Domain-driven Architecture Modeling and Rapid Prototyping with Context Mapper

Stefan Kapferer and Olaf Zimmermann

University of Applied Sciences of Eastern Switzerland (OST),
Oberseestrasse 10, 8640 Rapperswil, Switzerland
stefan@kapferer.ch, olaf.zimmermann@ost.ch

Abstract. Strategic Domain-driven Design (DDD) has become an established practice for system decomposition and service identification in recent years. The trend towards microservices increased the popularity of DDD patterns such as Subdomain, Bounded Context, Aggregate and Context Map. In our previous work, we presented a Domain-Specific Language (DSL) providing a clear and concise interpretation of the DDD patterns and their combinations. As a machine-readable description of DDD, the DSL establishes a foundation for systematic service decomposition and DDD-based architecture descriptions that can be refactored and refined by model transformations. The DSL and supporting tools are implemented in the open source project Context Mapper. In this extended version of our previous paper we enhance the DSL grammar to allow domain-driven designers to prototype applications rapidly: they can specify user stories and/or use cases in the DSL, and model transformations can then derive Subdomains and Bounded Contexts automatically. The Context Mapper tool chain supports the continuous, iterative specification and evolution of Context Maps and other service design artifacts. Our validation activities included prototyping, action research, and case studies. This paper illustrates such a transformation chain on the basis of one of our case studies.

Keywords: Domain-driven Design · Domain-specific Language · Microservices · Model-driven Software Engineering · Service-oriented Architecture

1 Introduction

Domain-driven Design (DDD) was introduced in a practitioner book in 2003 [10]. Since then, the DDD patterns, especially tactical ones such as Entity, Value Object, Aggregate, and Repository, have been used in software engineering to model complex business domains. However, strategic DDD has gained even more attention during the last few years in the context of microservices and enterprise application integration [30]. A second generation of DDD experts such as Vernon [39] or Tune and Millet [38] provides advice how to apply the patterns of Evans book in practice.

The decomposition of an application into appropriately sized services is challenging. Achieving high cohesion within the services and loose coupling between them is crucial to keep the application scalable and maintainable. It is not well understood yet how service interfaces can be identified and which patterns and practices are suitable to analyze and design service-oriented systems. DDD can play a key role in answering this question: with patterns such as Bounded Context (an abstraction of systems and teams developing them) and Context Map, it provides an approach for decomposing a domain. Context mapping patterns such as Customer-Supplier, Shared Kernel or Open Host Service can define the relationships between the units of decomposition. However, the strategic patterns come with a certain ambiguity and different interpretations of how they shall be applied. The question how concrete (micro-)services shall be derived from a DDD-based model (and then composed into applications) has only been answered partially so far [31].

2 Context and Previous Work

How to decompose software systems into cohesive modules (or components and services) that are loosely coupled is one of the classic questions and challenges in software engineering. For instance, Parnas [29] already wrote about how to decompose software systems into modules in 1972. Research questions that have not been answered satisfyingly yet include a) which criteria are relevant to find good module boundaries and b) which patterns and practices can be applied to identify the modules or services? [30]. Practitioners in the microservices community suggest to apply the strategic DDD patterns to tackle the problem. They propose to model complex business domains in terms of Bounded Contexts – a sub-system or module that implements a specific part of the domain. A Bounded Context establishes a boundary around a domain model that consist of so-called Aggregates: a set of objects/classes such as Entities or Value Objects. While the terms of the domain may have different meanings outside that boundary, they are clearly defined within the boundary (the so-called "ubiquitous language"). As we described in our previous paper [23], the identification of suited Bounded Contexts is still challenging. Context Maps and context mapping as a practice shall support this process of finding Bounded Contexts. The strategic DDD patterns are used on Context Maps to define the relationships between the Bounded Contexts.

Our experience in the industry has shown that a clear understanding of how these patterns shall work together is often missing, and different stakeholders have different opinions on how these patterns shall be applied and combined. Based on this observation, we derived the following hypothesis [23]:

Software engineers and service designers benefit from a precise interpretation of – and advice on how to apply and combine – the strategic DDD patterns.

We further consider a Context Map an artifact that evolves iteratively. Software architects and DDD adopters develop a Context Map by increasing their knowledge about the problem domain step-by-step. This is why we believe that Context Maps written in a formal language such as our Context Mapper DSL (CML) can be beneficial, since we can offer automated transformation tools that support the evolution of the models. It is further possible to generate other representations such as Unified Modeling Language (UML) diagrams or graphical Context Maps. This has already led us to our second hypothesis [23]:

Adopters of DDD benefit from a tool which supports the creation of DDD pattern-based models in a rigorous and expressive way. They want to transform and evolve such models iteratively.

In our previous paper [23] we presented a meta-model based on the DDD patterns and our CML language that implements that model. We illustrated how Context Mapper users can represent Domains, Subdomains, and Bounded Contexts in CML. We further proposed a set of semantic rules that reflect our interpretation of how the strategic DDD patterns can be combined. Those semantic rules have been implemented as validators for the CML language.

In this paper, we present an extended version of the CML Domain-specific Language (DSL) [23] that allows to prototype Domains and Bounded Contexts rapidly on the basis of use cases [8] and/or user stories [2]. Furthermore, we demonstrate how we validated the usefulness of the language and our hypothesis above by implementing model transformations that support the rapid prototyping. This paper illustrates such a process on an exemplary case.

The remainder of the paper is structured in the following way. Section 3 explains important DDD concepts briefly. It further introduces the meta-model behind the CML language [23] and discusses related work. Section 4 explains our first contribution: the DSL syntax including the latest extensions for feature modeling with use cases and user stories. In Section 5 we introduce a set of model transformations that support rapid prototyping of Domains and Bounded Contexts explained with an exemplary case. This section does a) suggest transformations to derive Bounded Contexts automatically with tool-support, and b) validate whether our language can serve as a foundation for evolving DDD Context Maps step-by-step. Section 6 discusses further validation activities and outlines pros and cons of the presented approach. Section 7 concludes and outlines future work.

3 Domain-driven Design (DDD) Essence, Meta-Model

Since Evans has published his original DDD book [10], other – mostly gray – literature on this topic has been published. Our analysis and interpretation of the patterns is based on the books of Evans [10] and Vernon [39]. Our personal professional experience [20] has influenced the meta-model as well. Additional patterns of Evans' DDD reference [11], which has been published a fews years after his first book, were also considered. We further studied publications of context mapping experts such as Brandolini [5] and Plöd [31,32].

3.1 Motivating Example

Strategic Domain-driven Design (DDD) can be used to decompose the problem domain of a software system into multiple Subdomains and the so-called *Bounded Contexts*. It also allows architects to define the relationships between Bounded Contexts, e.g., how they work together. To explain pattern concepts (and also, in Section 4, the DSL syntax) we use a fictitious insurance software scenario. Figure 1 illustrates the Context Map of the scenario inspired by the visualizations of Vernon [39], Brandolini [5] and Plöd [31].

