Software-Engineering 2

Prof. Dr.-Ing. Gerhard Wanner
Email: wanner@hft-stuttgart.de

DDD - DOMAIN DRIVEN DESIGN

Overview

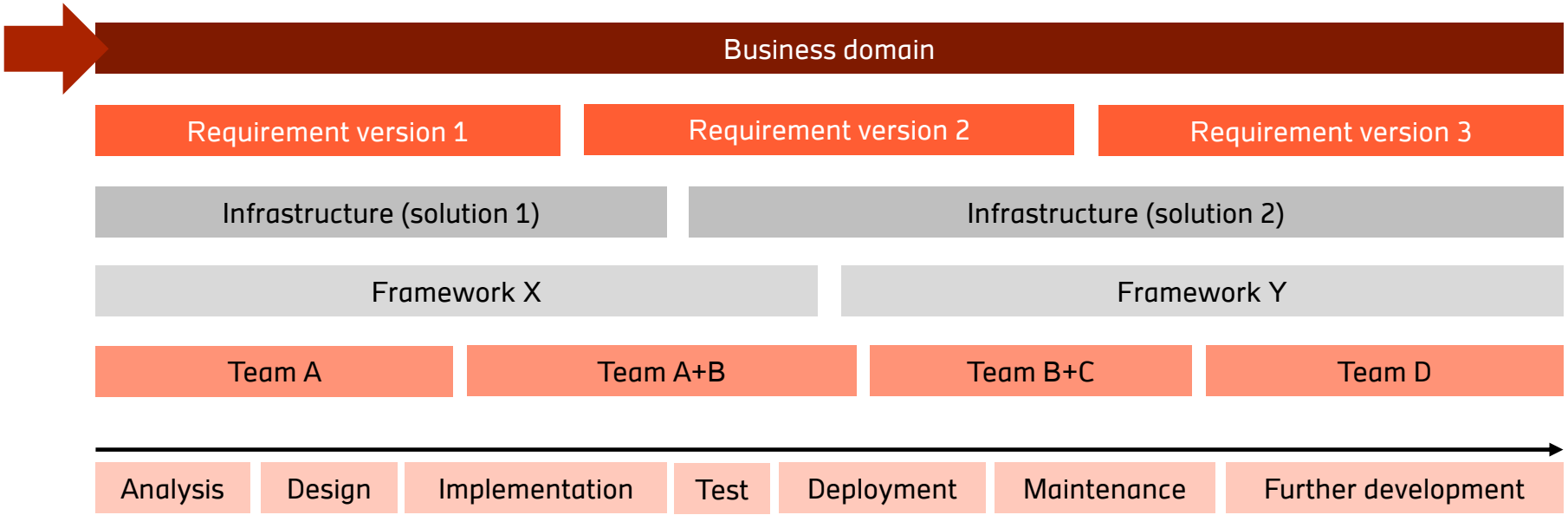
- Introduction
- DDD Strategic Design
- DDD Tactical Design / Pattern language.

DDD - Motivation

- Common problems in real-life projects
  - The **database is given a too high priority** and most discussions revolve around the database and the data model rather than business processes
  - Developers do **not take enough effort to name the objects** and operations in such a way that the names match the technical tasks they perform. This leads to a large gap between the mental model that users have and the software that developers deliver to users
    - This problem is usually the result of poor cooperation with the business department. Often, business experts spend too much time working alone on specifications that nobody uses or that are only partially read and therefore considered by developers
  - Developers **place business logic in components that belong to the user interface or data management**. In addition, database-operations are executed in the middle of the domain logic
  - The developers **create false abstractions to cover all present and possible future needs** with overly generalized solutions instead of satisfying the actual concrete business needs.

