DIMENSIONS OF SUCCESSFUL WEB API DESIGN AND EVOLUTION: CONTEXT, CONTRACTS, COMPONENTS

20th International Conference on Web Engineering (ICWE)

Online, June 11, 2020

Prof. Dr. Olaf Zimmermann (ZIO)
Distinguished (Chief/Lead) IT Architect
Institute for Software, HSR FHO OST
ozimmerm@hsr.ch

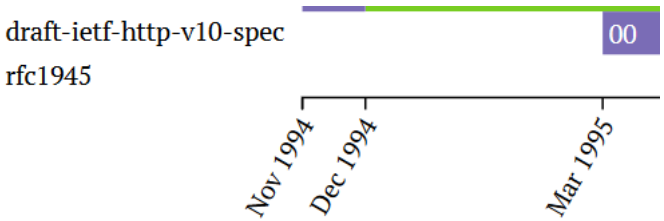
Blog: https://ozimmer.ch/blog

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz

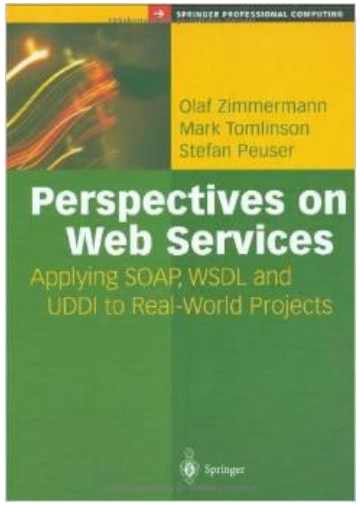
Happy Birthday ICWE (and HTTP 1.0 turns 25!)

- **At the time of the first conference edition:**
  - WWW 1.0: "e-business" CGI, servlets etc.
  - Amazon was an online book store
  - Shop software was a market, some POX APIs



- **Until 2010:** https://www.visualcapitalist.com/30-year-timeline-world-wide-web/
  - Web 2.0: Wikis, blogs, mashups (what happened to them?)
  - Twitter had started, Amazon now selling many things
  - Cloud? Not sure yet. Web APIs? Sure! RESTful? A few (if any).
- **Since 2010:**
  - Cloud and containers all over the place; AWS leading the cloud market
  - Many social networks, games; public APIs a must, hypermedia in the media
- **This year's conference themes:**
  - Performance, Testing, Machine Learning, Open Data, Sentiment Analysis, Emotion Detection, Location-Awareness, and more

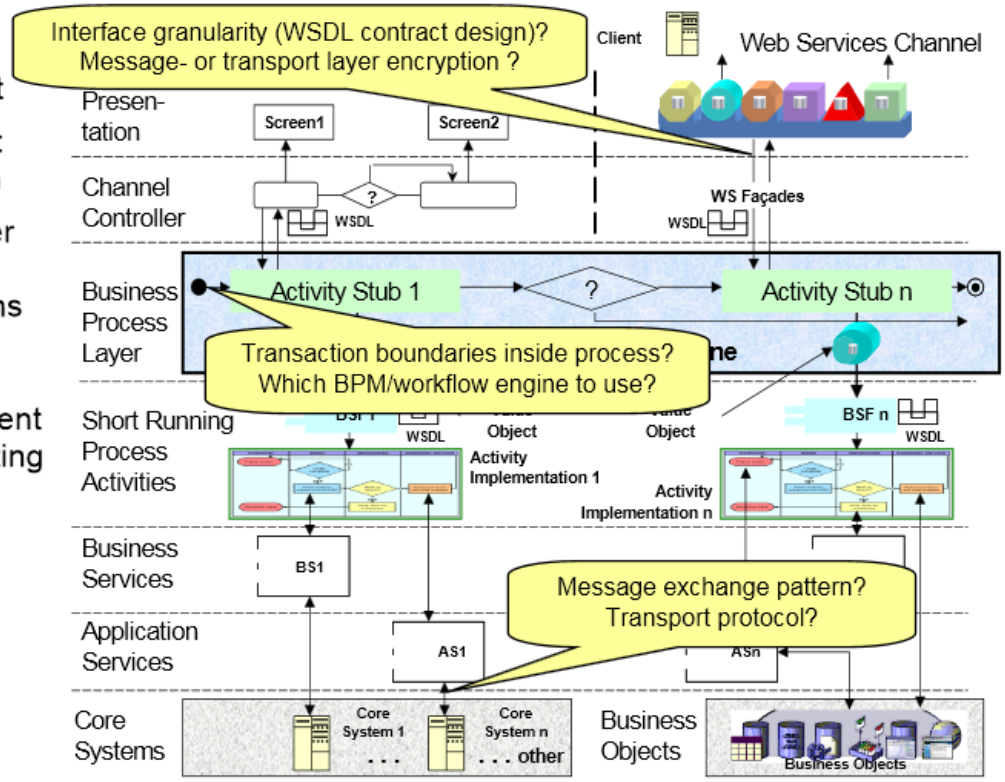
A SOA 1.0: Order Management (Telecommunications Domain)



Multi-Channel Order Management SOA in the Telecommunications Industry (in production since Q1/2005) [OOPSLA 2005]

Reference: IBM, ECOWS 2007

- **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



Agenda (and Take Away Messages)

1. ***Context* matters**
   - One size does not fit all
   - Strategic and tactic Domain-Driven Design (DDD) to the remedy
   - *Context Mapper DDD DSL and supporting tools available*
2. ***Contracts* rule**
   - A unified interfaces is great for browsers, but not enough for application integration
   - Protocol choice depends (on context, on requirements)
   - *Microservice Domain-Specific Language (MDSL) and tools released*
3. ***Components* contain (cost and risk)**
   - Web API designs do not have to be reinvented on every project
   - Much focus on infrastructure design so far, what about API endpoints, service contracts, message representation elements?
   - *Microservice API Patterns (MAP) structure the solution space*

Part 1: Context

Debunking Myths 1: Technology First, One Size Fits All

Context

Digitalization: Development of open source software or commercial project.

Myths

a) Information Technology (IT) must be at the heart of innovation;
users can be educated about their wants and needs later.
b) A single software design can serve all user contexts and requirements.

Rectification

Observe/listen to users to carve out the "business" value of the new software.
Establish an ubiquitous, application domain-driven language.
Apply proven engineering methods and empirical validation techniques.
No cargo cults either please… *context matters!*

What is Context? It depends.

- **Latin roots: *contexere*: "weave together", *contextus:* "tight coupling" (!)**
- **"A frame that surrounds […] event and provides resources for its appropriate interpretation"**
- **Dimensions in Software and Web Engineering:**
  - Location (for instance, of a mobile phone or robot)
  - P. Kruchten's project octopus and decision making context (incl. ethics, compliance)
  - Trigger and precondition for patterns usage
  - User experience ("operating range")
  - Developer experience (papers by G. Murphy et al.)
  - System context in systems of systems (interfaces)
  - Modeling context
    - Conway's Law vs. The Matrix": universal data model


INSIGHTS

Editor: Cesare Pautasso
University of Lugano
c.pautasso@ieee.org

Editor: Olaf Zimmerman
University of Applied Sciences of Eastern Switzerland, Rapperswil
ozimmerm@hsr.ch

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. —*Cesare Pautasso and Olaf Zimmerman*

THEY SAY THAT experience is what you get when you were expecting something else. This happened to me some years ago while I was studying human–machine interface concepts for a software system for spacecraft operations. One initial project activity involved analyzing the existing user interface to identify improvement areas and new concepts to prototype and explore. To that end, my colleagues and I designed a questionnaire on usability aspects, which a sample population of the users answered. Some of the answers were counterintuitive, to say the least. In time, and after giving it some thought, it suddenly dawned on me that I had lost sight of one key element: context. With context in mind, everything made more sense.

This lesson about context applies to not only GUI-related topics but also many other software-engineering areas, such as processes and tools. They all have a nominal operating range, so to speak. More on this later, but first I'll walk you through some of the GUI attributes our study explored. In each case, I'll brief you on the feedback we received through the questionnaire and explain why, despite not matching my expectations, the users' point of view ultimately made sense.

When the Stakes Are Too High

The questionnaire asked the respondents to assess how intuitive and user friendly the various displays were. We got some answers along the lines that some displays were "not user friendly," "far from intuitive," and "clumsy in the extreme." However, one theme also recurred in many of the answers. Learnability and intuitiveness were less of an issue because the users had been formally trained to operate the system, including rehearsals and simulations.

One respondent stated that it wasn't possible to use the system simply intuitively, some a priori knowledge was required. Another one even proposed to redesign some of the displays, trading intuitiveness for efficient screen real es-

0740-7459/15/$31.00 © 2015 IEEE

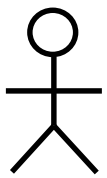
SEPTEMBER/OCTOBER 2015 | IEEE SOFTWARE 9

PL: Published Language **D:** Downstream, **U:** Upstream **ACL:** Anti-Corruption Layer

Methods and Practices – Old and New (?)