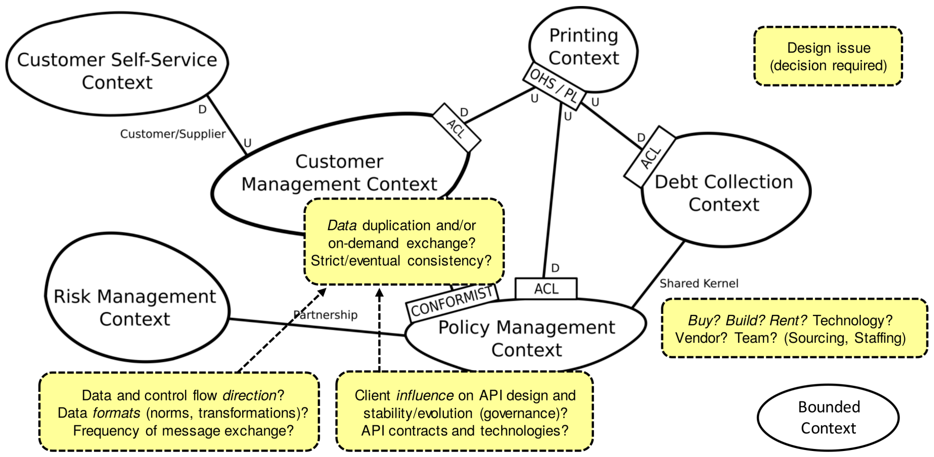
Customer Self-Service Context
D
Customer/Supplier
U
Customer Management Context
ACL
D
Printing Context
OHS / PL
U
U
U
D
ACL
Debt Collection Context
Design issue (decision required)
Data duplication and/or on-demand exchange? Strict/eventual consistency?
Risk Management Context
Partnership
CONFORMIST
ACL
D
Shared Kernel
Policy Management Context
Buy? Build? Rent? Technology? Vendor? Team? (Sourcing, Staffing)
Data and control flow direction? Data formats (norms, transformations)? Frequency of message exchange?
Client influence on API design and stability/evolution (governance)? API contracts and technologies?
Bounded Context

Fig. 1. Insurance Scenario Example Context Map

Figure 1 also highlights a number of design issues that arise when refining the Context Map and domain design. For instance, for each component (or context), it has to be decided whether to buy a software product (or install free software), rent the desired functionality as a cloud service offered by a cloud provider or build it. Connectors (here: relationships between contexts) may have a direction and require integration technologies such as message exchange formats and protocols (such as JSON over HTTP, XML over a mesage queue, etc.). Many data management decisions are required as well (copy or access patters, ownership, update frequencies, etc.). DDD and Context Maps can help identify the need for such decisions, and can also document the decision outcome.

3.2 DDD Patterns

A Bounded Context defines an explicit boundary within which a particular domain model, implementing parts of Subdomains, applies. This boundary affects team organization as well as physical manifestations such as code bases and database schemata. The internal design of a Bounded Context is specified with the tactic DDD patterns, including the *Aggregate* pattern. An Aggregate is a cluster of domain objects (such as *Entities*, *Value Objects*, and *Services*) which is kept consistent with respect to specific invariants and typically also represents a unit of work regarding system (database) transactions. A *Context Map* provides a global view over all Bounded Contexts which are related to the one a team is working on.

The DDD relationship patterns allow modelers to describe how two Bounded Contexts and their development teams work together. The *Partnership* relationship describes an intimate mutual relationship between two Bounded Contexts, since the resulting product of the two can only fail or success as a whole. A *Shared Kernel* relationship indicates that two contexts are very closely related and the two domain models overlap at many places. This pattern is often implemented as a shared library that is maintained by both teams.

Upstream-downstream relationships are marked with a *U* for upstream and a *D* for downstream in our illustration in Figure 1. The terms *upstream* and *downstream* are used to describe relationships in which only one Bounded Context influences the other; the upstream influences the downstream. Thus, the downstream Bounded Context depends on the domain model of the upstream Bounded Context, but not vice versa. A *Customer-Supplier* relationship is given if the downstream Bounded Context in an upstream-downstream relationship has power regarding the implementation decisions of the upstream. The supplier respects the requirements of the downstream in its development plans.

The patterns *Published Language (PL)*, *Open Host Service (OHS)*, *Anticorruption Layer (ACL)* and *Conformist (CF)* are used to describe the interaction between Bounded Contexts in an upstream-downstream relationship. Figure 1 shows them as labels of relationship ends. A Bounded Context can offer an OHS to grant access to a subsystem as a set of open APIs if multiple other Bounded Contexts require access to the same functionality. The PL pattern advises to use a well-documented shared language for communication and translation. Serving as a wrapper, an ACL protects the domain model of a Bounded Context from changes to another one it depends on. In contrast to an ACL, a context applying CF decides to simply conform to the domain model of the other context and must therefore always adjust its model to follow changes of the other context. Due to space limitations we do not explain all pattern details here, but refer the reader to the literature [10,11,31,39].

3.3 DDD Meta-Model for Context Mapper

The meta-model presented in this section is based on the previously explained DDD patterns and our own analysis and understanding regarding how they can be combined. The model is illustrated in Figure 2. It is implemented by our DSL and the Context Mapper tool introduced in Section 4.

The most central element in our meta-model is the Context Map. A Context Map shows Bounded Contexts and their relationships. A Bounded Context implements parts of one or many Subdomains, which can be *Core Domains*, *Supporting Domains* or *Generic Subdomains.* Both a Subdomain and a Bounded

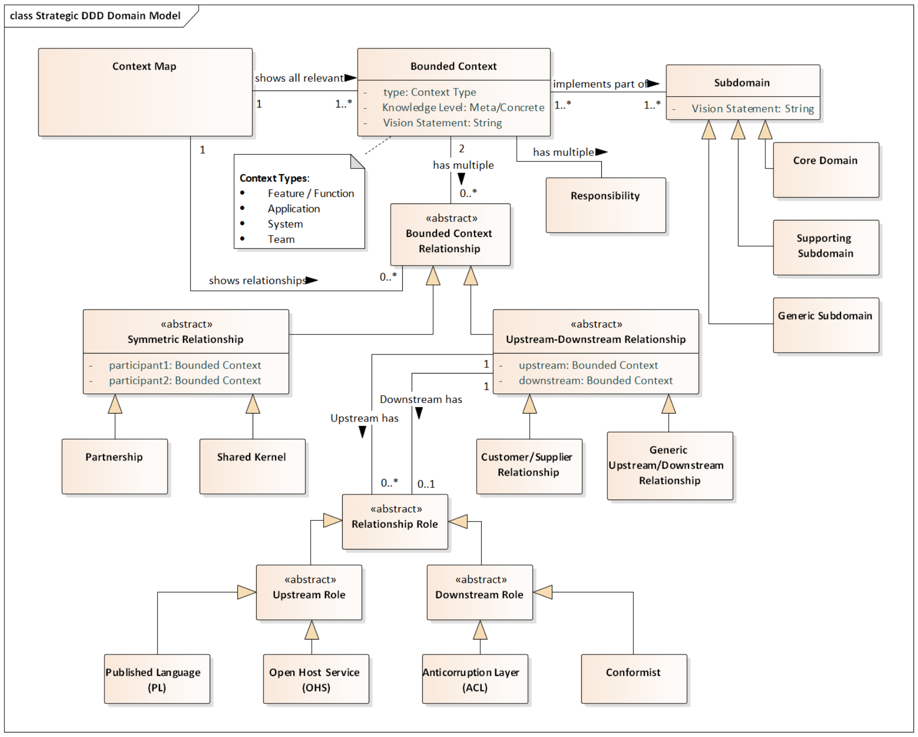
class Strategic DDD Domain Model
Context Map
shows all relevant
1
1..*
Bounded Context
- type: Context Type
- Knowledge Level: Meta/Concrete
- Vision Statement: String
implements part of
1..*
1..*
Subdomain
- Vision Statement: String
Core Domain
Supporting Subdomain
Generic Subdomain
1
2
has multiple
0..*
has multiple
Responsibility
Context Types:
• Feature / Function
• Application
• System
• Team
«abstract» Bounded Context Relationship
shows relationships
0..*
«abstract» Symmetric Relationship
- participant1: Bounded Context
- participant2: Bounded Context
«abstract» Upstream-Downstream Relationship
- upstream: Bounded Context
- downstream: Bounded Context
1
1
Downstream has
Upstream has
Partnership
Shared Kernel
Customer/Supplier Relationship
Generic Upstream/Downstream Relationship
0..*
0..1
«abstract» Relationship Role
«abstract» Upstream Role
«abstract» Downstream Role
Published Language (PL)
Open Host Service (OHS)
Anticorruption Layer (ACL)
Conformist

