Domain-driven Service Design
Context Modeling, Model Refactoring and Contract Generation

Stefan Kapferer and Olaf Zimmermann

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

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 them is challenging. Domain-driven Design (DDD) is a popular approach to identify (micro-)services by modeling so-called Bounded Contexts and Context Maps. In our previous work, we proposed a Domain-specific Language (DSL) that leverages the DDD patterns to support service modeling and decomposition. The DSL is implemented in Context Mapper, a tool that allows software architects and system integrators to create domain-driven designs that are both human- and machine-readable. However, we have not covered the tool architecture, the iterative and incremental refinement of such maps, and the transition from DDD pattern-based models to (micro-)service-oriented architectures yet. In this paper, we introduce the architectural concepts of Context Mapper and seven model refactorings supporting decomposition criteria that we distilled from the literature and own industry experience; they are grouped and serve as part of a service design elaboration method. We also introduce a novel service contract generation approach that leverages a new, technology-independent Microservice Domain-Specific Language (MDSL). These research contributions are implemented in Context Mapper and being validated using empirical methods.

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

1 Introduction

Domain-driven Design (DDD) was introduced in a practitioner book in 2003 [8]. Tactical DDD patterns such as Aggregate, Entity, Value Object, Factory, and Repository have been used in software engineering to model complex domains in an object-oriented way since then. While these tactical patterns focus on the domain model of an application, strategic ones such as Bounded Context and

Context Map establish domain model scopes as well as the relationships between such scopes. The strategic DDD patterns have gained attention through the popularity of microservices recently [33].

The tactical part within a Bounded Context can be modeled with UML or existing Domain-Specific Languages (DSLs) for DDD such as Sculptor[1]. Existing modeling tools have not supported the strategic patterns explicitly; Context Maps that use these relationships have had to be created manually so far. Hence, Context Mapper[2] [18] proposes and implements a DSL that allows business analysts, software architects, and system integrators to describe such models in a precise and expressive way.

The identification of Bounded Contexts is still a difficult task. Conflicting criteria have to be applied when splitting a domain into Bounded Contexts or monolithic systems into services. Many such criteria have been proposed by practitioners and researchers. However, not many concrete practices and systematic approaches how to decompose the models exist to date. Thus, we researched the criteria to be used for service decomposition and derived a series of Architectural Refactorings (ARs) [34] to decompose DDD-based models written in the Context Mapper DSL (CML) iteratively and incrementally.

Furthermore, the DDD patterns do not define how to realize the modeled contexts in (micro-)service-oriented architectures. We propose a mapping from the DDD patterns to Microservice API patterns (MAP) concepts[3] and Microservice Domain-Specific Language (MDSL)[4], a novel, technology-independent service contract and API description language.

In summary, the contributions of this paper are: a) a novel, layered and extensible tool architecture for DDD-based architecture modeling and discovery, b) a model refactoring catalog, tool, and method and c) a meta-model mapping (from DDD to service-oriented architectures) that allows Context Mapper to generate technology-independent service contracts (MDSL).

The remainder of this paper is structured as follows. Section 2 introduces key DDD concepts and patterns, and explains them in a fictitious insurance example. The section also establishes the context and vision of our work. Section 3 introduces our three research contributions: the modular and extensible tool architecture, a model refactoring catalog and method, and a service contract generation approach for DDD Context Maps. In Section 4 we outline our validation activities including prototyping, action research and case studies. Section 5 covers related work. Finally, Section 6 summarizes the paper and outlines future work. Appendix A introduces MDSL.

[1] `http://sculptorgenerator.org/`
[2] `https://contextmapper.org/`
[3] `https://microservice-api-patterns.org/`
[4] `https://microservice-api-patterns.github.io/MDSL-Specification/`

2 Context, Vision, and Previous Work

2.1 Domain-driven Design (DDD) Pattern Essentials

Since the publication of the first DDD book by Evans [8], other – mostly gray – literature has been published [33]. Our interpretation of the patterns primarily follows the guidelines from Evans and Vernon. The CML language [18] supports the strategic as well as the tactical patterns based on these books. Strategic DDD is used to decompose a domain into Subdomains and so-called Bounded Contexts (i.e., abstractions of (sub-)systems and teams). A Bounded Context establishes a boundary within which a particular domain model applies. The domain model has to be consistent within this boundary and the terms of the domain have to be clearly defined to build a *ubiquitous language*. The Context Map describes the relationships between these Bounded Contexts. Figure 1 illustrates a Context Map of the fictitious insurance application called Lakeside Mutual[5]. The strategic patterns Partnership and Shared Kernel describe symmetric relationships where two Bounded Contexts have organizational or domain model related interdependencies. In Upstream-Downstream relationships on the other hand only one Bounded Context is dependent on the other. The upstream-downstream metaphor indicates the *influence flow* between teams and systems as discussed by Plöd [26].

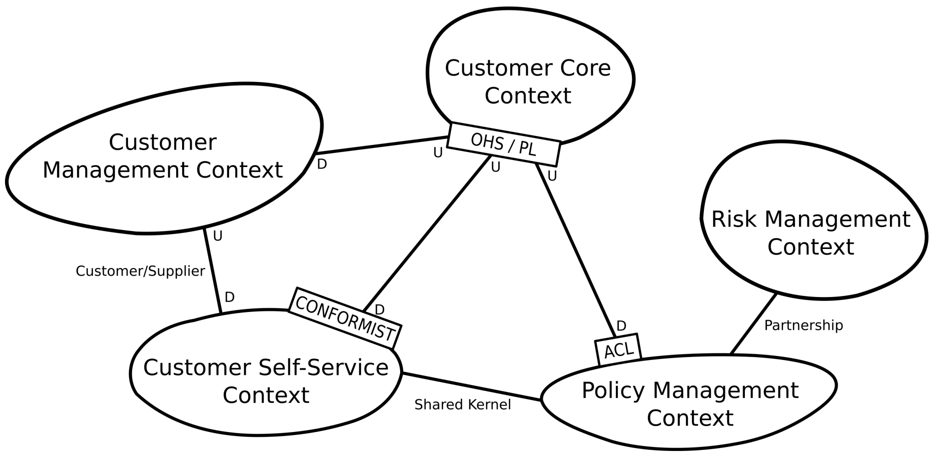


Fig. 1. Context Map: Lakeside Mutual Microservice Project

The patterns Open Host Service (OHS), Published Language (PL), Anticorruption-Layer (ACL) and Conformist allow the modelers to specify the roles of the upstream and downstream Bounded Contexts. Table 1 introduces the strategic DDD patterns relevant for this paper and depicted in Figure 1.

