OST
Eastern Switzerland
University of Applied Sciences

Domain-driven Service Design

Context Modeling, Model Refactoring and Contract Generation

Stefan Kapferer and Prof. Dr. Olaf Zimmermann

SummerSoC 2020,
September 14, 2020

Institute for Software

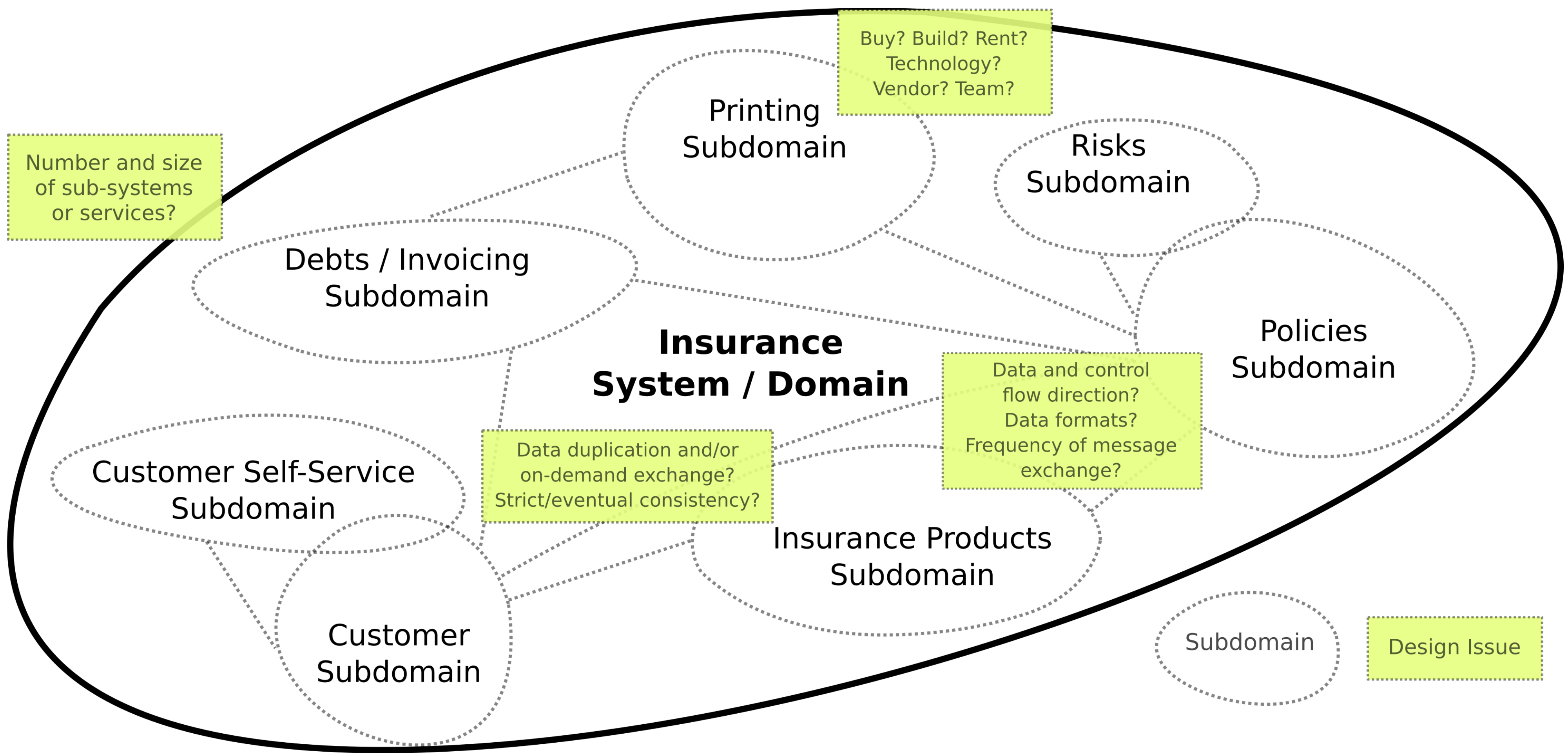
SummerSoC
Service Oriented Computing

Agenda

- **Motivation**
- **Strategic Domain-driven Design (DDD)**
- **Open Source Project: Context Mapper**
  - Domain-specific Language (DSL)
  - Framework Architecture
- **Selected Paper Contributions**
  - Architectural Refactorings (ARs)
  - Stepwise Service Decomposition Method
- **Short Context Mapper Demo**
- **Summary and Outlook**
- **Q&A**

© Stefan Kapferer, Olaf Zimmermann, 2020

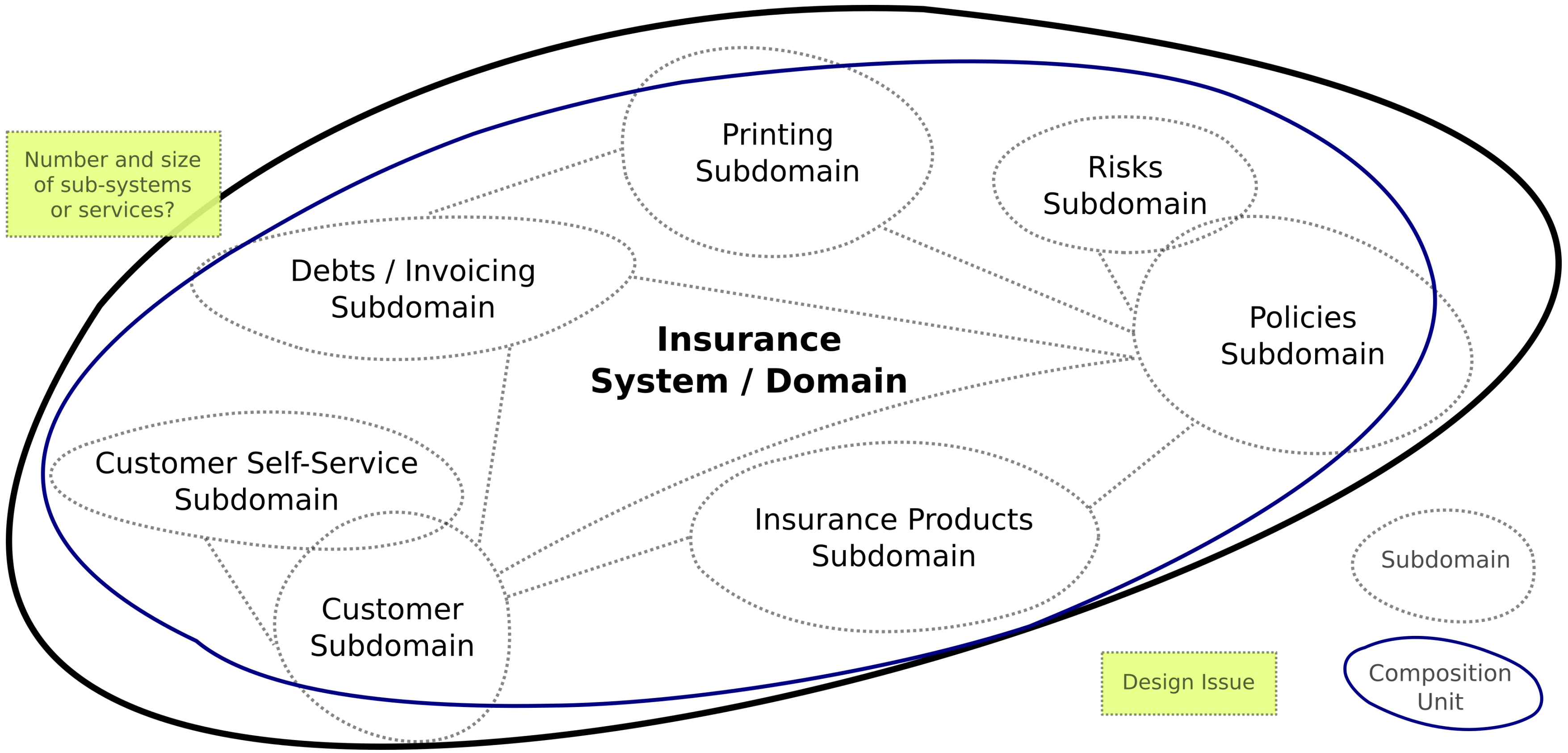
Motivation: Fictitious Insurance Software