Fig. 2. Context Mapper: Strategic DDD Meta-Model (UML class diagram) [23]

Context benefit from a statement regarding the vision and purpose of their own part of the domain. Hence, we apply the *Domain Vision Statement* pattern. We further include the *Knowledge Level* pattern on the level of a Bounded Context. The *Responsibility Layers* pattern is implemented by assigning single responsibilities to Bounded Contexts.

We distinguish between *symmetric* and *asymmetric* relationships between Bounded Contexts: We call asymmetric relationships *upstream-downstream* relationships in our meta-model. This is in line with the terminology in the DDD literature. In an upstream-downstream relationship only one context depends on the other. Likewise, only one Bounded Context influences the other; the upstream-downstream metaphor indicates an *influence flow* between teams and systems as discussed by [31]. The Partnership and Shared Kernel patterns, on the other hand, describe symmetric relationships. The Bounded Contexts involved in such relationships are mutually dependent on another.

The remaining patterns Published Language (PL), Open Host Service (OHS), Anticorruption Layer (ACL) and Conformist (CF) are roles taken by the upstream or downstream context within an upstream-downstream relationship. OHS and PL are patterns implemented by the upstream, which exposes parts of the model to be used by the downstream. The CF and ACL patterns are implemented by the downstream, which decides to either conform to the model exposed by the upstream or protect itself from changes (ACL).

According to our analysis, the Customer-Supplier pattern is a special case of upstream-downstream. We indicated this in Figure 2 by distinguishing between Customer-Supplier and *generic* upstream-downstream relationships.

In addition to this meta-model we presented a set of semantic rules in our previous paper [23]. Those rules reflect our own interpretation of the DDD patterns and state which combinations are allowed and which are not allowed according to this interpretation.

3.4 Architectural Viewpoints

Bounded Contexts are created for different reasons and can be seen from different perspectives. Brandolini [5] presents a comprehensive introduction into context mapping and explains different scenarios for the evolution of Bounded Contexts. In our DSL we implemented an additional attribute *context type* to reflect different reasons for creation. We see these types as different viewpoints corresponding to the 4+1 view model of software architecture [25]. Simon Brown's C4 model [6] is another but very similar approach to visualize software architecture from different perspectives. Table 1 lists the four context types, **F**eature, **A**pplication, **S**ystem and **T**eam and compares them with the perspectives of 4+1 and C4.

Table 1: FAST Context Types

Type	**Description and Mapping to Related Work**
Feature or **F**unction	This is a Bounded Context representing a feature or requirement which has been identified by the Object-oriented Analysis (OOA). In terms of the 4+1 model [25], it represents a context from the *Scenario* viewpoint. The system context view (level 1) of the C4 model shows such contexts and their relationships.
Application	Such a Bounded Context represents a certain application. It is evolved by Object-oriented Design (OOD) and from our understanding reflects the *Logical* and *Development* viewpoint in terms of 4+1 [25]. The C4 model does not differentiate between features or applications. Therefore *application* contexts map to the system context view as well (level 1). Its tactic DDD content (Aggregates with their Entities, Services, etc.) can be seen as C4 components.
System	A Bounded Context representing an physical system, container, or application tier. This type maps to the *physical* and/or *process* viewpoint in the 4+1 model [25]. The latter perspective is concerned with the way systems communicate and integrate with each other, for example by implementing Enterprise Integration Patterns (EIP) [18]. *System* Bounded Contexts correspond to the containers in the container diagram of C4.

Table 1: FAST Context Types (continued)

Type	**Description**
Team	A Team context represents a small organisational unit. A new context of this type might be created when a team has to be split to scale the company. This cross-cutting perspective is inspired by Conway's Law [9], stating that a systems design copies the communication structures of an organization. There are no corresponding concepts in 4+1 or C4.

The model transformations presented in Section 5 make use of the types *Feature* and *System*; the design of *Application* contexts remains manual work (requiring creativity and problem solving skills). The rapid prototyping process leads from user requirements to a Feature Bounded Context first. Later, the context specifications get more detailed and we switch the perspective to systems. Section 5 explains the process in detail.

3.5 Related Work

Decomposing monolithic systems into microservice architectures [42] is a topic with a huge attention within the last years not only in the industry but in the academic field as well [4,13,14,17,19,28]. Furthermore, DDD with its Bounded Contexts promises to ease this challenging task [13,19,26,28,30,33]. 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 [34] 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 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 the contexts are not mentioned explicitly.

Le et al. [27] 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 for tactic DDD patterns exist, such as *Sculptor*[1], *fuin.org's DDD DSL*[2] and *DSL Platform*[3]. Further approaches and

[1] `http://sculptorgenerator.org/`
[2] `https://github.com/fuinorg/org.fuin.dsl.ddd`
[3] `https://docs.dsl-platform.com/dsl-concepts`

projects based on annotations exist as well. None of these covers the strategic DDD patterns concerning the relationships between Bounded Contexts.

Informal graphical representations of Context Maps and the strategic DDD patterns were introduced by Brandolini [5] and Vernon [39]. Plöd proposed a formal graphical notation for Context Maps [31], which has not been implemented in a tool yet.

A less formal approach towards the identification of Bounded Context is "Event Storming", invented by Brandolini[4]. In our online documentation[5] we discuss how event storming results can be formalized with Context Mapper. More advice how to decompose a system into Bounded Contexts can be found in the gray literature[6] [7] [8]. However, the authors of these online resources focus on providing advice, best practices and heuristics, but do not offer formal approaches and concrete transformation tools as Context Mapper does.

4 Context Mapper DSL (CML)

We implemented the *Context Mapper*[9] tool that allows software architects to model systems according to the DDD meta-model introduced in the previous section. The following DSL examples are based on the insurance scenario introduced in Section 3. The complete example can be found in our examples repository[10].

4.1 Domains and Subdomains

Before thinking in terms of Bounded Contexts, DDD practitioners typically start discovering and analyzing a domain by decomposing it into Subdomains. As we explain in Section 5, we call this the *domain analysis* phase.

Domains and Subdomains in CML are declared as illustrated in Listing 4.1 and Listing 4.2. A Subdomain is of the type *Core Domain*, *Supporting Subdomain* or *Generic Subdomain* according to our meta-model and [10].