[5] `https://github.com/Microservice-API-Patterns/LakesideMutual/`

Table 1: Strategic Domain-driven Design (DDD) Pattern Overview

Pattern	**Description**
Subdomain	A subdomain is a part(ition) of the functional domain that is analyzed and designed. DDD differentiates between *core domains*, *supporting subdomains*, and *generic subdomains* [8]. Subdomains (or parts of them) are realized by one or more Bounded Contexts.
Bounded Context	A Bounded Context establishes a boundary within which a particular domain model is valid. The concepts of the domain must be defined clearly and distinctively within this boundary, constituting a *ubiquitous language* [8] for it. As abstractions of (sub-)systems and teams, Bounded Contexts realize parts of one or more subdomains.
Context Map	A Context Map specifies the relationships between the Bounded Contexts and how they interact with each other.
Customer/Supplier (C/S)	A Customer/Supplier (C/S) relationship is an Upstream-Downstream relationship in which the downstream Bounded Context influences the upstream a lot. The upstream supplier respects the requirements of its downstream customer and adjusts its planning accordingly.
Open Host Service (OHS)	If an upstream team has to provide the same functionality to multiple downstreams, it can implement a unified and open API, an Open Host Service (OHS).
Published Language (PL)	A context which offers functionalities to other contexts has to expose some parts of its own domain model. Direct translations and exposing internals of the domain model impose coupling. Industry standards or organization-internal specifications establish a well-documented and agreed-upon model subset, a Published Language (PL). This allows providers to guarantee language stability.
Anticorruption Layer (ACL)	A downstream Bounded Context has to integrate with the exposed model of the upstream which may not harmonize with its own domain model. In this case the downstream team can decide to implement an Anticorruption Layer (ACL) that translates between the two models and provides the upstream's functionality in terms of the downstream's own domain model to reduce coupling.
Conformist (CF)	A downstream may decide to simply run with the domain model of the upstream (and not implement an ACL). Rather than translating between two different domain models, the downstream adjusts its own design so that it fits to the upstream domain model (tight coupling).

Table 1: Strategic Domain-driven Design (DDD) Pattern Overview

Pattern	**Description**
Partnership (P)	A Partnership is a cooperative, symmetric relationship in which the two Bounded Contexts can only succeed or fail together. The pattern advises to establish a process for coordinated planning of development and joint management of integration in case two contexts are mutually dependent subsystems or teams.
Shared Kernel (SK)	A Shared Kernel is a very intimate relationship between two Bounded Contexts that share a part of their domain models. Shared Kernel is a symmetric relationship that is typically implemented as a shared library maintained by the two teams in charge of the two contexts.

The tactical DDD patterns are used to design the domain model within a Bounded Context. An Aggregate is a cluster of domain objects which is kept consistent with respect to invariants. It typically represents a unit of work regarding system (database) transactions. Each Bounded Context consists of a set of Aggregates that cluster Entities, Value Objects, Services, and Domain Events. Entities have an identity and a life cycle (mutable state); Value Objects are immutable and faceless. Both Entities and Value Objects may define attributes and methods; Services expose methods only. Domain Events record things that have happened and are worth reporting (for instance, changes to an Entity's state).

2.2 Vision, Goals, and our Previous Work

Decomposing a software system into modules, components or services has long been an open research question and challenging problem in practice. In 1972 Parnas wrote a seminal paper [24] on the criteria to be used for decomposing systems. Since then, many other researchers and practitioners proposed criteria and approaches to tackle the challenge. Many practitioners, especially in the microservice community, suggest to use the strategic DDD patterns to answer the decomposition question. Systems shall be modeled in terms of Bounded Contexts in order to implement one (micro-)service per Bounded Context later. Context Maps and *context mapping* as a practice shall support the DDD adopters in identifying Bounded Contexts and in modeling the relationships between them. However, the identification of the contexts is still challenging. A clear understanding of how the DDD patterns can be combined is often missing, and the hand-drawn models do not offer the possibility to apply concrete refactoring steps or systematic decomposition approaches. Hence, our first hypothesis is:

Software architects and system integrators can benefit from a modeling language that lets them describe Context Maps in a precise manner and offers the possibility to apply systematic decompositions and model transformations.

The language can further support business analysts describing a problem domain and its subdomains in a natural, yet precise and ubiquitous language.

Motivated by this hypothesis we realized the Context Mapper open source tool and proposed the CML Domain-specific Language (DSL) [18] to describe such models. Software architectures evolve and Context Maps must emerge iteratively. Brandolini [4] has shown how Context Maps can evolve and how Bounded Contexts can be identified step by step. A precise, machine-readable modeling approach allows us to offer transformations to improve the architecture in an agile way and generate other representations such as (micro-) service contracts out of the models. Our second hypothesis captures this vision:

Adopters of DDD who model Context Maps in a precise manner benefit from tools that allow them to evolve the architecture semi-automatically (i.e., supported by service decomposition heuristics and model refactorings), document the architecture, and generate other representations of the models such as Unified Modeling Language (UML) diagrams and service API contracts.

In our previous work [18] we introduced the CML language. Listing 2.1 illustrates the Lakeside Mutual Context Map modeled in CML. The symbol -> depicts directed Upstream-Downstream relationships; <-> depicts symmetric relationships. The relationship patterns from Table 1 appear in square brackets `[]` (this is optional). The language reference can be found online[6].

Listing 2.1. Context Map Syntax in CML

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

  CustomerCore [OHS,PL] -> CustomerManagement // influence flow: left to right

  CustomerSelfService [C ] <- [S] CustomerManagement // flow: right to left

  CustomerCore [OHS, PL] -> [CF] CustomerSelfService

  CustomerCore [OHS, PL] -> [ACL] PolicyManagement

  PolicyManagement [P] <-> [P] RiskManagement // Partnership (symmetric)

