FROM DOMAIN-DRIVEN DESIGN TO MICROSERVICE APIS OF QUALITY AND STYLE: CONTEXT, CONTRACTS, COMPONENTS

GI-Arbeitskreis Microservices und DevOps

Berlin, March 9, 2020

Prof. Dr. Olaf Zimmermann (ZIO)
Certified Distinguished (Chief/Lead) IT Architect
Institute für Software, HSR FHO
ozimmerm@hsr.ch
You have been tasked to develop a RESTful HTTP API for a master data management system that stores customer records and allows sales staff to report and analyze customer behavior. The system is implemented in Java and Spring. A backend B2B channel uses message queues (RabbitMQ).

What do you do?

a) I hand over to my software engineers and students because all I manage to do these days is attend meetings and write funding proposals.

b) I annotate the existing Java interfaces with @POST and @GET, as defined in Spring MVC, JAX-RS etc. and let libraries and frameworks finish the job.

c) I install an API gateway product in Kubernetes and hire a sys admin, done.

d) I design a service layer (Remote Facade with Data Transfer Objects) and publish an Open API Specification (f.k.a. Swagger) contract. I worry about message sizes, transaction boundaries, error handling and coupling criteria while implementing the contract. To resolve such issues, I create my own novel solutions. Writing infrastructure code and test cases is fun after all!

e) _____________________________
1. **Context matters**
   - One size does not fit all (top-level design heuristic: "it depends")
   - Strategic and tactic Domain-Driven Design (DDD)
   - Context Mapper DSL and tools

2. **Contracts rule**
   - Unified interfaces are great, but not enough
   - More SOA and microservices myth busting
   - Microservice Domain-Specific Language (MDSL)

3. **Components contain (cost and risk)**
   - Towards a context-driven, contract-first service identification method
   - Microservice API Patterns (MAP) to structure the solution space
   - (time permitting) Industry trends and resulting research questions
     - Microfrontends, containerization, cloud-native 12-factor applications
Multi-Channel Order Management SOA in the Telecommunications Industry (in production since Q1/2005) [OOPSLA 2005]

- **Functional domain**
  - Order entry management
  - Two business processes: new customer, relocation
  - Main SOA drivers: deeper automation grade, share services between domains

- **Service design**
  - Top-down from requirement and bottom-up from existing wholesaler systems
  - Recurring architectural decisions:
    - Protocol choices
    - Transactionality
    - Security policies
    - Interface granularity

Reference: IBM, ECOWS 2007
Agile practices

Professional services methods

Context Matters

IT Solution Requirements Analysis
- Use Case Model
- System Context
- Non-functional requirements

IT Solution Design
- Reference Architectures
- Architecture Overview Diagram
- Current IT Environment
- Operational Model
- Deployment Units

Component Model

Most work products

Business Requirements Specification

Automation

IT Solution Design

Development

Requirements on IT

Experience reports

Context Is King
What's Your Software’s Operating Range?
Francisco Torres

Talking with users might change how you see the context of your software project, often in unexpected ways. Drawing from his experience on spacecraft operations software projects, Francisco Torres shares stories on how listening to users taught him to stop making assumptions and helped him define his software’s operating range, the set of quality properties in which a software system can successfully run. —Claire Pasin and Olaf Zimmermann

Many design issues, typically recurring
- per system/team, per relationship, per interface

- System decomposition?
- Data duplication and/or on-demand exchange? Strict/eventual consistency?
- 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?
- Subdomain, System, Team

Design issue (decision required)
Domain-Driven Design (DDD) Overview

- 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, example model from [https://contextmapper.org/](https://contextmapper.org/)

D: Downstream, U: Upstream; ACL: Anti-Corruption Layer, OHS: Open Host Service

© Stefan Kapferer, Olaf Zimmermann, 2020.
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 and Master thesis @ HSR FHO
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”)

Plugin update site: https://dl.bintray.com/contextmapper/context-mapping-dsl/updates/
**Context Mapper: Domain-Specific Language**

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

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

- **Bounded Contexts** (systems or teams)
- **DDD relationship patterns** (role of endpoint)
  - Influence/data flow direction: ->, <-> (upstream-downstream or symmetric)

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

- **Context Mapper architecture**
  - Modelled with Context Mapper DSL
  - UML generated

The reverse engineering and discovery component can generate CML Context Maps from existing source code. This allows to reverse engineer the architecture model in projects with existing monoliths or microservices.

