Service Cutter constraints it is possible to assess that the generated user representations have an influence on the service cuts.

4.4.4 Service Cutter Results to CML Transformation

The results produced by Service Cutter can be transformed back to CML. Within the Service Cutter UI it is possible to export the result as a JSON file. A converter provided by the Context Mapper tool can be used to transform this file

into CML. However, the current implementation only uses the data which are available in the Service Cutter output file and therefore the feature has its limitations. Figure 4.11 illustrates the Service Cutter output model.

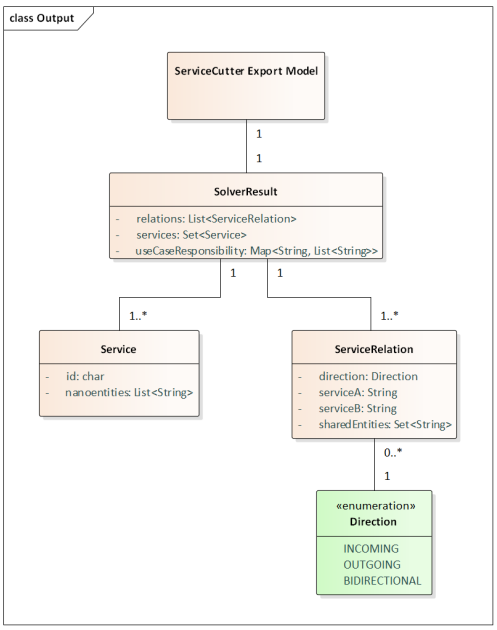

FIGURE 4.11: Service Cutter Output Model

The result contains services and service relations which are mapped to bounded contexts and bounded context relationships in CML. Since Service Cutter does not know aggregates, we just create one aggregate for every bounded context. Within this aggregate we create the entities and nanoentites. Service relations of the types INCOMING and OUTGOING are mapped to corresponding Upstream/Downstream relationships. If a service relation has the direction BIDIRECTIONAL, we create a Shared Kernel relationship. The Listings 17 and 18 illustrate the produced CML for the DDD sample result of Figure 4.9.

```
ContextMap {
  contains ServiceA
  contains ServiceB
  contains ServiceC
  ServiceA Shared-Kernel ServiceB
  ServiceC Upstream-Downstream ServiceB
}
BoundedContext ServiceA {
    Aggregate AggregateA {
        Entity A_CarrierMovement {
            Nanoentity departureTime
            Nanoentity arrivalTime
        }
        Entity A_VoyageNumber {
            Nanoentity number
        }
    }
}
```

LISTING 17: DDD Sample produced by Service Cutter (1)

```
19 BoundedContext ServiceB {
20     Aggregate AggregateB {
21         Entity B_Cargo {
22             Nanoentity trackingId
23         }
24         /* other entities removed here (saving space) */
25     }
26 }
27 BoundedContext ServiceC {
28     Aggregate AggregateC {
29         Entity C_UnLocode {
30             Nanoentity unLocode
31         }
32     }
33 }
```

LISTING 18: DDD Sample produced by Service Cutter (2)

One limitation is that Service Cutter does not know the datatypes of the nanoentities. Thus, as you can see in Listing 18, the generated type for all attributes is *Nanoentity*. Additionally, the entities are currently prefixed by the service identifier and an underscore, since the entities can be part of multiple bounded contexts after the cutting process and the Sculptor [46] DSL does not allow multiple entities with the same name.

This implementation has to be seen as a proof of concept. It illustrates that it is possible to fully integrate Service Cutter and generated results can be transfered back to CML. Obviously, the process could be improved by using the data from the original CML file which for example contains the datatypes. The generated bounded context relationships may have to be customized manually after the cutting process.

4.5 Graphical Representation with PlantUML

An additional delivery of this project is a transformation of the DSL model into a visual representation. As a proof of concept, a PlantUML [42] generator has been implemented. Two types of UML diagrams are generated out of a context map. On the one hand, a component diagram representing the bounded contexts and their relationships is generated. Each bounded context is visualized as a component. On the other hand, a class diagram is generated for each bounded context. The class diagram illustrates the aggregates, modules and all domain objects of a bounded context.

4.5.1 Component Diagram

The component diagram contains a component for every bounded context on the context map. Figure 4.12 illustrates the generated component diagram for the insurance example.

The bounded context name is mapped to the component name. Relationships of the types Partnership or Shared Kernel are illustrated with a simple

bidirectional connector (<–>) in PlantUML, as illustrated by the examples between the risk management context, policy management context and debt collection context. The connector is labeled with the relationship name and the implementation technology (*implementationTechnology* property in CML) in brackets:

- Partnership ({implementationTechnology})
- Shared Kernel ({implementationTechnology})

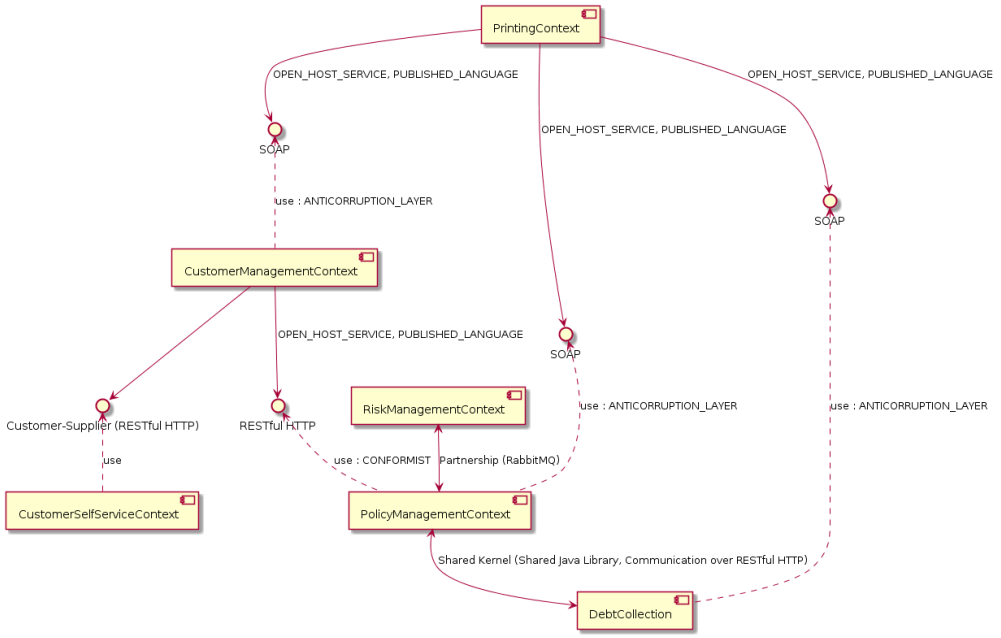

FIGURE 4.12: Insurance Example Component Diagram

Upstream/Downstream relationships produce an interface which is provided by the upstream and consumed by the downstream. The consumer arrow towards the interface is marked with the keyword «use». The interface is named after the *implementationTechnology* in CML. In the special case of a Customer/Supplier relationship the name is «Customer-Supplier» and the implementation technology is mentioned in brackets:

- Customer-Supplier ({implementationTechnology})

The upstream and downstream relationship patterns OHS, Published Language, ACL and Conformist are added to the arrow label towards the interface, as illustrated in Figure 4.12.

4.5.2 Class Diagram

The class diagram generator creates a rectangle for each aggregate. Inside the rectangle, all domain objects of the types entity, value object and domain event are created as classes. Domain objects of the type Enum are mapped to enum's

in PlantUML. The class or enum name is given by the corresponding name of the domain object in the CML model.

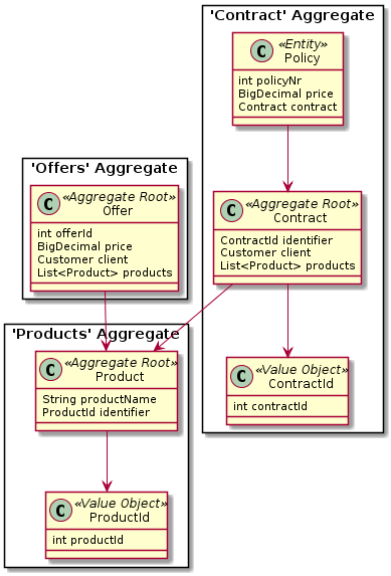

FIGURE 4.13: Insurance Example Class Diagram (1)

All attibutes of the domain objects are mapped to corresponding attributes in PlantUML. References are listed as attributes within the classes as well. If the type of a reference is equal to a domain object within the same bounded context, a plantUML reference is created. Figure 4.13 illustrates an example, concretely the policy management context of the insurance example. The tactic DDD patterns entity, value object and domain event are mapped to the class stereotype. A special case is the aggregate root. The aggregate root's stereotype is «Aggregate Root» instead of «Entity».