  CustomerSelfService [SK] <-> [SK] PolicyManagement // Shared Kernel
}
```

We also proposed a meta-model for the strategic DDD patterns previously [18]. It clarifies how the patterns can be combined in Context Maps. We also presented a set of semantic rules that outline pattern combinations that do not make sense according to our interpretation. The Context Mapper tool implements these semantic rules to enforce that they are met by CML models.

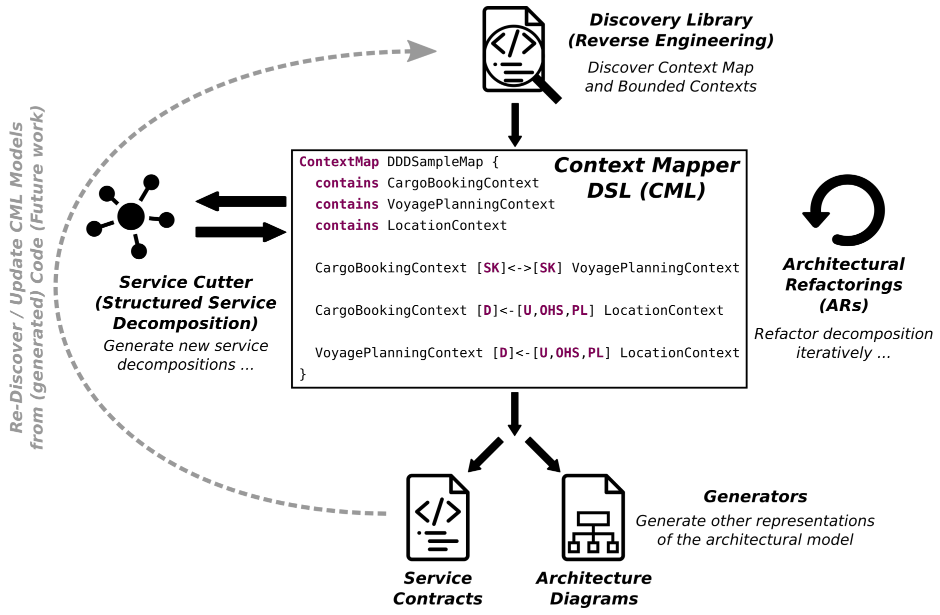
[6] `https://contextmapper.org/docs/language-reference/`

3 Context Mapper Concepts

In this section, we first introduce the Context Mapper tool architecture, which combines elements from language tool design (editors, linters) with a layered organization of refactorings and transformations. Next we establish a method for the stepwise refinement of DDD models via model refactoring, which supports common decomposition criteria from the literature and is implemented in Context Mapper. Finally, we specify how to map DDD models to service contracts; this mapping is also implemented in Context Mapper.

3.1 Modular and Extensible Tool Architecture

The Context Mapper framework architecture illustrated in Figure 2 includes multiple components that allow users to *discover*, *systematically decompose*, and *refactor* DDD Context Maps. In addition, the generators support transforming the models into other representations. The Context Mapper DSL (CML) grammar and tool constitute the hub of the three-stage architecture; all other components are integrated as spokes.


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

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

  CargoBookingContext [SK]<->[SK] VoyagePlanningContext

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

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

Fig. 2. Context Mapper Framework Architecture: Three Stages, DSL as Hub

The Discovery Library shall support users in brownfield projects to reverse engineer Context Maps and Bounded Contexts from source code. The design of the library supports the users in implementing arbitrary discovery mechanisms by applying the Strategy pattern [14]. The Architectural Refactorings (ARs) [34] (a.k.a. model refactorings) represent one component of the architecture and shall assist the users in decomposing a system step by step. In Section 3.2 we will present the AR component in detail. The Structured Service Decomposition component integrates Service Cutter [10] to derive completely new Context Maps by using coupling criteria and graph clustering algorithms. In addition, it is possible to generate PlantUML diagrams, graphical Context Maps (.svg, .png), or technology-independent service contracts (MDSL) with the generators. Discovery Library and Service Cutter integration are out of scope of this paper.

All black arrows in Figure 2 are implemented and open sourced. In the future we plan to close the "model-code" gap indicated by the dashed arrow in Figure 2. Through an enhanced Discovery Library it shall be possible to update the CML model if generated artifacts such as contracts or code are changed manually.

3.2 Model Refactoring: Catalog, Tool Integration, and Method

In this section we propose a series of model refactorings that allow modelers to decompose DDD Context Maps, written in CML, in an iterative manner. They allow to improve the modeled architectures and/or decompose a monolithic system by splitting up Bounded Contexts and Aggregates step by step. The ARs are derived from decomposition criteria (DCs) researched in mostly gray literature and our own practical experience. They are implemented as model transformations for the CML language in the Context Mapper tool.

a) Distillation of Decomposition Criteria. The refactorings proposed in this paper are based on criteria to be used for service decomposition. We researched such criteria from literature, the already existing coupling criteria catalog of Gysel et al. [10], and our own professional experience [3,17]. The conducted literature covered research papers such as the one of Parnas [24] and practitioners articles and online posts from DDD experts such as Brandolini [4], Plöd [27], Tune and Millet [32] or Tigges [31].