Provides the Context Mapper DSL (CML) modeling language to express architectures on the basis of Strategic Domain-driven Design (DDD) patterns.

The Service Cutter integration into Context Mapper allows to analyze the Context Map with respect to coupling criteria and supports to suggest improved Context Maps. The Service Cutter library exposes an API (Open Host Service and Published Language) used by Context Mapper to generate the new decompositions.

The generators allow to generate other representations of the architecture derived by a given CML Context Map.
1. **Context matters**
   - One size does not fit all (top-level design heuristic: "it depends")
   - Strategic and tactic Domain-Driven Design (DDD)
   - Context Mapper DSL and tools

2. **Contracts rule**
   - *Unified interfaces are great, but not enough*
   - *More SOA and microservices myth busting*
   - *Microservice Domain-Specific Language (MDSL)*

3. **Components contain (cost and risk)**
   - Towards a context-driven, contract-first service identification method
   - Microservice API Patterns (MAP) to structure the solution space
   - (time permitting) Industry trends and resulting research questions
     - Microfrontends, containerization, cloud-native 12-factor applications
Our focus: Microservices!

Middleware less popular, often custom build (term also used in deployment and clustering context)

Middleware

(data) contracts

Conversaion

Optional (then and now)

Service Registry

Service Endpoint

Application

Application Microservices!
Myth: SOA and microservices solve different problems, not comparable

- Application boundaries blurred in the Web age
- See [Microservices Tenets](#) article, see OOPSLA practitioner reports

Myth: Traditional SOA is "heavyweight" and requires centralization and enterprise-wide data normalization in an Enterprise Service Bus (ESB)

- What is heavyweight (definition)? Resource usage? Maintenance?
  - SOAP also uses HTTP by default; JSON not much lighter than "nice" XML
  - Have a look at the dependencies of services meshes (example: Istio)
- Most practices recommended today already appeared in the (good) SOA tutorials in the 2000s
  - e.g. no canonical data model, no single point of failure, no business logic in ESB
  - Yes, poor SIA implementations did occur (but that also holds for microservices)

Myth: SOA and XML-based "Web" services are coupled with each other

- Actually, they are less related that REST and HTTP are
  - Although REST claims to be an architectural style (only implemented once)
**Myth: REST is a protocol**
- It is an architectural style defined by abstract constraints
- So asking for a “REST API” is like asking for “Gothic window” (material?)

**Myth: SOAP is a protocol**
- It is a message exchange format, HTTP typically used for message transfer
  - Other protocols (theoretically) possible

**Myth: REST and SOAP can be compared**
- Can the Gothic style and concrete building materials/norms be compared?

**Myth: Thought leaders are objective and independent**
- There is an “industrial NN complex” (NN = Agile, REST, …)
  - To paraphrase M. Fowler at Agile Australia
- Book authors and consultants do have commercial agendas (lie vendors)
  - And should not reference their own papers/books only (SOA Design Patterns?)
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), IDEALIly 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)

Mythbusting (3/4): Microservices (since 2014)

- **Myth:** Self-Contained Systems are new, different form MS(A) and “monolith”
  - Evidence: e.g., S. Brown: [Modular Monolith](#)

- **Myth:** Distributed service mesh sidecars are easier to create, configure, manage than SOA-days ESBs
  - Evidence: notion of federated ESBs, EIP pattern mapping
  - Open source lock in replacing vendor lock in

- **Myth:** RESTful HTTP is the only protocol that is required and permitted
  - MOM and even RPC have their place
    - Evidence: Google gRPC, S. Newman first book on Microservices

- **Myth:** Unified interface is sufficient as contract
  - The success of Swagger/Open API Specification suggests that more elaborate [API Descriptions](#) are required
    - Data contract, pre- and postconditions, error handling, …
OpenAPI Specification (OAS): An Interface Definition Language (IDL)

- **Wikipedia** lists (only) 23 IDLs
  - OAS is one of them
  - Bound to HTTP

**TOOLS**

**SWAGGER UI**
Use a Swagger specification to drive your API documentation. [Demo](#) and [Download](#).

**SWAGGER EDITOR**
An editor for designing Swagger specifications from scratch, using a simple YAML structure. [Demo](#) and [Source](#).

**SDK GENERATORS**
Turn an API spec into client SDKs or server-side code with [Swagger Codegen](#).
How does this notation compare to Swagger/JSON Schema and WSDL/XSD?
Myth: Services always must be small/fine-grained
- How to measure? How to observe?
- What about dependencies? They increase.