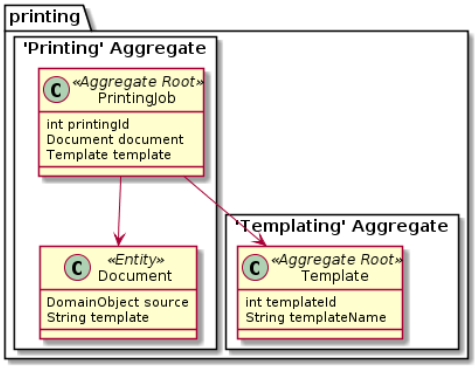

FIGURE 4.14: Insurance Example Class Diagram (2)

If modules are used within a bounded context they are mapped to packages in PlantUML, as shown in Figure 4.14. Note that the current implementation of the PlantUML generator is a proof of concept and not all concepts and possibilities of the Sculptor tactic DDD DSL are used and transformed into PlantUML. For example methods, services and repositories are currently not converted.

4.5.3 PlantUML Evaluation

As illustrated in the last sections, the implemented PlantUML transformation has been tested and evaluated with the the insurance example. It was further tested with the sample DDD application and the generator produced reasonable results in both cases. However, both diagrams, the class and the component diagram, could be further enhanced with CML data which are currently not used.

Component Diagram

The component diagram which illustrates the bounded contexts and their relationships currently does not use the relationship name. Further it does not illustrate the subdomains which are implemented by the bounded contexts. The domain vision statement could be incorporated with an upcoming release as well. Even if there is no specific UML notation for those concepts, a future release of the generator could add these information by using comments/notes.

Class Diagram

The class diagram currently describes a bounded context with all the domain objects such as entities, value objects and domain events with their attributes and references. This already illustrates the structure of the bounded context and provides a satisfying solution for this proof of concept. However, the Sculptor DSL [46] provides additional features which could be used to enhance the diagrams. For example, the domain objects are also allowed to specify methods. Further, the generator currently does not use services and repositories.

With the graphical representation transformation all implementation details of this project have been introduced. This chapter explained the CML language with its syntax and implemented semantics. It further documented the transformations for the service decomposition approach using Service Cutter and evaluated the results. The next chapter will summarize and evaluate the projects results overall. It further discusses potential future work.

Chapter 5

Evaluation, Conclusion and Future Work

This last chapter summarizes the results of the project, evaluates the fulfillment of the requirements and gives an outlook to future work.

5.1 Results & Contributions

This work presented a Domain-specific Language (DSL) which allows the creation of models based on Domain-driven Design (DDD) patterns. While the individual patterns already have been introduced and explained by the literature [13, 14, 52, 38], this work aims to contribute one particular interpretation how the patterns could work together within one meta-model. The design of the language with the presented domain model in Chapter 2 and the resulting semantics provide concise rules stating which patterns are allowed to be combined. With this approach we try to tackle predominant ambiguities regarding these patterns and possible pattern combinations.

Besides this conceptual contribution, the implemented DSL provides a tool to specify context maps. The models are written in a form which can be processed and transformed. Thus, the Context Mapper DSL (CML) language offers a foundation for any tools which intend to visualize context maps or apply transformations on them. With the implemented PlantUML [42] diagram generator we provided a proof of concept illustrating such a transformation into a graphical representation. Further, this work suggested architectural refactorings [55] inspired by a context mapping approach [3] which could be implemented by processing CML.

A major goal achieved by this project is the DSL as a basis for structured service decomposition approaches. Future approaches aiming for automatic decomposition or for algorithmic service decomposition suggestions, such as Service Cutter [20], may use our DDD-based DSL. The implemented Service Cutter integration provides a proof of concept for such a DSL processing towards service decomposition.

5.2 Requirements Evaluation

The critical success factors which were defined at the beginning of this project required a proof of concept consisting of the DSL grammar, the implementation of semantic validators, a transformation into a graphical representation

and the Service Cutter integration. These requirements are fulfilled and documented in-depth in Chapter 4. Example applications of the DSL were given by the insurance example throughout the whole report and the DDD sample, both completely available in Appendix B and [9]. From the project definition and its goals functional requirements represented as user stories and Non-Functional Requirements (NFRs) were derived. These are presented in Chapter 3. The following sections aim to evaluate these requirements briefly. Section 5.2.2 further summarizes the feedback we have received of the first users, for which we are very grateful.

5.2.1 User Stories

With the user stories and personas in Chapter 3, we presented the values our DSL should offer to the users. This section compares the promised values with the actual results. It highlights the contributions this project has achieved regarding the functional requirements and what the open issues for future projects are.

US-1: Understanding and Analyzing the Domain

The implemented DSL allows a user to create a domain model in terms of the tactic DDD patterns within every bounded context. It further provides the possibility to create models within subdomains. With these features a project team is able to model the knowledge of the every part of the domain in terms of their ubiquitous language.

US-2: Describing and Communicating the Architecture

A software architect is able to model a system and to decompose it into smaller components by using bounded contexts. He can communicate the architecture by using CML or the generated Unified Modelling Language (UML) diagrams. The modeled bounded contexts allow the architect to illustrate the boundaries of the individual domain models and to assign the implementation of single bounded contexts to corresponding teams (team map).

US-3: Generating other Representations of the Model

With this project we implemented two model transformations as a proof of concept. The first transformation converts a CML model into Service Cutter [20] input while the second transformation creates PlantUML [42] diagrams. However, this proof of concept might not fulfill the requirements of this user story completely. Future transformations may provide other representations of the model and/or generate code from CML. By providing more alternative representations of the model, the value of this user story can certainly be improved.

US-4: Modeling the Design of Components

With the integrated Sculptor [46] DSL a software engineer is able to model the design of a component using all tactic DDD patterns. This helps the engineer to strengthen the understanding of the domain and to manage the complexity

of the component. However, a transformation to automatically generate code out of a bounded context CML model would probably increase the acceptance of such a modeling tool among software engineers.

US-5: Analyzing Existing Architectures and Finding Problems

The current tool at least provides the possibility to model an existing system and thereby find problems manually. With the Service Cutter [20] integration we demonstrated how the DSL can be used to analyze an existing system with automated transformations and according tools. However, other automated approaches to inspect existing architectures will be part of the next project.

US-6: Compare Alternative Design Specifications

With the current implementation of Context Mapper a software architect or engineer is able to model multiple alternative design specifications. However, the comparison of the models is manual work and no tool support exists to simplify this process. The comparison of two model instances can be done manually on the basis of CML or by comparing the generated PlantUML [42] diagrams. The Service Cutter [20] integration at least provides the possibility to compare models in terms of coupling between multiple bounded contexts.

US-7: Transforming Models

We have mentioned the idea of implementing architectural refactorings [55] as model transformations based on CML within this report. With the paper [28], we have already realized a small proof of concept towards processing our DSL with a model transformation approach. This user story is not fulfilled yet, but will be a major focus of the next project.

5.2.2 Evaluators Feedback

Note that we have not conducted a representative user test to measure the fulfillment of these requirements. However, the tool has been used by the supervisor of this project as part of an excercise lesson of an application architecture lecture at our university[1]. This gave us the opportunity to collect first feedbacks regarding the tool and the DSL examples from a senior software architect (supervisor) and nearly 20 exercise participants. The following table 5.1 summarizes their impressions and the feedback we have got.

TABLE 5.1: Evaluators Feedback

Topic	Summarized Feedback
Examples & Syntax	All the users mentioned that it is possible to understand the examples written in the DSL within 15 to 20 minutes. Note that we specificly asked for that to have an input for evaluating the NFR «Simplicity of the DSL». However, we received a few critical feedbacks regarding the syntax as well.

[1]Thanks to Prof. Dr. Olaf Zimmermann for using the Context Mapper tool in one of his application architecture exercise lessons and providing the valuable feedback.

TABLE 5.1: Evaluators Feedback (continued)

Topic	Summarized Feedback
Examples & Syntax (continued)	The alternative syntax using the pattern names between the contexts instead of the arrows (see Appendix A for details) can be understood faster. The variant with the arrows tends to be unclear in the beginning. Additionally, the declaration of the relationship roles might be more clear if the keywords *upstream implements* and *downstream implements* are extended by adding the name of the corresponding context. With the current syntax a beginner might not directly recognize which context is upstream and which is downstream.
Features & Tools	An issue mentioned and already identified before is the IDE support. Users are not willing to install Eclipse if it is not the IDE they use in general. To increase the value for the users, additional transformations into other representations and/or code should be implemented. If a user for example is only interested in UML diagrams, it is probably easier to use a graphical tool specifically for that purpose. An interesting idea might be a transformation which generates application stubs out of bounded contexts. Generating CML from existing applications has been mentioned as another potential feature.
General Feedback	In summary, the feedback regarding the syntax and the examples was satisfying, taking into account that the goal of this project was to implement a proof of concept and not a fully-fledged product. However, providing a good documentation explaining the language concepts seems to be important in order to start beeing productive with the language quickly.

5.2.3 Non-Functional Requirements

Besides the functional requirements, we want to briefly evaluate the fulfillment of the NFRs as well.

Simplicity of the DSL

As already mentioned in the last section, a representative user test has not been conducted. However, the results and feedbacks from the evaluators mentioned in Section 5.2.2 indicates that it is possible to achieve the defined 15 to 20 minutes time limit to understand the examples. Even though, we can not finally evaluate this requirement without a representative test.

Size of Specification

The specification of the language can be found in Appendix A. The Context Mapper DSL without the Sculptor [46] integration consists of 16 grammar rules.