We describe our decomposition criteria and how we derived them in [16]. We selected a set of five DCs in order to derive and implement prototypical ARs from the collected criteria. To do so, we applied the following selection criteria:

- *Relevance in practice*: We chose criteria that are relevant for all software projects and not only in specific contexts.
- *Representativeness*: Criteria which are mentioned by multiple sources were preferred in our selection process.
- *Generality*: The set of criteria and derived ARs was chosen so that others could be implemented in a similar way.

Applying the above criteria yielded the following DCs (details appear in [16]):

- *DC-1: Business entities*: It is common to group attributes and entities which belong to the same part of the domain. These areas or parts of the domain form *linguistic* or *domain expert* boundaries [32].

– *DC-2: Use cases*: It is often recommended to group domain objects which are used by the same use cases together.
– *DC-3: Code owners / development teams*: Bounded Contexts are often built around the owners of the code (development teams).
– *DC-4: Likelihood for change*: According to Parnas [24], one should isolate things that change often from things that do not.
– *DC-5: Generalized Non-functional Requirement (NFR)*: NFRs often differ by subsytem or component. In Context Mapper, model elements that are similar with respect to an NFR can be grouped. Such NFRs could be *mutability*, *storage similarity*, *availability*, or *security*.

From the DCs listed above we then derived ARs that allow refactoring DDD Context Maps. For the prototypical implementation in our Context Mapper tool[7] we focused on Aggregates and Bounded Contexts. To be able to (de-)compose these objects, we realized ARs that are able to *split* or *merge* Aggregates or Bounded Contexts. Other ARs offer the possibility to *extract* Aggregates from a Bounded Context and build a new context based on them. Figure 3 illustrates the resulting ARs derived.

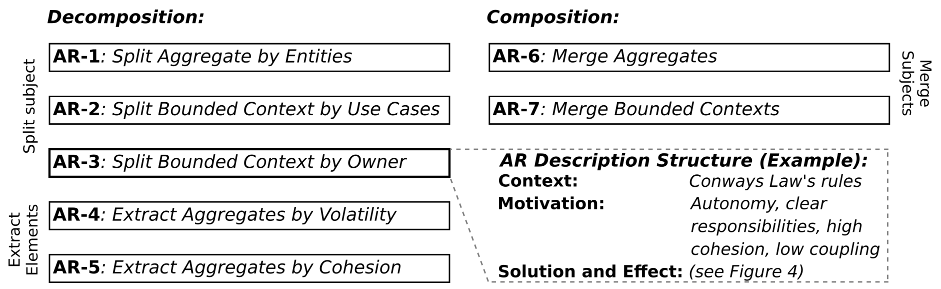

Decomposition:
Composition:
Split subject
Extract Elements
Merge Subjects
AR-1: Split Aggregate by Entities
AR-2: Split Bounded Context by Use Cases
AR-3: Split Bounded Context by Owner
AR-4: Extract Aggregates by Volatility
AR-5: Extract Aggregates by Cohesion
AR-6: Merge Aggregates
AR-7: Merge Bounded Contexts
AR Description Structure (Example):
Context: Conways Law's rules
Motivation: Autonomy, clear responsibilities, high cohesion, low coupling
Solution and Effect: (see Figure 4)

Fig. 3. Architectural/Model Refactorings by Operation and Element

In our technical report [16] we elaborated *context*, *motivation*, and *solution and effect* for all ARs presented in this paper. Figure 3 exhibits this template structure for one exemplary refactoring (AR-3).

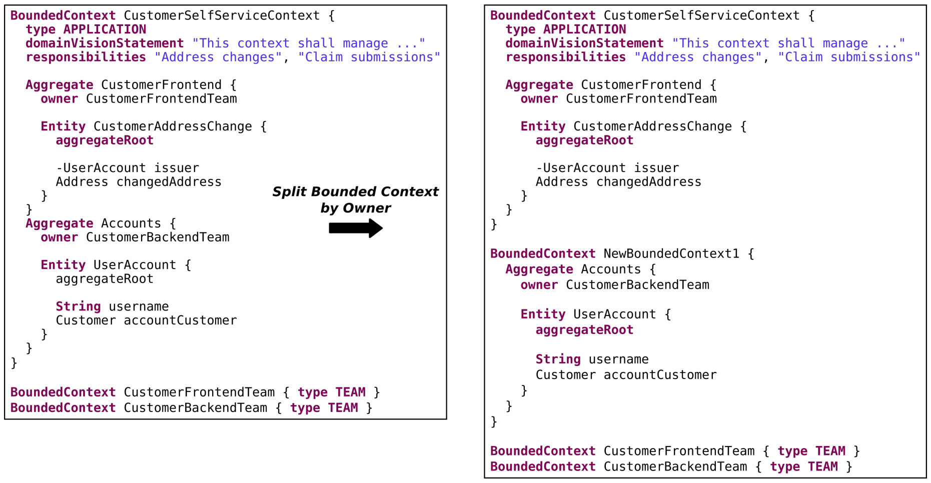
b) Realization of Refactorings as Model Transformations for CML. The CML language leverages Xtext[8], a DSL framework building on the Eclipse Modeling Framework (EMF) [30]. As described in [15], we implemented the ARs presented above as model transformations for CML. ARs are applied to a CML model in three steps:

1. DSL Text $\xrightarrow{parsing}$ Abstract Syntax Tree (AST) $\rightarrow$ EMF Model

[7] `https://contextmapper.org/`
[8] `https://www.eclipse.org/Xtext/`

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

Figure 4 shows an example Bounded Context, written in CML, on which the refactoring *Split Bounded Context by Owner* can be applied. The illustrated Bounded Context contains two Aggregates which are owned by different teams. Striving for autonomous teams with clear responsibilities, an application of this AR ensures that only one team works on a Bounded Context. Applied to this example, the AR creates a new Bounded Context for one of the Aggregates.