Buy? Build? Rent? Technology? Vendor? Team?
Number and size of sub-systems or services?
Printing Subdomain
Risks Subdomain
Debts / Invoicing Subdomain
Policies Subdomain
Insurance System / Domain
Data and control flow direction? Data formats? Frequency of message exchange?
Customer Self-Service Subdomain
Data duplication and/or on-demand exchange? Strict/eventual consistency?
Insurance Products Subdomain
Customer Subdomain
Subdomain
Design Issue

© Stefan Kapferer, Olaf Zimmermann, 2020

Motivation: How to decompose the system? (1/3)

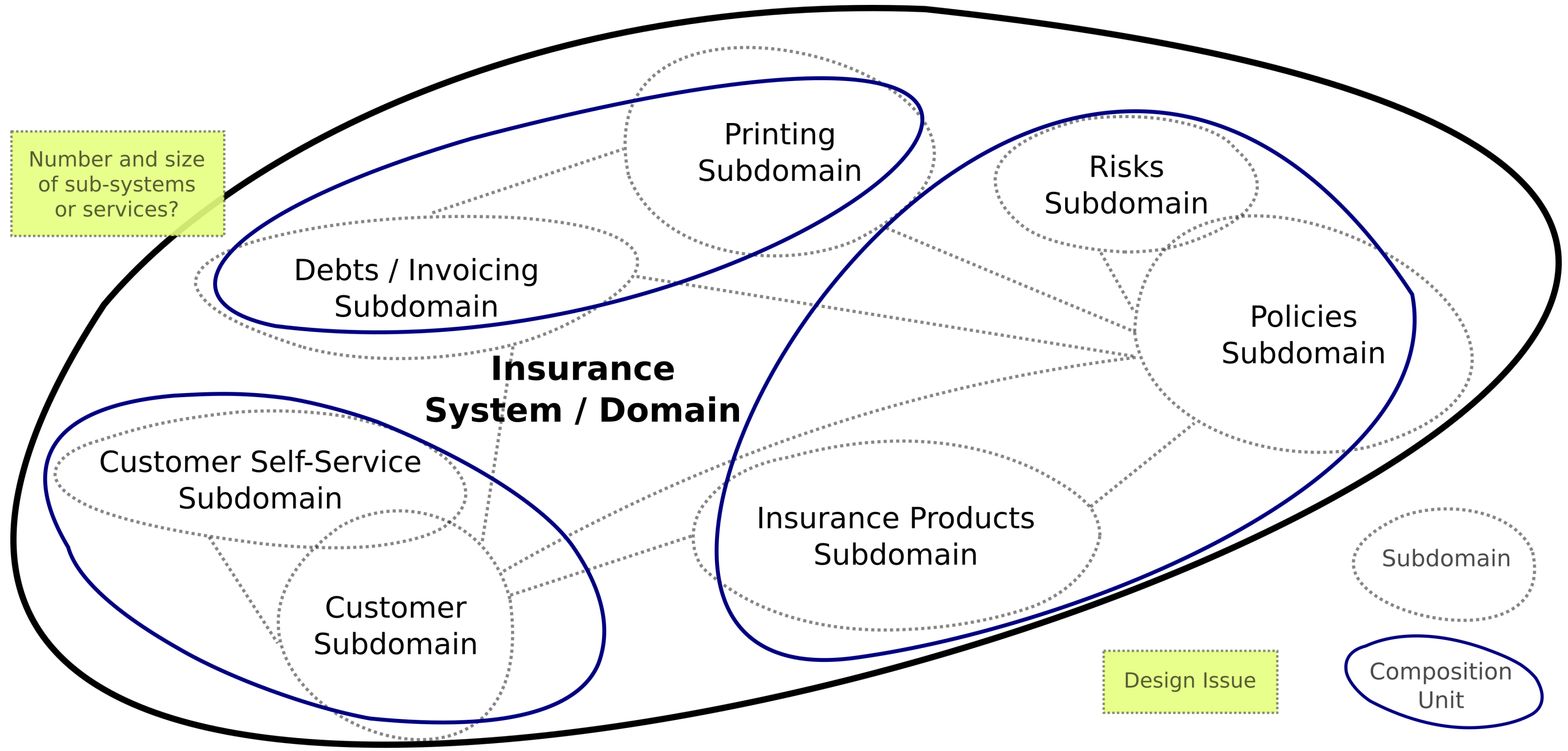
- Implementation as one single system? («Monolith») 🤔


Number and size of sub-systems or services?
Printing Subdomain
Risks Subdomain
Debts / Invoicing Subdomain
Insurance System / Domain
Policies Subdomain
Customer Self-Service Subdomain
Insurance Products Subdomain
Customer Subdomain
Subdomain
Design Issue
Composition Unit

© Stefan Kapferer, Olaf Zimmermann, 2020

Motivation: How to decompose the system? (2/3)