The integrated Sculptor DSL for the tactic DDD patterns consists of 57 grammar rules. With 73 grammar rules in total, the required limit of 10 to maximal 100 grammar rules is fulfilled.

Representativeness and Expressiveness

Within Chapter 2 all the strategic DDD patterns have been analyzed. Further, a decision if it is needed for the language has been made for each pattern. The decisions have been justified as well. The DSL supports all the patterns according to the mentiond resource[2] in the NFR.

Future-oriented Use of Tools and Frameworks

All tools used by our implementation are well established and open. However, the sustainability of the Xtext framework is difficult to judge. Nevertheless, the produced Abstract Syntax Tree (AST) is stored in the ECore format which can be said to be an sustainable format, due to its usage. Further, the Xtext [12] tool generates an ANTLR [4] parser, which might simplify a technological switch.

Reliability and Performance

The tool is tested with unit and integration tests and currently (v1.0.2) a test coverage of 89 percent is measured. During the last weeks of the project, manual tests have been made to ensure the stability of the current release. All model transformations using models of the size of our examples can be performed in less then 7 seconds.

Licences

This NFR is fulfilled since only tools and libraries with «Apache licence 2.0» and «Eclipse Public Licence 1.0» were used.

Supportability and Maintainability

Tools and mechanisms promoting a good code quality have been set up. Direct commits to the master branch of the repository are not allowed and always have to be made via a pull request. A continuous integration pipeline builds every commit, executing the unit and integration tests. The test coverage is measured by a corresponding tool as well. Finally, no special language features have been used and a junior software engineer should be able to understand the code, as required by this NFR.

5.3 Conclusion

In summary, the critical requirements and goals of this project were fulfilled. Besides the concrete deliverables, this project offered the opportunity to strengthen the personal knowledge regarding the DDD patterns and software architecture in general. Even if DDD was introduced many years ago, it is a very current

[2] Application Architecture DDD lectures by Olaf Zimmermann (2017), HSR FHO

topic and different opinions regarding their interpretation and how they can be combined exist. The presented DSL which describes our interpretation of the patterns is open source and publicly available[3]. It is currently still a work in progress and it would be a pleasure to discuss feedback and different opinions with other experts in the field. Contributions[4] are very welcome as well. The achieved results with Service Cutter [20] and our example CML models as input have confirmed that the DDD-based model is suitable for designing components with low coupling and high cohesion. Unfortunately, the poor IDE support is a downside we have to acknowledge. DSLs in general do not seem to be extremely popular anymore, since no other development tools exist which really offer a broad IDE support. However, we have identified this issue and it will hopefully be addressed in one of the next projects. In the next project we will focus on other service decomposition approaches based on DDD and bounded contexts. We further have many ideas for other exciting future projects. The next section presents an overview over potential future work based on our DSL.

5.4 Future Work

The syntax of the DSL currently provides multiple options for expressing certain patterns, especially regarding the relationships between bounded contexts. Identifying the best solution in such a case needs experience and future applications may provide deeper insights. Through the experiences of our first evaluators we have already received valuable feedback and hopefully more users will share their opinions to improve the language. Through experience and user feedback the syntax can be improved and certain variants may be removed in the future. Note that the project is open source and corresponding issues, feature requests and ideas for improvements can be placed there [8].

The support of other Integrated Development Environments (IDEs) than Eclipse seems to be a crucial issue to win more users. Current trends indicate that more and more users switch from Eclipse to other poplar IDEs on the market, such as IntelliJ IDEA [24]. Decoupling the DSL from the current Eclipse plugin and providing support for multiple IDEs would probably increase the acceptance of a novel modeling tool considerably. In Chapter 4, Section 4.1, we summarized the current technological situation by December 2018 and explain why an IntelliJ [24] integration has not been implemented within this proof of concept. However, it is highly desirable to address this issue in a future project.

Within the next project we want to investigate and propose other approaches towards service decomposition which use our DSL. The analysis in Chapter 2 already implied an idea for decomposition using architectural refactorings [55]. This idea could be realized by processing the DSL using model transformation [28]. Further, approaches similar to Service Cutter [20] could be implemented by finding other algorithms and heuristics. Another potential feature is the generation of code or «microservice project stubs». Given a model written in our DSL, project templates using microservice API patterns inspired by Lakeside Mutual [37] could be generated for all given bounded contexts.

[3] `https://contextmapper.github.io/`

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

Appendix A

Language Reference

This appendix contains a reference for the Context Mapper DSL (CML) language, listing all supported Domain-driven Design (DDD) patterns and the corresponding syntax. Note that this reference is based on the Context Mapper release **v1.0.2**[1], released in December 2018. The tactic DDD pattern syntax is not documented within this reference since it is based on the existing Sculptor [46] Domain-specific Language (DSL), except the additionally added Aggregate pattern. For an impression of the most common tactic DDD patterns used in CML we refer to the examples in Appendix B.

Note that this language reference does not explain the DDD patterns itself. We refer to Evans DDD reference [14] for pattern details.

A.1 Language Design

The design of our DSL and its rules is based on the domain model presented in Chapter 2. We also understand this model as the semantic model of our DSL, inspired by Fowler [16]. He introduces the term *semantic model* as a special form of a DSL-related domain model.

Currently we use the Ecore [49] model populated by the Xtext [12] framework as input for our generators directly. If this model which basically corresponds to the Abstract Syntax Tree (AST) tends to deviate from the semantic model substantially in the future, we have to develop a separate implementation of the semantic model on code-level. However, we currently implemented the grammar and thus the AST in a way that almost completely matches the semantic model presented in Chapter 2.

Thereby the domain concepts, concretely the DDD patterns in our case, are getting the first class citizens of the language. With this approach we also comply with Voelter [53] who mentions that a good DSL should be declarative and provide linguistic abstractions for relevant domain concepts. By following this concept it should be possible to understand the domain semantics of a model without sophisticated analysis of the code.

[1] `https://github.com/ContextMapper/context-mapper-dsl/tree/v1.0.2`

A.2 Terminals

The grammar snippets within the language reference use the terminals defined in Listing 19.

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

LISTING 19: Xtext CML Terminals

A.3 Context Map

The context maps grammar rule is shown in Listing 20. 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. At the end of the grammar rule body the bounded context relationships can be added.

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

LISTING 20: Xtext Context Map Grammar Rule

Listing 21 illustrates an example for the context map rule. Note that the order of the *state* and *type* does not matter.

```
ContextMap {
  type = SYSTEM_LANDSCAPE
  state = AS_IS

  contains CargoBookingContext
  contains VoyagePlanningContext
  contains LocationContext

  CargoBookingContext <-> VoyagePlanningContext : Shared-Kernel
}
```

LISTING 21: Syntax example for the ContextMap rule

Listing 22 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 22: 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 23.

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

LISTING 23: Xtext: Relationship Rule

The SymmetricRelationship rule further allows the application of the rules Partnership or SharedKernel (Listing 24).

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

LISTING 24: Xtext: SymmetricRelationship Rule

For the syntax of the Partnership rule we refer to Section A.7. The SharedKernel rule is explained in Section A.8.

The rule UpstreamDownstreamRelationship shown in Listing 25 allows either the application of the CustomerSupplierRelationship rule or directly writing a generic Upstream/Downstream relationship.

```
UpstreamDownstreamRelationship:
  CustomerSupplierRelationship |
  (('@'name=ID)?
    (((upstream = [BoundedContext] 'Upstream-Downstream' downstream = [BoundedContext] |
    (upstream = [BoundedContext] '<-' downstream = [BoundedContext] ':'
                                                          'Upstream-Downstream') |
    (downstream = [BoundedContext] '->' upstream = [BoundedContext] ':'
                                                          'Upstream-Downstream')
  ))
  (OPEN
    ('implementationTechnology' '=' implementationTechnology=STRING)?
    (('upstream' 'implements' (upstreamRoles+=UpstreamRole)
                                              ("," upstreamRoles+=UpstreamRole)*)? &
    ('downstream' 'implements' (downstreamRoles+=DownstreamRole)
                                            ("," downstreamRoles+=DownstreamRole)*)?)
  CLOSE)?))
;
```

LISTING 25: Xtext: UpstreamDownstreamRelationship Rule

With an «@» followed by an ID, the relationship can be named, optionally. As declared in the grammar rule, there are three alternative syntaxes which allow the specification of the same Upstream/Downstream relationship. The Listings 26, 27 and 28 show a corresponding example in all possible ways.

```
1 CargoBookingContext -> LocationContext : Upstream-Downstream
```

LISTING 26: Xtext: Upstream/Downstream Variant 1

```
1 LocationContext <- CargoBookingContext : Upstream-Downstream
```

LISTING 27: Xtext: Upstream/Downstream Variant 2

```
1 LocationContext Upstream-Downstream CargoBookingContext
```

LISTING 28: Xtext: Upstream/Downstream Variant 3

Note that if one of the variants with the arrows (-> or <-) are used, the arrow always points from the downstream towards the upstream, reflecting the dependency (the downstream depends on the upstream).