```
BoundedContext CustomerSelfServiceContext {
  type APPLICATION
  domainVisionStatement "This context shall manage ..."
  responsibilities "Address changes", "Claim submissions"

  Aggregate CustomerFrontend {
    owner CustomerFrontendTeam

    Entity CustomerAddressChange {
      aggregateRoot

      -UserAccount issuer
      Address changedAddress
    }
  }
  Aggregate Accounts {
    owner CustomerBackendTeam

    Entity UserAccount {
      aggregateRoot

      String username
      Customer accountCustomer
    }
  }
}

BoundedContext CustomerFrontendTeam { type TEAM }
BoundedContext CustomerBackendTeam { type TEAM }
```

```
BoundedContext CustomerSelfServiceContext {
  type APPLICATION
  domainVisionStatement "This context shall manage ..."
  responsibilities "Address changes", "Claim submissions"

  Aggregate CustomerFrontend {
    owner CustomerFrontendTeam

    Entity CustomerAddressChange {
      aggregateRoot

      -UserAccount issuer
      Address changedAddress
    }
  }
}

BoundedContext NewBoundedContext1 {
  Aggregate Accounts {
    owner CustomerBackendTeam

    Entity UserAccount {
      aggregateRoot

      String username
      Customer accountCustomer
    }
  }
}

BoundedContext CustomerFrontendTeam { type TEAM }
BoundedContext CustomerBackendTeam { type TEAM }
```

Fig. 4. Example Refactoring: Split Bounded Context by Owner (Team)

Once the architecture model has been refactored, one can use our generators to transform the Context Map into other representations. In previous work [18] we introduced our PlantUML[9] generator. Figure 5 illustrates an example of a PlantUML diagram generated with Context Mapper. It shows another simple Aggregate of our fictitious insurance scenario (two Entities, one Value Object).

c) Stepwise Application of the Refactorings. The ARs presented above are designed to decompose an application incrementally. Figure 6 outlines a logical process to decompose a system including a suggested order of AR applications. Note that the logical order does not mandate a strict, one-time chronological execution of the steps; other decomposition strategies can be applied as well. The process might be repeated multiple times.

One has to analyze the domain and identify an initial set of subdomains early in a(ny) project. Context Mapper allows the users to model the entities that a

[9] https://plantuml.com/

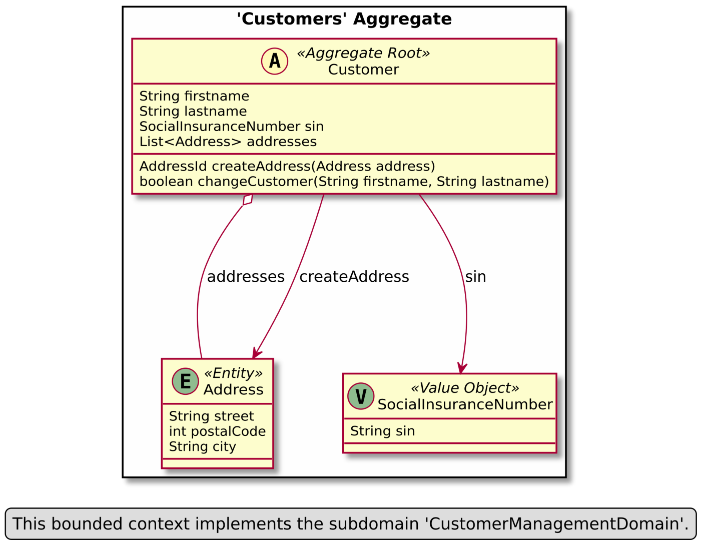


Fig. 5. Generated PlantUML Diagram for an Aggregate

subdomain contains. From use cases, user stories, or techniques such as Event Storming [5], a set of initial Bounded Contexts has to be identified.

The given contexts can then be decomposed iteratively. As indicated in Figure 6, Bounded Contexts are typically formed around the features or use cases (AR-2). If teams are getting too big during the project the contexts might be split accordingly (AR-3). During the implementation and maintenance of the software it may become clear which parts of the software exhibit increased volatility. This may lead to additional decompositions (AR-4). AR-5 can then be further adapt the decomposition according to quality attributes (QAs) and non-functional requirements (NFRs). AR-7 allows inverting the mentioned operations and merging split Bounded Contexts.

Figure 6 further illustrates that a Bounded Context is defined by one or multiple Aggregates that can be decomposed by Entities (AR-1). They can be merged again with AR-6, similar to Bounded Contexts. The Context Map relationship definitions and their knowledge which Aggregates are exposed in such relationships finally allow us to generate MDSL (micro-)service contracts.

3.3 Metamodel-Based Service Contract Generation

As another contribution, we developed a service contract generator that maps the DDD patterns to MAP patterns, and therefore specifies how such Context Map models can be transformed into a (micro-)service-oriented architecture.

The architecture models using Domain-driven Design (DDD) patterns allow users to describe the decomposition of a software system. However, these models do not indicate how such a system shall be implemented as microser-