- Decompose into three subsystems? 🤔



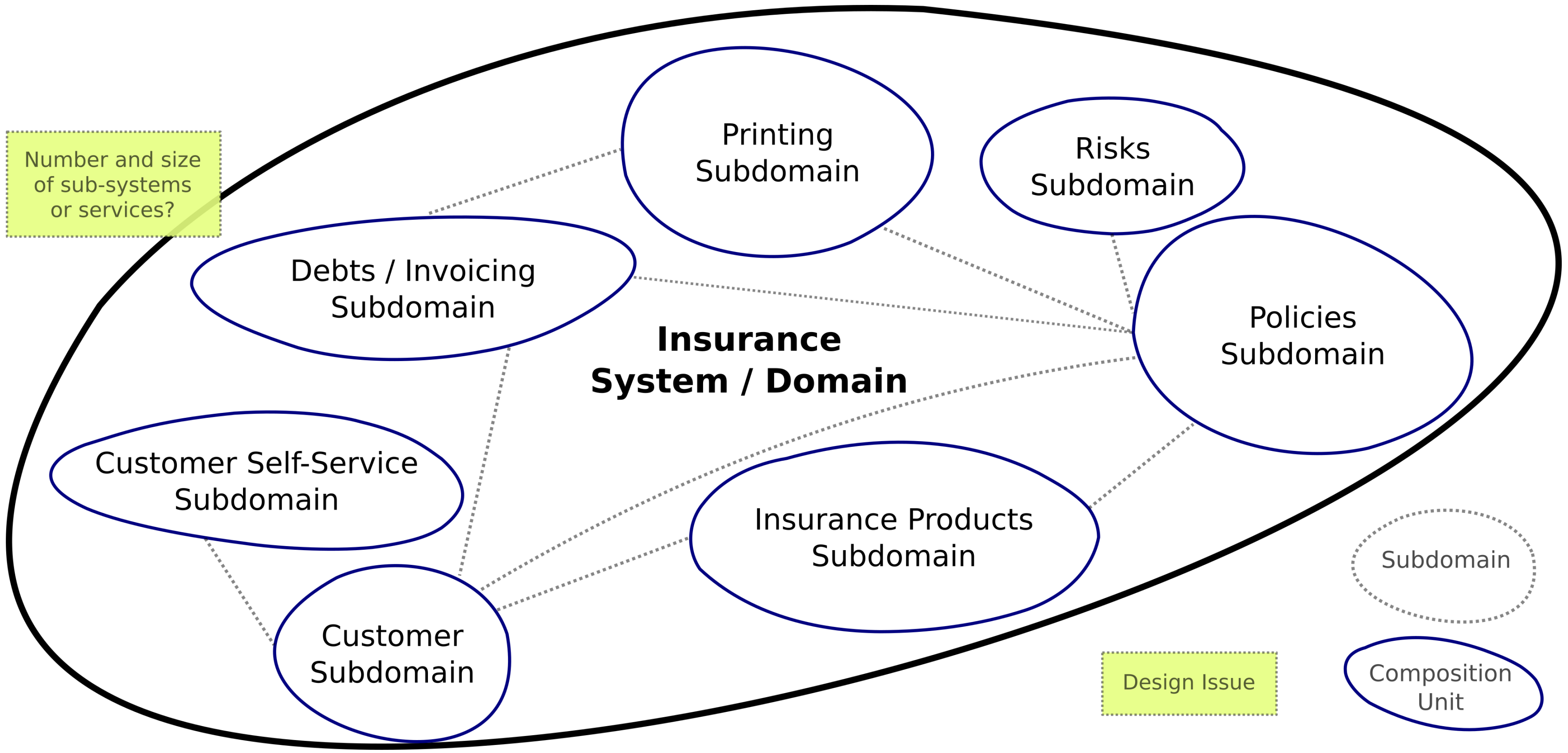
Number and size of sub-systems or services?
Printing Subdomain
Risks Subdomain
Debts / Invoicing Subdomain
Policies Subdomain
Insurance System / Domain
Customer Self-Service Subdomain
Insurance Products Subdomain
Customer Subdomain
Subdomain
Design Issue
Composition Unit

© Stefan Kapferer, Olaf Zimmermann, 2020

IFS INSTITUTE FOR SOFTWARE

Motivation: How to decompose the system? (3/3)

- Or one system per subdomain?


Number and size of sub-systems or services?
Printing Subdomain
Risks Subdomain
Debts / Invoicing Subdomain
Insurance System / Domain
Policies Subdomain
Customer Self-Service Subdomain
Insurance Products Subdomain
Customer Subdomain
Subdomain
Design Issue
Composition Unit

© Stefan Kapferer, Olaf Zimmermann, 2020

Motivation: Project Vision as User Story

As a **software architect** *I want to*

model the subsystems and components of my architecture and how they interact

so that

I can evolve the architecture semi-automatically (i.e, supported by model refactorings and service decomposition heuristics), communicate the architecture, and generate other representations of the models such as Unified Modeling Language (UML) diagrams and service API contracts (or even code).

Strategic Domain-driven Design (DDD)

- A popular answer these days:
  Domain-driven Design (DDD)
- Emphasizes need for modeling and communication
  - Ubiquitous language (vocabulary) - the domain model
- **Tactic DDD**
  - Decomposes a domain model
  - Entities, Value Objects, Services, Repositories
  - Grouped into Aggregates
- **Strategic DDD**
  - Defining boundaries around and between domain models
  - Teams, subsystems, components modeled as «Bounded Contexts»

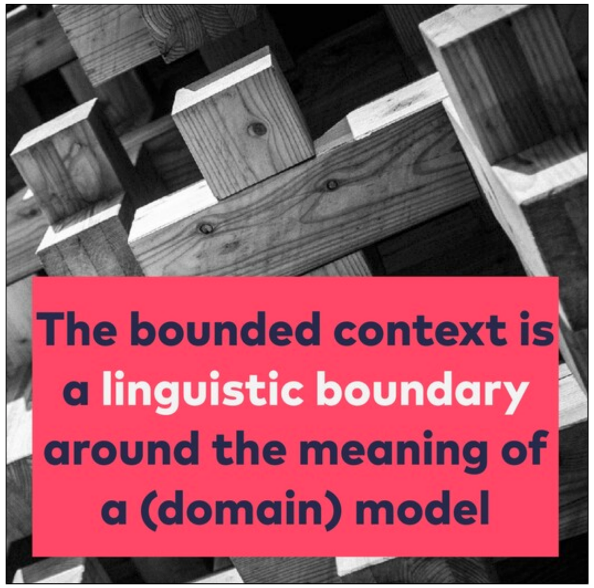

The bounded context is a linguistic boundary around the meaning of a (domain) model

Image reference: Michael Plöd, *Aligning organization and architecture with strategic DDD* (Slides)

Bounded Context and Context Mapping

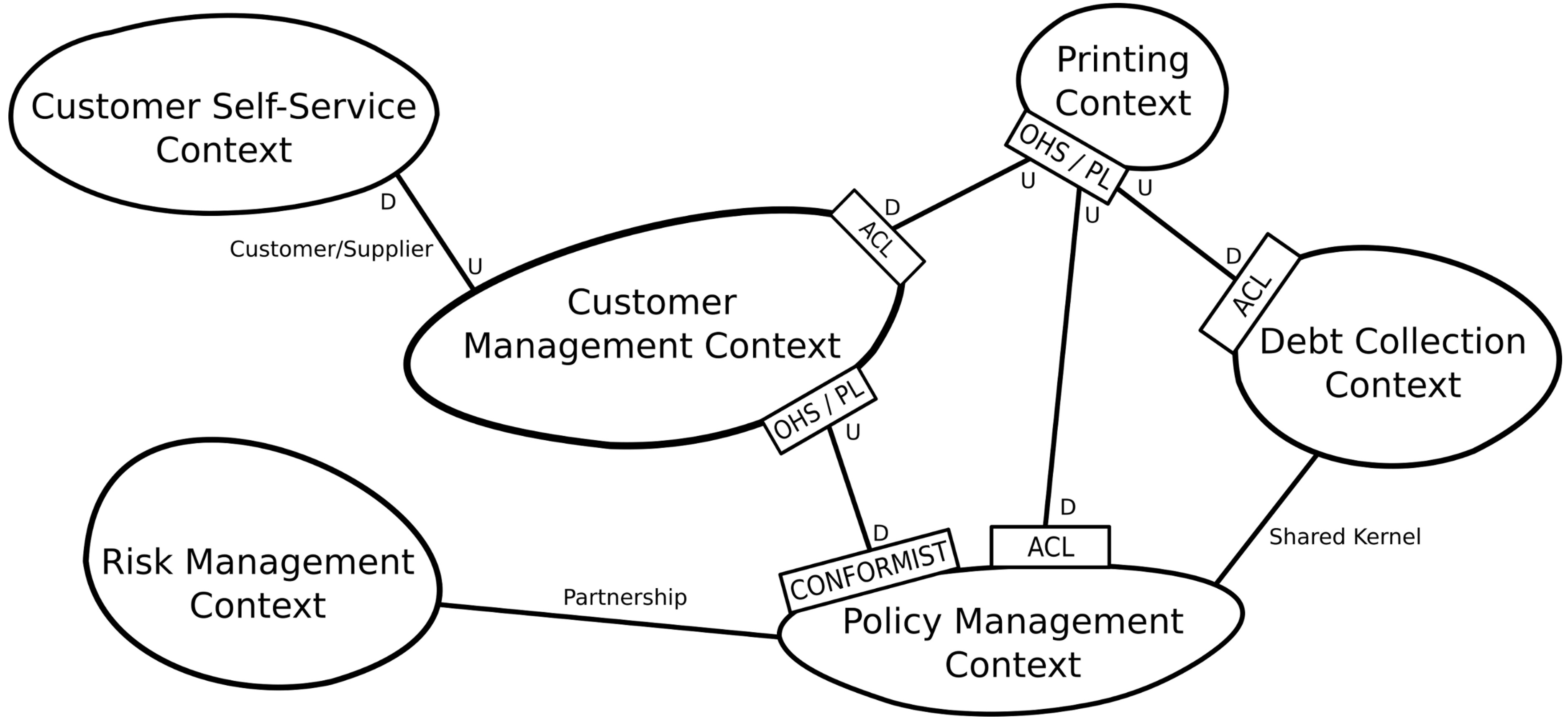
- **Bounded Context**
  - Establishes a boundary within which a particular domain model is valid.
  - The concepts within a Bounded Context must be defined clearly and distinctively: «*ubiquitous language*»[2]
  - Abstractions of (sub-) systems and teams.
  - Realize parts of one or multiple subdomains.
  - Heuristic: implement one (micro-)service per Bounded Context.[3] [4]
- **Context Map**
  - Define how Bounded Contexts integrate.
  - «Information flow»

