Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling

Stefan Kapferer and Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland (HSR FHO), Oberseestrasse 10, 8640 Rapperswil, Switzerland

Keywords: DSL, Enterprise Application Integration, Model-driven Software Engineering, Service Design, Patterns.

Abstract: Service-oriented architectures and microservices have gained much attention in recent years; companies adopt these concepts and supporting technologies in order to increase agility, scalability, and maintainability of their systems. Decomposing an application into multiple independently deployable, appropriately sized services and then integrating such services is challenging. With strategic patterns such as Bounded Context and Context Map, Domain-driven Design (DDD) can support business analysts, (enterprise) architects, and microservice adopters. However, existing architecture description languages do not support the strategic DDD patterns sufficiently; modeling tools for DDD primarily focus on its tactical patterns. As a consequence, different opinions on how to apply strategic DDD exist, and it is not clear how to combine its patterns. Aiming for a clear and concise interpretation of the patterns and their combinations, this paper distills a meta-model of selected strategic DDD patterns from the literature. It then introduces Context Mapper, an open source project that a) defines a Domain-specific Language (DSL) expressing the strategic DDD patterns and b) provides editing, validation, and transformation tools for this DSL. As a machine-readable description of DDD, the DSL provides a modeling foundation for (micro-)service design and integration. The models can be refactored and transformed within an envisioned tool chain supporting the continuous specification and evolution of Context Maps. Our validation activities (prototyping, action research, and case studies) suggest that the DDD pattern clarification in our meta-model and the Context Mapper tool indeed can benefit the target audience.

1 INTRODUCTION

Domain-driven Design (DDD) was introduced in a practitioner book in 2003 (Evans, 2003). 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. Strategic DDD has gained even more attention during the last few years in the context of microservices and enterprise application integration (Pautasso et al., 2017). 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.

How to decompose software systems into smaller, more maintainable units indeed has been an open research question for many years. For instance, Parnas already wrote about module decomposition in 1972 (Parnas, 1972).

Despite the large body of existing work, it is not understood well yet how service interfaces can be identified and which patterns and practices are suitable to analyze and design service-oriented systems (Pautasso et al., 2017):

Which criteria are relevant to find adequate service boundaries? Which patterns and practices can be applied to identify candidate services?

DDD can play a key role in answering these questions: with patterns such as Bounded Context (an abstraction of systems and teams developing them, defining a model boundary), it provides an approach for structuring a domain. Context mapping patterns such as Customer-Supplier, Shared Kernel or Open Host Service (Evans, 2003) can define the relationships between the units of decomposition. However, the strategic patterns come with some ambiguity and different interpretations of how they shall be applied.

The microservices community suggests to leverage DDD patterns to answer the above design questions. Advocates of this field suggest to model software-intensive systems in terms of Bounded Contexts, and then implement one microservice for each Bounded Context.

Kapferer, S. and Zimmermann, O.
Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling.
DOI: 10.5220/0008910502990306
In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pages 299-306
ISBN: 978-989-758-400-8; ISSN: 2184-4348

However, the identification of suited Bounded Contexts is still challenging. This is where Context Map models and diagrams, context mapping as a practice and the strategic DDD patterns to define the relationships between Bounded Contexts come into play. From our experience, a clear understanding of how these patterns shall work together is often missing, and different stakeholders have different opinions on how these patterns should be applied and combined. From these observations we derived our first hypothesis:

Software engineers and service designers benefit from clarification and advice on how to combine the strategic DDD patterns in Context Maps.

We further believe that Context Maps are artifacts which evolve iteratively. Software engineers can use them to analyze and understand a domain. They can also serve as an instrument to describe and communicate the architecture of a system. However, it is also beneficial if models can be transformed seamlessly in order to improve the architecture in an agile way or to generate other representations upon demand. A machine-readable definition of a model offers the possibility to automate certain steps, for instance to generate abstract or concrete service contracts. This leads us to our second hypothesis:

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 refactor, transform and evolve such models iteratively.