Listing 4.1. Subdomain Syntax in CML

```
Domain Insurance {
  Subdomain CustomerManagementDomain {
    type = CORE_DOMAIN
    domainVisionStatement = "Customer-related entities..."

    Entity Customer
    Entity Address
```

[4] https://ziobrando.blogspot.com/2013/11/introducing-event-storming.html
[5] https://contextmapper.org/docs/event-storming/
[6] https://leanpub.com/ddd-by-example/
[7] https://medium.com/nick-tune-tech-strategy-blog/
[8] https://github.com/ddd-crew/
[9] https://contextmapper.org/
[10] https://github.com/ContextMapper/context-mapper-examples/

Listing 4.2. Subdomain Syntax in CML (continued)

```
    Service CustomerService {
      createCustomer;
      changeAddress;
    }
  }

  Subdomain PolicyManagementDomain {
    type = CORE_DOMAIN

    Entity Contract
    Entity Policy
  }

  Subdomain PrintingDomain {
    type = SUPPORTING_DOMAIN
  }
}
```

The CML language allows users to specify which Entities (domain objects) are part of which Subdomains. With Services it is further already possible to declare operations that will be required.

4.2 Bounded Contexts

From the *domain analysis* with the Subdomains as result we typically move onto the *stratgic DDD* phase where the models become more concrete and organized within Bounded Contexts. Listing 4.3 shows the declaration of the *Customer-ManagementContext* as an example for a Bounded Context in CML. A Bounded Context has a type as already explained in Section 3. The following attributes are implementations of the Domain Vision Statement and the Responsiblity Layers patterns. The user can further specify the implementation technology of a Bounded Context. A Bounded Context consists of one or more Aggregates. Inside the Aggregates the language supports the usage of all tactical DDD patterns to fully specify the domain model of the Bounded Context. The implementation of CML inside the Aggregates is based on the Sculptor[11] project.

Listing 4.3. Bounded Context Syntax in CML

```
BoundedContext CustomerManagementContext implements CustomerManagementDomain {
  type = FEATURE
  domainVisionStatement = "The customer context ..."
  responsibilities = "Collects and exposes customer data",
                     "Manages the customers addresses"
  implementationTechnology = "Java, JEE Application"

  Aggregate Customers {
    Entity Customer {
      aggregateRoot

      String firstname
      String lastname
    }
  }
}
```

[11] http://sculptorgenerator.org/

With the *implements* keyword we refer back to the analysis part and specify which Subdomains are implemented by a specific Bounded Context. Note that a Bounded Context not necessarily implements a complete Subdomain.

4.3 The Context Map

The central and most important structure of CML is the Context Map which specifies the relationships between Bounded Contexts. Listing 4.4 shows a small example of a Context Map written in CML. The *contains* keyword indicates the Bounded Contexts that are added to the Context Map. They can then be used to declare relationships.

Listing 4.4. Context Map Syntax in CML

```
ContextMap {
  contains CustomerManagementContext, PolicyManagementContext

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

Listing 4.4 also features an exemplary upstream-downstream relationship. The endpoints of this relationship apply three more patterns, Open Host Service (OHS), Published Language (PL) and Conformist (CF).

4.4 Relationship Syntax

For symmetric relationships the syntax uses an arrow directing to both Bounded Contexts ($< - >$), whereas asymmetric relationships use an arrow ($- >$ or $< -$) pointing from the upstream towards the downstream. In all cases, the relationship roles are declared within brackets as illustrated in Listing 4.4. Note that the declaration of the implementation technology is optional and we omit it in the following examples.

Partnership Listing 4.5 shows an example for the Partnership (P) pattern, which is a symmetric relationship.

Listing 4.5. Partnership Pattern Syntax in CML

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

Shared Kernel The second symmetric relationship is the Shared Kernel (SK). The syntax is identical to the Partnership. Listing 4.6 illustrates an example.

Listing 4.6. Shared Kernel Pattern Syntax in CML

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

Generic Upstream-Downstream Relationship As already mentioned, the upstream-downstream (or asymmetric) relationships use an arrow from the upstream towards the downstream, expressing the influence flow. This syntax states which Bounded Context is upstream and which one is downstream in an expressive way. The arrowhead can be placed either on the left or on the right. Thus, the declaration examples in Listings 4.7 and 4.8 are semantically equal.

Listing 4.7. Upstream-Downstream Relationship in CML (1)

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

Listing 4.8. Upstream-Downstream Relationship in CML (2)

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

Upstream-Downstream Roles The upstream and downstream roles Open Host Service (OHS), Published Language (PL), Anticorruption Layer (ACL) or Conformist (CF) are listed within the brackets after the upstream (U) and downstream (D) specification. Listing 4.9 illustrates an example with the OHS and PL patterns on the upstream side and the ACL pattern on the downstream side.

Listing 4.9. Upstream-Downstream Relationship with Roles

```
PrintingContext [U,OHS,PL]->[D,ACL] PolicyMgmtContext
```

Customer-Supplier Relationship The customer-supplier relationship is a special case of an upstream-downstream relationship in which the upstream is called supplier and the downstream is called customer. The syntax is therefore almost identical to the generic upstream-downstream relationship; to state that the upstream-downstream relationship is a customer-supplier relationship the user has to add the abbreviations S for supplier and C for customer. These abbreviations must appear behind the U/D, but before the relationship roles, as shown in Listing 4.10.

Listing 4.10. Customer-Supplier Relationship in CML (1)

```
SelfServiceContext [D,C,ACL]<-[U,S,PL] CustomerMgmtContext
```

However, since the upstream in a customer-supplier relationship is always the supplier and the downstream is always the customer, it is also possible to omit the U and D abbreviations in this case. Thus, the declaration in Listing 4.11 is semantically equal to the one in Listing 4.10.

Listing 4.11. Customer-Supplier Relationship in CML (2)

```
SelfServiceContext [C,ACL]<-[S,PL] CustomerMgmtContext
```

4.5 Expressing User Requirements

In addition to the CML concepts presented above and in our previous publication [23], we enhanced Context Mapper to express features in the form of use cases [8] or user stories [2]. This grammar feature allowed us to realize the rapid prototyping process introduced in Section 5.

Listing 4.12 illustrates a user story written in CML. The syntax corresponds to the "role-feature-reason" format invented at Connextra in the UK and published by the Agile Alliance [2]. Note that we extended the template with the "with its" and "for a" parts so that one can model attributes and references to other entities. These elements are not part of the original template [2]. However, they are optional in CML; we hypothesize that both domain experts and software designers can adopt such an extension (which is subject to validation).

Listing 4.12. User Story in CML

```
UserStory ManageCustomers {
  As an "Insurance Employee"
    I want to "create" a "Customer" with its "firstname", "lastname"
    I want to "update" an "Address" for a "Customer"
    I want to "create" a "Contract" for a "Customer"
  so that "I am able to manage the customer data and offer them contracts."
}
```

In addition, to reduce code duplication CML allows modellers to add multiple "I want to" parts per user story as shown in Listing 4.12. This is a slight deviation from the original template [2] as well. Listing 4.13 illustrates how the same user requirement can be formulated as a use case in CML.

Listing 4.13. Use Case in CML