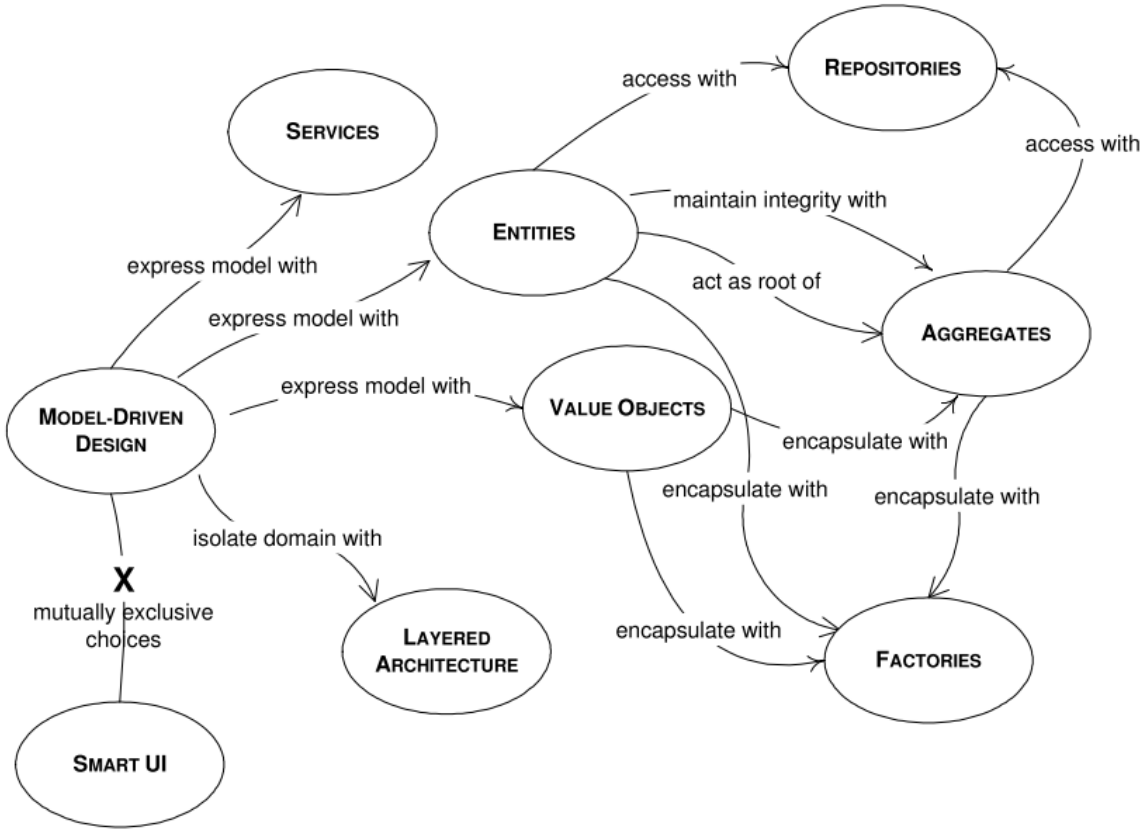
DDD - Motivation

- What makes software stable?



DDD - Overview

- Origin: Eric Evans, Domain-Driven Design - Tackling Complexity in the Heart of Software, Addison-Wesley, 2004.



Abel Avram, Floyd Marinescu: Domain-Driven Design Quickly, http://sosa.ucsd.edu/teaching/cse294/fall2007/dddbook.pdf

Architecture-centric development approaches
Domain Driven Design (DDD)

- **Architectural styles are not sufficient**
- Software with high structural quality requires **architecture work close to the code** (architecture-centered development approaches)
- Domain Driven Design is a **collection of principles and patterns** that assist in designing **high quality software**
- Domain Driven Design also addresses **aspects of organization and collaboration** between architect, developer, domain expert and other teams
- Domain Driven Design is divided into **strategic and tactical design**.

Domains and subdomains

- Domain
  - Independent problem and solution area
  - Hierarchy formation through subdomains
  - Each domain or subdomain has a **bounded context**

- Types of domains
  - Core domain - Core domain functionality of the system
  - Generic Domain - Not part of the Core Domain but important for the business
  - Supporting Domain - Supporting, subordinate functions that must be separated from the core domain.

> Domain:
> A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.
>
> Eric Evans

Overview

- Introduction
- DDD Strategic Design
- DDD Tactical Design / Pattern language.