In this paper, we present a Domain-specific Language (DSL) for the strategic DDD patterns and as another contribution, we distill a meta-model providing a concise interpretation of these patterns and their applicability. The DSL is implemented in Context Mapper. Domain-driven designs modelled in our tool can be used to identify services with reasonable cohesion and coupling. Our DSL also supports the refinement of Bounded Contexts with tactic DDD patterns as supported in the Sculptor DSL[1]. Generator tools producing graphical Context Maps, PlantUML[2] diagrams, and (micro-) service contracts illustrate how the language can be used to transform the Context Maps into other representations. This paper focuses on the DSL and addresses the second hypothesis only partially. All other parts of our framework are documented online[3] and will be elaborated upon in future work.

[1]http://sculptorgenerator.org/

[2]http://plantuml.com/

[3]https://contextmapper.org/

The remainder of the paper is structured as follows. In Section 2 we present our first research contribution, a strategic DDD meta-model with semantic rules that describe our interpretation of the patterns. Section 3 introduces the DSL syntax. Section 4 discusses usage scenarios and the applicability of the presented approach. Section 5 concludes and outlines future work.

2 DOMAIN-DRIVEN DESIGN (DDD) ESSENTIALS/ANALYSIS

Since Evans has published his original DDD book (Evans, 2003), other – mostly gray – literature on this topic has been published. Our analysis and interpretation of the patterns is based on the books of Evans (Evans, 2003) and Vernon (Vernon, 2013). Our personal professional experience (Kapferer, 2017) has influenced the meta-model as well. Additional patterns of Evans' DDD reference (Evans, 2015), 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 (Brandolini, 2009) and Plöd (Plöd, 2018; Plöd, 2019).

2.1 Example

Strategic Domain-driven Design (DDD) can be used to decompose the problem domain of a software system into multiple sub-domains 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 3, 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 (Vernon, 2013), Brandolini (Brandolini, 2009) and Plöd (Plöd, 2018).

A Bounded Context defines an explicit boundary within which a particular domain model, implementing parts of sub-domains, 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* that have identifiers and lifecycles, *Value Objects* that are stateless and immutable, 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 related to the one a team is working on.

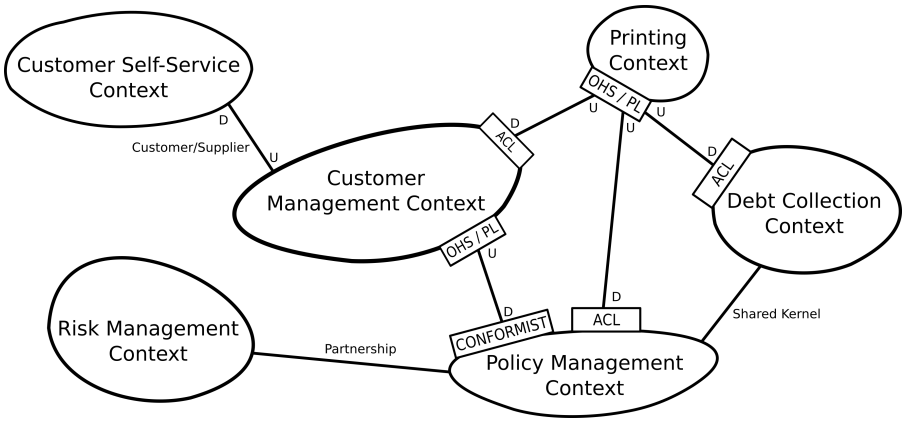

Figure 1: Insurance Scenario Example Context Map (Kapferer, 2018).

DDD offers several relationship patterns allowing modelers to describe how two Bounded Contexts and the corresponding 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 succeed 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 in DDD 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 and plans the development accordingly.

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. In Figure 1 they are added to the rectangles either on the upstream or on the downstream side. A Bounded Context can offer an OHS, which provides access to a subsystem as a set of open services, 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 of 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 and refer to the corresponding literature (Evans, 2003; Evans, 2015; Plöd, 2018; Vernon, 2013).

2.2 Our Meta-model for Strategic DDD

The meta-model presented in this section is based on the previously mentioned strategic DDD patterns and our own analysis and understanding regarding how they can be combined. The model is illustrated in Figure 2. It is implemented in our DSL and the Context Mapper tool introduced in Section 3.