```
UseCase ManageCustomers {
  actor "Insurance Employee"
  interactions
    "create" a "Customer" with its "firstname", "lastname",
    "update" an "Address" for a "Customer",
    "create" a "Contract" for a "Customer"
  benefit "Being able to manage the customers data and offer them contracts."
  scope "Insurance Application"
  level "Summary"
}
```

The attributes *actor*, *interactions*, and *benefit* cover the same information as the user story format seen before. With the additional attributes *scope* and *level* we support expressing use cases according to the brief or casual format suggested by A. Cockburn [8].

We have shown the core concepts of CML Context Maps above. Due to space limitations we cannot present all abilities of our language. CML currently also supports an alternative syntax to declare relationships for A/B testing purposes. All language features are documented online[12] and the complete insurance example can be found in our examples repository[13]. In the next section we introduce one approach how we validated our modeling language and our hypothesis by

[12] https://contextmapper.org/docs
[13] https://github.com/ContextMapper/context-mapper-examples

providing transformation tools that allow users to prototype an application in terms of DDD patterns rapidly.

5 Language and Tool Extension: Rapid Prototyping

The Context Mapper DSL (CML) is based on the Xtext[14] language framework. The models behind the textual representation are Eclipse Modeling Framework (EMF) [37] models. Therefore, we can support the evolution of CML models by providing model transformations [22]. Starting from the user story [2] or use case [7] syntax introduced in Section 4, we designed and implemented three novel model transformations that support rapid prototyping. The transformations do not aim at replacing human design work but capture some proven analysis and design heuristics from the literature and online resources.

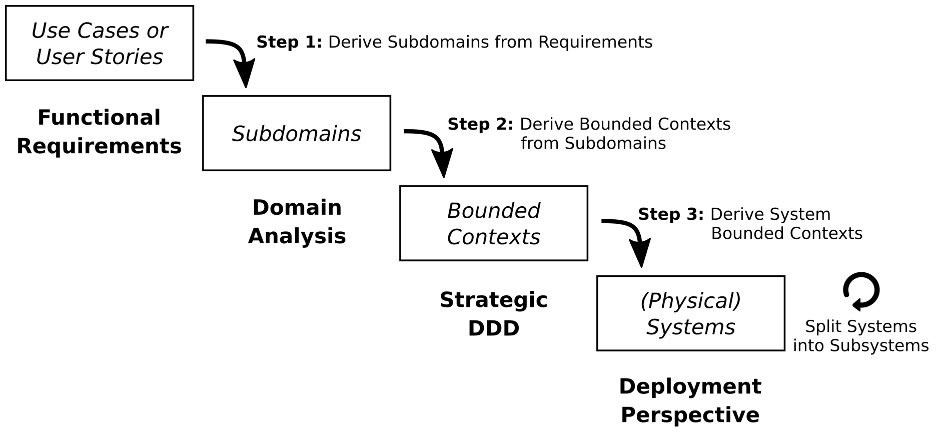
Use Cases or User Stories
Step 1: Derive Subdomains from Requirements
Functional Requirements
Subdomains
Step 2: Derive Bounded Contexts from Subdomains
Domain Analysis
Bounded Contexts
Step 3: Derive System Bounded Contexts
Strategic DDD
(Physical) Systems
Split Systems into Subsystems
Deployment Perspective

Fig. 3. Rapid Prototyping Transformation Steps

Figure 3 illustrates the steps and provided transformations. A domain modeler can specify requirements in the form of user stories [2] or use cases [8] as an initial step. The following model transformations support him/her in deriving Subdomains and Bounded Contexts from these requirements. Hence, the CML language is able to represent all stages of the process: requirements (use cases and/or user stories), Subdomains, and Bounded Contexts (of the different architectural viewpoints explained in Section 3).

The Exemplary Case In order to validate our use case grammar we modeled a case of A. Cockburn's book [8] in CML. The following Listing 5.1 shows the use case "Get paid for car accident" (we stay in the insurance domain) written

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

in our DSL. The interactions in the CML use case correspond to the six steps described by Cockburn [8].

Listing 5.1. "Get paid for car accident" in CML

```
UseCase Get_paid_for_car_accident {
  actor "Claimant"
  interactions
   "submit" a "Claim" with its "date", "amountClaimed", "desc" for a "Policy",
   "verifyExistanceOf" "Policy" with its "startDate", "endDate" for a "Contract",
   "assign" an "Agent" with its "personalID", "firstName", "lastName" for "Claim",
   "verify" "Policy" for a "Contract",
   "pay" "Claimant" with its "firstName", "lastName",
   "close" "Claim" for "Claimant"
  benefit "Claimant submits claim and and gets paid from the insurance company."
  scope "Insurance company"
  level "Summary"
}
```

Step 1: Derive Subdomains From Requirements Context Mapper offers a model transformation that produces a Subdomain definition given a set of requirements or features as shown in Listing 5.1 as input. From the use case in Listing 5.1 the transformation creates the Subdomain illustrated by Listing 5.2.

Listing 5.2. Subdomain Derived From Use Case

```
Domain Insurance_Application {
  Subdomain ClaimsManagement {
    domainVisionStatement "Aims at promoting: A claimant submits a claim and ..."
    Entity Claim {
      Date date
      Double amountClaimed
      String description
      - Agent agent
    }
    Entity Policy {
      Date startDate
      Date endDate
      - List<Claim> claims
    }
    Entity Contract {
      - List<Policy> policies
    }
    Entity Agent {
      Long personalID
      String firstName
      String lastName
    }
    Entity Claimant {
      String firstName
      String lastName
      - List<Claim> claims
    }
    Service AccidentService {
      submitClaim;
      verifyExistanceOfPolicy;
      assignAgent;
      verifyPolicy;
      payClaimant;
      closeClaim;
    }
  }
}
```

The transformation uses the verbs, Entity names, and attributes mentioned in the interactions to derive the elements of the Subdomain. The user selects the use cases and user stories that will be jointly mapped to a single Subdomain, thereby controlling the placement of Entities and Services. This makes it possible to group by high cohesion, low coupling criteria at an early stage, while still analyzing the domain and the requirements. These placement decisions can be revised later on. The goal of this step is to break the domain down into sets of Entities and operations that belong together according to the business/domain. For example: typical Subdomains in the insurance domain are customers, contracts/policies, claims.

The mapping of the transformation from Listing 5.1 to Listing 5.2 (Step 1) is trivial. Figure 4 illustrates it more explicit. The single interactions contain a *verb*, an *entity name*, *entity attributes*, and optionally a *reference to another Entity*. Based on this information we derive Entities with attributes and references, and Services with operations for the Subdomain. For example, the interaction shown in Figure 4 leads to the Entity called *Claim* and the Service operation called *submitClaim* in the resulting Subdomain.

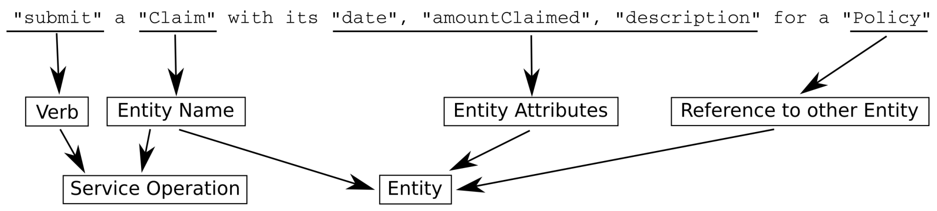
"submit" a "Claim" with its "date", "amountClaimed", "description" for a "Policy"
Verb
Entity Name
Entity Attributes
Reference to other Entity
Service Operation
Entity

Fig. 4. Model transformation mapping: Use Case to Subdomain (Entities and Services)

Step 2: Derive Feature Bounded Context This step is performed by application designers and software architects when transitioning from analysis to design. Context Mapper can derive a Bounded Context of the type *Feature* (see Architectural Viewpoints in Section 3) automatically from the Subdomain illustrated above (Step 2).

In this transformation step the user can select a set of Subdomains to be mapped into one Bounded Context; one Bounded Context can implement parts of multiple Subdomains [39]. In case multiple Subdomains are involved, we map each Subdomain into one Aggregate of the Bounded Context. The transformation further increases the level of detail in the Service operations and introduces parameters and return types. The resulting Bounded Context for our example use case is shown in Listing 5.3. Thus, the input of the transformation are multiple Subdomains and the output is one Bounded Context. The user is in control again and decides which Subdomains shall be implemented as one Bounded Context. The purpose of this step is to organize the implementation of the Subdomains.

Listing 5.3. Bounded Context (Feature) Derived From Subdomain