[2] **Reference:** Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley (2003)

[3] **Reference:** Jan Stenberg, *Vaughn Vernon on Microservices and Domain-Driven Design*, InfoQ (Link)

[4] **Reference:** Nick Tune, *Domain-Driven Design: Hidden Lessons from the Big Blue Book*, Craft Conf 2019 (Slides)

Context Map for Insurance Example



Legend: Upstream (U), Downstream (D), Open Host Service (OHS), Published Language (PL), Anticorruption-Layer (ACL)

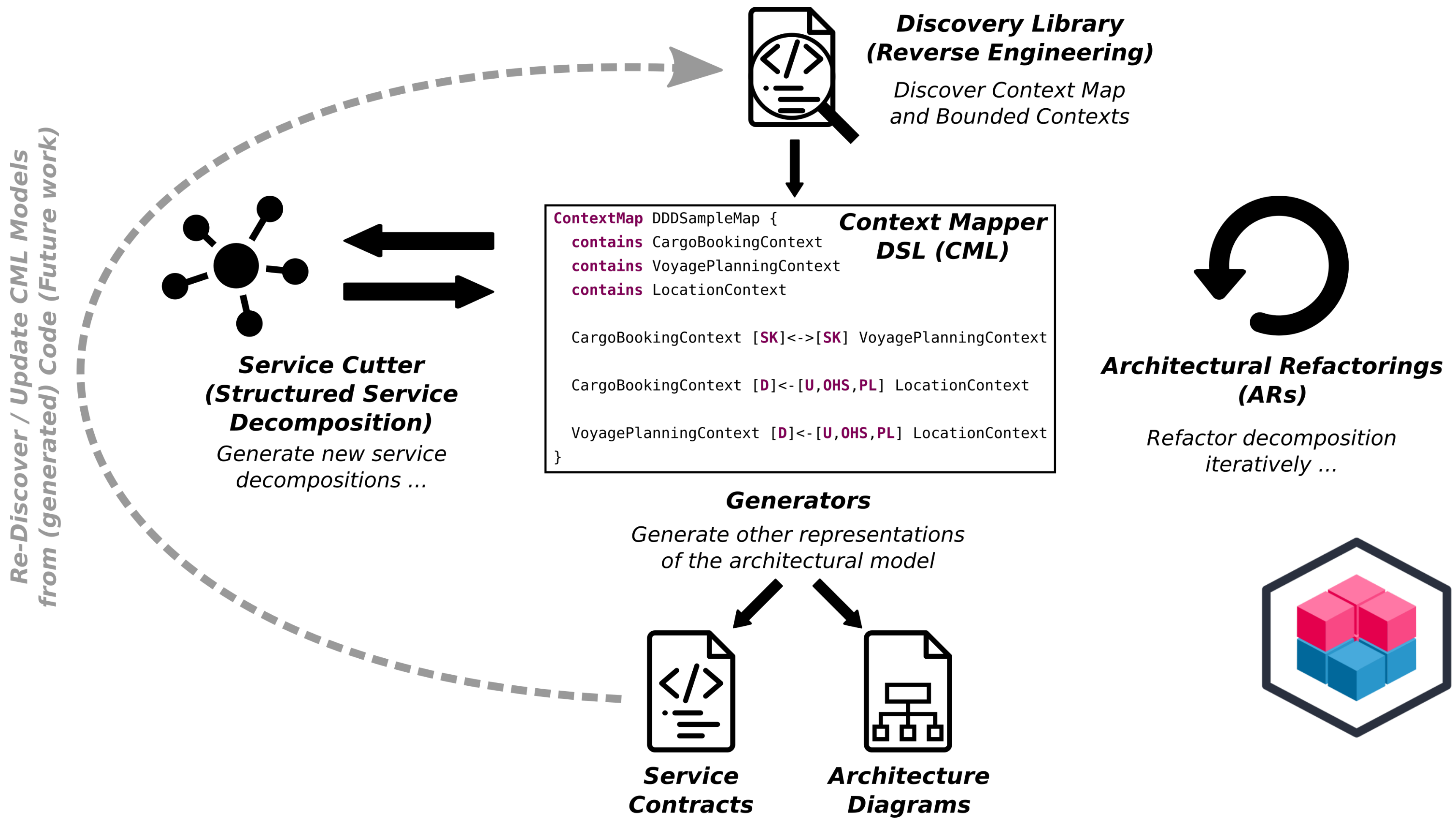
Context Mapper: A DSL for Strategic DDD

- Human- and machine-readable language for writing Context Maps[5]