The most central element in our meta-model is the Context Map. A Context Map shows Bounded Contexts and their relationships. A Bounded Context itself consists of a well-defined and delimited domain model which is decomposed into multiple Aggregates. It implements parts of one or many subdomains, which can be *Core Domains*, *Supporting Domains* or *Generic Subdomains*. Both a subdomain and a Bounded 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

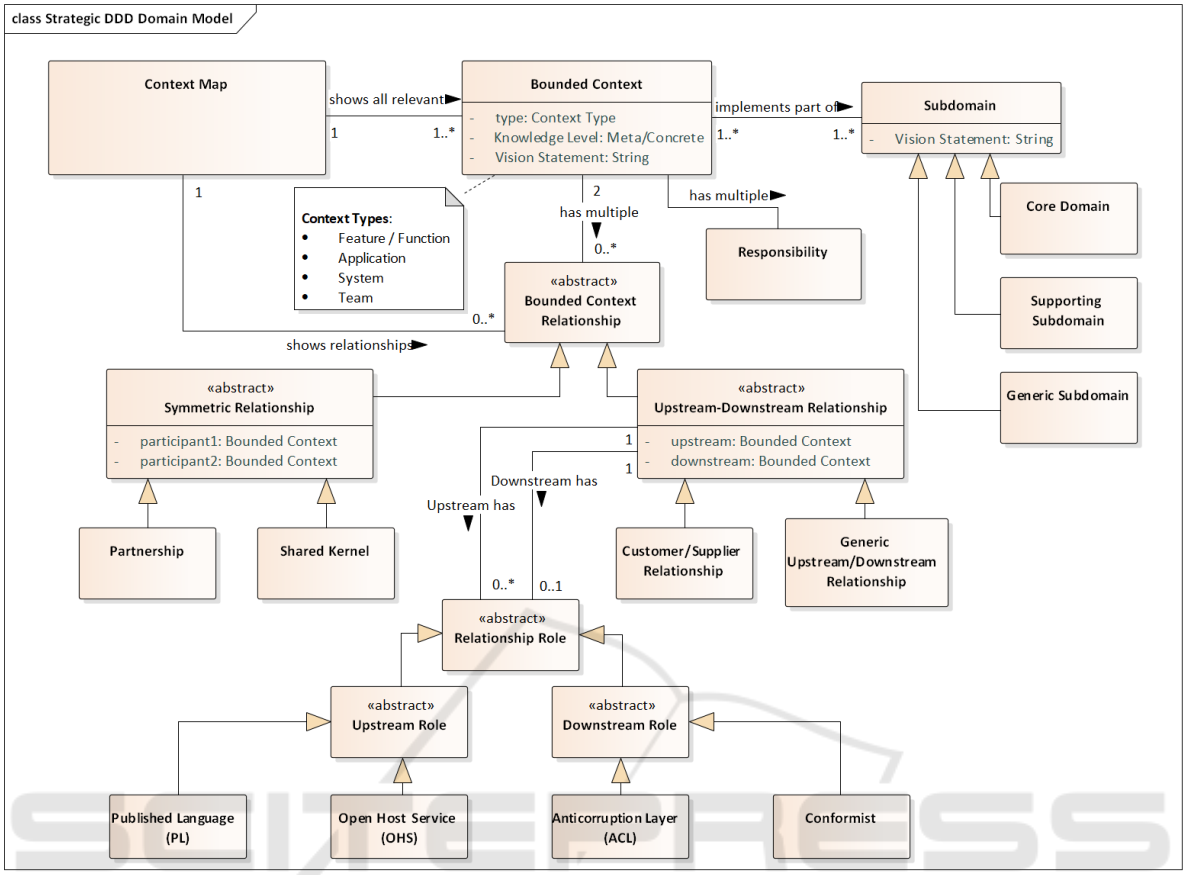

Figure 2: Context Mapper: Strategic DDD Meta-Model (UML class diagram)

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 (Plöd, 2018). 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 an upstream-downstream relationship. We indicated this in Figure 2 by distinguishing between customer-supplier relationships and *generic* upstream-downstream relationships.

2.3 Semantic Rules

There is no consensus in the DDD community on how the patterns are related and how they can be combined. We evolved a set of semantic rules in addition to our meta-model. These rules define the valid relationships between the patterns as well as combinations not permitted in our language. Note that these rules mainly reflect the authors interpretation based on empirical evidence (e.g., (Kapferer, 2017)). However, all rules have been conceptualized considering the literature of DDD and Context Mapping experts (Vernon, 2013; Plöd, 2019; Brandolini, 2009). Several rules are already implicitly given by the meta-model in Figure 2, others extend the model.