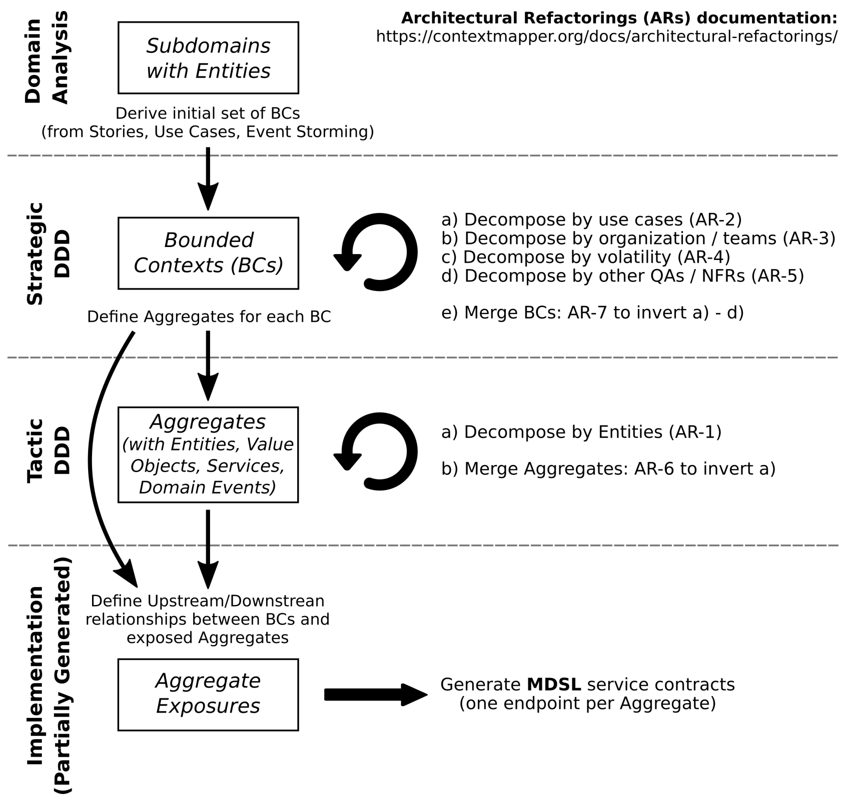

Architectural Refactorings (ARs) documentation:
https://contextmapper.org/docs/architectural-refactorings/
Domain Analysis
Subdomains with Entities
Derive initial set of BCs (from Stories, Use Cases, Event Storming)
Strategic DDD
Bounded Contexts (BCs)
a) Decompose by use cases (AR-2)
b) Decompose by organization / teams (AR-3)
c) Decompose by volatility (AR-4)
d) Decompose by other QAs / NFRs (AR-5)
e) Merge BCs: AR-7 to invert a) - d)
Define Aggregates for each BC
Tactic DDD
Aggregates (with Entities, Value Objects, Services, Domain Events)
a) Decompose by Entities (AR-1)
b) Merge Aggregates: AR-6 to invert a)
Implementation (Partially Generated)
Define Upstream/Downstrean relationships between BCs and exposed Aggregates
Aggregate Exposures
Generate MDSL service contracts (one endpoint per Aggregate)

Fig. 6. Incremental, Stepwise Architectural Refactoring (AR) Application Process

vices. The Microservice API Patterns (MAP)[10] discuss how such microservices shall be implemented and offer corresponding design patterns. As a DSL to denote (micro-)service contracts[11], the Microservice Domain-Specific Language (MDSL)[12] supports the *API Description*[13] pattern.

A Context Mapper generator produces MDSL contracts from CML Context Maps, hence proposing a mapping between the two concepts that specifies how DDD-based models can be implemented as microservices. Lübke et al. [20] presented the domain abstractions and their relationships that are part of such microservice API descriptions. Figure 7 illustrates a simplified version of this

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

[11] A *contract* is not a pair of *precondition and postcondition* here, but an *API description* that specifies and governs the message exchange between two services or subsystems.

[12] `https://microservice-api-patterns.github.io/MDSL-Specification/`

[13] `https://microservice-api-patterns.org/patterns/foundation/APIDescription`

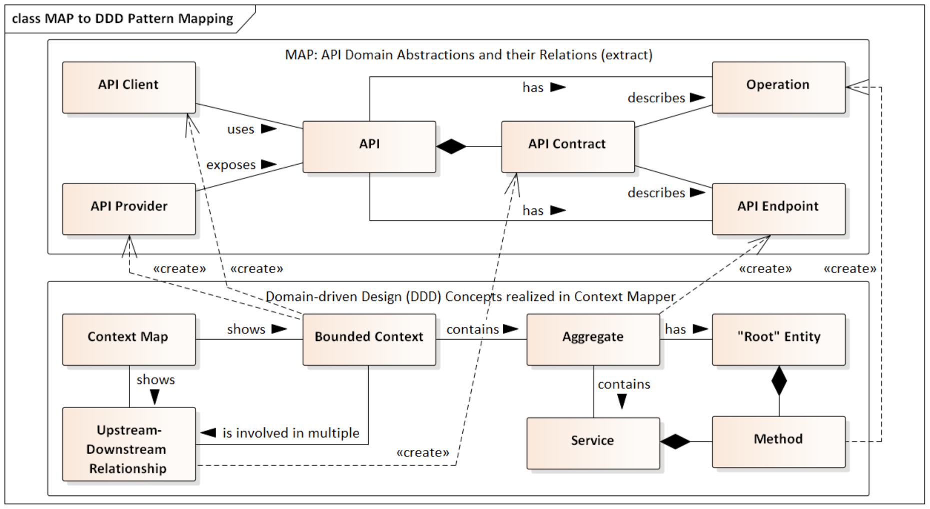


Fig. 7. MAP API Domain Model [20] to Strategic DDD Mapping

domain model and the mapping. An API description [20] describes the exposed endpoints with its operations. An operation expects and delivers specific data types. The described API is used by an API client and exposed by an API provider. Table 2 describes the mappings introduced in Figure 7 in detail; Appendix A then introduces MDSL including its MAP decorators.

Table 2: DDD Context Map to Service Contract Mapping

DDD Concept	MAP Concept	Description
Bounded Contexts (that are upstream in a relationship)	API contract	As many DDD experts we suggest to implement one microservice per Bounded Context. Therefore, we create one API contract per context.
Aggregates exposed by Upstream	API endpoint	We suggest to create a separate endpoint for every Aggregate.
Methods in Aggregate root Entities or Services	Operation	From methods defined in root Entities or Services that are part of the Aggregate we derive operations.
Parameters and return types of the methods mentioned above	Data type definitions (referenced in request and response messages of operations)	The methods that are mapped to API operations as described above declare parameters and return types. From these types we derive corresponding data type definitions.

Table 2: DDD Context Map to Service Contract Mapping

DDD Concept	MAP Concept	Description
Upstream Bounded Context	API provider	The upstream context in an Upstream-Downstream relationship exposes parts of his domain model to be used by the downstream. It is therefore the API provider.
Downstream Bounded Context	API client	The downstream context uses parts of the domain model that are exposed by the upstream. It is therefore mapped to the API client.

The MDSL generator in Context Mapper is documented online[14]. The following Listing 3.1 shows an excerpt of an example MDSL contract generated from the Lakeside Mutual insurance project we modeled in CML.

Listing 3.1. MDSL Example Contract

