CONTEXT MAPPER: DSL AND TOOLS FOR DOMAIN-DRIVEN SERVICE DESIGN – BOUNDED CONTEXT MODELING AND MICROSERVICE DECOMPOSITION

JUG St. Gallen, Switzerland
September 10, 2019

Stefan Kapferer
Prof. Dr. Olaf Zimmermann (ZIO)
HSR FHO
stefan.kapferer@hsr.ch
ozimmerm@hsr.ch
Abstract

Service-oriented architectures and microservices have gained much attention in recent years; many companies adopt them in order to increase agility, maintainability and scalability of their systems. Decomposing an application into multiple independently deployable, appropriately sized services is challenging. With strategic patterns such as Bounded Context and Context Map, Domain-Driven Design (DDD) can support software architects and domain experts during service decomposition. However, existing architecture description languages, methods and tools do not support strategic DDD sufficiently. As a consequence, different interpretations and opinions regarding pattern applicability can be observed, and it is not always clear how the patterns can be combined. Context modeling is an ad-hoc, error-prone activity.

In this talk we present Context Mapper, an open source project providing a Domain-Specific Language (DSL) for DDD. Aiming for a clear and concise interpretation of the patterns and their combinations, we distilled a meta-model of the DDD patterns from community input. The DSL provides a light syntax to express the patterns and model DDD context maps. An Eclipse editor supports syntax highlighting, code completion, and model validation. Other tools allow designers to refactor and continuously evolve the models and generate lower-level artifacts such as service contracts. DSL and supporting tools promote iterative, incremental modeling and agile practices.
Session Outline

- **Presentation Part 1 (20 mins)**
  - Motivation
  - Brief introduction to Microservice Architectures (MSA)
  - Domain-Driven Design (DDD) and service decomposition
  - Context Mapper overview

- **Context Mapper Demo (20 mins)**

- **Presentation Part 2 (20 mins)**
  - Architectural refactoring
  - Next steps in the BizDevOps tool & practice chain:
    - Microservice Domain-Specific Language (MDSL)
    - Microservice API Patterns (MAP)

- **Q&A (15 mins)**
  - *Input and feedback appreciated – this is ongoing research!*

© Stefan Kapferer, Olaf Zimmermann, 2019.
Session Outline

- **Presentation Part 1 (20 mins)**
  - Motivation
  - Brief introduction to Microservice Architectures (MSA)
  - Domain-Driven Design (DDD) and service decomposition
  - Context Mapper overview

- **Context Mapper Demo (20 mins)**

- **Presentation Part 2 (20 mins)**
  - Architectural refactoring
  - Next steps in the BizDevOps tool & practice chain:
    - Microservice Domain-Specific Language (MDSL)
    - Microservice API Patterns (MAP)

- **Q&A (15 mins)**
  - Input and feedback appreciated – this is ongoing research!
Many design issues, typically recurring

- per system/team, per relationship, per interface

Data duplication and/or on-demand exchange?  Strict/eventual consistency?

Risk Management

Customer Self-Service

Policy Management

Customer Management

Printing

Debt Collection


System or Dev/Ops team

Data and control flow direction? Data formats (norms, transformations)? Frequency of message exchange?

Client influence on API design and stability/evolution (governance)? API contracts and technologies?
A Consolidated Definition of Microservices

- Microservices architectures evolved from previous incarnations of Service-Oriented Architectures (SOAs) to promote agility and elasticity
  - Independently deployable, scalable and changeable services, each having a single responsibility
  - Modeling business capabilities


- Often deployed in lightweight containers
- Encapsulating their own state, and communicating via message-based remote APIs (HTTP, queueing), IDEALLY in a loosely coupled fashion
- Facilitating polyglot programming and persistence
- Leveraging DevOps practices including decentralized continuous delivery and end-to-end monitoring (for business agility and domain observability)
On the Criteria To Be Used in Decomposing Systems into Modules

D.L. Parnas
Carnegie-Mellon University

Open Problem: Service Decomposition

Research and Development Questions

How can systems be decomposed and cut into services (forward engineering)?
How do the applied criteria and heuristics differ from software engineering and software architecture “classics” such as separation of concerns and single responsibility principle?

© Olaf Zimmermann, 2019.
Decomposition Heuristics that do not suffice

- Two-pizza rule (team size)
- Lines of code (in service implementation)
- Size of service implementation in IDE editor
- Simple if-then-else rules of thumb
  - E.g. “If your application needs coarse-grained services, implement a SOA; if you require fine ones, go the microservices way” (I did not make this up!)
- Non-technical traits, including “products not projects”

What is wrong with these “metrics” and “best practice” recommendations?