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

    CargoBookingContext [SK]<->[SK] VoyagePlanningContext

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

    LocationContext [U,OHS,PL]->[D] VoyagePlanningContext
}

BoundedContext CargoBookingContext { /* tactic DDD */ }
BoundedContext VoyagePlanningContext { /* tactic DDD */ }
BoundedContext LocationContext { /* tactic DDD */ }
```

[5] **Website:** contextmapper.org

Context Mapper: Framework Architecture



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

  CargoBookingContext [SK]<->[SK] VoyagePlanningContext

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

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

CONTEXT
MAPPER

© Stefan Kapferer, Olaf Zimmermann, 2020

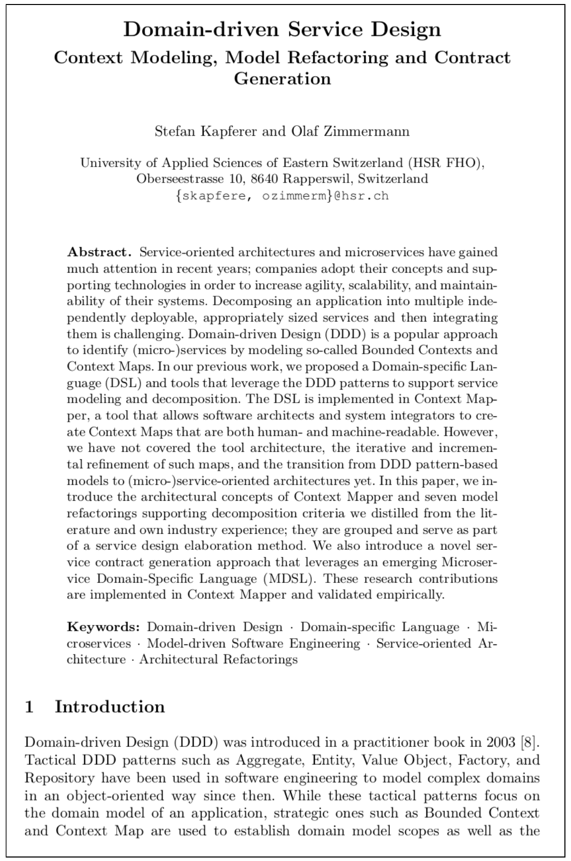
Context Mapper: DSL Benefits

Machine-readable approach allows us to:

- Generate different architecture diagrams/visualizations
- Generate service (API) contracts
- Generate code
- Apply model transformations[6]
  - Implement model (architectural) refactorings as model transformations

[6] **Reference:** Stefan Kapferer, *Model Transformations for DSL Processing*, Term Project (2019), eprints.hsr.ch/819

Scope of SummerSoC 2020 Paper


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

Stefan Kapferer and Olaf Zimmermann

University of Applied Sciences of Eastern Switzerland (HSR FHO),
Oberseestrasse 10, 8640 Rapperswil, Switzerland
{skapfere, ozimmerm}@hsr.ch

Abstract. Service-oriented architectures and microservices have gained much attention in recent years; companies adopt their 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) and tools that leverage 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 Context Maps 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 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 an emerging Microservice Domain-Specific Language (MDSL). These research contributions are implemented in Context Mapper and validated empirically.

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 are used to establish domain model scopes as well as the

1. Service Decomposition with Strategic Domain-driven Design (DDD) patterns
2. Context Mapper (framework): a machine-readable approach to Strategic DDD
3. Decomposition Criteria
4. Architectural Refactorings (AR)
5. An incremental method to decompose services «step by step»
6. Service contract generation out of DDD models

Unfortunately we cannot cover all our topics in this short presentation.

Scope of SummerSoC 2020 Paper

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

Stefan Kapferer and Olaf Zimmermann

University of Applied Sciences of Eastern Switzerland (HSR FHO),
Oberseestrasse 10, 8640 Rapperswil, Switzerland
{skapfere, ozimmerm}@hsr.ch

Abstract. Service-oriented architectures and microservices have gained much attention in recent years; companies adopt their 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) and tools that leverage 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 Context Maps 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 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 an emerging Microservice Domain-Specific Language (MDSL). These research contributions are implemented in Context Mapper and validated empirically.

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 are used to establish domain model scopes as well as the

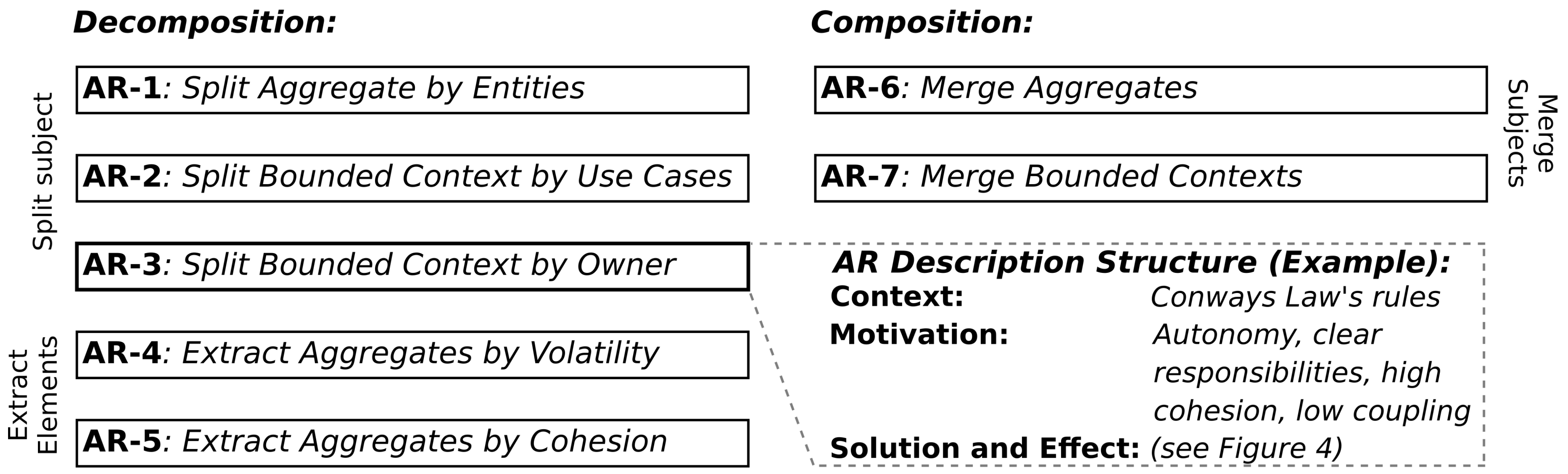
1. Service Decomposition with Strategic Domain-driven Design (DDD) patterns
2. Context Mapper (framework): a machine-readable approach to Strategic DDD
3. Decomposition Criteria
4. **Architectural Refactorings (AR)**
5. **An incremental method to decompose services «step by step»**
6. Service contract generation out of DDD models

So, let us give you a glimpse into the incremental refactoring method ...

© Stefan Kapferer, Olaf Zimmermann, 2020