Example of an Agile Practice: User Stories

https://www.agilealliance.org/glossary/user-story-template/



Business Analyst

***As a* business analyst (specializing on a particular business or technical domain),**

***I would like to* describe the problem domain and its subdomains in a natural, yet precise and ubiquitous language (i.e., domain concepts, their properties and relations)**

***so that* project sponsor, team and other stakeholders can develop and share a common understanding about these concepts and their intricacies in the given domain – in line with Agile values and principles.**

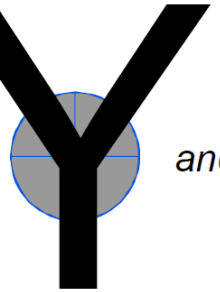
```
UserStory PaperArchiving {
   As a "Researcher"
   I want to create a "PaperItem" with its "title", "authors", "venue" in a "PaperCollection"
   so that "other researchers can find and cite the referenced paper easily, and my h-index goes up."
}
```

Key Tools in my ITA Toolbox: Lean/Light Templates, Checklists

- **Towards an "open and lean architecting framework" (ECSA SAGRA 16):**
  - Agile practices such as *user stories* and *definition of done* (done-deciding)
  - SMART *nonfunctional requirements* a.k.a. desired qualities:
    - Specific, Measurable, Agreed upon, Realistic, Time bound
  - CRC cards to specify components and their collaborations
  - Y-statements to capture *architectural decisions* and their rationale:

In the context of <use case uc and/or component co>,

… facing <non-functional concern c>,



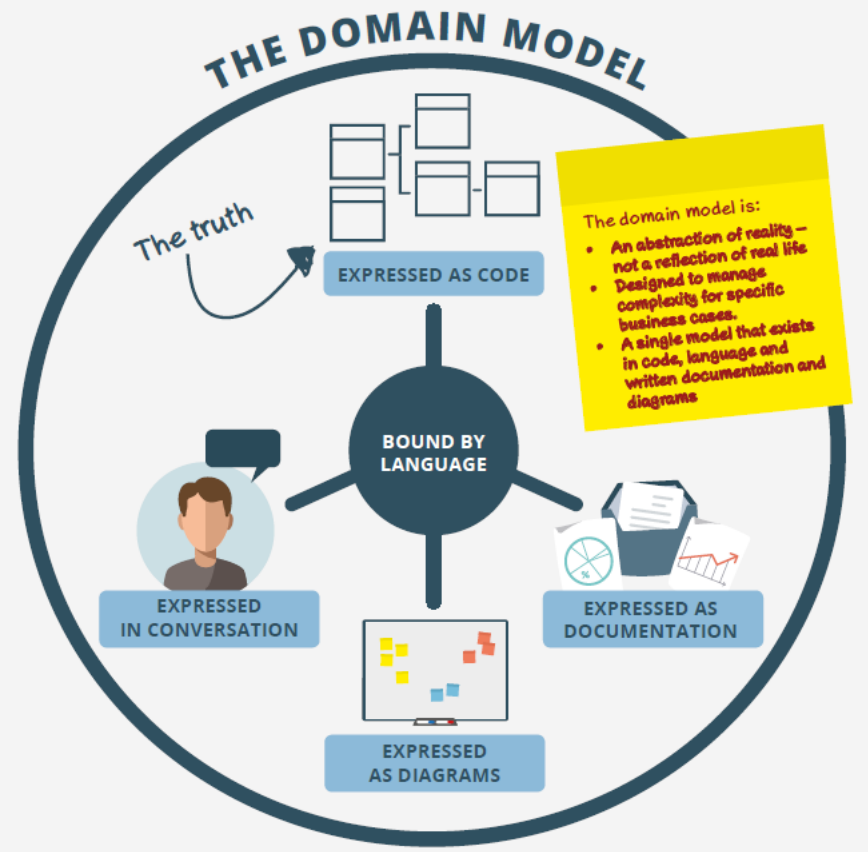
.. we decided for <option o1>

and neglected <options o2 to on>,

… to achieve <quality q>,

… accepting downside <consequence c>.

Domain-Driven Design (DDD): Domain Model in the Center



- **Strategic: Bounded Contexts**
  - Model boundaries and their relations (Web APIs)
  - Abstraction of team or (sub-) system
    - E.g. payment, product catalog, shipping
- **Tactic DDD: Aggregates**
  - Object clusters as storage units, consistency invariants
    - E.g. order and its items
- **Event storming to find the domain model elements**
  - Huge momentum right now
  - Similar techniques have been around since 1990s

Figure reference: https://leanpub.com/theanatomyofdomain-drivendesign

A DSL and Tools for DDD: Context Mapper

CONTEXT MAPPER

- **Eclipse plugin (v5.12) , VSC Extension (new!) or Web IDE/GitPod (new!)**

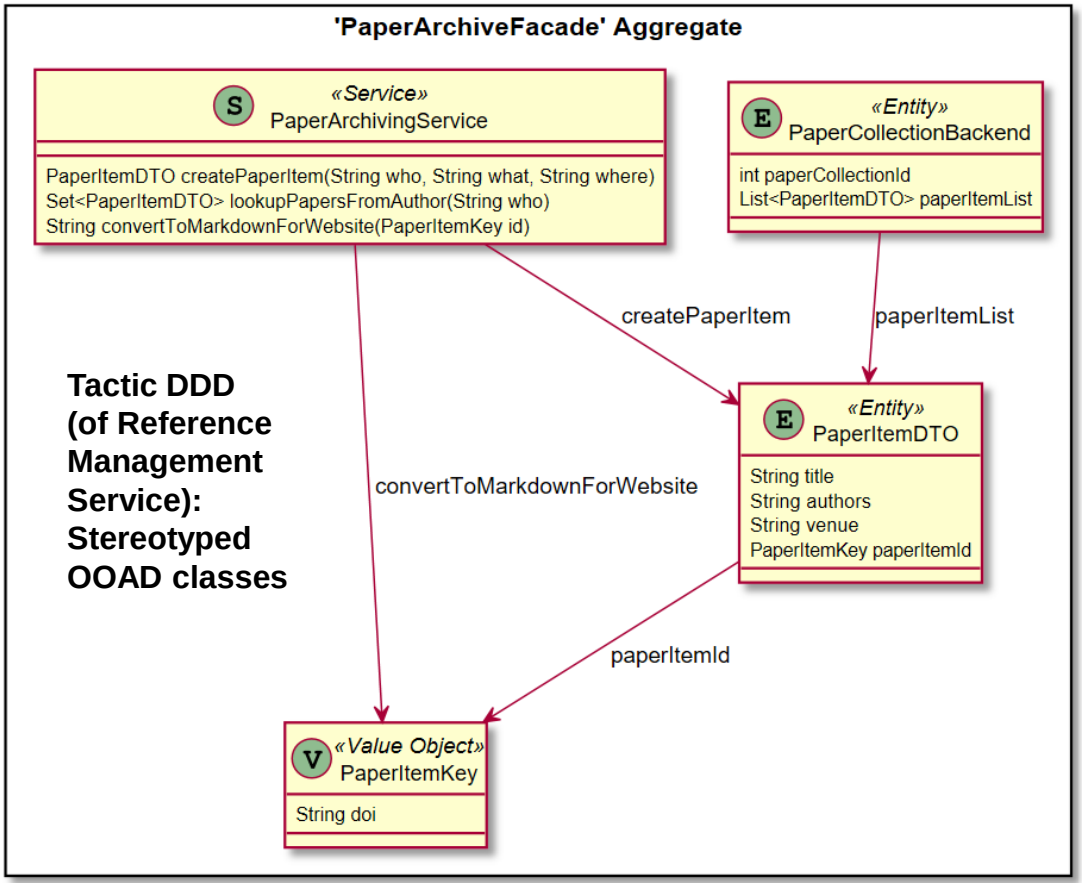
Strategic DDD: Each bounded context has *its own* domain model (aggregates etc.)

```
ContextMap {
  contains ReferenceManagementFrontend
  contains ReferenceManagementService

  ReferenceManagementService [PL]
    -> [CF] ReferenceManagementFrontend {
      implementationTechnology "HTTP"
      exposedAggregates PaperArchiveFacade
  }
}
```

Relations: PL: Published Language, CF: Conformist (and more)

Tactic DDD (of Reference Management Service): Stereotyped OOAD classes

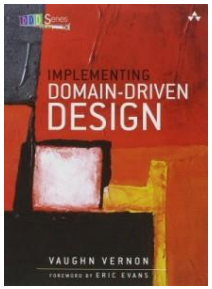


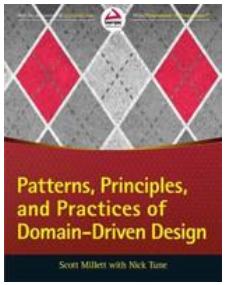
Rapid OOAD/DDD with Context Mapper (and MDSL)

Step-by-step instructions: https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html

User Story

- in Context Mapper (manual)
- Use cases also possible



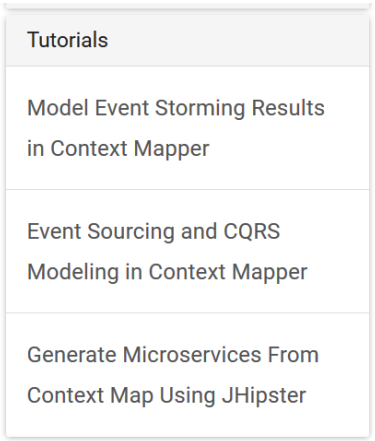


generate (initially) → **Analysis Model**

- Subdomains, Entities

generate (initially) → **Design Model**

- Feature BCs with Aggregates
- System BCs
- JHipster JDL



generate (fully) → **Service Contract**

- MDSL
- Open API

generate (stubs) → **Clients, Servers**

- JHipster
- Spring Boot

Part 2: Contracts (in Context of Principles, Styles, Protocols)

Debunking Myths 2: HTTP and Microservices Confirmation Biases

Context

Web API Design and Evolution. Container as a Service, Serverless. Services.