Context matters, as M. Fowler pointed out at Agile Australia 2018 (or: one size does not fit all)
Domain-Driven Design (DDD) to the Remedy

- Emphasizes need for modeling and communication
  - Ubiquitous language (vocabulary) – the *domain model*

- **Tactic DDD** – “Object-Oriented Analysis and Design (OOAD) done right”
  - Emphasis on business logic in layered architecture
  - Decomposes *Domain Model* pattern from M. Fowler
  - Patterns for common roles, e.g. Entity, Value Object, Repository, Factory, Service; grouped into *Aggregates*

- **Strategic DDD** – “agile Enterprise Architecture and/or Portfolio Management”
  - Models have boundaries
  - Teams, systems and their relations shown in *Context Maps of Bounded Contexts*

---

Books (Selection, Reverse Chronological Order)
- M. Plöed, *Hands-on Domain-driven Design - by example*, Leanpub
- *Domain-Driven Design: The First 15 Years*, Leanpub
- V. Vernon, *DDD Distilled*; a German translation is available: *DDD Kompakt*
Insurance scenario modelled at https://contextmapper.github.io/
M. Ploed is one of the “go-to-guys” here (find him on Speaker Deck)

Applies and extends DDD books by E. Evans and V. Vernon
N. Tune and S. Millett: **Designing Autonomous Teams and Services**
- Describe how to coevolve organizational and technical boundaries to architect autonomous applications and teams based on DDD Bounded Contexts and (micro-)services.

O. Tigges: **How to break down a Domain to Bounded Contexts**
- Presents criteria to be used to identify Bounded Contexts.

R. Steinegger et al.: **Overview of a Domain-Driven Design Approach to Build Microservice-Based Applications**
- Describes a development process to build MSA applications based on the DDD concepts, emphasizing the importance of decomposing a system in several iterations.

A. Brandolini: **Introducing Event Storming**
- Proposes a workshop-based technique to analyze a domain and discover bounded contexts, following events through the system/business process and detecting commands, entities (and more) along the way.
“Implementing DDD” book by V. Vernon (and blog posts, presentations):

- No 1:1 pass-through (interfaces vs. application/domain layer)
- **Bounded Contexts (BCs)** realized by API provider: one service API and IDE project for each team/system BC (a.k.a. microservice)
- **Aggregates** supply API resources (or responsibilities) of service endpoints
- **Services** donate top-level (home) resources in BC endpoint as well
- The Root Entity, the Repository and the Factory in an Aggregate suggest top-level resources; contained entities yield sub-resources
- **Repository** lookups as paginated queries (GET with search parameters)

Additional rules of thumb (from our experience and additional sources):

- Master data and transactional data go to different contexts/aggregates
- Creation requests to Factories become POSTs
- Entity modifiers become PUTs or PATCHes
- Value Objects appear in the custom mime types representing resources
What is Context Mapper?

Context Mapper provides a DSL to create Context Maps based on strategic Domain-driven Design (DDD). DDD with its Bounded Contexts offers an approach for decomposing a domain or system into multiple independently deployable (micro-)services. With our Architectural Refactorings (ARs) we provide transformation tools to refactor and decompose a system in an iterative way. The tool further allows you to generate MDSL (micro-)service contracts providing assistance regarding how your system can be implemented in an (micro-)service-oriented architecture. In addition, PlantUML diagrams can be generated to transform the Context Maps into a graphical representation. With Service Cutter you can generate suggestions for new services and Bounded Contexts.

- **Eclipse plugin, based on:**
  - Xtext, ANTLR
  - Sculptor (tactic DDD DSL)