```
BoundedContext ClaimsManagement implements ClaimsManagement {
  domainVisionStatement "Realizes the following subdomains: ClaimsManagement"
  type FEATURE
  /* Contains the entities and services of the 'ClaimsManagement' subdomain.
   * TODO: You can now refactor the aggregate, for example by ...
   * TODO: Add attributes and operations to the entities.
   * TODO: Add operations to the services.
   * Find examples and further instructions on our website:
   * https://contextmapper.org/docs/rapid-ooad/ */
  Aggregate ClaimsManagementAggregate {
    Service Get_paid_for_car_accidentService {
      boolean submitClaim (@Claim claim);
      boolean verifyExistanceOfPolicy (@Policy policy);
      boolean assignAgent (@Agent agent);
      boolean verifyPolicy (@Policy policy);
      boolean payClaimant (@Claimant claimant);
      boolean closeClaim (@Claim claim);
    }
    Entity Claim {
      String date
      String amountClaimed
      String description
      ClaimId claimId
      - List<Agent> agentList
    }
    Entity Policy {
      String startDate
      String endDate
      PolicyId policyId
      - List<Claim> claimList
    }
    Entity Contract {
      ContractId contractId
      - List<Policy> policyList
    }
    Entity Agent {
      String personalID
      String firstName
      String lastName
      AgentId agentId
    }
    Entity Claimant {
      String firstName
      String lastName
      ClaimantId claimantId
      - List<Claim> claimList
    }
  }
}
```

The generated Bounded Context contains ”TODO” hints/comments that help the modeler to refine and detail the design. Note that the transformation produces generic parameter and return types in case they cannot be mapped to Entities automatically. Context Mapper users can indicate that they refined the Bounded Context setting its type to *Application*. The transformation in the next step supports contexts of the type *Feature* as well as *Application* as input.

Given such a Bounded Context of the type *Feature*, Context Mapper is already able to generate a running Java application in a few steps. We do not discuss code generation in this paper, but we documented how users can generate a Java application using Context Mapper and JHipster[15] in our online tutorial[16]. The tool generates one Microservice for each Bounded Context in the CML model.

Step 3: Derive System Bounded Contexts Bounded Contexts of the type *Feature* represent a boundary around specific features as already explained in Section 3. In this chain described here, we map Bounded Contexts of the type *Feature* one-to-one to Bounded Contexts of the type *Application*. Therefore, Step 3 in our transformation process already changes the architectural viewpoint to physical systems; Bounded Contexts of the type *System*. Currently, Context Mapper offers a transformation to transform a *Feature* Bounded Context (or *Application* Bounded Context) into two *System* Bounded Context: a frontend and a backend system. Listing 5.4 illustrates the result for our use case. Note that we do not repeat the contents of the Aggregates to save space at this point. Based on the domain model seen in Listing 5.3 this transformation generates an Aggregate in the backend context and a view model (technically an Aggregate as well) in the frontend context. The transformation takes one Bounded Context as input and produces two new Bounded Contexts. The goal of this step is to break an application down into its deployment units, tiers, or technical building blocks.

Listing 5.4. Bounded Context (System) Derived From Feature Context

```
ContextMap {
  contains ClaimsManagementFrontend, ClaimsManagementBackend

  ClaimsManagementBackend [ PL ] -> [ CF ] ClaimsManagementFrontend {
    implementationTechnology "RESTful HTTP"
    exposedAggregates ClaimsManagementAggregate
  }
}

BoundedContext ClaimsManagementBackend implements ClaimsManagement {
  domainVisionStatement "Realizes the following subdomains: ClaimsManagement"
  type SYSTEM
  implementationTechnology "Java, Spring Boot"
  Aggregate ClaimsManagementAggregate {
    // removed contents to save space
  }
}