Within the body of the rule (inside the terminals OPEN and CLOSE), the implementation technology, the upstream roles and the downstream roles can be defined. The corresponding keywords are *implementationTechnology*, *upstream implements* and *downstream implements*. An example is shown in Listing 29.

```
1 VoyagePlanningContext -> LocationContext : Upstream-Downstream {
2     implementationTechnology = "RESTful HTTP"
3     upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
4     downstream implements ANTICORRUPTION_LAYER
5 }
```

LISTING 29: Xtext: Upstream/Downstream Example

The allowed UpstreamRole's and DownstreamRole's are specified by the enums UpstreamRole and DownstreamRole shown in Listing 30.

```
1 enum UpstreamRole:
2   PUBLISHED_LANGUAGE | OPEN_HOST_SERVICE
3 ;
4
5 enum DownstreamRole:
6   ANTICORRUPTION_LAYER | CONFORMIST
7 ;
```

LISTING 30: Xtext: UpstreamRole & DownstreamRole

The alternative UpstreamDownstreamRelationship defined by the role CustomerSupplierRelationship is explained in Section A.9.

A.3.1 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).

A.4 Bounded Context

A bounded context can be defined according to the BoundedContext grammar rule, shown in Listing 31.

```
1  BoundedContext:
2    'BoundedContext' name=ID (('implements' (implementedSubdomains+=[Subdomain])
3    ("," implementedSubdomains+=[Subdomain])*)? & ('realizes' bcRealizedByTeam =
4                                                          [BoundedContext])?)
5    (
6      OPEN
7        (('domainVisionStatement' '=' domainVisionStatement=STRING)? &
8        ('type' '=' type=BoundedContextType)? &
9        (('responsibilities' '=' responsibilities+=Responsibility)
10                                ("," responsibilities+=Responsibility)*)? &
11       ('implementationTechnology' '=' implementationTechnology=STRING)? &
12       ('knowledgeLevel' '=' knowledgeLevel=KnowledgeLevel)?)
13       modules += Module*
14       aggregates += Aggregate*
15     CLOSE
16   )?
17 ;
```

LISTING 31: Xtext: BoundedContext rule

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.

The allowed values for the enum's BoundedContextType and KnowledgeLevel are given by the rules in Listing 32.

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

LISTING 32: Xtext: BoundedContextType & KnowledgeLevel

The Responsibility rule is explained in Section A.14.

The bounded context further allows to contain Modules and Aggregates. Modules are not further explained within this language reference since it is a Sculptor [46] concept. However it is modified and can contain Aggregates in addition to the other Sculptor [46] elements. Aggregates are explained in Section A.16.

With the *implements* keyword it is further possible to define which subdomains the bounded context implements. Figure 33 shows an example for a bounded context specification.

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

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

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

LISTING 33: Xtext: Bounded Context Example

If the bounded context is of the type *TEAM*, it is allowed to use the *realizes* keyword and specify which bounded context is implemented by the team. Listing 34 shows an example for this use case.

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

LISTING 34: Xtext: *realizes* Keyword Example

A.4.1 Bounded Context Semantic Rules

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

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

A.5 Subdomain

The Subdomain pattern is defined by the grammar rule in Listing 35. As on a bounded context (A.4) 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 36.

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

LISTING 35: Xtext: Subdomain Rule

The subdomain further offers the possibility to add entities (Sculptor [46], 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 36: Xtext: SubDomainType enum

A.6 Domain Vision Statement

The Domain Vision Statement pattern is implemented as a description attribute (String) on bounded contexts (A.4) and subdomains (A.5). For the corresponding grammar rules, we refer to Section A.4 and Section A.5. Listing 37 shows an example bounded context with a domain vision statement, and Listing 38 a subdomain accordingly.

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

LISTING 37: Xtext: Domain Vision Statement on Bounded Context

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

LISTING 38: Xtext: Domain Vision Statement on Subdomain

A.7 Partnership

The Partnership relationship pattern is defined by the grammar rule illustrated in Listing 39. With an «@» followed by an ID, the relationship can be named, optionally.

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

LISTING 39: Xtext: Partnership Rule

Note that there are two possible syntax variants to declare the relationship. Listing 40 shows an example for the first syntax variant and Listing 41 for the second respectively.

```
ContractsContext <-> ClaimsContext : Partnership {
  implementationTechnology = "Messaging"
}
```

LISTING 40: Xtext: Partnership Syntax Variant 1

```
ContractsContext Partnership ClaimsContext {
  implementationTechnology = "Messaging"
}
```

LISTING 41: Xtext: Partnership Syntax Variant 2

As the Listings illustrate, both syntax variants allow to declare the implementation technology for the relationship.

A.8 Shared Kernel

The Shared Kernel relationship pattern is defined by the grammar rule illustrated in Listing 42. With an «@» followed by an ID, the relationship can be named, optionally.

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

LISTING 42: Xtext: Partnership Rule

Note that there are two possible syntax variants to declare the relationship. Listing 43 shows an example for the first syntax variant and Listing 44 for the second respectively.

```
CargoBookingContext <-> VoyagePlanningContext : Shared-Kernel {
  implementationTechnology = "Java Library"
}
```

LISTING 43: Xtext: Shared Kernel Syntax Variant 1

```
CargoBookingContext Shared-Kernel VoyagePlanningContext {
  implementationTechnology = "Java Library"
}
```

LISTING 44: Xtext: Shared Kernel Syntax Variant 2

As the Listings illustrate, both syntax variants allow to declare the implementation technology for the relationship.

A.9 Customer/Supplier

The Customer/Supplier relationship pattern is defined by the grammar rule illustrated in Listing 45. With an «@» followed by an ID, the relationship can be named, optionally. Note that Customer/Supplier is a special case of a Upstream/Downstream relationship. Thus, the syntax is principally the same besides the keywords. The *Upstream-Downstream* keyword is replaced with *Customer-Supplier*, the *upstream implements* keyword with *supplier implements* and *downstream implements* is replaced with *customer implements*.

```
CustomerSupplierRelationship:
  ('@'name=ID)?
  (((downstream = [BoundedContext] 'Customer-Supplier' upstream = [BoundedContext]) |
    (downstream = [BoundedContext] '->' upstream = [BoundedContext] ':'
                                                        'Customer-Supplier') |
    (upstream = [BoundedContext] '<-' downstream = [BoundedContext] ':'
                                                        'Customer-Supplier')
  )
  (OPEN
    ('implementationTechnology' '=' implementationTechnology=STRING)?
    (('supplier' 'implements' (upstreamRoles+=UpstreamRole)
                                          ("," upstreamRoles+=UpstreamRole)*)? &
    ('customer' 'implements' (downstreamRoles+=DownstreamRole)
                                        ("," downstreamRoles+=DownstreamRole)*)?)
  CLOSE)?)
;
```

LISTING 45: Xtext: Customer/Supplier Rule

As declared in the grammar rule, there are three alternative syntaxes which allow the specification of the same Customer/Supplier relationship. The Listings 46, 47 and 48 show a corresponding example in all possible ways.

```
CustomerSelfServiceContext -> CustomerManagementContext : Customer-Supplier
```

LISTING 46: Xtext: Customer/Supplier Variant 1

```
CustomerManagementContext <- CustomerSelfServiceContext : Customer-Supplier
```

LISTING 47: Xtext: Customer/Supplier Variant 2

```
CustomerSelfServiceContext Customer-Supplier CustomerManagementContext
```

LISTING 48: Xtext: Customer/Supplier Variant 3

Note that if one of the variants with the arrows (-> or <-) are used, the arrow always points from the customer towards the supplier, reflecting the dependency (the customer depends on the supplier).

Within the body of the rule (inside the terminals OPEN and CLOSE), the implementation technology, the upstream roles and the downstream roles can

be defined. The corresponding keywords are *implementationTechnology*, *supplier implements* and *customer implements*. An example is shown in Listing 49.

```
@Customer_Frontend_Backend_Relationship // Relationship name is optional
CustomerSelfServiceContext -> CustomerManagementContext : Customer-Supplier {
  implementationTechnology = "RESTful HTTP"
  supplier implements PUBLISHED_LANGUAGE
  customer implements ANTICORRUPTION_LAYER
}
```

LISTING 49: Xtext: Customer/Supplier Example

The allowed UpstreamRole's and DownstreamRole's are specified by the enums UpstreamRole and DownstreamRole shown in Listing 50.

```
enum UpstreamRole:
  PUBLISHED_LANGUAGE | OPEN_HOST_SERVICE
;

enum DownstreamRole:
  ANTICORRUPTION_LAYER | CONFORMIST
;
```

LISTING 50: Xtext: UpstreamRole & DownstreamRole

A.9.1 Customer/Supplier Semantic Rules

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

- The Conformist pattern (DownstreamRole) is not applicable in a Customer/Supplier relationship.
- The Open Host Service pattern (UpstreamRole) is not applicable in a Customer/Supplier relationship.
- The Anticorruption Layer pattern (DownstreamRole) shall not be used in a Customer/Supplier relationship.
  - Note that this rule produces a **Warning** only.