- **Creator: S. Kapferer**
  - Term projects @ HSR FHO

```plaintext
ContextMap DDD_CargoSample_Map {
    type = SYSTEM_LANDSCAPE
    state = AS_IS

    contains CargoBookingContext
    contains VoyagePlanningContext
    contains LocationContext

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

**SK:** Shared Kernel, **PL:** Published Language
**D:** Downstream, **U:** Upstream
**ACL:** Anti-Corruption Layer, **OHS:** Open Host Service
Session Outline

- **Presentation Part 1 (20 mins)**
- **Context Mapper Demo (20 mins)**
  - Part 1: DSL (Editing, Validations)
  - Part 2: Code Generation (PlantUML, MDSL)
  - Part 3: Architectural Refactorings (ARs)
- **Presentation Part 2 (20 mins)**
  - Architectural refactoring
  - Next steps in the BizDevOps tool & practice chain:
    - Microservice Domain-Specific Language (MDSL)
    - Microservice API Patterns (MAP)
- **Q&A (15 mins)**
Goal: provide clear and concise interpretation of the strategic DDD patterns – and valid combinations of them

Reference: https://contextmapper.github.io/docs/language-model/
Context Mapper: DSL implements Meta-Model and Semantics

- **A Domain-Specific Language (DSL) for DDD:**
  - Formal, machine-readable DDD Context Maps via *editors and validators*
  - Model/code *generators* to convert models into other representations
  - Model transformations for *refactorings* (e.g., “Split Bounded Context”)
Context Mapper: Generators (DDD DSL as Input)

- **PlantUML** generator
  - Generate graphical representations of model

- **Service Cutter** input generator
  - Use structured approach to identify service candidates
  - Term project/bachelor thesis at HSR FHO

- **MDSL** service contract generator
  - Generate technology-agnostic (micro-)service contracts from Bounded Contexts/Aggregates

http://servicecutter.github.io/
Session Outline

- Presentation Part 1 (20 mins)
- Context Mapper Demo (20 mins)
- Presentation Part 2 (20 mins)
  - Architectural refactoring
  - Next steps in the BizDevOps tool/practice chain:
    - Microservice Domain-Specific Language (MDSL)
    - Microservice API Patterns (MAP)
- Q&A (15 mins)
Open Problem: Refactoring to Microservices

Research and Development Questions

How to migrate a modular monolith to a services-based cloud application (a.k.a. cloud migration, brownfield service design)? Can “micro-migration/modernization” steps be called out?

Which techniques and practices do you employ? Are you content with them?
Code Refactoring vs. Architectural Refactoring

- Refactoring are “small behavior-preserving transformations” (M. Fowler 1999)

- Code refactorings such as “extract method”:
  - Operate on Abstract Syntax Tree (AST)
  - Based on compiler theory, so well understood and automation possible (e.g., in Eclipse Java/C++)
  - Catalog and commentary:

- Architectural refactorings are different:
  - Resolve one or more bad architectural smells, have impact on quality attributes
    - Bad architectural smell: suspect that architecture is no longer adequate (“good enough”) under current requirements and constraints (may differ from original ones)
  - Are carriers of reengineering knowledge (patterns?)
  - Can only be partially automated
From Biz and Dev to Ops: Bad Smells and Refactorings

As a first step, we collected Decomposition Criteria (DC):
- From literature and own experience; criteria catalog in Service Cutter

Reference: Service Decomposition as a Series of Architectural Refactorings, Stefan Kapferer, student research project HSR FHO 2019 (thesis PDF)
Architectural Refactorings (ARs) then derived from mined/observed Decomposition Criteria (DC)

- Compiled from literature and own experience
- Decompose (split, extract) and compose (merge) DDD bounded contexts and aggregates.

Selected Decomposition Criteria:

- **DC-1**: Business entities (which belong together)
- **DC-2**: Use Cases
- **DC-3**: Business areas & development teams
- **DC-7**: Likelihood for change (volatility)
- **DC-{8-12}**: Generalized non-functional requirement

Derived Architectural Refactorings:

- **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

Reference: Service Decomposition as a Series of Architectural Refactorings, Stefan Kapferer, student research project HSR FHO 2019 (thesis PDF)
Context

We have decided to go the SOA and/or microservices way. We use DDD for domain modeling and agile practices for requirements elicitation.

Research and Development Problems:

How to identify/specify an adequate number of API endpoints and operations?

How to design message representation structures so that API clients and API providers are loosely coupled and meet their (non-)functional requirements IDEALlly?

Which patterns, principles, and practices do you use (code first, contract first)? Do they work well?
How does this notation compare to Swagger/JSON Schema and WSDL/XSD?

- **Data contract**
  - Compact, technology-neutral