BoundedContext ClaimsManagementFrontend implements ClaimsManagement {
  domainVisionStatement "Realizes the following subdomains: ClaimsManagement"
  type SYSTEM
  implementationTechnology "Angular"
  Aggregate ViewModel {
    // removed contents to save space
  }
}
```

In addition, the transformation in Step 3 creates a Context Map with a relationship that illustrates the information flow between the frontend and the backend system.

[15] https://www.jhipster.tech/

[16] https://contextmapper.org/docs/jhipster-microservice-generation/

Steps 4 to n: Continue Decomposing Into Subsystems Finally, we offer a transformation "Split System Context Into Subsystems" that allows users to further decompose a system into more subsystems or deployment units. The input for this transformation is always one Bounded context, and the output are two Bounded Contexts (split one into two). For example: one could split the backend tier into a *domain logic* and a *database* tier.

Besides the transformations presented above we realized a set of Architectural Refactorings (ARs)[17] [36] that support the continuous improvement of the design. They allow Context Mapper users to further *split* or *merge* Bounded Contexts and Aggregates, or *extract* parts of the domain model into new Bounded Contexts. We discuss our ARs in a separate paper [24] in more detail. All these transformation tools supported us in applying the presented modeling language in case studies and self experiments with the goal to validate the practicability of the DSL. The next section lists all our validation activities in more detail and discusses strengths and weaknesses of the approach.

6 Validation and Discussion

Goals and Techniques We validated our approach according to Shaw's recommendations [35] with the goal to demonstrate correctness, usefulness and effectiveness according to the validation type "experience" [35]. Having designed our meta-model we implemented a prototype, the first version of our DSL, to validate the model. We made the tool available for download allowing practitioners to evaluate it (including ourselves when working in industry projects). To validate the implementation we applied empirical validation techniques such as prototyping, case study [40], and action research [3].

Conducted Validation Activities The prototypical *implementation* of the tools allowed us to evaluate the language, its abilities, and our hypothesis that the DSL can provide a foundation for service design and system decomposition.

We conducted several self-experiments and *action research*, including modeling Cockburn's sample use case [8] explained in the previous Section 5. We also demonstrated the tool to DDD thought leaders [39], peers and interested practitioners; one of the authors demonstrated another end-to-end example of the rapid prototyping chain at ICWE 2020[18]. Feedback from these demonstrations was continuously incorporated into our research cycles and development sprints. Since November 2018 we published 50 Context Mapper releases.

Next, the rapid OOAD/DDD toolchain was used in a two-hour service design workshop with five software architects with multiple years of experience in professional services (enterprise application development and integration); they were familiar with strategic and tactic DDD. One of the authors received a list of

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

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

three service design questions (also outlining one user story/use case) two days before the workshop: a) should services be flexible and generic/broad or specific and narrow? b) how does database design and the service autonomy tenet influence service granularity? c) should system or business transactions (sagas) be used? He was able to model the story and sample DDD designs for a) and b) within one hour (supported by the transformations). Question c) was also discussed but pertains to the service implementation rather than the API, so was deemed out of scope. The draft model was shown in the workshop and another story modeled. This helped ground the discussion and focus on a concrete example.

During the implementation of the Context Mapper tool we also applied action research to validate and improve the DSL iteratively and with short feedback cycles. One of the authors modeled the *Lakeside Mutual*[19] project, an example application for microservice API patterns (MAP) [43], with CML to validate the tool with a practical application. As another case study we modeled the "Cargo Tracking" sample application [10] to validate the tool and its compatibility with the original DDD concepts. Furthermore, we conducted a *case study* on a real-world project in the health-care sector [16]. The hardened syntax was also used to model another case, the microservices in an existing production system for document management. An architect of the system and one of the paper authors cooperated for this validation activity. Previous models were updated to feature and re-validate the revised syntax. We further used the tool as part of an exercise accompanying the DDD lesson of the software architecture course at our institution and collected the feedback of the nearly 20 exercise participants. Thereby we were able to evaluate the simplicity of the DSL and improve the syntax and tooling. The observations conducted by modeling these applications influenced the improvements of our DSL substantially. The CML syntax introduced in Section 4 is a revised version which improved *writability*, *readability* and *consistency with meta-model and DDD patterns* in comparison to the first version [21].

In addition to our own validation activities, we made the Context Mapper tool available to the DDD community and collected feedback via issues on GitHub. Context Mapper is available for the Eclipse IDE[20] as well as for Visual Studio Code[21]. Via Gitpod[22], Eclipse Theia[23], and the Visual Studio Code extension, we can even offer a Web IDE running in the browser. According to the Eclipse marketplace Context Mapper has been installed over 40 times per month in the last three months (March, April, and May 2020). We only released the Visual Studio Code extension recently but already had 40 downloads within the first two weeks, according to the marketplace statistics.

[19] https://github.com/Microservice-API-Patterns/LakesideMutual
[20] https://marketplace.eclipse.org/content/context-mapper
[21] https://marketplace.visualstudio.com/items?itemName=contextmapper.context-mapper-vscode-extension
[22] https://www.gitpod.io/
[23] https://theia-ide.org/

Validation Results The five architects in the service design workshop challenged whether a graphical representation would be better suited, whether application services (DDD pattern) should be placed inside or outside aggregates, whether such a tool could be used as an excuse for not engaging with end users (the whole point of DDD: establish a conversation and a common language). The conclusion from this validation activity was that the general approach can be useful in education and early project stages, but should not replace careful business analysis and coding in Java or other languages. Support for roundtripping and a careful synchronization of manual and tool supported steps in agile (iterative, incremental) practices were seen to be critical success factors for a broader adoption. Attendees appreciated the representation of the patterns in the DSL; the story extension with attributes and relationships was accepted.

In general, our intermediate validation activities so far suggest that both our hypothesis mentioned in Section 2 hold true. Discussions with DDD experts have further confirmed that controversial debates regarding the original pattern definitions and how the patterns can be combined exist among the practitioners, which supports our first hypothesis stating that architects and adopters benefit from a precise interpretation. The validation results gained from our case studies also support our second hypothesis that a modeling language such as CML can be helpful to model (micro-)service-oriented architectures with strategic DDD.

Threats to Validity Regarding *construct validity* [40] there might be a risk that questions in our workshops or exercise lessons were misinterpreted by the participants. We tried to mitigate this threat by selecting experienced architects that are familiar with the topic and DDD. However, in our exercise lessons with the less experienced students there might be a risk for misunderstandings. We consider it unlikely that the opinions of the workshop participants, exercise participants, and DDD experts were influenced by factors unrelated to our approach (*internal validity* [40]). Threats to *external validity* [40] do exist, since we mainly relied on feedback of users that are familiar with DDD. Therefore, the validation results could vary in case we validate with other potential users (not familiar with DDD). We mitigated this threat a bit by using the tool with students at our university. However, future validation activities should include even more potential users that are experienced with software architecture but not DDD specifically. In addition, many of our experiments were self-experiments; since Context Mapper is an open source project, we do not know all our users, but have received direct feedback from six companies and teaching institutions located in different European countries (so we can consider the diversity threat and possible interest bias to be mitigated somewhat).

Analysis of Validation Results: Pros and Cons of DSL and Tools We consider the conformance of the language and our terminology with the original DDD patterns to be a strength of the proposed approach. DDD adopters can familiarize themselves with the language easily. Our validation activities further indicate that the tool can increase the productivity in context mapping, especially when the map has to be improved iteratively. The model transformations can improve such a process in comparison to drawing by hand. This support for iterative model evolution is also a reason why we consider the approach conform with agile practices [1]. However, members of the agile community may argue that the approach is non-conforming with “working software over comprehensive documentation” [1]. Therefore, we can consider this a weakness and strength at the same time. Another strength is that we are able to generate architecture visualizations on different levels of abstraction out of the DSL-based models. Communicating software architecture always requires different perspectives and levels of abstraction depending on the audience. The “model-code” gap [12] can be considered a weakness of DSL- and generator-based approaches is general. Generated code typically changes and the original architecture descriptions tend to become outdated quickly. In addition, the approach requires an Integrated Development Environment (IDE) with editor support. This can be costly, especially if multiple IDEs have to be supported. However, we still consider the approach based on DDD future-proof, since technology-independent domain modeling is always relevant in software engineering. The presented approach is independent of any programming languages, architectural styles, or frameworks.