A.10 Conformist

The Conformist pattern is implemented as a value of the DownstreamRole enum, as shown in Listing 51.

```
enum DownstreamRole:
  ANTICORRUPTION_LAYER | CONFORMIST
;
```

LISTING 51: Xtext: Conformist implementation

The CONFORMIST role can be used as a role for the downstream context in any Upstream/Downstream relationship. Listing 52 illustrates an example.

```
1 PolicyManagementContext -> CustomerManagementContext : Upstream-Downstream {
2   implementationTechnology = "RESTful HTTP"
3   upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
4   downstream implements CONFORMIST
5 }
```

LISTING 52: Xtext: Conformist Example

A.10.1 Conformist Semantic Rules

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

- The Conformist pattern (DownstreamRole) is not applicable in a Customer/Supplier relationship.

A.11 Open Host Service

The Open Host Service pattern is implemented as a value of the UpstreamRole enum, as shown in Listing 53.

```
1 enum UpstreamRole:
2   PUBLISHED_LANGUAGE | OPEN_HOST_SERVICE
3 ;
```

LISTING 53: Xtext: Open Host Service implementation

The OPEN_HOST_SERVICE role can be used as a role for the upstream context in any Upstream/Downstream relationship. Listing 54 illustrates an example.

```
1 PrintingContext <- PolicyManagementContext : Upstream-Downstream {
2   implementationTechnology = "SOAP"
3   upstream implements OPEN_HOST_SERVICE
4   downstream implements ANTICORRUPTION_LAYER
5 }
```

LISTING 54: Xtext: Open Host Service Example

A.11.1 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.

A.12 Anticorruption Layer

The Anticorruption Layer pattern is implemented as a value of the DownstreamRole enum, as shown in Listing 55.

```
enum DownstreamRole:
  ANTICORRUPTION_LAYER | CONFORMIST
;
```

LISTING 55: Xtext: Anticorruption Layer implementation

The ANTICORRUPTION_LAYER role can be used as a role for the downstream context in any Upstream/Downstream relationship. Listing 56 illustrates an example.

```
DebtCollection -> PrintingContext : Upstream-Downstream {
  implementationTechnology = "SOAP"
  upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
  downstream implements ANTICORRUPTION_LAYER
}
```

LISTING 56: Xtext: Anticorruption Layer Example

A.12.1 Anticorruption Layer Semantic Rules

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

- The Anticorruption Layer pattern (DownstreamRole) shall not be used in a Customer/Supplier relationship.
  - Note that this rule produces a **Warning** only.

A.13 Published Language

The Published Language pattern is implemented as a value of the UpstreamRole enum, as shown in Listing 57.

```
enum UpstreamRole:
  PUBLISHED_LANGUAGE | OPEN_HOST_SERVICE
;
```

LISTING 57: Xtext: Published Language implementation

The PUBLISHED_LANGUAGE role can be used as a role for the upstream context in any Upstream/Downstream relationship. Listing 58 illustrates an example.

```
PolicyManagementContext -> CustomerManagementContext : Upstream-Downstream {
  implementationTechnology = "RESTful HTTP"
  upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
  downstream implements CONFORMIST
}
```

LISTING 58: Xtext: Published Language Example

A.14 Responsibility Layers

A Responsibility is given by the rule in Listing 59 and is either a single ID or an ID with a description string in brackets.

```
Responsibility:
        name=ID ('{' description=STRING '}')?
;
```

LISTING 59: Xtext: Responsibility rule

The responsibilities can be used on bounded contexts and on aggregates, as the Listings 60 and 61 illustrate.

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

LISTING 60: Xtext: Responsibility Layers Example (1)

```
Aggregate Customers {
  responsibilities = Customers, Addresses { "Address description ..." }

  Entity Customer {
    aggregateRoot

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

LISTING 61: Xtext: Responsibility Layers Example (2)

A.15 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 62.

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

LISTING 62: Xtext: KnowledgeLevel enum

Listing 63 shows an example on a bounded context and Listing 64 on an aggregate.

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

LISTING 63: Xtext: Knowledge Level Example (1)

```
Aggregate Customers {
  responsibilities = Customers, Addresses { "Address description ..." }
  knowledgeLevel = CONCRETE

  Entity Customer {
    aggregateRoot

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

LISTING 64: Xtext: Knowledge Level Example (2)

A.16 Aggregate

The Aggregate rule shown in Listing 65 has been added to the Sculptor [46] grammar in order to extend the tactic DDD DSL accordingly. Note that all other tactic DDD patterns are not documented here. We refer to the Sculptor project [46] and their documentation [45].

```
Aggregate :
  (doc=STRING)?
  "Aggregate" name=ID "{"
    ((('responsibilities' '=' responsibilities+=Responsibility)
                    ("," responsibilities+=Responsibility)*)? &
    ('knowledgeLevel' '=' knowledgeLevel=KnowledgeLevel)?)
    ((services+=Service) |
     (resources+=Resource) |
     (consumers+=Consumer) |
     (domainObjects+=SimpleDomainObject))*
  "}"
;
```

LISTING 65: Xtext: Aggregate rule

The aggregate supports the Responsibility Layers pattern (A.14) and the Knowledge Level pattern (A.15) explained in Section A.14 and Section A.15 respectively. 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 [46] DSL, as already mentioned. However, Listing 66 illustrates an example of an aggregate to provide an impression how the rule can be used.

```
Aggregate Contract {
  responsibilities = Contracts, Policies
  knowledgeLevel = CONCRETE

  Entity Contract {
    aggregateRoot

    - ContractId identifier
    - Customer client
    - List<Product> products
  }

  ValueObject ContractId {
    int contractId key
  }

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

LISTING 66: Xtext: Aggregate Example

A.17 Complete CML Grammar

The following Listings 67, 68 and 69 contain the complete CML grammar in the version v1.0.2.

```
grammar org.contextmapper.dsl.ContextMappingDSL with org.contextmapper.tactic.dsl.
                                                                    TacticDDDLanguage

generate contextMappingDSL "http://www.contextmapper.org/dsl/ContextMappingDSL"

ContextMappingModel:
  (
    (map = ContextMap)? &
    (boundedContexts += BoundedContext)* &
    (subdomains += Subdomain)*
  )
;

ContextMap:
  {ContextMap} // make sure there is always a context map
  'ContextMap'
  OPEN
    (('state' '=' state=ContextMapState)? &
    ('type' '=' type=ContextMapType)?)
    ('contains' boundedContexts += [BoundedContext])*
    relationships += Relationship*
  CLOSE
;

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

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

Relationship:
  SymmetricRelationship | UpstreamDownstreamRelationship
;
```

LISTING 67: Xtext: Complete CML grammar (1)

```
57  SymmetricRelationship:
58    Partnership | SharedKernel
59  ;
60
61  Partnership:
62    ('@'name=ID)?
63    ((participant1 = [BoundedContext] 'Partnership' participant2 = [BoundedContext]) |
64    (participant1 = [BoundedContext] '<->' participant2 = [BoundedContext] ':'
65                                                              'Partnership'))
66    (OPEN
67      ('implementationTechnology' '=' implementationTechnology=STRING)?
68    CLOSE)?
69  ;
70
71  SharedKernel:
72    ('@'name=ID)?
73    ((participant1 = [BoundedContext] 'Shared-Kernel' participant2 = [BoundedContext]) |
74    (participant1 = [BoundedContext] '<->' participant2 = [BoundedContext] ':'
75                                                              'Shared-Kernel'))
76    (OPEN
77      ('implementationTechnology' '=' implementationTechnology=STRING)?
78    CLOSE)?
79  ;
80
81  UpstreamDownstreamRelationship:
82    CustomerSupplierRelationship |
83    (('@'name=ID)?
84      (((upstream = [BoundedContext] 'Upstream-Downstream' downstream = [BoundedContext] |
85      (upstream = [BoundedContext] '<-' downstream = [BoundedContext] ':'
86                                                          'Upstream-Downstream') |
87      (downstream = [BoundedContext] '->' upstream = [BoundedContext] ':'
88                                                          'Upstream-Downstream')
89    ))
90    (OPEN
91      ('implementationTechnology' '=' implementationTechnology=STRING)?
92      (('upstream' 'implements' (upstreamRoles+=UpstreamRole)
93                                              ("," upstreamRoles+=UpstreamRole)*)? &
94      ('downstream' 'implements' (downstreamRoles+=DownstreamRole)
95                                          ("," downstreamRoles+=DownstreamRole)*)?)
96    CLOSE)?))
97  ;
98
99  CustomerSupplierRelationship:
100   ('@'name=ID)?
101   (((downstream = [BoundedContext] 'Customer-Supplier' upstream = [BoundedContext]) |
102     (downstream = [BoundedContext] '->' upstream = [BoundedContext] ':'
103                                                        'Customer-Supplier') |
104     (upstream = [BoundedContext] '<-' downstream = [BoundedContext] ':'
105                                                        'Customer-Supplier')
106   )
107   (OPEN
108     ('implementationTechnology' '=' implementationTechnology=STRING)?
109     (('supplier' 'implements' (upstreamRoles+=UpstreamRole)
110                                             ("," upstreamRoles+=UpstreamRole)*)? &
111     ('customer' 'implements' (downstreamRoles+=DownstreamRole)
112                                         ("," downstreamRoles+=DownstreamRole)*)?)
113   CLOSE)?)
114 ;
```

LISTING 68: Xtext: Complete CML grammar (2)