Architectural Refactorings for Context Map Models

- We distilled «Decomposition Criteria» empirically (see Appendix for more)
- Based on those criteria, we derived «Architectural Refactorings»[7] (model transformations):



Decomposition:

Split subject
- **AR-1**: *Split Aggregate by Entities*
- **AR-2**: *Split Bounded Context by Use Cases*
- **AR-3**: *Split Bounded Context by Owner*

Extract Elements
- **AR-4**: *Extract Aggregates by Volatility*
- **AR-5**: *Extract Aggregates by Cohesion*

Composition:

Merge Subjects
- **AR-6**: *Merge Aggregates*
- **AR-7**: *Merge Bounded Contexts*

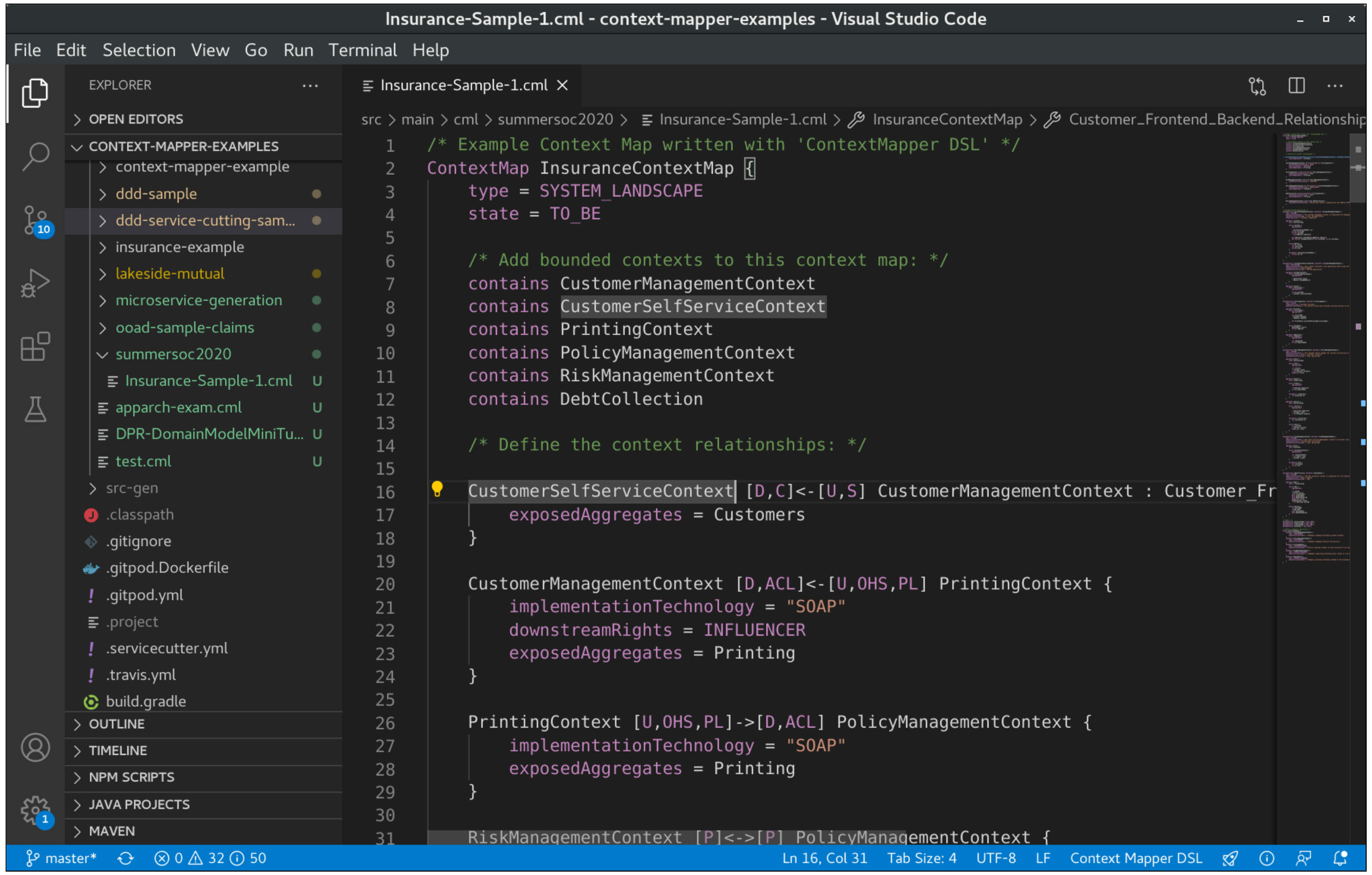
AR Description Structure (Example):

Context:	*Conways Law's rules*
Motivation:	*Autonomy, clear responsibilities, high cohesion, low coupling*
Solution and Effect:	*(see Figure 4)*

7 **Reference:** Neri D., Soldani, J., Zimmermann, O., Brogi, A: *Design Principles, Architectural Smells and Refactorings for Microservices. A Multivocal Review.* In: SICS Software-Intensive Cyber-Physical Systems (Springer 2019). (PDF)
Reference: Zimmermann, O.: *Architectural Refactoring for the Cloud: a Decision-Centric View on Cloud Migration*. In: Springer Computing, 2017, pp 129–145. (PDF)

Context Mapper: Demo

- **Short Context Mapper demonstration:**
  - Editor support
  - One architectural refactoring (AR)
- **Download links:**
  - Visual Studio Code
  - Online (Browser) IDE
  - Eclipse



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

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

    /* Define the context relationships: */

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

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

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

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

© Stefan Kapferer, Olaf Zimmermann, 2020

Stepwise Service Decomposition Method

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)

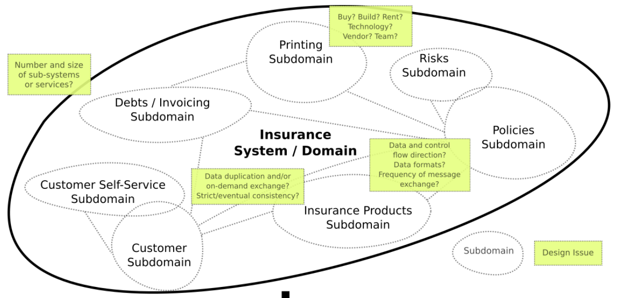
Define Aggregates for each BC

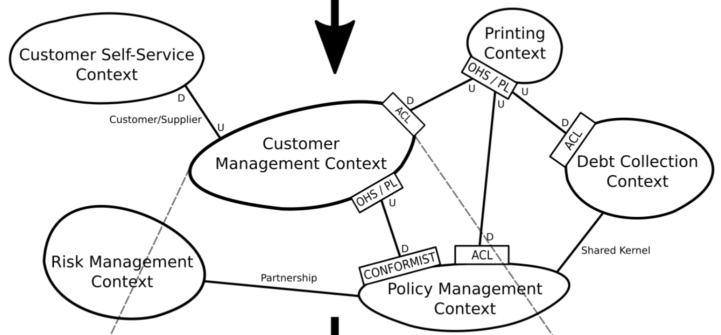
Tactic DDD

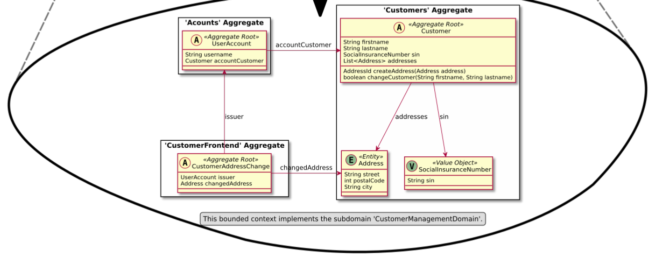
Aggregates (with Entities, Value Objects, Services, Domain Events)

a) Decompose by Entities (AR-1)

b) Merge Aggregates: AR-6 to invert a)


Buy? Build? Rent? Technology? Vendor? Team?
Number and size of sub-systems or services?
Printing Subdomain
Risks Subdomain
Debts / Invoicing Subdomain
Insurance System / Domain
Policies Subdomain
Data and control flow direction? Data formats? Frequency of message exchange?
Customer Self-Service Subdomain
Data duplication and/or on-demand exchange? Strict/eventual consistency?
Insurance Products Subdomain
Customer Subdomain
Subdomain
Design Issue




'Acounts' Aggregate
A «Aggregate Root» UserAccount
String username
Customer accountCustomer
accountCustomer
'Customers' Aggregate
A «Aggregate Root» Customer
String firstname
String lastname
SocialInsuranceNumber sin
List<Address> addresses
AddressId createAddress(Address address)
boolean changeCustomer(String firstname, String lastname)
issuer
addresses
sin
'CustomerFrontend' Aggregate
A «Aggregate Root» CustomerAddressChange
UserAccount issuer
Address changedAddress
changedAddress
E «Entity» Address
String street
int postalCode
String city
V «Value Object» SocialInsuranceNumber
String sin
This bounded context implements the subdomain 'CustomerManagementDomain'.

© Stefan Kapferer, Olaf Zimmermann, 2020

Summary and Outlook

- DDD is a trending **approach for service decomposition**
- **Context Mapper**: a **machine-readable** approach for DDD models
- Supports systematic and **stepwise decomposition** through **architectural refactorings**
- Approach allows us to **generate** architecture diagrams and other representations of models:
  - PlantUML: component and class diagrams
  - Graphical Context Maps
  - Service contracts (MDSL)
  - Code: Spring Boot applications via JHipster