Myth: A business capability has to be a function
- And Entity Service (always) are an anti pattern
- Archive? Logbook? File share?

Myth: The DDD patterns fully solve the decomposition problem
- Process required (and related knowledge/patterns), see here and here
- Subdomains and Aggregates and Bounded Contexts (BCs) are as hard to find as services, so "turn BC into microservice" only delegates the problem

Myth: “Hello World” implementations are suited to demonstrate the value and price of microservices
- Domain model needs to have a certain size and complexity e.g., to see ramifications of replication, eventual consistency (see Lakeside Mutual)
1. **Context matters**
   - One size does not fit all (top-level design heuristic: "it depends")
   - Strategic and tactic Domain-Driven Design (DDD)
   - Context Mapper DSL and tools

2. **Contracts rule**
   - Unified interfaces are great, but not enough
   - More SOA and microservices myth busting
   - Microservice Domain-Specific Language (MDSL)

3. **Components contain (cost and risk)**
   - *Towards a context-driven, contract-first service identification method*
   - *Microservice API Patterns (MAP) to structure the solution space*
   - *(time permitting) Industry trends and resulting research questions*
     - Microfrontends, containerization, cloud-native 12-factor applications
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

Reference: JUGS presentation, Bern/CH, Jan 9, 2020
Input: analysis model, NFRs

Output: API contracts (here: MDSL)

Tasks: Select pattern, refine design, refactor

Step 0: Baseline (Starting Point)

Challenges (Tasks)

Step 1: Identification and Foundation Patterns

Step 2: Roles and Responsibilities (R)

Step 3: Basic and Composite Structures (S)

Step 4: Quality Enhancements (Q)

Step 5: Evolution Patterns

API description LakesideMutual

data type StatusInformation (V<bool>,L)

event type CustomerManagement serves as INFORMATION HOLDER RESOURCE

exposes

operation findCustomer with responsibility RETRIEVAL_OPERATION

expecting payload V<void> // no payload
delivering payload "customerIDList":ID

operation readCustomer with responsibility RETRIEVAL_OPERATION

expecting payload "customerID":ID
delivering payload "customerDTO":V?

operation updateCustomer with responsibility EVENT_PROCESSOR

expecting payload "customerDTO":V?
delivering payload StatusInformation
Calls to Service Operations are EIP-style Messages

Sample request message (note: PUTs and POSTs would look different)

Response message structure

{[…]} -- some JSON (or other MIME type)

Introducing… Microservice API Patterns (MAP)

- **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
API Description Pattern

- Which knowledge should be shared between an API provider and its clients?
- How should this knowledge be documented?

https://microservice-api-patterns.org/patterns/foundation/APIDescription.html
MAP Example: Pagination (1/2)

- **Context**
  - An API endpoint and its calls have been identified and specified.

- **Problem**
  - *How can an API provider optimize a response to an API client that should deliver large amounts of data with the same structure?*

- **Forces**
  - Data set size and data access profile (user needs), especially number of data records required to be available to a consumer
  - Variability of data (are all result elements identically structured? how often do data definitions change?)
  - Memory available for a request (both on provider and on consumer side)
  - Network capabilities (server topology, intermediaries)
  - Security and robustness/reliability concerns
**Solution**

- Divide large response data sets into manageable and easy-to-transmit chunks.
- Send only partial results in the first response message and inform the consumer how additional results can be obtained/retrieved incrementally.
- Process some or all partial responses on the consumer side iteratively as needed; agree on a request correlation and intermediate/partial results termination policy on consumer and provider side.

**Variants**

- Cursor-based vs. offset-based

**Consequences**

- E.g. state management required

**Know Uses**

- Public APIs of social networks
Mini-Exercise: Can MAP serve as a map/guide to API design?

Let’s have a look at the language organization and selected patterns…

- http://microservice-api-patterns.org
  - Website public since 2/2019; experimental preview site available to beta testers
  - Sample patterns (suggestions):
    - Request Bundle, Embedded Entity, Wish List, API Key, Two in Production

Microservice API Patterns

Microservice API Patterns (MAP) take a broad view on API design and evolution, primarily focusing on message representations – the payloads exchanged when APIs are called. These payloads have structure. The representation elements in the payloads differ in their meanings as API endpoints and their operations have different architectural responsibilities. Furthermore, the chosen representation structures strongly influence the design time and runtime qualities of an API.