```
115 enum UpstreamRole:
116   PUBLISHED_LANGUAGE | OPEN_HOST_SERVICE
117 ;
118
119 enum DownstreamRole:
120   ANTICORRUPTION_LAYER | CONFORMIST
121 ;
122
123 enum ContextMapState:
124   AS_IS | TO_BE
125 ;
126
127 enum ContextMapType:
128   SYSTEM_LANDSCAPE | ORGANIZATIONAL
129 ;
130
131 enum BoundedContextType:
132   FEATURE | APPLICATION | SYSTEM | TEAM
133 ;
134
135 enum SubDomainType:
136   CORE_DOMAIN | SUPPORTING_DOMAIN | GENERIC_SUBDOMAIN
137 ;
138
139 // define terminals
140 terminal OPEN: '{';
141 terminal CLOSE: '}';
```

LISTING 69: Xtext: Complete CML grammar (3)

Appendix B

Examples

This appendix contains the complete Context Mapper DSL (CML) source code of the examples according to the Context Mapper version v1.0.2 (state December 2018). The current versions of all examples are available in our examples repository [9].

B.1 Insurance Example (Context Map)

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

  /* Add bounded contexts to this context map: */
  contains CustomerManagementContext
  contains CustomerSelfServiceContext
  contains PrintingContext
  contains PolicyManagementContext
  contains RiskManagementContext
  contains DebtCollection

  /* Define the contexts relationships */
  @Customer_Frontend_Backend_Relationship // Relationship name is optional
  CustomerSelfServiceContext -> CustomerManagementContext : Customer-Supplier

  CustomerManagementContext -> PrintingContext : Upstream-Downstream {
    implementationTechnology = "SOAP"
    upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
    downstream implements ANTICORRUPTION_LAYER
  }

  PrintingContext <- PolicyManagementContext : Upstream-Downstream {
    implementationTechnology = "SOAP"
    upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
    downstream implements ANTICORRUPTION_LAYER
  }

  RiskManagementContext <-> PolicyManagementContext : Partnership {
    implementationTechnology = "RabbitMQ"
  }

  PolicyManagementContext -> CustomerManagementContext : Upstream-Downstream {
    implementationTechnology = "RESTful HTTP"
    upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
    downstream implements CONFORMIST
  }
```

LISTING 70: Examples: Insurance Example - Context Map (1)

```
39    DebtCollection -> PrintingContext : Upstream-Downstream {
40      implementationTechnology = "SOAP"
41      upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
42      downstream implements ANTICORRUPTION_LAYER
43    }
44
45    PolicyManagementContext <-> DebtCollection : Shared-Kernel {
46      implementationTechnology = "Shared Java Library, Communication over RESTful HTTP"
47    }
48
49  }
50
51  /* Bounded Context Definitions */
52  BoundedContext CustomerManagementContext implements CustomerManagementDomain {
53    type = FEATURE
54    domainVisionStatement = "The customer management context is responsible for managing
55                             all the data of the insurance companies customers."
56    implementationTechnology = "Java, JEE Application"
57    responsibilities = Customers, Addresses
58
59    Aggregate Customers {
60      Entity Customer {
61        aggregateRoot
62
63        - SocialInsuranceNumber sin
64        String firstname
65        String lastname
66        - List<Address> addresses
67      }
68
69      Entity Address {
70        String street
71        int postalCode
72        String city
73      }
74
75      ValueObject SocialInsuranceNumber {
76        String sin key
77      }
78    }
79
80  }
81
82  BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
83    type = APPLICATION
84    domainVisionStatement = "This context represents a web application which allows the
85                             customer to login and change basic data records like the
86                             address."
87    responsibilities = AddressChange
88    implementationTechnology = "PHP Web Application"
89
90    Aggregate CustomerFrontend {
91      DomainEvent CustomerAddressChange {
92        aggregateRoot
93
94        - UserAccount issuer
95        - Address changedAddress
96      }
97    }
```

LISTING 71: Examples: Insurance Example - Context Map (2)

```
  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 to the
                           other Bounded Contexts."

  Aggregate Printing {
    DomainEvent PrintingJob {
      aggregateRoot

      int printingId
      - Document document
      - Template template
    }

    Entity Document {
      DomainObject source
      String template
    }
  }

  Aggregate Templating {
    Entity Template {
      aggregateRoot

      int templateId
      String templateName
    }
  }
}

BoundedContext PolicyManagementContext implements PolicyManagementDomain {
  type = FEATURE
  domainVisionStatement = "This bounded context manages the contracts and policies of
                           the customers."
  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
    }
```

LISTING 72: Examples: Insurance Example - Context Map (3)

```
    ValueObject ProductId {
      int productId key
    }
  }

  Aggregate Contract {
    Entity Contract {
      aggregateRoot

      - ContractId identifier
      - Customer client
      - List<Product> products
    }

    ValueObject ContractId {
      int contractId key
    }

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

BoundedContext RiskManagementContext implements RiskManagementDomain {
  type = FEATURE
  domainVisionStatement = "Uses data from PolicyManagement context to calculate
                           risks."
  responsibilities = Customer_Risk_Calculation
  implementationTechnology = "Java, Spring App"

  Aggregate Risks {
    Entity CustomerRiskFactor {
      aggregateRoot

      int totalRiskFactor
      - List<Risk> risks
      - Customer client
    }

    ValueObject Risk {
      int likelihood
      String risk
    }
  }
}

BoundedContext DebtCollection implements DebtsDomain {
  type = FEATURE
  domainVisionStatement = "The debt collection context is responsible for the
                           financial income of the insurance company (the debts)
                           which depend on the corresponding contracts and policies."
  responsibilities = Debts, Dunning
  implementationTechnology = "JEE"

  Aggregate Debts {
    Entity Debt {
      aggregateRoot
```

LISTING 73: Examples: Insurance Example - Context Map (4)

```
      int debtNr
      - Policy policy
      Date creationDate
      Date paymentDate
      Date paymentDeadline
      BigDecimal price
      PaymentStatus status
      - List<Dunning> dunnigs
    }

    Entity Dunning {
      int dunningNr
      - Debt debt
      Date dunningDate
      Date paymentDeadline
    }
  }
}

/* Subdomain Definitions */
Subdomain CustomerManagementDomain {
  type = CORE_DOMAIN
  domainVisionStatement = "Subdomain managing everything customer-related."
}
Subdomain PolicyManagementDomain {
  type = CORE_DOMAIN
  domainVisionStatement = "Subdomain managing contracts and policies."
}
Subdomain PrintingDomain {
  type = SUPPORTING_DOMAIN
  domainVisionStatement = "Service (external system) to solve printing for all
                           kinds of documents (debts, policies, etc.)"
}
Subdomain RiskManagementDomain {
  type = GENERIC_SUBDOMAIN
  domainVisionStatement = "Subdomain supporting everything which relates to
                           risk management."
}
Subdomain DebtsDomain {
  type = GENERIC_SUBDOMAIN
  domainVisionStatement = "Subomain including everything related to the
                           incoming money (debts, dunning, etc.)"
}
```

LISTING 74: Examples: Insurance Example - Context Map (5)

B.2 Insurance Example (Team Map)

```
/* Example Context Map written with 'ContextMapper DSL' */
ContextMap {
  type = ORGANIZATIONAL
  state = TO_BE

  /* Add teams to this organizational map: */
  contains CustomersFrontofficeTeam
  contains CustomersBackofficeTeam
  contains ContractsTeam
  contains ClaimsTeam

  /* Define the team relationships */
  @CustomerTeamsRelationship // name of relationship (optional)
  CustomersFrontofficeTeam -> CustomersBackofficeTeam : Customer-Supplier

  ContractsTeam -> CustomersBackofficeTeam : Upstream-Downstream

  ContractsTeam <-> ClaimsTeam : Partnership
}

/* Team Definitions */
BoundedContext CustomersBackofficeTeam implements CustomerManagementDomain realizes
                                                              CustomerManagementContext {
  type = TEAM
  domainVisionStatement = "This team is responsible for implementing the
                           customers module in the back-office system."
}

BoundedContext CustomersFrontofficeTeam implements CustomerManagementDomain realizes
                                                            CustomerSelfServiceContext {
  type = TEAM
  domainVisionStatement = "This team is responsible for implementing the
                           front-office application for the insurance
                           customers."
}

BoundedContext ContractsTeam implements PolicyManagementDomain realizes
                                                                PolicyManagementContext {
  type = TEAM
  domainVisionStatement = "This team is responsible for implementing the
                           contract- and policy-management modules in the
                           back-office system."
}

BoundedContext ClaimsTeam implements RiskManagementDomain realizes
                                                                  RiskManagementContext {
  type = TEAM
  domainVisionStatement = "This team is responsible for for implementing
                           the claims module and providing customer
                           risks information."
}
```