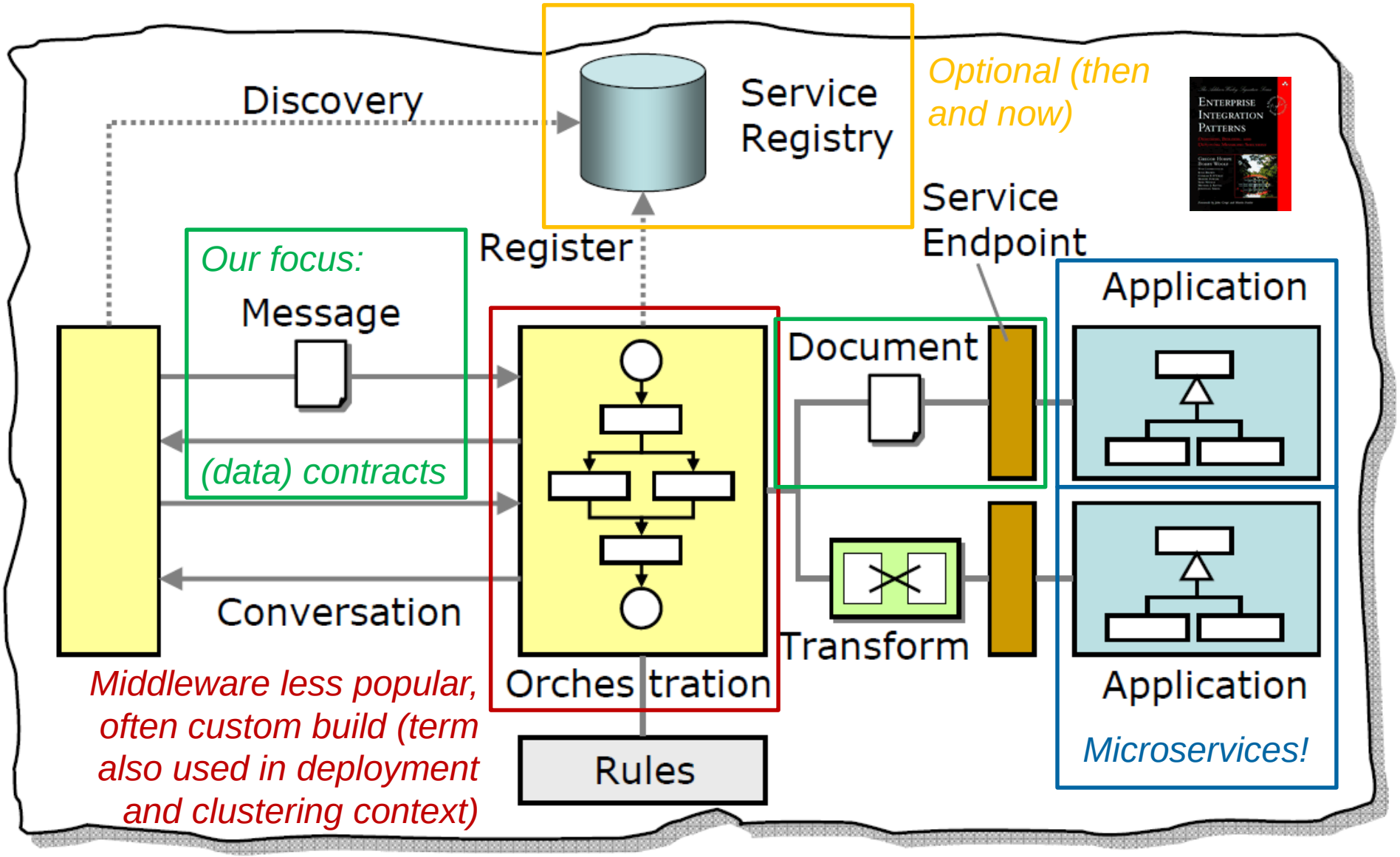
Myths

a) SOA and microservices are competing styles. Micro means really small.
b) Any (micro-)service must expose an HTTP resource API.
c) The REST principle of a unified interface (GET, POST, PUT, etc.) is suited and sufficient to express rich domain model semantics.

Rectification: *Contracts rule (polyglot integration, data type precision)!*

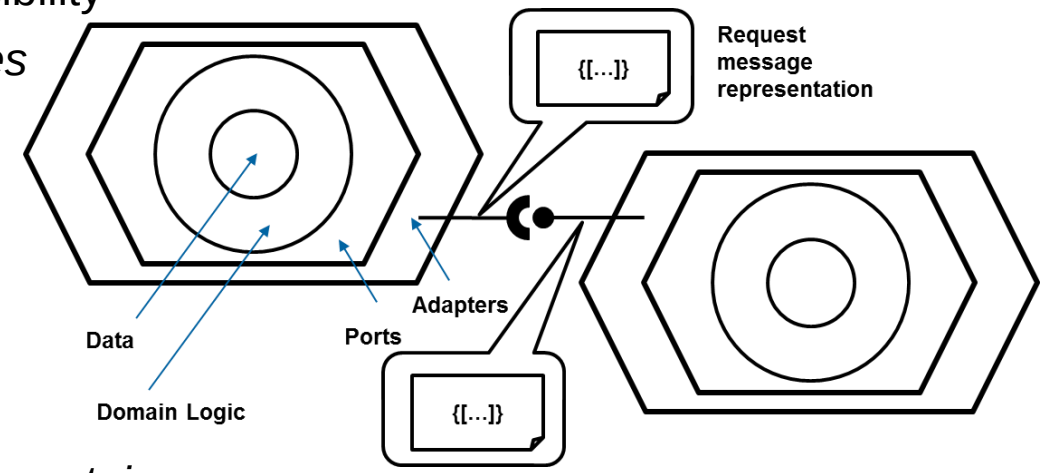
a) Microservices are an implementation approach to SOA. Service size varies.
b) IDEAL, FROSTT, CCP Two-Tier; event sourcing and streaming, Kafka
c) DDD, MDSL, `operationId` in Open API Specification (OAS)

"Napkin Sketch" of SOA Realizations (Adopted from G. Hohpe)



A Consolidated Definition of Microservices

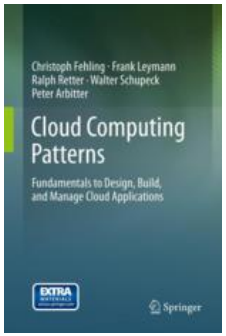
- **Microservices architectures evolved from previous incarnations of Service-Oriented Architectures (SOAs) to promote agility and elasticity**
  - *Independently deployable, scalable* and *changeable* services, each having a single responsibility
  - Modeling *business capabilities*

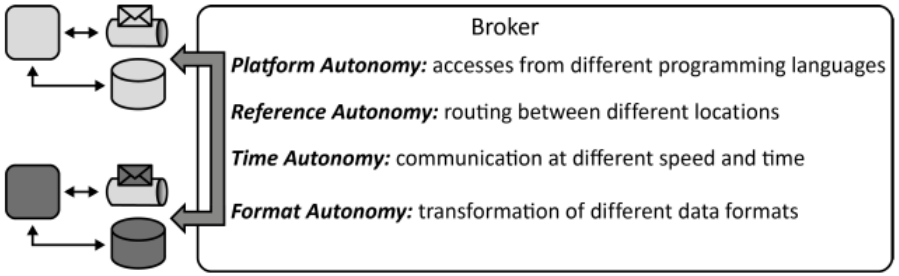


Detailed analysis: Zimmermann, O., *Microservices Tenets: Agile Approach to Service Development and Deployment*, Springer Journal of Computer Science Research and Development (2017)

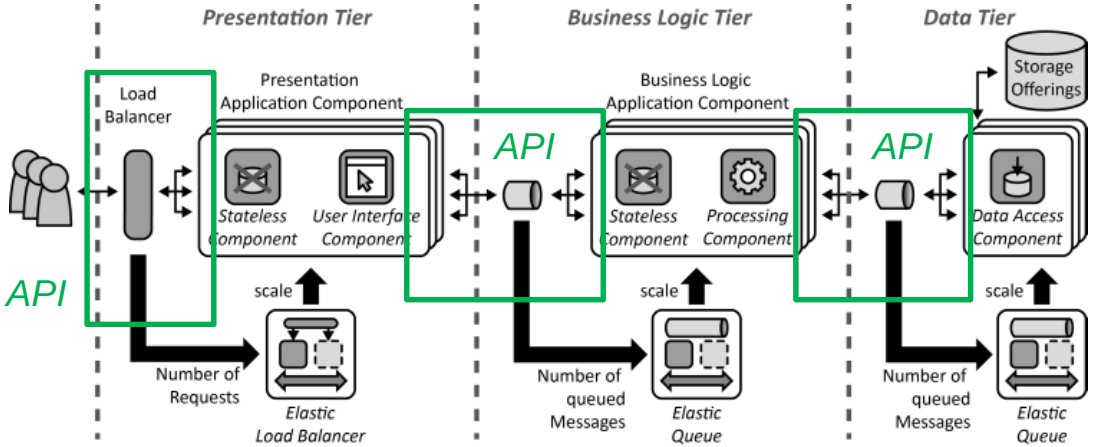
  - Often deployed in *lightweight containers*
  - Encapsulating their *own state*, and communicating via *message-based remote APIs* (HTTP, queueing), IDEALly in a loosely coupled fashion
  - Facilitating *polyglot programming and persistence*
  - Leveraging DevOps practices including decentralized *continuous delivery* and *end-to-end monitoring* (for business agility and domain observability)

Cloud-native Application Architectures are API-centric





IDEAL: Isolated State, Distribution/Decomposition, Elasticity, Automation, Loose Coupling



Cloud Application Architectures

Fundamental Cloud Architectures
- Loose Coupling
- Distributed Application

Cloud Application Components
- Stateful Component
- Stateless Component
- User Interface Component
- Processing Component
- Batch Processing Component
- Data Access Component
- Data Abstractor
- Idempotent Processor
- Transaction-based Processor
- Timeout-based Message Processor
- Multi-Component Image

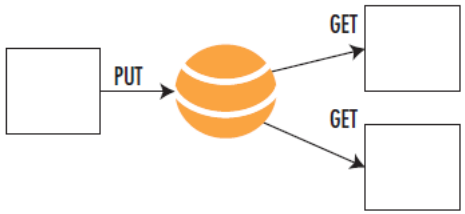
Multi-Tenancy
- Shared Component
- Tenant-isolated Component
- Dedicated Component

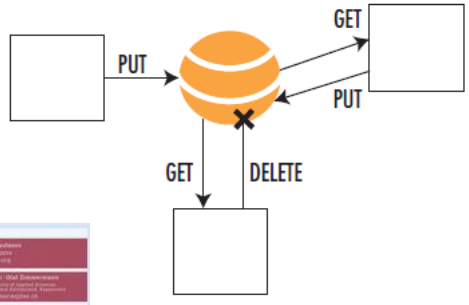
Cloud Integration
- Restricted Data Access Component
- Message Mover
- Application Component Proxy
- Compliant Data Replication
- Integration Provider