Our Microservice API Patterns capture proven solutions to design problems commonly encountered when specifying and implementing message-based APIs in terms of their structure, responsibilities, and quality.
Key Messages of this Talk

- It is the API contract (and its implementations) that make or break projects – not (or not only) middleware and tools
  - Frameworks and infrastructures come and go, APIs stay

- **Microservice API Patterns (MAP) language/components**
  - [Public MAP website](#) now available in Version 1.2.1
  - 20+ patterns, sample implementation in public repo, supporting tools

- **Microservices Domain-Specific Language (MDSL)**
  - Uses MAPs in service contracts (as decorators)
  - Can be generated from DDD bounded contexts

- **Context Mapper** tool supporting strategic Domain-Driven Design (DDD) and architectural refactoring
  - Other tools emerging

- **Research areas (ZIO):**
  - Service modeling, identification, decomposition, refactoring
You had been tasked to develop a RESTful HTTP API for a master data management system that stores customer records and allows sales staff to analyze customer behavior. The system is implemented in Java and Spring. A backend B2B channel uses message queues (RabbitMQ).

What do you do (now)?

a) I hand over to my software engineers and students because all I manage to do these days is attend meetings and write funding proposals.

b) I annotate the existing Java interfaces with @POST and @GET, as defined in Spring MVC or JAX-RS etc. and let libraries and frameworks finish the job.

c) I install an API gateway product in Kubernetes and hire a sys admin, done.

d) I design a service layer (Remote Facade with Data Transfer Objects) and publish an Open API Specification (f.k.a. Swagger) contract. I worry about message sizes, transaction boundaries, error handling and coupling criteria while implementing the contract. To resolve such issues, I create my own novel solutions. Writing infrastructure code and test cases is fun after all!

e) I leverage Context Mapper, MDSL, MAP for API design and evolution 😊
FROM DOMAIN-DRIVEN DESIGN TO MICROSERVICE APIS OF QUALITY AND STYLE – BACKUP CHARTS

GI-Arbeitskreis Microservices und DevOps
Berlin, March 9, 2020

Prof. Dr. Olaf Zimmermann (ZIO)
Certified Distinguished (Chief/Lead) IT Architect
Institute für Software, HSR FHO
ozimmerm@hsr.ch
DDD Applied to (Micro-)Service Design ctd., Source:

- **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
SOA 1.0: WSDL (XML Language for Service Descriptions)

Web Services Description Language (WSDL)

- **WSDL document elements**
  - Type definitions and imports
  - Interface description (Port Type, Operations, Messages)
  - Extensible binding section
  - Implementation description (Ports)
- **WSDL SOAP binding**
  - Defines header and fault support
  - Extensibility element for addressing
- **HTTP binding also defined**
XML elements for operation parameters

a.k.a. message parts

XML complex types for nontrivial DTOs

XML basic types for scalar DTOs
A Software Architect’s Dilemma….

How do I split my system into services?

Step 1: Analyze System
- Entity-relationship model
- Use cases
- System characterizations
- Aggregates (DDD)

Coupling information is extracted from these artifacts.

Step 2: Calculate Coupling
- Data fields, operations and artifacts are nodes.
- Edges are coupled data fields.
- Scoring system calculates edge weights.
- Two different graph clustering algorithms calculate candidate service cuts (=clusters).

Step 3: Visualize Service Cuts
- Priorities are used to reflect the context.
- Published Language (DDD) and use case responsibilities are shown.

Technologies:
Java, Maven, Spring (Core, Boot, Data, Security, MVC), Hibernate, Jersey, JHipster, AngularJS, Bootstrap
https://github.com/ServiceCutter
### Coupling Criteria (CC) in “Service Cutter” (Ref.: ESOCC 2016)

**Cohesiveness**
- Semantic Proximity
- Identity & Lifecycle Commonality
- Security Contextuality
- Shared Owner
- Latency

**Compatibility**
- Structural Volatility
- Consistency Criticality
- Storage Similarity
- Content Volatility
- Availability Criticality
- Security Criticality

**Constraints**
- Consistency Constraint
- Security Constraint
- Predefined Service Constraint

**Communication**
- Mutability
- Network Traffic Suitability


- **E.g. Semantic Proximity can be observed if:**
  - Service candidates are accessed within same use case (read/write)
  - Service candidates are associated in OOAD domain model

- **Coupling impact (note that coupling is a relation not a property):**
  - Change management (e.g., interface contract, DDLs)
  - Creation and retirement of instances (service instance lifecycle)
Open Research Problem: Refactoring to Microservices

Research 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?