LISTING 75: Examples: Insurance Example - Team Map (1)

```
/* Bounded Context Definitions */
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  domainVisionStatement = "The customer management context is responsible for managing
                           all the data of the insurance companies customers."
  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
    }

    ValueObject SocialInsuranceNumber {
      String sin key
    }
  }

}

BoundedContext CustomerSelfServiceContext implements CustomerManagementDomain {
  type = APPLICATION
  domainVisionStatement = "This context represents a web application which allows
                           the customer to login and change basic data records like
                           the address."
  responsibilities = AddressChange
  implementationTechnology = "PHP Web Application"

  Aggregate CustomerFrontend {
    DomainEvent CustomerAddressChange {
      aggregateRoot

      - UserAccount issuer
      - Address changedAddress
    }
  }

  Aggregate Acounts {
    Entity UserAccount {
      aggregateRoot

      String username
      - Customer accountCustomer
    }
  }
}

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

LISTING 76: Examples: Insurance Example - Team Map (2)

```
  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
    }

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

BoundedContext RiskManagementContext implements RiskManagementDomain {
  type = FEATURE
  domainVisionStatement = "Uses data from PolicyManagement context to calculate risks."
  responsibilities = Customer_Risk_Calculation
  implementationTechnology = "Java, Spring App"

  Aggregate Risks {
    Entity CustomerRiskFactor {
      aggregateRoot

      int totalRiskFactor
      - List<Risk> risks
      - Customer client
    }

    ValueObject Risk {
      int likelihood
      String risk
    }
  }
}
```

LISTING 77: Examples: Insurance Example - Team Map (3)

```
/* Subdomain Definitions */
Subdomain CustomerManagementDomain {
  type = CORE_DOMAIN
  domainVisionStatement = "Subdomain managing everything customer-related."
}
Subdomain PolicyManagementDomain {
  type = CORE_DOMAIN
  domainVisionStatement = "Subdomain managing contracts and policies."
}
Subdomain RiskManagementDomain {
  type = GENERIC_SUBDOMAIN
  domainVisionStatement = "Subdomain supporting everything which relates to risk
                           management."
}
Subdomain DebtsDomain {
  type = GENERIC_SUBDOMAIN
  domainVisionStatement = "Subomain including everything related to the
                           incoming money (debts, dunning, etc.)"
}
```

LISTING 78: Examples: Insurance Example - Team Map (4)

B.3 DDD Sample

```
/* The DDD Cargo sample application modeled in CML. Note that we split the application
                                                    into multiple bounded contexts. */
ContextMap {
  contains CargoBookingContext
  contains VoyagePlanningContext
  contains LocationContext

  /* As Evans mentions in his book (Bounded Context chapter): The voyage planning
   * can be seen as separated bounded context. However, it still shares code with
   * the booking application (CargoBookingContext). Thus, they are in a
   * 'Shared-Kernel' relationship.
   */
  CargoBookingContext <-> VoyagePlanningContext : Shared-Kernel

  /* Note that the splitting of the LocationContext is not mentioned in the
   * original DDD sample of Evans. However, locations and the management around them,
   * can somehow be seen as a separated concept which is used by other
   * bounded contexts. But this is just an example, since we want to demonstrate our
   * DSL with multiple bounded contexts.
   */
  CargoBookingContext -> LocationContext : Upstream-Downstream {
    upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
  }
  VoyagePlanningContext -> LocationContext : Upstream-Downstream {
    upstream implements OPEN_HOST_SERVICE, PUBLISHED_LANGUAGE
  }
}
```

LISTING 79: Examples: DDD Sample - Context Map (1)

```
/* The original booking application context */
BoundedContext CargoBookingContext {
  Module cargo {
    basePackage = se.citerus.dddsample.domain.model

    Aggregate CargoItineraryLegDeliveryRouteSpecification {
      Entity Cargo {
        aggregateRoot

        TrackingId trackingId
        - LocationShared origin
        - RouteSpecification routeSpecification
        - Itinerary itinerary
        - Delivery delivery

        Repository CargoRepository {
          @Cargo find(TrackingId trackingId) throws CargoNotFoundException;
          List<@Cargo> findAll;
          store(@Cargo cargo);
          TrackingId nextTrackingId();
        }
      }

      ValueObject Delivery {
        - TransportStatus transportStatus;
        - LocationShared lastKnownLocation;
        - Voyage currentVoyage;
        boolean misdirected;
        Date eta;
        - HandlingActivity nextExpectedActivity;
        boolean isUnloadedAtDestination;
        - RoutingStatus routingStatus;
        Date calculatedAt;
        - HandlingEvent lastEvent;
      }

      ValueObject HandlingActivity {
        HandlingEvent.Type handlingEventType
        - LocationShared location
        - Voyage voyage
      }

      ValueObject Itinerary {
        ItineraryNumber itineraryNumber
        - List<Leg> legs
      }

      ValueObject Leg {
        - Voyage voyage
        - LocationShared loadLocation
        - LocationShared unloadLocation
        Date loadTime
        Date unloadTime
      }

      ValueObject RouteSpecification {
        - LocationShared origin
        - LocationShared destination
        Date arrivalDeadline
      }

      enum TransportStatus {
        NOT_RECEIVED, IN_PORT, ONBOARD_CARRIER, CLAIMED, UNKNOWN
      }
```

LISTING 80: Examples: DDD Sample - Context Map (2)

```
92        enum RoutingStatus {
93          NOT_ROUTED, ROUTED, MISROUTED
94        }
95
96        Service RoutingService {
97          List<@Itinerary> fetchRoutesForSpecification(@RouteSpecification
98          routeSpecification) throws LocationNotFoundException;
99        }
100
101     }
102   }
103   Module handling {
104     basePackage = se.citerus.dddsample.domain.model
105
106     Aggregate Handling {
107       DomainEvent HandlingEvent {
108         aggregateRoot
109         persistent
110
111         Type handlingType;
112         - Voyage voyage;
113         - LocationShared location;
114         Date completionTime;
115         Date registrationTime;
116         - Cargo cargo;
117
118         Repository HandlingEventRepository {
119           @HandlingHistory lookupHandlingHistoryOfCargo(TrackingId trackingId);
120         }
121       }
122
123       ValueObject HandlingHistory {
124         - List<HandlingEvent> handlingEvents
125       }
126     }
127   }
128 }
129
130 /* We split the Voyage Planning into a separate bounded context as Evans proposes
131  * it in his book. */
132 BoundedContext VoyagePlanningContext {
133   Module voyage {
134     basePackage = se.citerus.dddsample.domain.model
135
136     Aggregate Voyage {
137       Entity Voyage {
138         aggregateRoot
139
140         - VoyageNumber voyageNumber;
141           - Schedule schedule;
142
143         Repository VoyageRepository {
144
145         }
146       }
147
148       ValueObject CarrierMovement {
149         - LocationShared departureLocation;
150         - LocationShared arrivalLocation;
151         Date departureTime;
152         Date arrivalTime;
153       }
154
155       ValueObject Schedule {
156         - List<CarrierMovement> carrierMovements
157       }
```

LISTING 81: Examples: DDD Sample - Context Map (3)

```
      ValueObject VoyageNumber {
        String number
      }
    }
  }
}

/* Separate bounded context for managing the locations. */
BoundedContext LocationContext {
  Module location {
    basePackage = se.citerus.dddsample.domain.model

    Aggregate Location {
      Entity Location {
        aggregateRoot

        PortCode portcode
        - UnLocode unLocode;
        String name;

        Repository LocationRepository {
          @Location find(@UnLocode unLocode);
          List<@Location> findAll();
        }
      }