2.3.1 Rule #1: Permitted Upstream Roles

The patterns OHS and PL can only be implemented by the upstream context in an upstream-downstream relationship. The upstream context always provides and exposes a certain functionality. The downstream context uses and consumes this services and does not

expose parts of his/her own domain model. If this was the case and the upstream used this functionality, the definition that the upstream is independent of the downstream would be contradicted.

2.3.2 Rule #2: Permitted Downstream Roles

The patterns ACL and CF can only be applied by the downstream context in an upstream-downstream relationship. These patterns solve a downstream problem, namely how to deal with a dependency to another context. It is always the downstream context that has to integrate the upstream model.

2.3.3 Rule #3: Protect or Conform

The patterns ACL and CF cannot be applied jointly, but provide alternatives. The downstream either conforms (CF) *or* protects itself with an ACL.

2.3.4 Rule #4: Integrity of Symmetric Relationships

The patterns OHS, PL, ACL and CF are not applicable in symmetric relationships (Partnership and Shared Kernel), since doing so would lead to contradictions with the pattern definitions. In a Shared Kernel relationship, the two contexts communicate over shared code such as a library. Both contexts manage the shared code together, which clearly contradicts with the mentioned four pattern definitions. An OHS indicates a directed provider/consumer behavior which is not the case here. There is no need for a common inter-context language (PL), since the two contexts simply share the same model. An ACL is not required either since the two participants share the model anyway. And neither context has to conform to the model of the other since it is one *shared* model. In a Partnership relationship both contexts depend on each other, which means they can only succeed or fail together.

2.3.5 Rule #5: Customer vs. Conformist

The CF pattern is not applicable within a customer-supplier relationship. In a customer-supplier relationship the customer has influence on the supplier and can at least negotiate regarding priorities of the requirements and the implementation. A conformist in contrast has no influence and simply decides to conform to what the upstream provides.

2.3.6 Rule #6: Generic vs. Custom Service

The OHS pattern is not applicable within a customer-supplier relationship. Whereas the customer-supplier pattern implies that the involved teams work closely together, meaning that the upstream respects the downstreams requirements in his planning sessions, the OHS pattern indicates that the upstream team decides to implement one API in a *one for all* approach. This is 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. From personal practical experience a customer-supplier relationship leads to individual requirements of single customers. As soon as the supplier implements a customer-specific API feature it is by pattern definition no longer an OHS.

2.3.7 Rule #7: Protect or Cooperate

The ACL pattern should not be used within a customer-supplier relationship. Changes of the supplier should be in-sync with the needs of the customer. Protection should be unnecessary. Note that this is only a *soft* rule since the combination is possible but not common. Our tool issues a warning rather than an error message if it detects a violation of the rule.

3 CONTEXT MAPPER DSL

To allow software architects to model systems according to our DDD meta-model, we implemented the *Context Mapper*[4] tool.

All of the following DSL examples are based on the insurance scenario introduced in Section 2. The complete example can be found in our examples repository[5].

3.1 Bounded Context, Subdomain and Context Map Syntax

We start with the Context Mapper DSL (CML) syntax for Bounded Contexts. Listing 1 shows the declaration of the *CustomerManagementContext* as an example. The attributes at the top of the declaration 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

[4]https://contextmapper.org/

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

Bounded Context. The implementation of CML inside the Aggregates is based on the Sculptor[6] project.

Listing 1: Bounded Context Syntax in CML.

```
BoundedContext CustomerManagementContext implements
                                CustomerManagementDomain {
  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
    }
  }
}
```

The user specifies the subdomains implemented by the Bounded Context behind the keyword *implements*. The subdomains are declared as illustrated in Listing 2 and must always be part of a domain. A subdomain is of the type *Core Domain*, *Supporting Subdomain* or *Generic Subdomain* according to our meta-model and (Evans, 2003). Note that a Bounded Context not necessarily implements a complete subdomain.

Listing 2: Subdomain Syntax in CML.