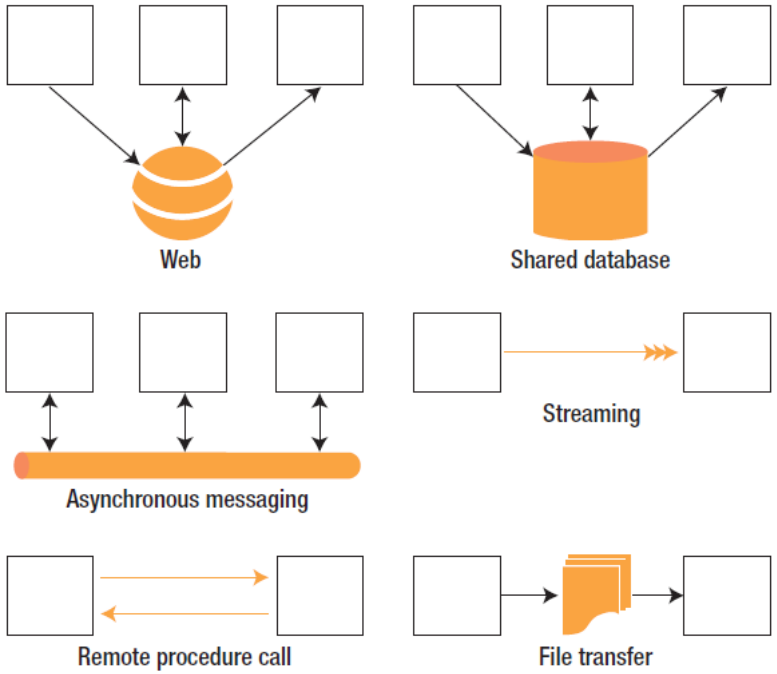
http://www.cloudcomputingpatterns.org

You can view the Web as an asynchronous connector technology

- **Resource takes place of queue in this view:**








INSIGHTS

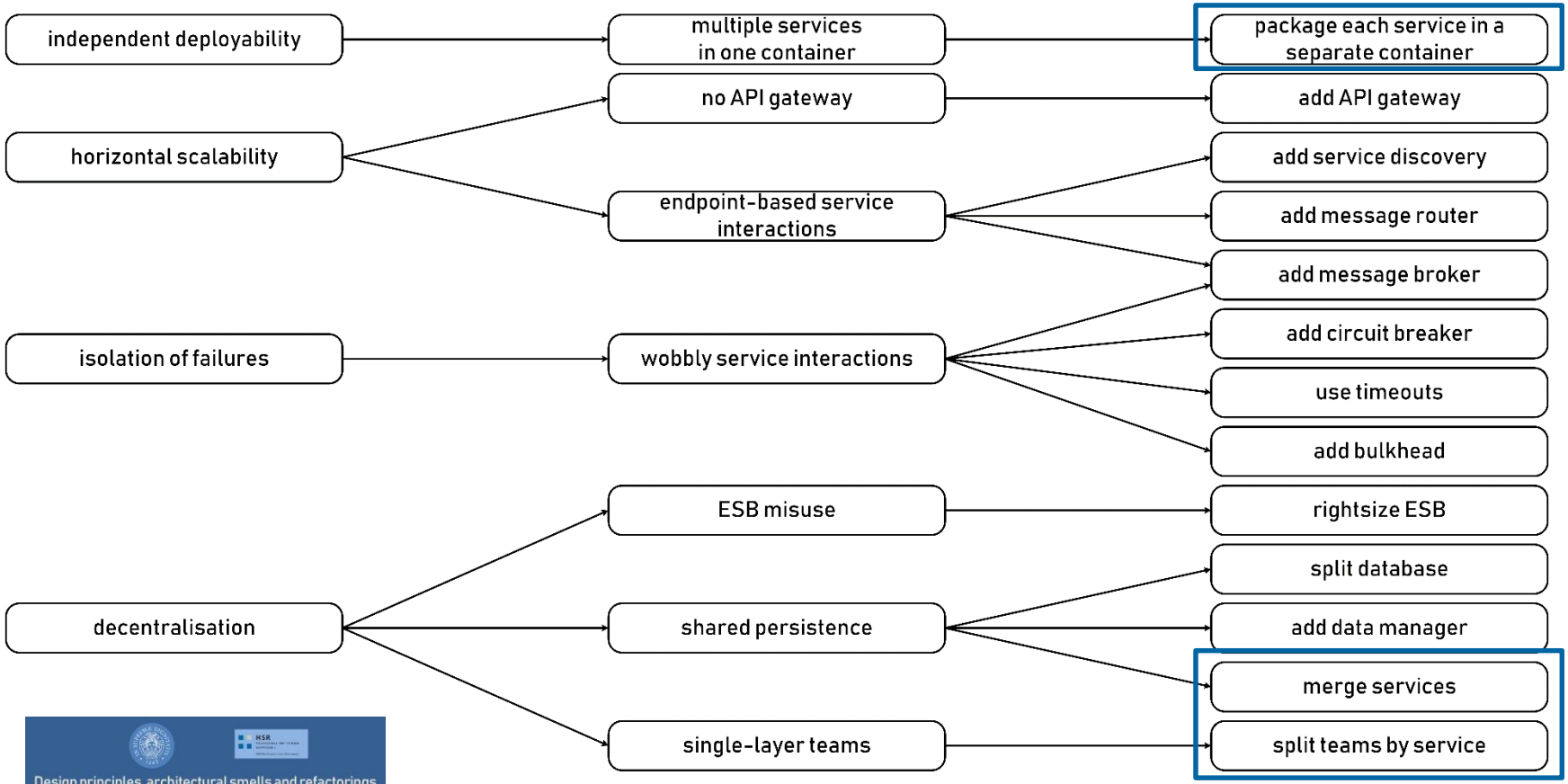
The Web as a Software Connector

Integration Resting on Linked Resources

https://ieeexplore.ieee.org/document/8239944?arnumber=8239944

From Biz and Dev to Ops: Bad Smells and Refactorings

In scope of DDD and this talk/my research



Reference: Brogi, A., Neri D., Soldani, J., Zimmermann, O., *Design Principles, Architectural Smells and Refactorings for Microservices*: A Multivocal Review. CoRR abs/1906.01553 and Springer SICS (2019) (online, report PDF, short presentation)

OpenAPI Specification (OAS): An Interface Definition Language (IDL)

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

```yaml
components:
  schemas:
    PaperItemDTO:
      type: object
      properties:
        title:
          type: string
        authors:
          type: string
        venue:
          type: string
        paperItemId:
          $ref: '#/components/schemas/PaperItemKey'
    PaperItemKey:
      type: object
      properties:
        doi:
          type: string
    createPaperItemParameter:
      type: object
      properties:
        who:
          type: string
        what:
          type: string
        where:
          type: string
```

```yaml
put:
  tags:
  - PaperArchiveFacade
  summary: write only
  description: This operation realizes the State Creation Operation
    pattern, described
    [on the MAP website](https://microservice-api-patterns.org
      /patterns/responsibility/operationResponsibilities
      /StateCreationOperation.html).
  operationId: createPaperItem
  requestBody:
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/createPaperItemParameter'
  responses:
    "200":
      description: response message payload (success case)
      content:
        application/json:
          schema:
            $ref: '#/components/schemas/PaperItemDTO'
```

Contracts in Microservice Domain-Specific Language (MDSL)

MDSL

```
data type PaperItemDTO {
 "title":D<string>, "authors":D<string>,
 "venue":D<string>, "paperItemId":PaperItemKey }
data type PaperItemKey {
 "doi":D<string> }
data type createPaperItemParameter {
 "who":D<string>, "what":D<string>, "where":D<string> }
```

"id": Role<type> triplet: Role=(Meta-)Data, Link, ID

```
endpoint type PaperArchiveFacade
  serves as INFORMATION_HOLDER_RESOURCE
  exposes
      operation createPaperItem
          with responsibility STATE_CREATION_OPERATION
          expecting
              payload createPaperItemParameter
          delivering
              payload PaperItemDTO
```

INFORMATION_HOLDER is a MAP decorator (role stereotype)

- **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, …)**

How does this notation compare to OpenAPI/JSON Schema (and other IDLs, WSDL/XSD)?

MDSL and OAS in Comparison

MDSL

Criterion	MDSL	OAS (f.k.a. Swagger)
Concrete syntax	DSL (made with Xtext)	YAML, JSON
Abstract syntax	SOA concepts, MAP domain model	HTTP concepts
Main use cases	Agile modeling, SOAD, contract first	Code first, testing, stub generation
Bindings	HTTP, Java; more possible and planned (gRPC, GSL, MOM/EIP)	HTTP only (sibling language AsyncAPI)
Size of specification	19 pages (v3.3.2; note: less complete!)	95 pages (v3.0.3)
Size of "Publication Management" Demo contract (blog post)	1,568 bytes, 132 words (1,321 characters)	3,263 bytes, 199 words (2,159 characters)
Tools	Few (editor, OAS generator), see here	Many, see here
Maturity	since 2018; just open sourced (ZIO)	since 2011 (SmartBear)
License	Apache License 2.0	Apache License 2.0

News on MDSL – v3 Grammar and Tools Open Sourced

MDSL

- **Editor, API linter, OpenAPI generator; tutorial [available on GitHub]**

Design Goals

A contract language for (micro-)service API design should/must:

- Support *agile modeling* practices, for instance in API design workshops:
  - Value and promote readability over parsing efficiency (in language design)
  - Support partial specifications as first-class language concepts to be refined iteratively (see above example `{ID, D}`)