```
API description CustomerCoreAPI // a.k.a. service  contract
usage context PUBLIC_API for BACKEND_INTEGRATION and FRONTEND_INTEGRATION

data type Customer { "firstname":D<string>, "lastname":D<string>,
                     "sin":SocialInsuranceNumber, "addresses":Address* }
data type SocialInsuranceNumber { "sin":D<string> }
data type Address { "street":D<string>, "postalCode":D<int>, "city":D<string> }
data type AddressId P // placeholder, AddressId not specified in detail
data type createAddressParameter { "customer":Customer, "address":Address }

endpoint type CustomersAggregate
  exposes
    operation createAddress
      expecting
        payload createAddressParameter
      delivering
        payload AddressId
    operation changeAddress
      expecting
        payload Address
      delivering
        payload D<bool>

API provider CustomerCoreProvider
  offers CustomersAggregate
  at endpoint location "http://localhost:8000"
    via protocol "RESTful HTTP"

API client CustomerSelfServiceClient
  consumes CustomersAggregate
API client CustomerManagementClient
  consumes CustomersAggregate
API client PolicyManagementClient
  consumes CustomersAggregate
```

The first block of the contract describes the data types that are used by the operations. The middle part lists the API endpoints with its operations. Each

[14] https://contextmapper.org/docs/mdsl/

operation declares the data types that it expects and delivers. The last part of the contract declares the API providers and clients, derived from the upstream and downstream Bounded Contexts.

DDD and Context Mapper promote SOA and cloud principles such as isolated state, distribution and loose coupling.

Our contract generator contributes a mapping to derive candidate service architectures from DDD Context Maps. Further automation is possible, for instance microservice project stubs (i.e., client and server code) can also be generated, as well as elastic infrastructure code.

4 Validation

We hypothesized that DDD adopters and service designers can benefit from Context Mapper as a modeling tool providing architectural/model refactorings in Section 2.2. The main objective of our ongoing validation activities is to show that the tool is indeed useful and beneficial for the target user group. We validate the tool according to the recommendations of Shaw [29] to demonstrate *correctness*, *usefulness* and *effectiveness*. Hence, we apply empirical validation strategies such as prototyping, action research [1] and case study.

Our Context Mapper implementation prototype uses the Xtext DSL framework in version 2.20.0 and is offered as plugin in the Eclipse marketplace[15] (it is compatible with Eclipse 4.8 and newer). The tool is developed iteratively using CI/CD pipelines[16]; at the time of writing, 55 releases have been made. Context Mapper is listed in our Design Practice Repository.[17]

Context Mapper implements the Architectural Refactorings (ARs) from Section 3 (and several more) as code refactorings for the Context Mapping DSL (CML). These ARs have to be validated w.r.t. their usefulness and effectiveness. During their implementation, we conducted action research to improve the concepts iteratively in short feedback cycles. We modeled larger, realistic sample projects such as *Lakeside Mutual*[18] with CML; an examples repository[19] is available. We also conducted a case study on a real-world project in the healthcare sector [11]. Another real-world use (case study, action research) of Context Mapper is an ongoing research collaboration with a fintech startup; requirements and technical designs of the startup and its clients are modelled in CML with the objective to be able to rapidly respond to business model changes on the API design level. 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 more than 20 exercise participants. We were able to evaluate the simplicity of the tool usage and could improve it according to the feedback.

[15] `https://marketplace.eclipse.org/content/context-mapper/`
[16] `https://travis-ci.com/github/ContextMapper/`
[17] `https://github.com/socadk/design-practice-repository/`
[18] `https://github.com/Microservice-API-Patterns/LakesideMutual/`
[19] `https://github.com/ContextMapper/context-mapper-examples/`

Using a DSL also has its weaknesses. For example: members of the agile community might argue that it does conform with agile practices ("working software over comprehensive documentation"). The "model-code" gap [9] is a weakness of most DSL tools and generators.

The validation results so far support our hypothesis that the target user group can indeed benefit from refactorings implemented as model transformations to ease the creation, evolution and usage of DDD in general and Context Maps in particular; we have to further validate the ARs and their systematic application.

5 Related Work

Domain-driven Design (DDD) Tools. DDD has not only been adopted by practitioner communities, but is picked up in academia as well [7,12,19,21,23,25,28]. However, very few tools specific to DDD exist; agile modeling on whiteboards is commonly practiced. UML tools can be profiled to support DDD as well. None of the existing tools uses a hub-and-spoke architecture.

Architectural Refactorings. The Architectural Refactorings (ARs) [34] proposed in this paper are derived from criteria that are known to be used to decompose software systems. Many of the criteria used to do this are mentioned by the coupling criteria catalog[20] of Service Cutter [10]. The first research paper regarding this topic was published by Parnas [24]. His approach separates parts that change often from other parts of the system. Tune and Millet [32] mention use cases and other domain heuristics such as language, domain expert boundaries, business process steps, data flow, or ownership as criteria that have to be considered. They also mention the importance of co-evolving organizational and technical boundaries which is known as "Conway's Law" [6].

A similar list of criteria has been proposed by Tigges [31]. Linguistic and model differences are the primary drivers for Bounded Context identification according to Plöd [27]. He emphasizes that microservice characteristics such as the organization around business capabilities[21], decentralized governance, and evolutionary design suit the idea behind Bounded Contexts. Brandolini [4] explains how Context Maps can evolve in iterative steps. He recommends *event storming* [5], a workshop technique to analyze domains and derive Bounded Contexts.

These and many other DDD experts propose criteria and practices for decomposing a software system. However, none of them propose concrete systematic or algorithmic solutions for the decomposition process. Our ARs aim at formalizing this process and offer concrete procedures and steps that can be realized as code refactorings for DDD-based modeling languages such as CML.

To the best of our knowledge, comparable architecture modeling tools that are based on strategic DDD patterns do not exist. As Mens and Tourwé [22]

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

[21] https://searchapparchitecture.techtarget.com/definition/business-capability/

mention, there has been a trend towards refactorings on design level. For example Boger et al. [2] discuss refactorings on the level of UML diagrams. Although no similar DSLs with refactorings exist, the technical concept behind them is not new. The ARs are implemented as model transformations [15] that are applied to the Eclipse Modeling Framework (EMF) models [30] behind our Xtext-based DSL [18]. Ivkovic and Kontogiannis [13] introduce another approach to refactor software architecture artifacts using model transformations.

Microservice Contract Generation. With our Microservices Domain-Specific Language (MDSL) contract generator we provide a tool that supports architects regarding the question how DDD-based architecture models can be implemented as microservices. The OpenAPI initiative[22] formerly known as Swagger[23] is a popular notation for HTTP resource API contracts; many tools exist, including editors, test tools and server stub and client proxy code gererators.

6 Summary and Outlook

In this paper we presented the concepts of Context Mapper and its open source tools. Context Mapper provides an architecture modeling language supporting strategic and tactic Domain-driven Design (DDD). As our research contributions, we proposed a) a modular and extensible framework architecture for DDD-based modeling tools, b) a tool-supported refactoring catalog and method for decomposing systems step by step from DDD Context Maps, and c) a mapping from DDD to the domain model of the Microservice API patterns (MAP) language that specifies how DDD models can be realized as microservice contracts, expressed in the emerging Microservice Domain-Specific Language (MDSL). The provided tool capabilities support DDD adopters in formalizing DDD Context Maps and evolving them in an iterative manner.

Our validation via implementation (prototyping), action research, and case studies already support our hypothesis that the modeling language and the transformation tools can support architects when modeling domain-driven designs and decomposing software systems. Additional validation activities are required to find out whether the proposed ARs are sufficient or will have to be extended.

In our future work we plan to evolve and enhance Context Mapper by extending its framework components. The tool shall evolve into a modeling framework that not only allows describing models and generating contracts from them, but also supports software maintainers that reverse engineer models from source code. The AR-based decomposition method will be validated and matured further. In addition, we could support additional analysis and design transformations and let the service contract generator create microservice project stubs. Using the JDL of JHipster[24], a popular rapid application development platform

[22] `https://www.openapis.org/`
[23] `https://swagger.io/`
[24] `https://www.jhipster.tech/jdl/`

(generating JavaScript frontends and Java/Spring backends from architectural input) is another direction towards generating client and server code. Last but not least, providing support for multiple IDEs (for instance, Visual Studio Code) can (and already has) increased our user group.

Acknowledgements: This work was supported by the Hasler Foundation[25] in the project "Domain-Driven Digital Service Engineering".

A Appendix: Introduction to MDSL

Microservice Domain-Specific Language (MDSL)[26] abstracts from technology-specific interface description languages such as OpenAPI/Swagger, WSDL, and Protocol Buffers. MDSL design principles are a) promote platform and protocol independence with a modular language structure separating abstract contracts from provider bindings, b) support agile modeling practices with partial specifications and c) promote readability over parsing efficiency.

The abstract syntax of MDSL is inspired and driven by the domain model and concepts of Microservice API Patterns (MAP), featuring endpoints, operations, and data representation elements [20]. The concrete syntax of *service endpoint contracts* is elaborate; the concrete syntax of *data contracts* is compact yet simple. It generalizes data exchange formats such as JSON and type systems in service-centric programming languages such as Ballerina and Jolie. Endpoint, operation, and one representation element can be decorated with patterns from MAP[20]; these decorator annotations are first-class language concepts that can be processed by tools later. For instance, API linters may validate them, and they could influence the output of cloud deployment scripts.

Most programming languages declare variables by name and type; MDSL uses a *name-role-type* triple (e.g., `"customerName": ID<String>`) to specify *Atomic Parameters*. The role can be any *element stereotype* from MAP (i.e., ID(entifier), Link, Data, or Metadata). A generic, unspecified placeholder `P` can replace role and type; the parameter name is optional if the role is defined. Implementing the *Parameter Tree* pattern from MAP, simple (yet powerful) nesting is supported in an object- and block-like curly brace syntax `{{...}, {...}}` known from data representation languages such as JSON. Cardinalities such as `?`, `*`, `+` can be specified as well. Listing A.1 gives an example:

[25] https://haslerstiftung.ch/

[26] https://microservice-api-patterns.github.io/MDSL-Specification/index

Listing A.1. MDSL Example: Data Contract and Endpoint Contract