7 Summary and Outlook

In our previous paper [23] we presented Context Mapper, our approach to describe integration architectures and service decompositions in terms of strategic DDD patterns. As our research contributions, we proposed a) a meta-model and semantic rules based on the DDD patterns aiming for a concise interpretation of the patterns and how they can be combined, and b) a DSL and supporting tools to model Bounded Contexts and their relationships as well as Aggregates.

This extended version of the original paper introduced language improvements and enhancements that allow users to start modeling on the level of use cases and user stories. Additionally, we introduced model transformations that a) support Context Mapper users in modeling DDD Subdomains and Bounded Contexts rapidly, and b) illustrate how the Context Mapper DSL (CML) can be used as a foundation for systematic service decomposition approaches. The Context Mapper tool further allows to generate code, visual Context Maps and other architecture diagrams (not presented in this paper). In addition, the rapid prototyping transformations demonstrate how we apply and validate the DSL in practical cases.

Besides the rapid prototyping transformations we implemented several Architectural Refactorings (ARs)[24] (discussed in another paper [24]) that support the users in improving the architecture models iteratively.

Validation results collected via implementation, action research, and case studies suggest that Context Mapper can support architects in their modeling work and decision making effectively and efficiently. The existing results and

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

user feedback further led to the syntax enhancements presented in this paper. However, additional validation activities will be required to finally confirm our hypothesis that Context Mapper can be beneficial in agile architecting and modeling environments.

In our future work we plan to further improve the language and tool so that software architects can evolve their designs with additional transformations and architecture refactorings [41] in an iterative and incremental manner. A reverse engineering library shall close the “model-code” gap [12] and provide model generation from existing or generated source code. This shall ease the application of the tool in brownfield projects that plan to refactor monoliths to microservices and/or migrate to the cloud. The integration of a systematic service decomposition approach similar to Service Cutter [15] shall propose new decompositions (Context Maps) that improve coupling and cohesion between contexts automatically.

References

1. Agile Alliance: Agile manifesto. https://www.agilealliance.org/agile101/the-agile-manifesto/ (2001)
2. Agile Alliance: User story template. https://www.agilealliance.org/glossary/user-story-template/ (2001)
3. Avison, D.E., Lau, F., Myers, M.D., Nielsen, P.A.: Action research. Commun. ACM **42**(1), 94–97 (Jan 1999). https://doi.org/10.1145/291469.291479
4. Baresi, L., Garriga, M., De Renzis, A.: Microservices identification through interface analysis. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) Service-Oriented and Cloud Computing. pp. 19–33. Springer International Publishing, Cham (2017)
5. Brandolini, A.: Strategic domain driven design with context mapping. `https://www.infoq.com/articles/ddd-contextmapping` (2009)
6. Brown, S.: The C4 model for visualising software architecture: Context, Containers, Components and Code. https://www.infoq.com/articles/C4-architecture-model/ (2018)
7. Cheesman, J., Daniels, J.: UML Components: A Simple Process for Specifying Component-Based Software. Addison-Wesley Longman Publishing Co., Inc. (2000)
8. Cockburn, A.: Writing Effective Use Cases. Agile Software Development Series, Addison-Wesley (2001)
9. Conway, M.: Conway’s law (1968)
10. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2003)
11. Evans, E.: Domain-driven design reference: Definitions and pattern summaries. `https://domainlanguage.com/ddd/reference` (2015)
12. Fairbanks, G.: Just enough software architecture: a risk-driven approach. Marshall & Brainerd (2010)
13. Francesco, P.D., Lago, P., Malavolta, I.: Migrating towards microservice architectures: An industrial survey. In: 2018 IEEE International Conference on Software Architecture (ICSA). pp. 29–2909 (April 2018). https://doi.org/10.1109/ICSA.2018.00012

14. Gouigoux, J., Tamzalit, D.: 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). pp. 62–65 (April 2017). https://doi.org/10.1109/ICSAW.2017.35
15. Gysel, M., Kölbener, L., Giersche, W., Zimmermann, O.: Service cutter: A systematic approach to service decomposition. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) Service-Oriented and Cloud Computing. pp. 185–200. Springer International Publishing, Cham (2016)
16. Habegger, M., Schena, M.: Cloud-Native Refactoring in a mHealth Scenario. Bachelor thesis, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2019)
17. Hassan, S., Ali, N., Bahsoon, R.: Microservice ambients: An architectural meta-modelling approach for microservice granularity. In: 2017 IEEE International Conference on Software Architecture (ICSA). pp. 1–10 (April 2017). https://doi.org/10.1109/ICSA.2017.32
18. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
19. Josélyne, M.I., Tuheirwe-Mukasa, D., Kanagwa, B., Balikuddembe, J.: Partitioning microservices: A domain engineering approach. In: Proceedings of the 2018 International Conference on Software Engineering in Africa. pp. 43–49. SEiA '18, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3195528.3195535
20. Kapferer, S.: Architectural Refactoring of Data Access Security. Semester thesis, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2017), `https://eprints.hsr.ch/564`
21. Kapferer, S.: A Domain-specific Language for Service Decomposition. Term project, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2018), `https://eprints.hsr.ch/722`
22. Kapferer, S.: Model Transformations for DSL Processing. Term project, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2019), `https://eprints.hsr.ch/819/`
23. Kapferer., S., Zimmermann., O.: Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,. pp. 299–306. INSTICC, SciTePress (2020). https://doi.org/10.5220/0008910502990306
24. Kapferer, S., Zimmermann, O.: Domain-driven service design – context modeling, model refactoring and contract generation. In: Proc. of the 14th Symposium and Summer School On Service-Oriented Computing - SummerSoC (September 13-19, 2020). Springer Communications in Computer and Information Science (CCIS) (to appear)
25. Kruchten, P.: The 4+1 view model of architecture. IEEE Software **12**(6), 42–50 (1995). https://doi.org/10.1109/52.469759
26. Landre, E., Wesenberg, H., Rønneberg, H.: Architectural improvement by use of strategic level domain-driven design. In: Companion to the 21st ACM OOPSLA. pp. 809–814. OOPSLA '06, ACM, New York, NY, USA (2006). https://doi.org/10.1145/1176617.1176728
27. Le, D.M., Dang, D.H., Nguyen, V.H.: Domain-driven design using meta-attributes: A dsl-based approach. In: 2016 Eighth International Conference on Knowledge and Systems Engineering (KSE). pp. 67–72 (Oct 2016). https://doi.org/10.1109/KSE.2016.7758031

28. Mazlami, G., Cito, J., Leitner, P.: Extraction of microservices from monolithic software architectures. In: 2017 IEEE International Conference on Web Services (ICWS). pp. 524–531 (June 2017). https://doi.org/10.1109/ICWS.2017.61
29. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (Dec 1972). https://doi.org/10.1145/361598.361623
30. Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., Josuttis, N.: Microservices in practice, part 1: Reality check and service design. IEEE Software **34**(1), 91–98 (Jan 2017). https://doi.org/10.1109/MS.2017.24
31. Plöd, M.: DDD Context Maps - an enhanced view. `https://speakerdeck.com/mploed/context-maps-an-enhanced-view` (2018)
32. Plöd, M.: Hands-on Domain-driven Design - by example. Leanpub (2019)
33. Rademacher, F., Sorgalla, J., Sachweh, S.: Challenges of domain-driven microservice design: A model-driven perspective. IEEE Software **35**(3), 36–43 (May 2018). https://doi.org/10.1109/MS.2018.2141028
34. Rademacher, F., Sachweh, S., Zündorf, A.: Towards a UML profile for domain-driven design of microservice architectures. In: Cerone, A., Roveri, M. (eds.) Software Engineering and Formal Methods. pp. 230–245. Springer International Publishing, Cham (2018)
35. Shaw, M.: Writing good software engineering research papers: Minitutorial. In: Proceedings of the 25th International Conference on Software Engineering. pp. 726–736. ICSE '03, IEEE Computer Society, Washington, DC, USA (2003), `http://dl.acm.org/citation.cfm?id=776816.776925`
36. Stal, M.: Software architecture refactoring. In: Tutorial in The International Conference on Object Oriented Programming, Systems, Languages and Applications (2007)
37. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Eclipse Series, Pearson Education (2008)
38. Tune, N., Millett, S.: Designing Autonomous Teams and Services: Deliver Continuous Business Value Through Organizational Alignment. O'Reilly Media (2017)
39. Vernon, V.: Implementing Domain-Driven Design. Addison-Wesley Professional, 1st edn. (2013)
40. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in Software Engineering. Springer Publishing Company (2012)
41. Zimmermann, O.: Architectural refactoring for the cloud: a decision-centric view on cloud migration. Computing **99**(2), 129–145 (2017). https://doi.org/10.1007/s00607-016-0520-y
42. Zimmermann, O.: Microservices tenets. Computer Science - Research and Development **32**(3), 301–310 (Jul 2017). https://doi.org/10.1007/s00450-016-0337-0
43. Zimmermann, O., Stocker, M., Lübke, D., Pautasso, C., Zdun, U.: Introduction to Microservice API Patterns (MAP). In: Cruz-Filipe, L., Giallorenzo, S., Montesi, F., Peressotti, M., Rademacher, F., Sachweh, S. (eds.) Joint Post-proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019). OpenAccess Series in Informatics (OASIcs), vol. 78, pp. 4:1–4:17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/OASIcs.Microservices.2017-2019.4