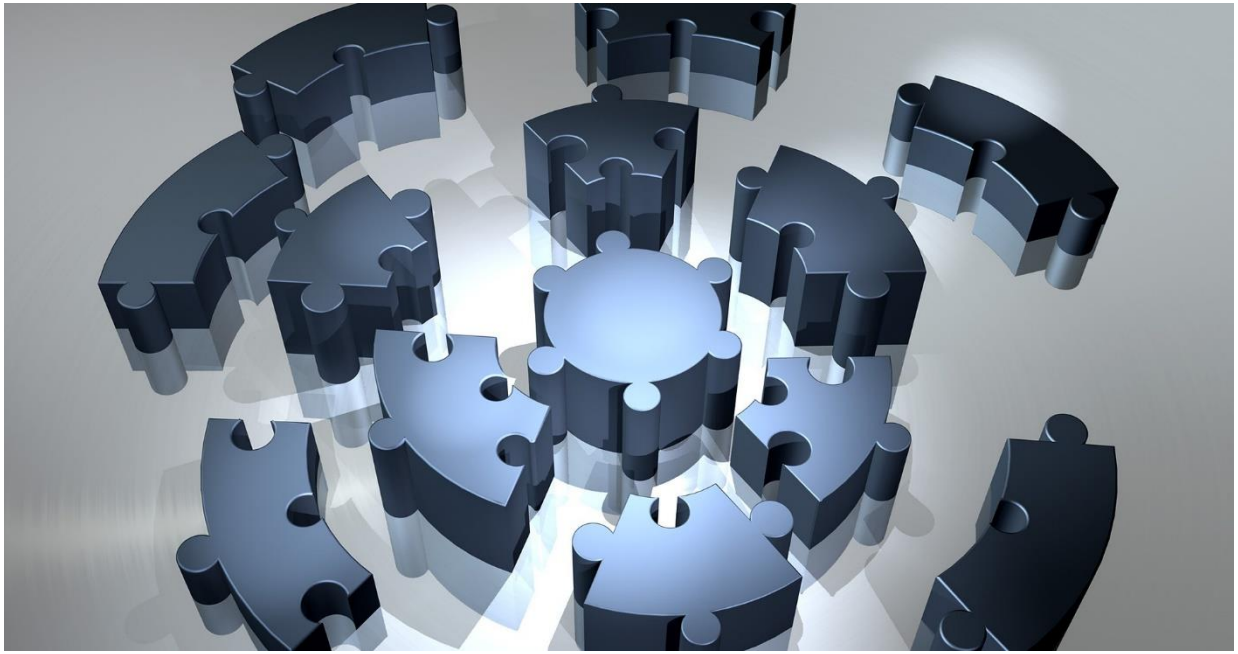
Getting Started

- Presentations featuring Context Mapper, MAP and MDSL can be found here.
- The GitHub Pages for MDSL provide a tutorial and language reference information.
- There is an Eclipse update site https://microservice-api-patterns.github.io/MDSL-Specification/updates/ for the MDSL editor.
- As a contributor, please consult the readme file of the dsl-core project for getting started information and prerequisites.

  - *Top-down* from requirements (for instance, user stories for integration scenarios)
  - *Bottom up* from existing systems (represented, for instance, as DDD-style context maps)

Design Principles

Part 3: Components (and Patterns)



Debunking Myths 3: Not-Invented-Here Syndrome

Context

Web/cloud application development (any project, actually).

Myth

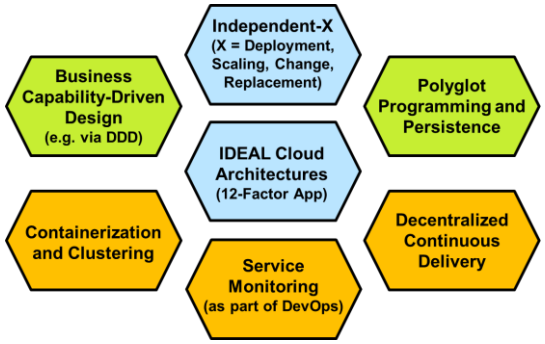
Our performance scalability requirements are more advanced than everybody else's (ok, except for those of the Internet giants). There is no point in reusing existing solutions; we have to invent our own library, framework, protocol, etc.

Rectification

Back to context and requirements (DDD!): Identify candidate services and make a conscious and candid buy-rent-build decision per service.
If "build" is required, apply patterns when doing so.
Components (captured as patterns, in frameworks) contain (cost and risk)!

Decomposition Heuristics that do not suffice

- **Two-pizza rule (team size)**
- **Lines of code (in service implementation)**
- **Size of service implementation in IDE editor**



Microservices tenets

- **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”**

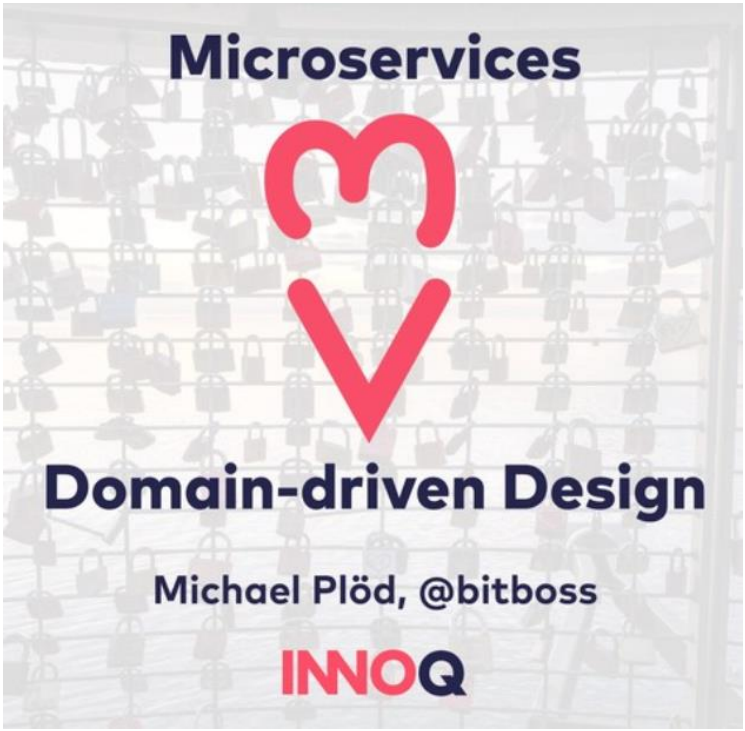
What is wrong with these “metrics” and “best practice” recommendations?

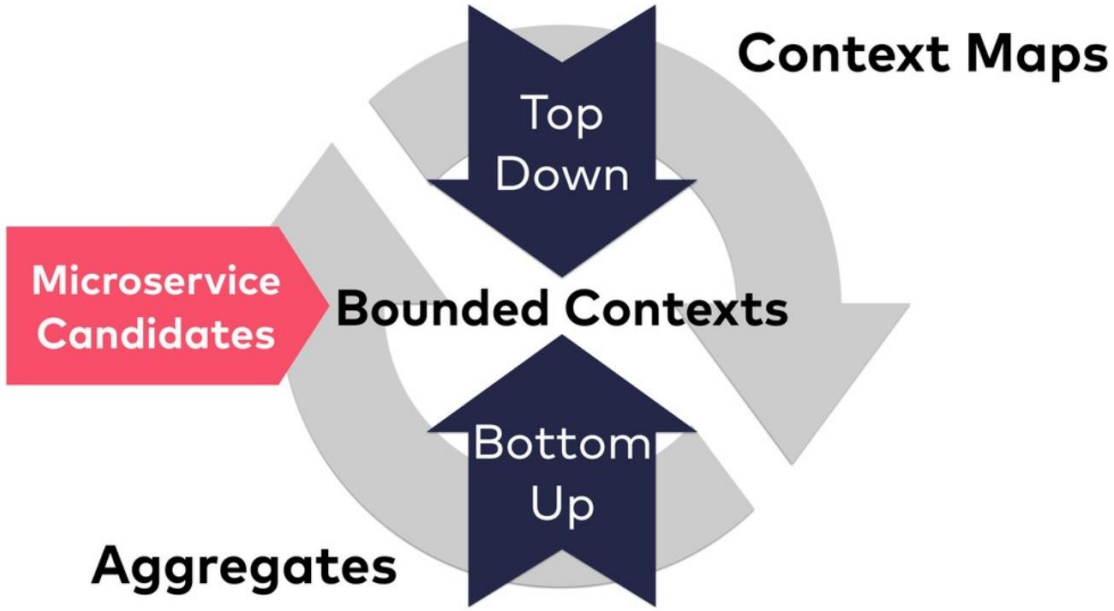
➡ Context matters here too – one size does *not* fit all,

(also pointed out in a keynote at Agile Australia)

DDD to the Remedy in (Micro-)Service Design

- **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

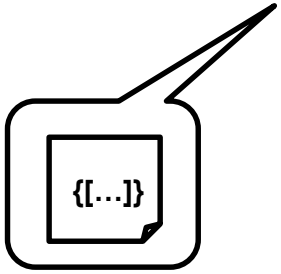




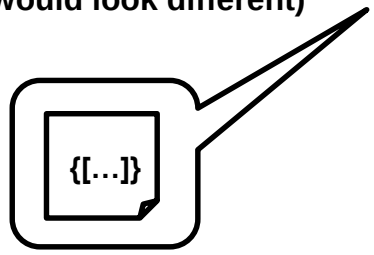
Reference: JUGS presentation, Bern/CH, Jan 9, 2020

Service Operations and Message Types (from Integration Domain)

```
curl -X GET "http://localhost:8080/customers/rgpp0wkpec" -H "accept: */*"
```



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

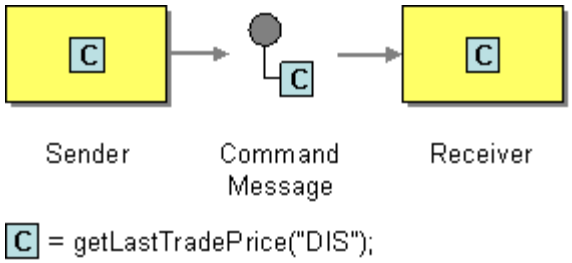


Response message structure

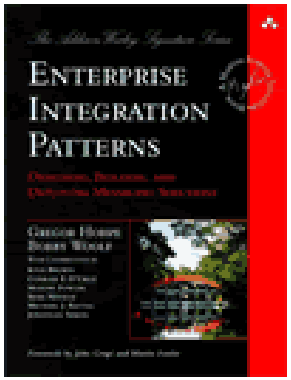
```
{
  "_links": [
    {
      "deprecation": "string",
      "href": "string",
      "hreflang": "string",
      "media": "string",
      "rel": "string",
      "templated": true,
      "title": "string",
      "type": "string"
    }
  ],
  "birthday": "2019-02-12T09:10:07.370Z",
  "city": "string",
  "customerId": "string",
  "email": "string",
  "firstname": "string",
  "lastname": "string",
  "moveHistory": [
    {
      "city": "string",
      "postalCode": "string",
      "streetAddress": "string"
    }
  ],
  "phoneNumber": "string",
  "postalCode": "string",
  "streetAddress": "string"
}
```

+/-?