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

The central and most important structure of CML is the Context Map which specifies the relationships between Bounded Contexts. Listing 3 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 3: Context Map Syntax in CML.

```
ContextMap {
  contains CustomerContext, PolicyContext

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

Listing 3 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).

[6]http://sculptorgenerator.org/

3.2 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 3. Note that the declaration of the implementation technology is optional and we omit it in the following examples.

3.2.1 Partnership

Listing 4 shows an example for the Partnership (P) pattern, which is a symmetric relationship.

Listing 4: Partnership Pattern Syntax in CML.

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

3.2.2 Shared Kernel

The second symmetric relationship is the Shared Kernel (SK). The syntax is identical to the Partnership. Listing 5 illustrates an example.

Listing 5: Shared Kernel Pattern Syntax in CML.

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

3.2.3 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 6 and 7 are semantically equal.

Listing 6: Upstream-downstream Relationship in CML (1).

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

Listing 7: Upstream-downstream Relationship in CML (2).

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

3.2.4 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 8 illustrates an example with the OHS and PL patterns on the upstream side and the ACL pattern on the downstream side.

Listing 8: Upstream-downstream Relationship with Roles.

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

3.2.5 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 9.

Listing 9: 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 10 is semantically equal to the one in Listing 9.

Listing 10: Customer-Supplier Relationship in CML (2).

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

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[7] and the complete insurance example can be found in our examples repository[8].

4 USAGE SCENARIOS AND APPLICABILITY DISCUSSION

Our DSL and tools have three primary usage scenarios and user roles: 1) business analysts can use the modeling language to analyze and understand a domain, and establish a common vocabulary for it. 2) architects can describe, communicate, and evolve system designs and connections with other systems (e.g., enterprise application integration). 3) adopters of microservices architectures can model system decompositions and individual services.

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

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

Our vision is to use the language as a tool to evolve and improve the architecture with refactorings and model transformations iteratively (note that these two topics are not in the scope of this paper). The language can be further used as input for generators which produce other representations of the models or input for other tools. With our PlantUML[9] generator, for instance, we demonstrate how the models can be transformed into graphical representations. It supports the generation of UML component diagrams out of Context Maps and class diagrams for individual Bounded Contexts (and their Aggregated and Entities). Figure 3 illustrates the Context Map introduced by Figure 1 as a component diagram generated within the Context Mapper tool. Another generator recently added to the tool produces graphical Context Maps identical to the illustration in Figure 1.

The service contract generator of Context Mapper assists architects designing service-oriented architectures. It supports the Microservices Domain Specific Language (MDSL)[10] format, an emerging DSL realizing the *API Description* pattern, which is part of the Microservice API Patterns (MAP) language (Zimmermann et al., 2019).