- **Future enhancements**:
  - Generate code for test automation (Test-driven Development)
  - More discovery (reverse engineering) strategies
  - *Feedback and ideas for improvements are always welcome!*

More Information and Links

- **Presentation**
  - Download this slides and example model: stefan.kapferer.ch/SummerSoC2020
- **Context Mapper**
  - contextmapper.org
    - Visual Studio Code plugin
    - Eclipse plugin
    - Online IDE via Gitpod
  - All open source: github.com/ContextMapper
  - Live Demo of *Online IDE*: contextmapper.org/demo
  - Example and case study models: github.com/ContextMapper/context-mapper-examples
- **Previous papers and presentations**:
  - contextmapper.org/background-and-publications

© Stefan Kapferer, Olaf Zimmermann, 2020

Q&A

Thank you very much for your attention.

Let's move on to Q&A and discussion...

© Stefan Kapferer, Olaf Zimmermann, 2020

Appendix

Bounded Context Identification

Which criteria can we use to decompose our domain? 🤔

© Stefan Kapferer, Olaf Zimmermann, 2020

Decomposition Criteria

- Some criteria typically mentioned by practitioners and DDD experts:
  - Use Cases
  - Language and domain expert boundaries
  - Business process steps
  - Business capabilities
  - Data flow
  - Ownership and teams (Conways Law)
  - Non-functional requirements (NFRs) such as security, availability, etc.

© Stefan Kapferer, Olaf Zimmermann, 2020

Decomposition Criteria

- DDD experts and practitioners that provide criteria and heuristics:
  - M. Plöd: *Hands-on Domain-driven Design - by Example* (Leanpub)
  - N. Tune and S. Millett: *Designing Autonomous Teams and Services: Deliver Continuous Business Value Through Organizational Alignment.*
  - O. Tigges: *How to break down a Domain to Bounded Contexts* (speakerdeck.com/otigges/how-to-break-down-a-domain-to-bounded-contexts)
  - A. Brandolini: *Strategic Domain Driven Design with Context Mapping* (infoq.com/articles/ddd-contextmapping/)
- A catalog of coupling criteria researched from literature and industry experience:
  - github.com/ServiceCutter/ServiceCutter/wiki/Coupling-Criteria

From DDD Models towards Service Implementation

Once a Context Map and the domain models inside the Bounded Contexts are created, another question arises:

How to implement the corresponding services (DDD Bounded Contexts)?

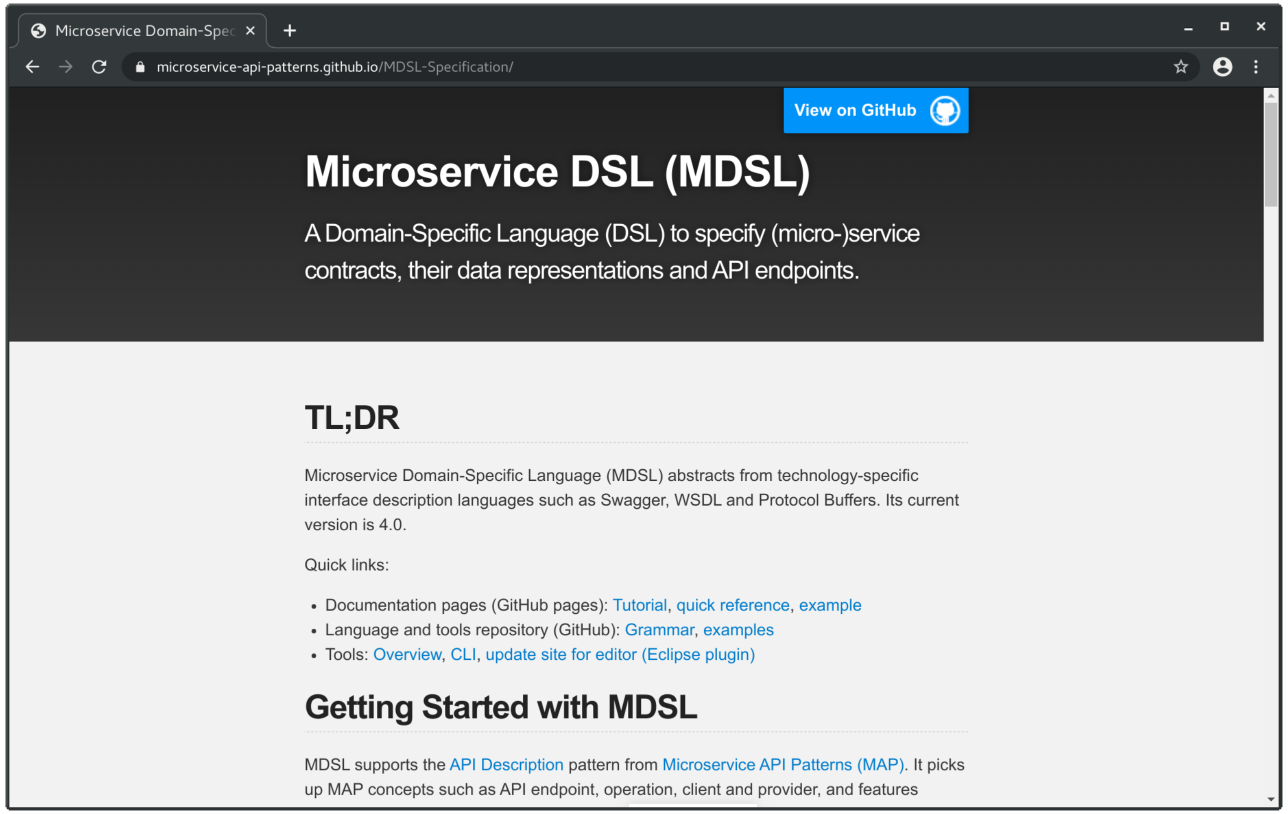
© Stefan Kapferer, Olaf Zimmermann, 2020

From DDD Models towards Service Implementation

- Microservice API Patterns (MAP) answer questions regarding how (micro-)services shall be implemented.
  - microservice-api-patterns.org
- Microservice Domain-Specific Language (MDSL) language implements *API Description* pattern of MAP.
- We developed a mapping between the meta models of DDD Context Maps and MDSL API descriptions.
- And: a generator in Context Mapper that generates MDSL service contracts out of CML Context Maps.

© Stefan Kapferer, Olaf Zimmermann, 2020

Microservice Domain-specific Language (MDSL)



Microservice DSL (MDSL)

A Domain-Specific Language (DSL) to specify (micro-)service contracts, their data representations and API endpoints.

TL;DR

Microservice Domain-Specific Language (MDSL) abstracts from technology-specific interface description languages such as Swagger, WSDL and Protocol Buffers. Its current version is 4.0.

Quick links:

- Documentation pages (GitHub pages): Tutorial, quick reference, example
- Language and tools repository (GitHub): Grammar, examples
- Tools: Overview, CLI, update site for editor (Eclipse plugin)

Getting Started with MDSL

MDSL supports the API Description pattern from Microservice API Patterns (MAP). It picks up MAP concepts such as API endpoint, operation, client and provider, and features

microservice-api-patterns.github.io/MDSL-Specification

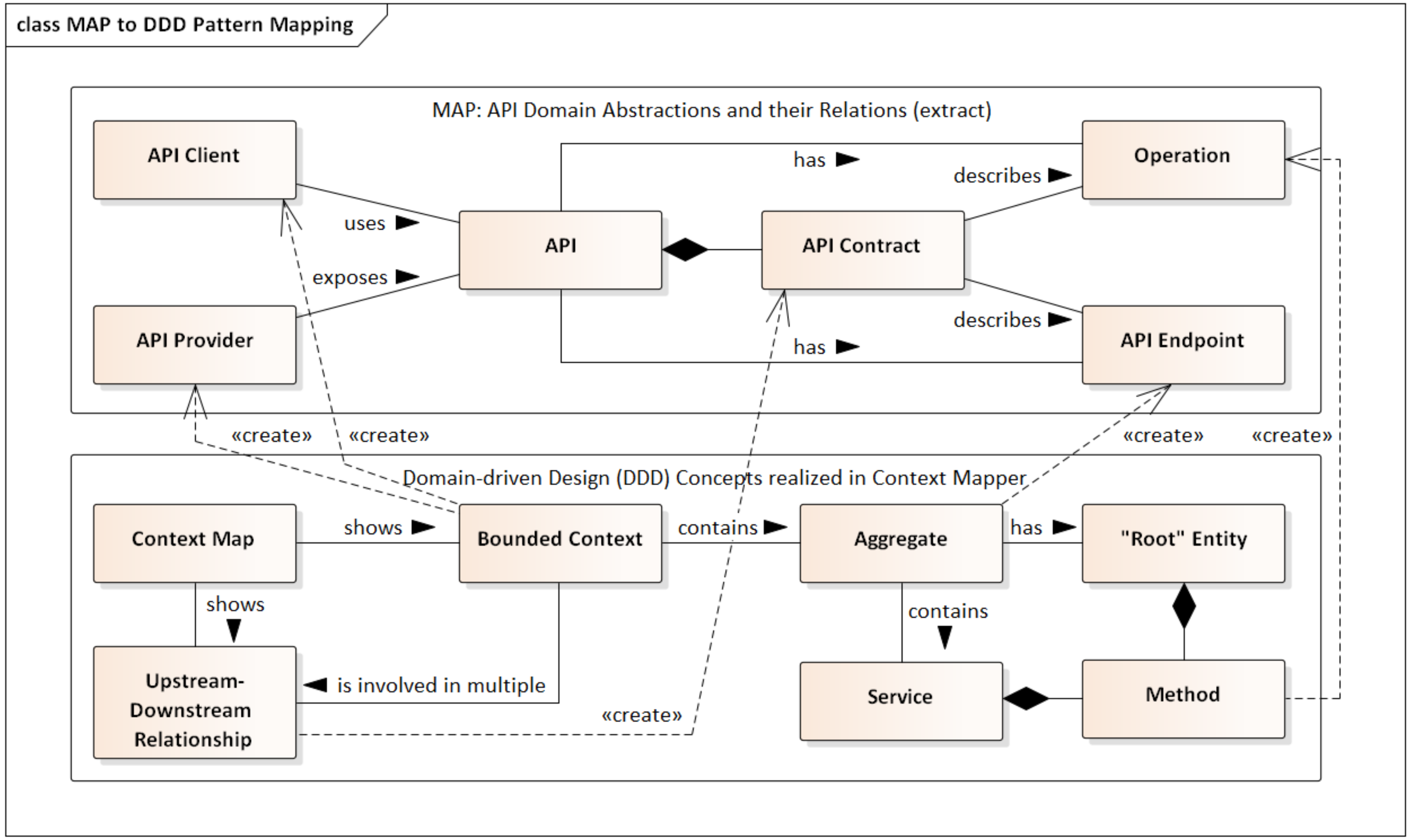
MDSL Service Contract Sample (1)