  - Inspired by JSON, regex

- **Endpoints and operations**
  - Elaborate, terminology from MAP domain model
  - Abstraction of REST resource
  - Abstraction of WS-* concepts

- **API client, provider, gateway; governance (SLA, version, …)**

Reference: https://socadk.github.io/MDSL/index
Microservice API Patterns (MAP) Categories

**Identification Patterns:**

- DDD as one practice to find candidate endpoints and operations

**Foundation Patterns**

- What type of (sub-)systems and components are integrated?
- Where should an API be accessible from?
- How should it be documented?

**Structure Patterns**

- What is an adequate number of representation elements for request and response messages?
- How are these elements structured?
- How can they be grouped and annotated with usage information?

**Quality Patterns**

- How can an API provider achieve a certain level of quality of the offered API, while at the same time using its available resources in a cost-effective way?
- How can the quality tradeoffs be communicated and accounted for?

**Responsibility Patterns**

- Which is the architectural role played by each API endpoint and its operations?
- How do these roles and the resulting responsibilities impact (micro-)service size and granularity?

**Evolution Patterns:**

- Recently workshopped (EuroPLoP 2019)

http://microservice-api-patterns.org
Microservices API Patterns (MAP): Patterns by Category

Responsibility
- Endpoint Roles
  - Processing Resource
  - Information Holder Resource

Structure
- Representation Elements
  - Atomic Parameter
  - Atomic Parameter List

Quality
- Quality Management and Governance
  - API Key
  - Rate Limit

Evolution
- Version Identifier
- Semantic Versioning
- Two In Production
- Aggressive Obsolescence
- Limited Lifetime Guarantee
- Experimental Preview
- Eternal Lifetime Guarantee

Reference Management
- Embedded Entity
- Linked Information Holder

Microservice API Patterns (MAP)

© Olaf Zimmermann, 2019.

http://microservice-api-patterns.org
Vision: Agile Tools for BizDevOps (in DDD and MSA Context)

Selected (Agile) Practices *(our focus here)*

- Enterprise Architecture/SAFe
  - Strategic DDD *(System Decomposition)*
- User Story Telling, Mapping, Splitting
- Event Storming, Tactic DDD
- *API Design*: abstract/conceptual, platform-specific (contract first, code first)
- Service Implementation and Integration
- CI/CD Pipelining, Monitoring, …

Tools *(our proposal)*

- Context Mapper
- Business as Usual (BaU): whiteboard/flipchart, C4, drawing tool, issue tracker
- MDSL Editor & Linter *(with MAP Decorators)*
- Open API Specification (f.k.a. Swagger), AsyncAPI, …
- BaU, e.g. Spring Boot, Spring MVC, RabbitMQ, Kafka, etc.
- BaU, e.g. GitLab, Cloud tools, Docker, Kubernetes
DDD Context Map for our Tools

ContextMapper

Support for Strategic and Tactical DDD

BoundedContextAndAggregatesToAPIEndpoints

use

MDSLEditor

Support for Microservice Domain-Specific Language (MDSL)

APIEndpointsToRESTfulHTTPContract

use

APIEndpointsToWebServicePortType

use

ToBeAnnounced

use

SwaggerTools

Editor, test tools, code generators for Open API Specification 3.0

WSDL_XSD_Tools

Web Services and XML Schema Tools

gRPC, GQL Schema Language, AsyncAPI (wrapping Kafka, RabbitMQ, etc.), to be decided

OtherTools

This PlantUML: generated with CONTEXT MAPPER

© Stefan Kapferer, Olaf Zimmermann, 2019.
Summary and Outlook

- Microservices have many predecessors (evolution not revolution)
  - Implementation approach for, and sub-style of, SOA (7 tenets)
    - More emphasis on autonomy and decentralization (of decisions, of data ownership), less vendor-driven
    - Automation advances and novel target environments
  - Context Mapper (open source/term thesis projects @ HSR)
    - DSL for modeling strategic DDD Context Maps
    - Tool support to evolve models iteratively (ARs)
    - PlantUML, Service Cutter, and MDSL generation
  - Microservice Domain-Specific Language (MDSL) for service contracts
  - Microservice API Patterns (MAP) language & website
    - 20+ patterns, sample implementation, tutorial

Thank you very much! Let’s move on to Q&A and discussion…
Feedback appreciated…

- Did we catch the essence of strategic DDD (context mapping)?
- Is the DDD DSL expressive enough, but also easy to understand?
- Is anything missing in terms of functionality?
  - Which decomposition criteria do you use when cutting/carving services?
  - Which architectural refactorings would you like to see in future versions?
  - Which model transformations and code generators would be valuable?
    - E.g. should we look into reverse transformations (from code to DSL)?

- Can you envision to apply Context Mapper, MDSL, MAP in practice?
  - Do tools and notations have the potential to improve productivity & quality?
  - What are critical success factors for adoption (NFRs)?
- Which API design patterns and contract language features are missing?