We have validated different versions of the DSL syntax with 20 exercise participants in a software architecture course at our institution. We evaluated *readability* and *writability*, for example by asking for the time needed to understand existing and write new models. Participants worked with the online documentation of Context Mapper, including the provided examples; they also were asked to model a real-world Context Map from the oil industry (Wesenberg et al., 2006). The feedback was generally positive; participants were able to complete the exercise tasks in the allocated time slots. Tool features such as hover help and constructive, detailed validation messages were appreciated, as they (re-)educate users about the pattern meanings and valid pattern combinations.

In addition we conceptualized a more verbose alternative syntax for A/B testing. In comparison to the presented version in this paper, which is very dense and optimized for *writability*, the alternative version improves the *readability*. Due to space limitations we were not able to introduce both variants in this paper; both versions are documented online.

5 SUMMARY AND OUTLOOK

In this paper we presented Context Mapper, a DSL and tools to describe application landscapes and ser-

[9]http://plantuml.com/

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

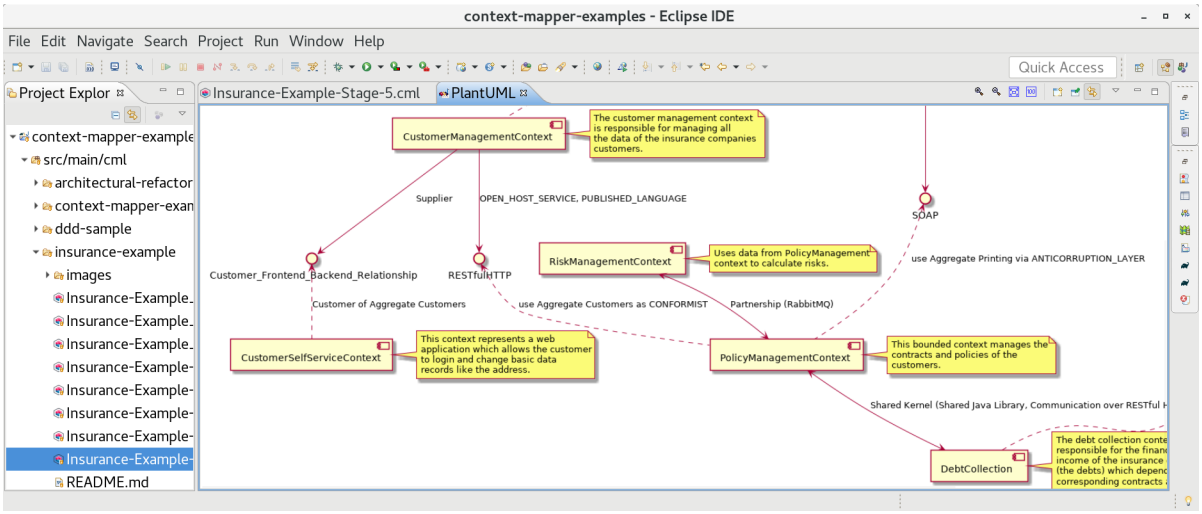

Figure 3: Context Mapper Eclipse Plugin: PlantUML Generator Example Output.

vice designs in terms of strategic DDD patterns. As our research contributions, we proposed a) a metamodel and semantic rules aiming for a concise specification of how DDD patterns can be combined, and b) a DSL and supporting tools to model Bounded Contexts and their relationships as well as tactic DDD patterns such as Aggregates. Being defined in a DSL, our Context Maps can be processed and transformed into other representations. Thus, the Context Mapper DSL (CML) provides a modeling foundation for service design approaches and tools aiming for visualizing Context Maps and transforming them.

In our future work we plan to further improve the tool so that software architects can evolve system architectures with more transformations and refactorings. Besides the already supported generation of MDSL service contracts, we may automatically generate microservice application stubs out of the Context Maps. An already prototyped reverse engineering tool to generate CML from existing source code can ease the application of the tool in brownfield projects that plan to refactor monoliths to microservices and/or migrate them to the cloud. Decoupling the language from Eclipse and providing other development environments on the basis of the approach proposed by Bünder (Bünder, 2019) may further increase the target user group.

REFERENCES

Brandolini, A. (2009). Strategic domain driven design with context mapping. https://www.infoq.com/articles/ddd-contextmapping.

Bünder, H. (2019). Decoupling language and editor - the impact of the language server protocol on textual domain-specific languages. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 131–142. INSTICC, SciTePress.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley.

Evans, E. (2015). Domain-driven design reference: Definitions and pattern summaries. https://domainlanguage.com/ddd/reference.

Kapferer, S. (2017). Architectural refactoring of data access security. Semester thesis, University of Applied Sciences of Eastern Switzerland (HSR FHO). https://eprints.hsr.ch/564.

Kapferer, S. (2018). A domain-specific language for service decomposition. Term project, University of Applied Sciences of Eastern Switzerland (HSR FHO). https://eprints.hsr.ch/722.

Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

Pautasso, C., Zimmermann, O., Amundsen, M., Lewis, J., and Josuttis, N. (2017). Microservices in practice, part 1: Reality check and service design. *IEEE Software*, 34(1):91–98.

Plöd, M. (2018). DDD Context Maps - an enhanced view. https://speakerdeck.com/mploed/context-maps-an-enhanced-view.

Plöd, M. (2019). *Hands-on Domain-driven Design - by example*. Leanpub.

Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley Professional, 1st edition.

Wesenberg, H., Landre, E., and Rønneberg, H. (2006). Using domain-driven design to evaluate commercial off-the-shelf software. In *Comp. to 21th Annual ACM SIGPLAN OOPSLA*, pages 824–829.

Zimmermann, O., Stocker, M., Zdun, U., Luebke, D., and Pautasso, C. (2019). Microservice API Patterns. https://microservice-api-patterns.org.