Embed nested entity data?
or
Link to separate resource?

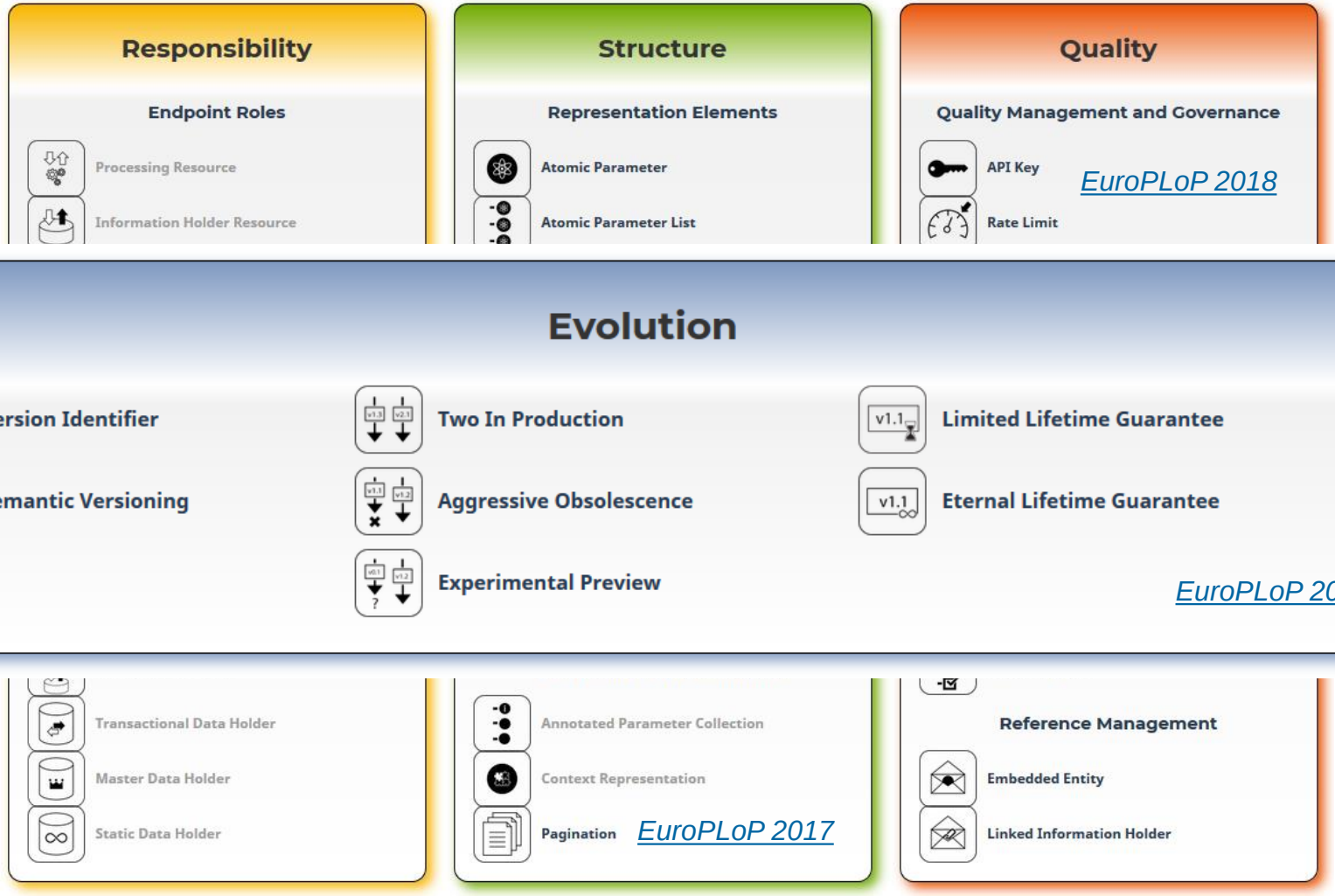


https://www.enterpriseintegrationpatterns.com/patterns/messaging/CommandMessage.html



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

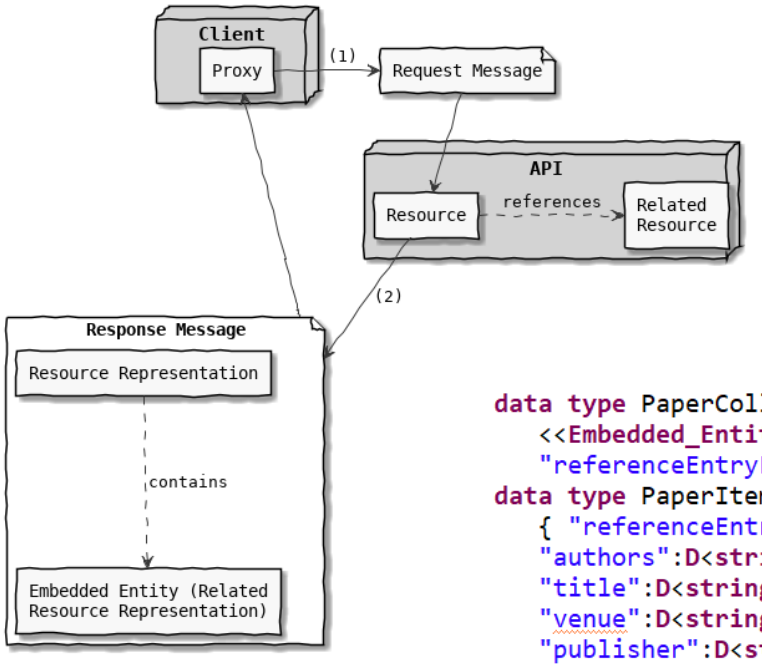
Microservices API Patterns (MAP): Overview



(not yet) EuroPLoP 2020

http://microservice-api-patterns.org

Embedded Entity Pattern



Problem: How can you avoid exchanging multiple messages when receivers require insights from multiple related information elements?

```
data type PaperCollectionDTO { "paperCollectionId":D<long>,
   <<Embedded_Entity>>
   "referenceEntryList":PaperItemDTO*}
data type PaperItemDTO
   { "referenceEntryId":D<int>,
   "authors":D<string>*,
   "title":D<string>,
   "venue":D<string>,
   "publisher":D<string>,
   "doi":D<string>}
```

Forces: Performance, scalability; flexibility and modifiability; data quality, freshness, consistency.

online version of pattern

Solution: For any relationship that the client has to follow, embed a Data Element in the message that contains the data of the target entity (instead of linking to the target entity).

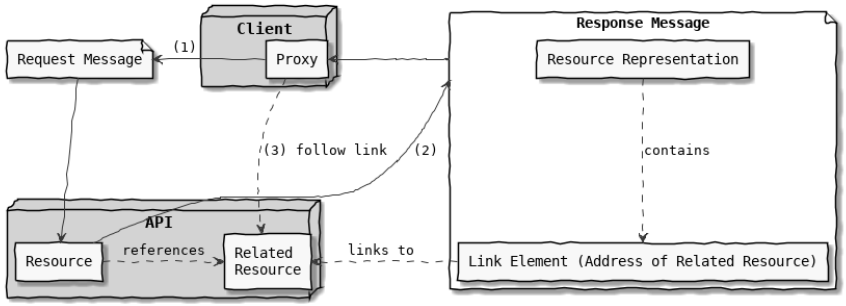
Linked Information Holder Pattern

Problem: When exposing structured, possibly deeply nested information elements in an API, how can you avoid sending large messages containing lots of data that is not always useful for the message receiver in its entirety?

Forces: Same as for Embedded Entity.

```
data type PaperDirectory { "paperCollectionId":D<long>,
   <<Linked_Information_Holder>>
   "referenceEntryPointer":PaperItemInformationHolderLink*}
data type PaperItemInformationHolderLink {
   <<Link_Element>> "uri":L<string>
}
```

[online version of pattern]



Solution: Add a Link Element to the message that references an API endpoint. Let this API endpoint represent the linked entity; for instance, use an Information Holder Resource for the referenced information element.

Key Messages of this Talk

- **It is the API contracts (and their implementations) that make or break projects – not (or not only) middleware and network protocols**
  - Frameworks and infrastructures come and go, APIs stay
  - Rich domain semantics call for higher-level design tools

- **Microservice API Patterns (MAP) language: website first**
  - Focus on message representation elements
  - 20+ patterns, sample implementation in public repo, supporting tools
- **Microservices Domain-Specific Language (MDSL) open sourced**
  - Uses MAPs in service contracts (as decorators)
  - Can be generated from DDD bounded contexts

MDSL

- **Context Mapper supports strategic Domain-Driven Driven Design (DDD), rapid OOAD/tactic DDD and architectural refactoring**
  - Other tools emerging: context discovery, application layer design

Teaser Question

- **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 ☺*

Selected Publications (click on screen caption to get them)

Olaf Zimmermann (ZIO) Home Blog Publications Projects About Type and enter...