      ValueObject UnLocode {
        String unLocode
      }

      ValueObject LocationShared {
        PortCode portCode
        - Location location
      }
    }
  }
}
```

LISTING 82: Examples: DDD Sample - Context Map (4)

List of Figures

List of Tables

List of Abbreviations

ACL Anticorruption Layer. 18, 40, 41, 57

API Application Programming Interface. 16, 18, 32, 41

AR Architectural Refactoring. 12

AST Abstract Syntax Tree. 31–33, 39, 40, 46, 48, 65, 67, 101

CML Context Mapper DSL. 32, 33, 36, 39, 42–51, 54, 57–59, 61–64, 67, 68, 83–85, 87

DDD Domain-driven Design. 1–4, 6–9, 16, 17, 19, 22–24, 26, 27, 29, 33, 39, 42, 43, 45, 50, 51, 53, 55, 58, 61, 62, 65–67, 81

DSL Domain-specific Language. 1–10, 12, 19, 20, 22–25, 29, 31–33, 35, 42–44, 48, 56, 58, 59, 61–67, 81, 82

ERD Entity Relationship Diagram. 45, 46, 50

GPL General Public License. 30, 42

IDE Integrated Development Environment. 31, 32, 66

JSON JavaScript Object Notation. 46, 48, 50

LSP Language Server Protocol. 32

NFR Non-Functional Requirement. 24, 25, 42, 62–65

OHS Open Host Service. 16–18, 39–41, 57

OOA Object-oriented Analysis. 20

OOD Object-oriented Design. 20

SCL Service Cutter DSL. 48–51

UML Unified Modelling Language. 1, 2, 45, 56, 62, 64

US User Story. 25

Bibliography

[1] Agile Alliance. *Role-Feature-Reason User Story Template*. https://www.agilealliance.org/glossary/role-feature/. [Online; Accessed: 2018-10-15].

[2] Agile Alliance. *User Stories*. https://www.agilealliance.org/glossary/user-stories. [Online; Accessed: 2018-10-15].

[3] Alberto Brandolini. *Strategic Domain Driven Design with Context Mapping*. https://www.infoq.com/articles/ddd-contextmapping. [Online; Accessed: 2018-10-12].

[4] ANTLR. *ANTLR (ANother Tool for Language Recognition)*. https://www.antlr.org/. [Online; Accessed: 2018-11-29].

[5] Luciano Baresi, Martin Garriga, and Alan De Renzis. "Microservices Identification Through Interface Analysis". In: *Service-Oriented and Cloud Computing*. Ed. by Flavio De Paoli, Stefan Schulte, and Einar Broch Johnsen. Cham: Springer International Publishing, 2017, pp. 19–33. ISBN: 978-3-319-67262-5.

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

[7] Citerus. *DDD Sample*. https://github.com/citerus/dddsample-core. [Online; Accessed: 2018-12-03].

[8] ContextMapper. *ContextMapper DSL: A Domain-specific Language for Context Mapping & Service Decomposition*. https://contextmapper.github.io/. [Online; Accessed: 2018-12-11].

[9] ContextMapper. *ContextMapper Examples*. https://github.com/ContextMapper/context-mapper-examples. [Online; Accessed: 2018-11-30].

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

[11] dsl-platform.com. *DSL Platform: Domain-Driven Design*. https://docs.dsl-platform.com/ddd-foundations. [Online; Accessed: 2018-12-03].

[12] Eclipse Xtext. *Xtext - Language Engineering Made Easy!* https://www.eclipse.org/Xtext/. [Online; Accessed: 2018-11-30].

[13] 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.

[14] 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.

[15] Brian Foote and Joseph Yoder. "Big Ball of Mud". In: *Pattern Languages of Program Design*. Addison-Wesley, 1999, pp. 653–692.

[16] Martin Fowler. *Domain Specific Languages*. 1st. Addison-Wesley Professional, 2010. ISBN: 0321712943, 9780321712943.

[17] P. Di Francesco, P. Lago, and I. Malavolta. "Migrating Towards Microservice Architectures: An Industrial Survey". In: *2018 IEEE International Conference on Software Architecture (ICSA)*. 2018, pp. 29–2909. DOI: 10.1109/ICSA.2018.00012.

[18] fuin.org. *DDD DSL: Xtext based DSL supporting Domain-driven design (DDD)*. https://github.com/fuinorg/org.fuin.dsl.ddd. [Online; Accessed: 2018-12-03].

[19] J. Gouigoux and D. Tamzalit. "From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture". In: *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. 2017, pp. 62–65. DOI: 10.1109/ICSAW.2017.35.

[20] 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.

[21] S. Hassan, N. Ali, and R. Bahsoon. "Microservice Ambients: An Architectural Meta-Modelling Approach for Microservice Granularity". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. 2017, pp. 1–10. DOI: 10.1109/ICSA.2017.32.

[22] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0321200683.

[23] intellij-lsp. *LSP Support for IntelliJ*. https://github.com/gtache/intellij-lsp. [Online; Accessed: 2018-11-30].

[24] JetBrains. *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*. https://www.jetbrains.com/idea/. [Online; Accessed: 2018-11-30].

[25] JetBrains. *MPS: The Domain-Specific Language Creator by JetBrains*. https://www.jetbrains.com/mps/. [Online; Accessed: 2018-11-30].

[26] Munezero Immaculée Josélyne et al. "Partitioning Microservices: A Domain Engineering Approach". In: *Proceedings of the 2018 International Conference on Software Engineering in Africa*. SEiA '18. Gothenburg, Sweden: ACM, 2018, pp. 43–49. ISBN: 978-1-4503-5719-7. DOI: 10.1145/3195528.3195535. URL: http://doi.acm.org/10.1145/3195528.3195535.

[27] Stefan Kapferer. "Architectural Refactoring of Data Access Security". Publication: https://eprints.hsr.ch/564/. Semester Thesis. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2017.

[28] Stefan Kapferer. "Model Transformations for DSL Processing". Seminar Paper [In progress]. University of Applied Sciences of Eastern Switzerland (HSR FHO), 2018.

[29] 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.

[30] Einar Landre, Harald Wesenberg, and Harald Rønneberg. "Architectural Improvement by Use of Strategic Level Domain-driven Design". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. OOPSLA '06. Portland, Oregon, USA: ACM, 2006, pp. 809–814. ISBN: 1-59593-491-X. DOI: 10.1145/1176617.1176728. URL: http://doi.acm.org/10.1145/1176617.1176728.

[31] C. Larman and P. Kruchten. *Applying UML and Patterns: An Introduction to Object-oriented Analysis and Design and the Unified Process*. Safari electronic books. Prentice Hall PTR, 2002. ISBN: 9780130925695.

[32] Duc Minh Le, Duc-Hanh Dang, and Viet-Ha Nguyen. "Domain-driven design using meta-attributes: A DSL-based approach". In: *2016 Eighth International Conference on Knowledge and Systems Engineering (KSE)*. 2016, pp. 67–72. DOI: 10.1109/KSE.2016.7758031.

[33] Ian XY Leung et al. "Towards real-time community detection in large networks". In: *Physical Review E* 79.6 (2009), p. 066107.

[34] G. Mazlami, J. Cito, and P. Leitner. "Extraction of Microservices from Monolithic Software Architectures". In: *2017 IEEE International Conference on Web Services (ICWS)*. 2017, pp. 524–531. DOI: 10.1109/ICWS.2017.61.

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

[36] Michael Plöd. *Michael Plöd's DDD Presentations*. https://speakerdeck.com/mploed. [Online; Accessed: 2018-12-13].

[37] Microservice-API-Patterns. *Lakeside Mutual: Example Application for Microservice API Patterns (MAP) and other patterns (DDD, PoEAA, EIP)*. https://github.com/Microservice-API-Patterns/LakesideMutual. [Online; Accessed: 2018-12-05].

[38] S. Millett. *Patterns, Principles and Practices of Domain-Driven Design*. Wiley, 2015. ISBN: 9781118714706.

[39] Mark EJ Newman and Michelle Girvan. "Finding and evaluating community structure in networks". In: *Physical review E* 69.2 (2004), p. 026113.

[40] Claus Pahl and Pooyan Jamshidi. "Microservices: A Systematic Mapping Study". In: *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*. CLOSER 2016. Rome, Italy: SCITEPRESS - Science and Technology Publications, Lda, 2016, pp. 137–146. ISBN: 978-989-758-182-3. DOI: 10.5220/0005785501370146. URL: https://doi.org/10.5220/0005785501370146.

[41] C. Pautasso et al. "Microservices in Practice, Part 1: Reality Check and Service Design". In: *IEEE Software* 34.1 (2017), pp. 91–98. ISSN: 0740-7459. DOI: 10.1109/MS.2017.24.

[42] plantuml.com. *Open-source tool that uses simple textual descriptions to draw UML diagrams.* http://plantuml.com/. [Online; Accessed: 2018-10-15].

[43] F. Rademacher, J. Sorgalla, and S. Sachweh. "Challenges of Domain-Driven Microservice Design: A Model-Driven Perspective". In: *IEEE Software* 35.3 (2018), pp. 36–43. ISSN: 0740-7459. DOI: 10.1109/MS.2018.2141028.

[44] Florian Rademacher, Sabine Sachweh, and Albert Zündorf. "Towards a UML Profile for Domain-Driven Design of Microservice Architectures". In: *Software Engineering and Formal Methods*. Ed. by Antonio Cerone and Marco Roveri. Cham: Springer International Publishing, 2018, pp. 230–245. ISBN: 978-3-319-74781-1.

[45] Sculptor Project. *Sculptor - Documentation*. http://sculptorgenerator.org/documentation/. [Online; Accessed: 2018-12-03].

[46] Sculptor Project. *Sculptor - Generating Java code from DDD-inspired textual DSL*. https://github.com/sculptor/sculptor. [Online; Accessed: 2018-12-03].

[47] Service Cutter. *Service Cutter Github Wiki*. https://github.com/ServiceCutter/ServiceCutter/wiki. [Online; Accessed: 2018-12-05].

[48] Spoofax. *The Spoofax Language Workbench*. http://www.metaborg.org. [Online; Accessed: 2018-11-30].

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

[50] The Eclipse Foundation. *Eclipse desktop & web IDEs*. https://www.eclipse.org/ide/. [Online; Accessed: 2018-11-30].

[51] The Eclipse Foundation. *IntelliJ IDEA Support For The Xtext Framework and the Xtend Programming Language*. https://github.com/eclipse/xtext-idea. [Online; Accessed: 2018-11-30].

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

[53] Markus Voelter et al. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013. ISBN: 978-1-4812-1858-0.

[54] Frank Witte. *Testmanagement und Softwaretest : theoretische Grundlagen und praktische Umsetzung*. ger. 1. Aufl. 2016. Wiesbaden: Springer Fachmedien Wiesbaden, 2016. ISBN: 978-3-658-09964-0.

[55] 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.

[56] Olaf Zimmermann. "Microservices tenets". In: *Computer Science - Research and Development* 32.3 (2017), pp. 301–310. ISSN: 1865-2042. DOI: 10.1007/s00450-016-0337-0. URL: https://doi.org/10.1007/s00450-016-0337-0.