```
API description HelloWorldAPI // a.k.a. service contract
data type SampleDTO {ID, D<int>} // partially specified tree (two leaves)

endpoint type HelloWorldEndpoint serves as PROCESSING_RESOURCE // MAP decorator
exposes
  operation sayHello with responsibility  COMPUTATION_FUNCTION // MAP decorator
    expecting payload "greeting": D<string>+ // one or more greetings
    delivering payload <<error_report>> SampleDTO // MAP decorator (<<pattern>>)

API provider HelloWorldAPIProvider1
  offers HelloWorldEndpoint
  at endpoint location "https://..." via protocol HTTP
API client HelloWorldAPIClient1
  consumes HelloWorldEndpoint
  from HelloWorldAPIProvider1 via protocol HTTP
```

`sayHello` accepts a scalar string value `D<string>` as input. This operation returns a Data Transfer Object (DTO) called `SampleDTO`, which is modeled explicitly so that its specification can be used elsewhere too. `SampleDTO` is specified incompletely: it pairs two atomic parameters, in ID and (D)ata roles, whose names and types have not been specified. This partial yet expressive specification supports early use and continuous refinement. For instance, MDSL specifications can be drafted in workshops with non-technical stakeholders and then completed iteratively and incrementally (e.g, adding data type information).

References

1. 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
2. Boger, M., Sturm, T., Fragemann, P.: Refactoring browser for uml. In: Objects, Components, Architectures, Services, and Applications for a Networked World. pp. 366–377. Springer Berlin Heidelberg (2003)
3. Brandner, M., Craes, M., Oellermann, F., Zimmermann, O.: Web services-oriented architecture in production in the finance industry. Informatik Spektrum **27**(2), 136–145 (2004). https://doi.org/10.1007/s00287-004-0380-2
4. Brandolini, A.: Strategic domain driven design with context mapping. `https://www.infoq.com/articles/ddd-contextmapping` (2009)
5. Brandolini, A.: Introducing EventStorming: An act of Deliberate Collective Learning. Leanpub (2018)
6. Conway, M.: Conway's law (1968)
7. Di Francesco, P., 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
8. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2003)
9. Fairbanks, G.: Just enough software architecture: a risk-driven approach. Marshall & Brainerd (2010)

10. 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)
11. Habegger, M., Schena, M.: Cloud-Native Refactoring in einem mHealth Szenario. Bachelor thesis, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2019), `https://eprints.hsr.ch/806/`
12. Hippchen, B., Giessler, P., Steinegger, R., Schneider, M., Abeck, S.: Designing microservice-based applications by using a domain-driven design approach. International Journal on Advances in Software (1942-2628) **10**, 432 – 445 (12 2017)
13. Ivkovic, I., Kontogiannis, K.: A framework for software architecture refactoring using model transformations and semantic annotations. In: Conference on Software Maintenance and Reengineering (CSMR'06). pp. 10 pp.–144 (March 2006). https://doi.org/10.1109/CSMR.2006.3
14. Johnson, R., Gamma, E., Vlissides, J., Helm, R.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
15. Kapferer, S.: Model Transformations for DSL Processing. Term project, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2019), `https://eprints.hsr.ch/819/`
16. Kapferer, S.: Service Decomposition as a Series of Architectural Refactorings. Term project, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2019), `https://eprints.hsr.ch/784/`
17. Kapferer, S., Jost, S.: Attributbasierte Autorisierung in einer Branchenlösung für das Versicherungswesen. Bachelor thesis, University of Applied Sciences of Eastern Switzerland (HSR FHO) (2017), `https://eprints.hsr.ch/602/`
18. Kapferer, S., Zimmermann, O.: Domain-specific language and tools for strategic domain-driven design, context mapping and bounded context modeling. In: Proc. of the 8th International Conference on MODELSWARD. pp. 299–306. INSTICC, SciTePress (2020). https://doi.org/10.5220/0008910502990306
19. Landre, E., Wesenberg, H., Rønneberg, H.: Architectural improvement by use of strategic level domain-driven design. In: Companion to the 21st ACM SIGPLAN OOPSLA. pp. 809–814. ACM, New York, NY, USA (2006). https://doi.org/10.1145/1176617.1176728
20. Lübke, D., Zimmermann, O., Pautasso, C., Zdun, U., Stocker, M.: Interface evolution patterns: Balancing compatibility and extensibility across service life cycles. In: Proc. of the 24th European Conference on Pattern Languages of Programs. ACM, New York, NY, USA (2019). https://doi.org/10.1145/3361149.3361164
21. 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
22. Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Transactions on Software Engineering **30**(2), 126–139 (Feb 2004). https://doi.org/10.1109/TSE.2004.1265817
23. Munezero, I.J., Mukasa, D., Kanagwa, B., Balikuddembe, J.: Partitioning microservices: A domain engineering approach. In: 2018 IEEE/ACM Symposium on Software Engineering in Africa (SEiA). pp. 43–49 (May 2018)
24. 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

25. 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
26. Plöd, M.: DDD Context Maps - an enhanced view. `https://speakerdeck.com/mploed/context-maps-an-enhanced-view` (2018)
27. Plöd, M.: Hands-on Domain-driven Design - by example. Leanpub (2019)
28. 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
29. Shaw, M.: Writing good software engineering research papers: Minitutorial. In: Proc. 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`
30. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Eclipse Series, Pearson Education (2008)
31. Tigges, O.: How to break down a domain to bounded contexts? speakerdeck.com/otigges/how-to-break-down-a-domain-to-bounded-contexts, [Online; Accessed: 2020-02-14]
32. Tune, N., Millett, S.: Designing Autonomous Teams and Services: Deliver Continuous Business Value Through Organizational Alignment. O'Reilly Media (2017)
33. Vernon, V.: Implementing Domain-Driven Design. Addison-Wesley (2013)
34. Zimmermann, O.: Architectural refactoring for the cloud: decision-centric view on cloud migration. Computing **99**(2), 129–145 (2017). https://doi.org/10.1007/s00607-016-0520-y, `http://rdcu.be/lFW6`