Research Papers and Practitioner Articles (Selection) [2]

Microservices and related topics (since 2016):

- Zimmermann, O.; Stocker, M.; Lübke, D.; Pautasso, C.; Zdun, U.: *Introduction to Microservice API Patterns (MAP)*, Joint Post-Proceedings of the First and Second International Conference on Microservices (Microservices 2017/2019), Dagstuhl OpenAccess Series in Informatics (OASIcs) 2020 (PDF)
  - The MAP website can be found here.
- Kapferer, S.; Zimmermann, O.: *Domain-specific Language and Tools for Strategic Domain-driven Design, Context Mapping and Bounded Context Modeling*. Proc. of MODELSWARD 2020 conference, pp. 299-306. (PDF)
  - More information on Context Mapper can be found here (presentations, papers).
- Lübke, D.; Zimmermann, O.; Pautasso, C.; Zdun, U.; Stocker, M.: *Interface Evolution Patterns: Balancing Compatibility and Extensibility across Service Life Cycles*, Proc. of EuroPLoP 2019 conference, ACM Digital Library (PDF)
  - All papers from the MAP project are listed here (5 so far).
- 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)
- Pardon, G.; Pautasso, C.; Zimmermann, O.: *Consistent Disaster Recovery for Microservices: the BAC Theorem*. In: IEEE Cloud Computing, 5(1) 2018, pp. 49-59. (PDF)
- Pahl, C.; Jamshidi, P.; Zimmermann, O.: *Architectural Principles for Cloud Software*. In: ACM Trans. on Internet Technology (TOIT), 18 (2) 2018, pp. 17:1-17:23. (PDF)

My New Blog: "The Concerned Architect"

https://ozimmer.ch/blog/

Slides, demo instructions: https://ozimmer.ch/practices/2020/06/10/ICWEKeynoteAndDemo.html

DIMENSIONS OF SUCCESSFUL WEB API DESIGN AND EVOLUTION: CONTEXT, CONTRACTS, COMPONENTS
BACKUP CHARTS

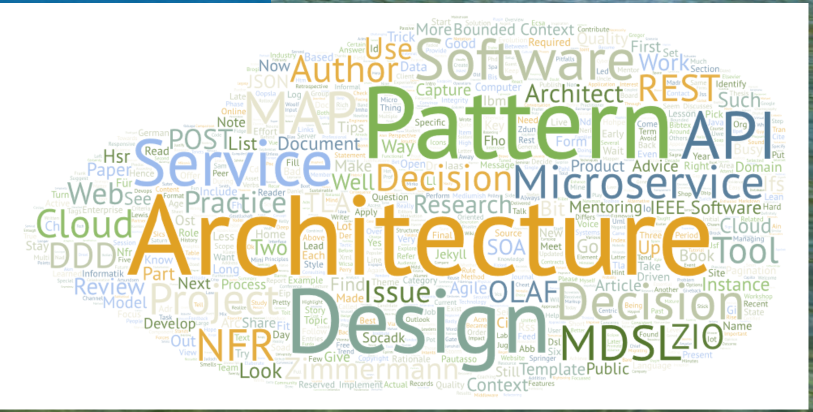
In collaboration with Microservices Community

20th International Conference on Web Engineering (ICWE)

Online, June 11, 2020

Prof. Dr. Olaf Zimmermann (ZIO)
Distinguished (Chief/Lead) IT Architect
Institute for Software, HSR FHO OST
ozimmerm@hsr.ch

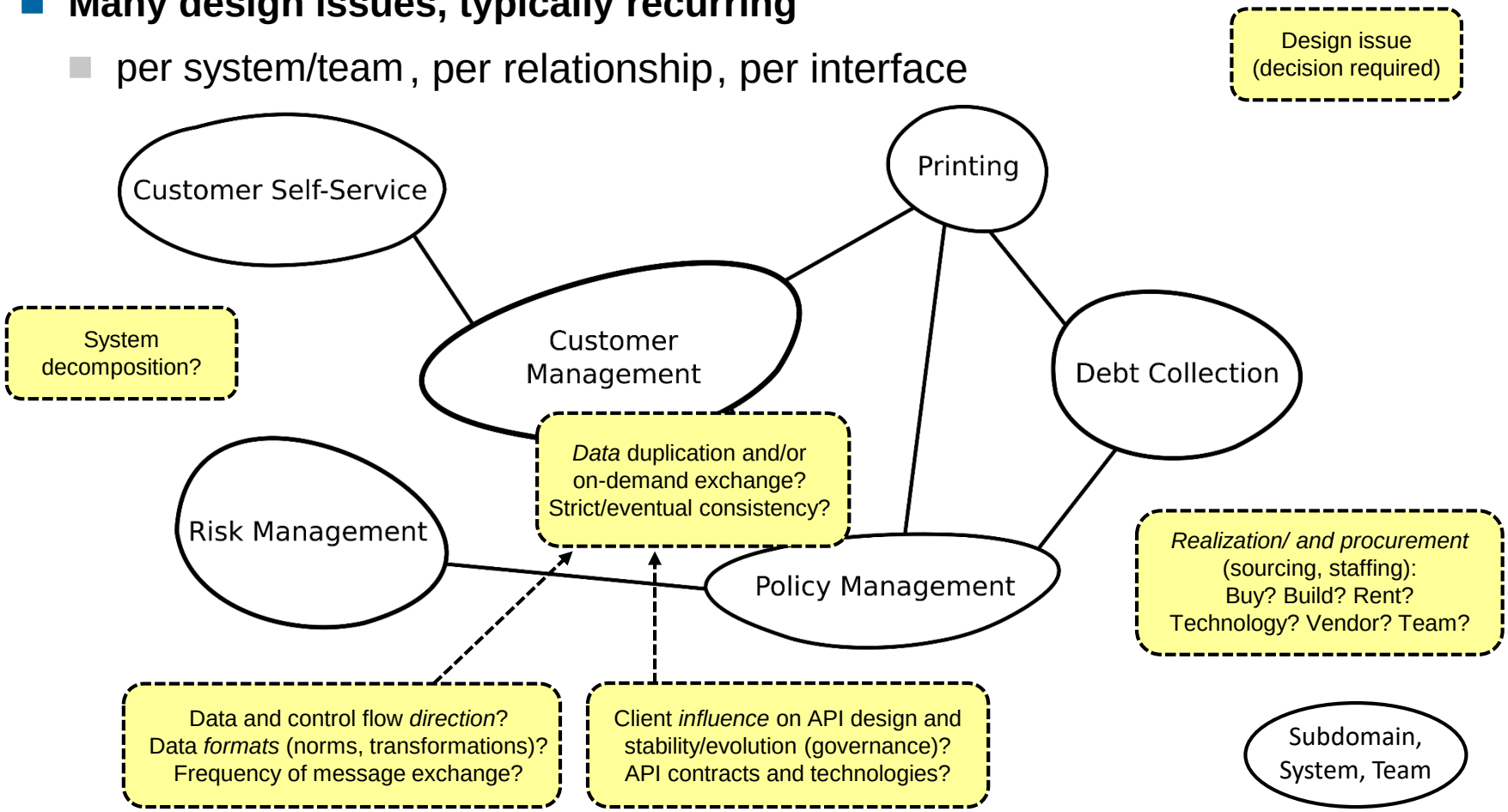
Blog: https://ozimmer.ch/blog



HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

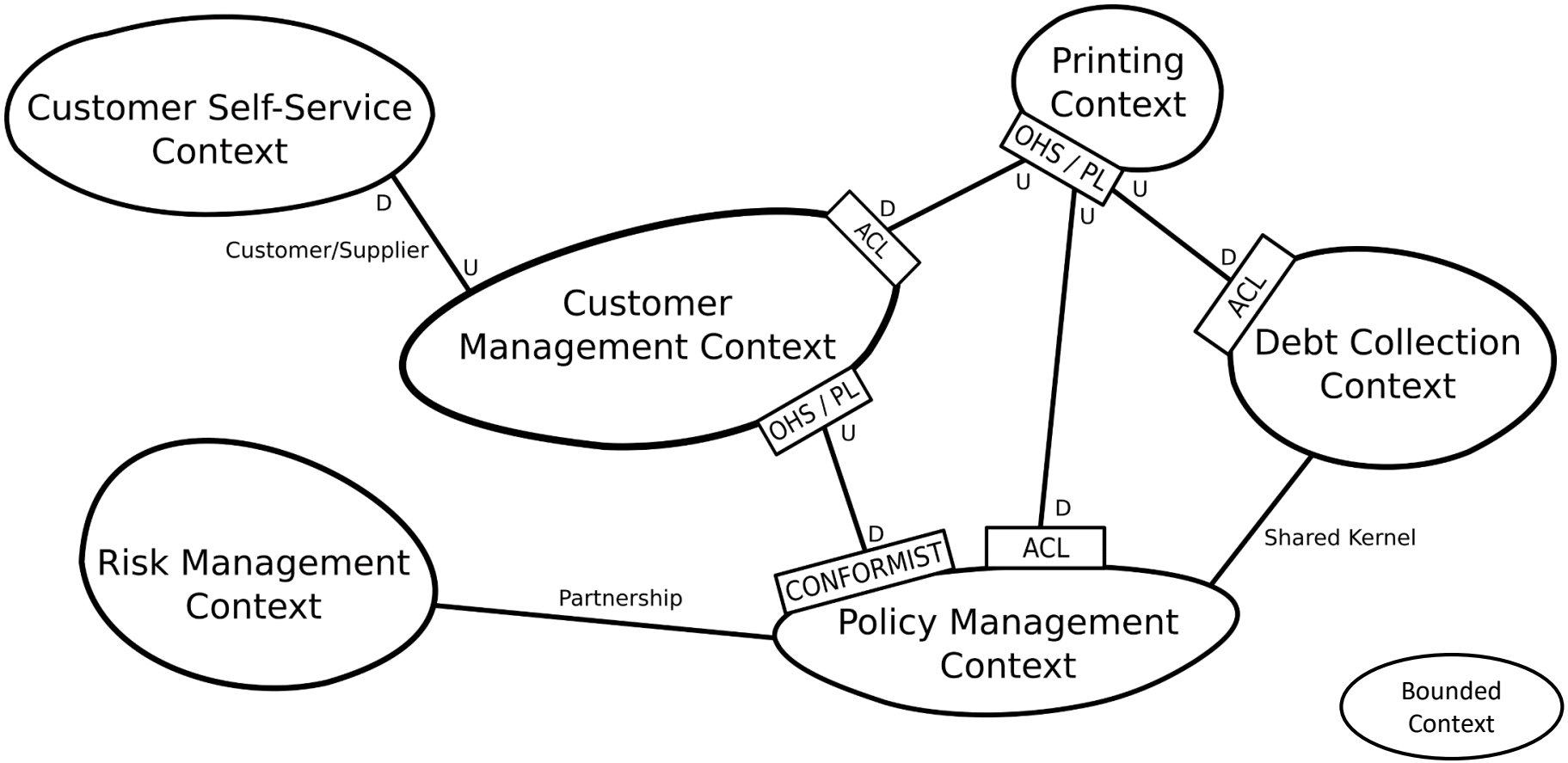
“Fictitious” Insurance Application/Integration Landscape