```
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
}
```

MDSL Service Contract Sample (2)

```
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
```

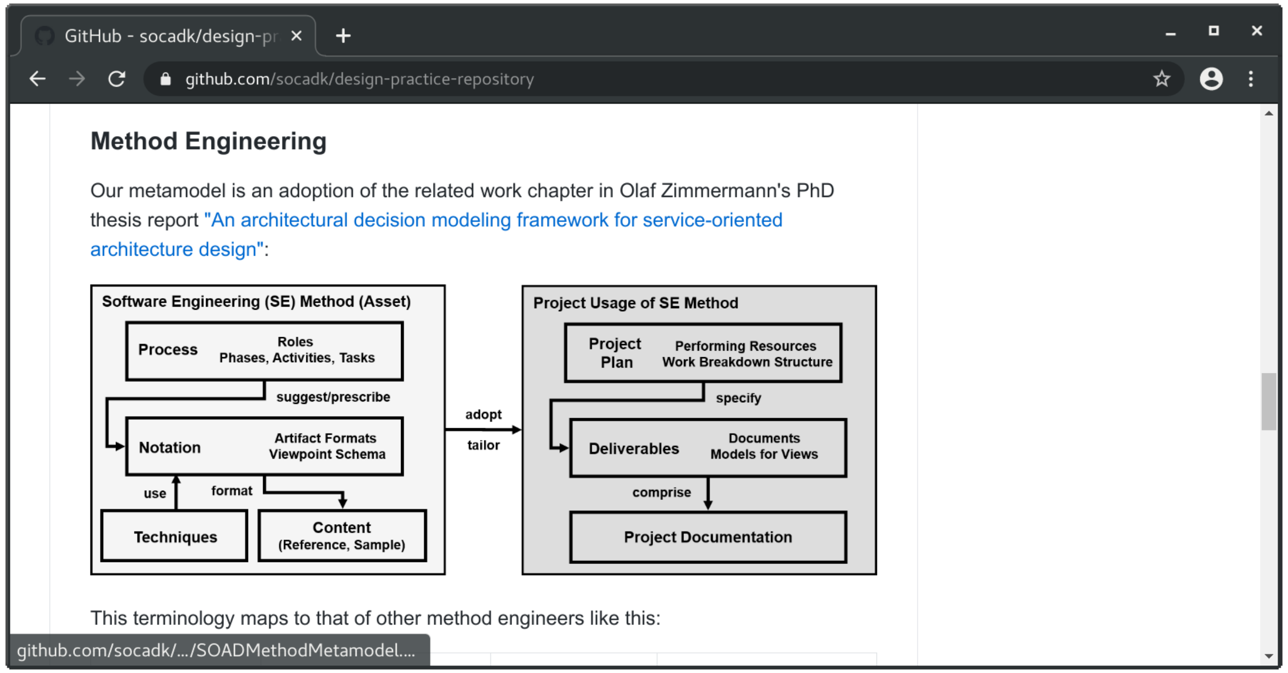
Context Mapper DSL (CML) to MDSL Mapping



© Stefan Kapferer, Olaf Zimmermann, 2020

Software/Service/API Design Practice Repository (DPR)

- DPR (pronounced «deeper») design practice repository by Olaf Zimmermann features Context Mapper:



GitHub - socadk/design-pr

github.com/socadk/design-practice-repository

Method Engineering

Our metamodel is an adoption of the related work chapter in Olaf Zimmermann's PhD thesis report "An architectural decision modeling framework for service-oriented architecture design":

This terminology maps to that of other method engineers like this:

github.com/socadk/.../SOADMethodMetamodel....

github.com/socadk/design-practice-repository

IFS INSTITUTE FOR SOFTWARE