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



A Strategic DDD Context Map with Relationships

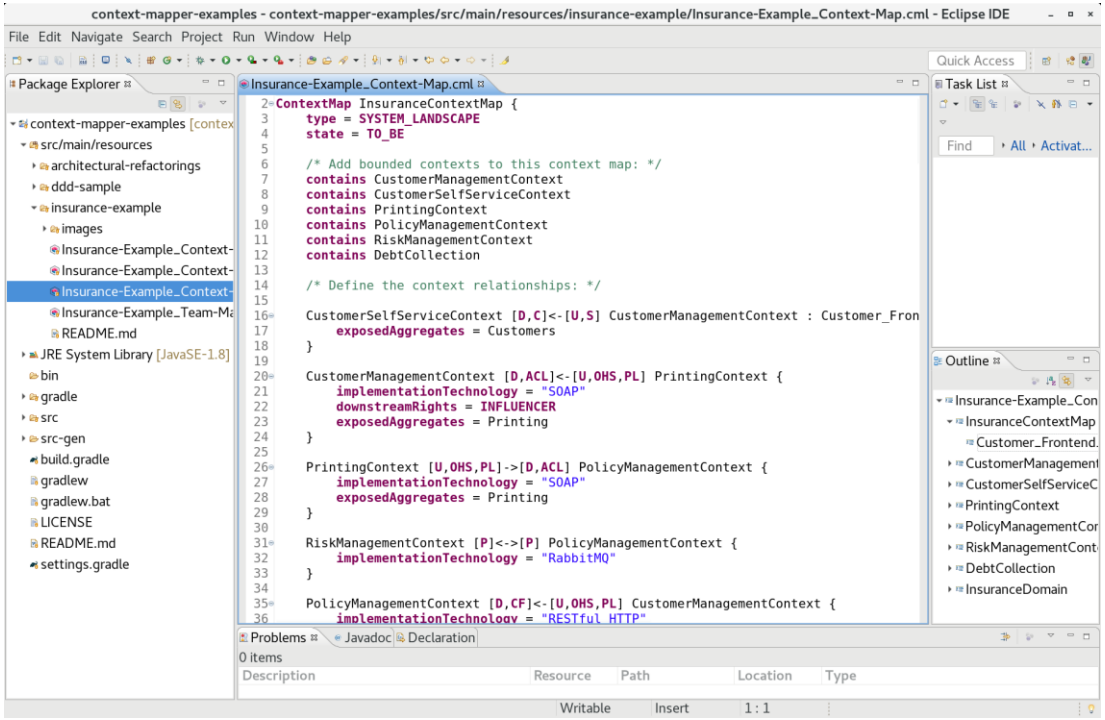
- **Insurance scenario, example model from https://contextmapper.org/**



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

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”)



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

    CargoBookingContext [SK]<->[SK] VoyagePlanningContext

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

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

Bounded Contexts (systems or teams)

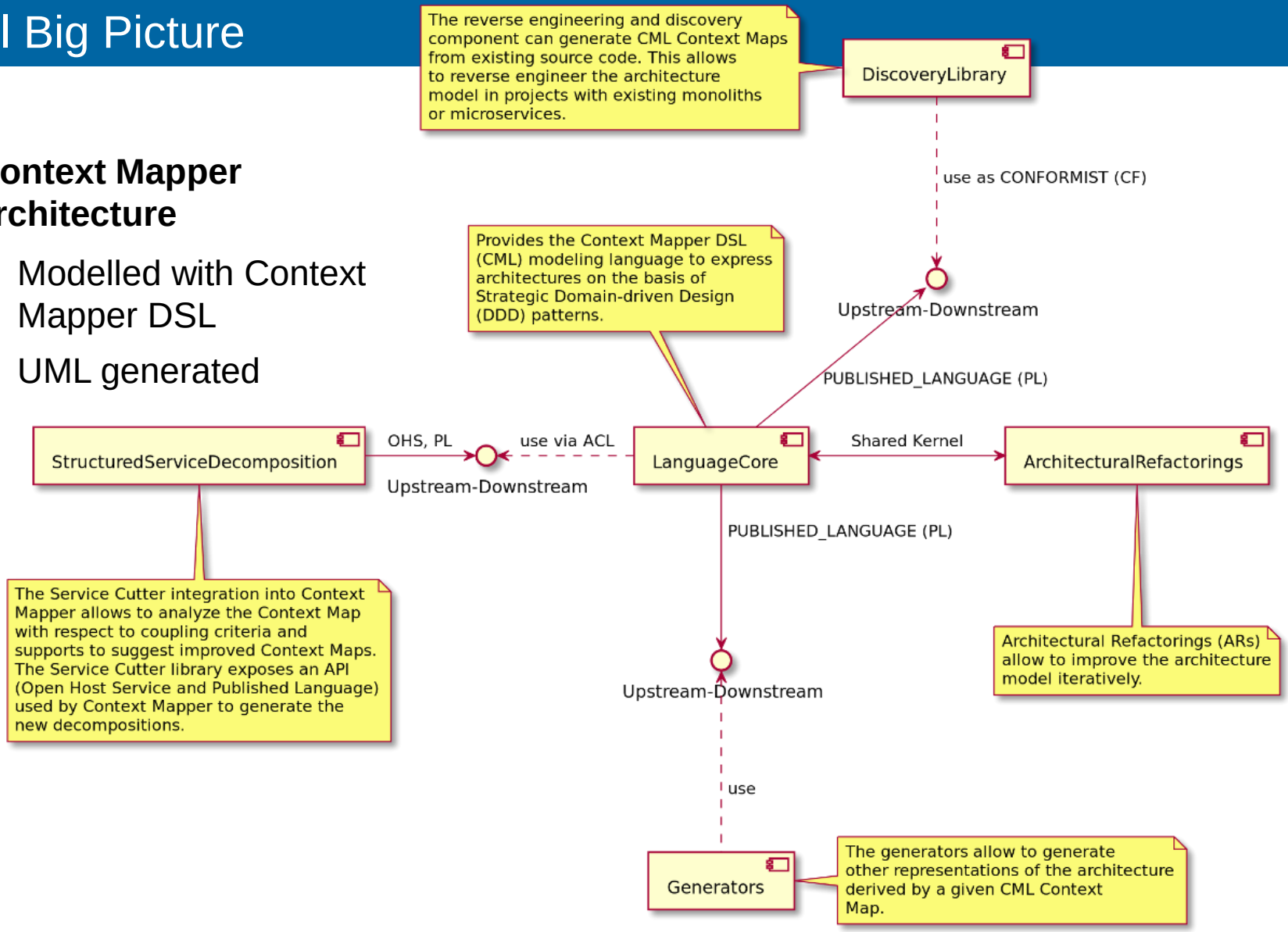
DDD relationship patterns (role of endpoint)

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



Microservice API Patterns (MAP) Categories

europlop

- **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?

READ MORE →

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?

READ MORE →

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?

READ MORE →

- **Evolution Patterns:**
  - Recently workshopped (EuroPLoP 2019)

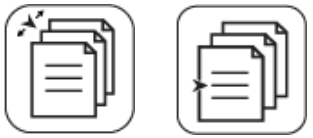
http://microservice-api-patterns.org

Microservice API Patterns (MAP)

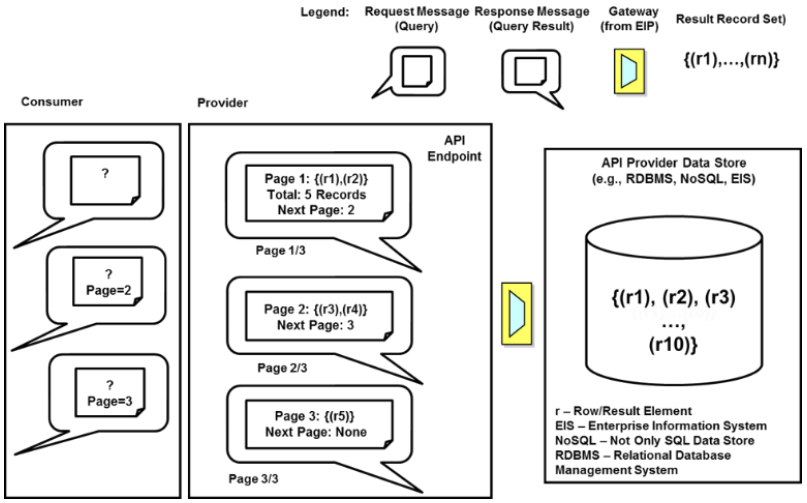
MAP Example: Pagination (1/2)

- **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

MAP Example: Pagination (2/2)



- **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



Microservice API Patterns (MAP)

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