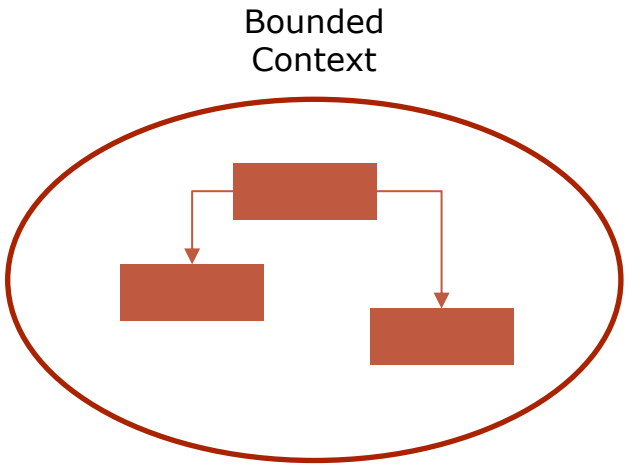
Domain Driven Design - Strategic Design

- Strategic design is the design of independent business domains and their business language

> [Strategic Design] lays out techniques for recognizing, communicating and choosing the limits of a model and its relationships to others.
>
> Eric Evans

Domain Driven Design – Strategic Design

- **Bounded Context**
  - Business delimited area
  - Elements of the context are motivated exclusively by business aspects
  - Has interfaces to other Bounded Contexts.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

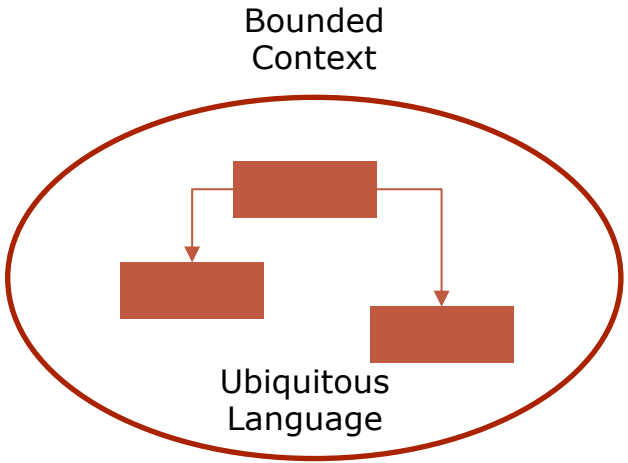
Domain Driven Design – Bounded Context

- Advantage of large Bounded Contexts
  - Larger unified model
  - Simple distribution of tasks
  - Less difficult translation between models
  - Clearer communication in the entire team through a common ubiquitous language.
- Advantage of smaller Bounded Contexts
  - Reduced communication overhead in each team
  - Smaller code base
  - Simplified Continuous Integration
  - Easier implementation of very specific requirements.

Domain Driven Design – Strategic Design

- Ubiquitous Language
  - Language of the context understood by all parties (developers, architects, domain experts, stakeholders)
  - Manifests itself in the source code through a model as well as in class and method names.

> Ubiquitous Language:
> To create a supple, knowledge-rich design calls for a versatile, shared team language and a lively experimentation with [that language]
>
> Eric Evans



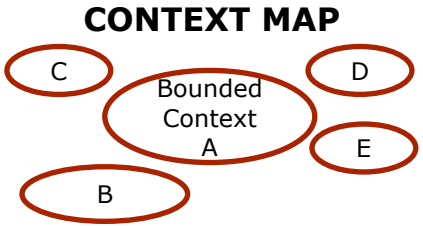
Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Bounded Context, Teams and Ownership

- Domain Driven Design meets different views of the business units on the business through the principle of Bounded Contexts
  - This prevents monolithic structures and systems!
- A bounded context...
  - is a modularization tool that helps to **keep defined limits and to concentrate on the core of your business** / system within the limits
  - has its **own source code repository**
  - has its **own database schema** (or database)
  - is a **standalone deployment artefact**
  - can only be **processed by one team**. One team can be responsible for several bounded contexts
    - This statement primarily addresses an enterprise architecture. When designing a software system there is often a (large) team available. A bounded context is used here as the modularization limit.

Context Maps

- When designing an enterprise- or system-architecture, there are usually several domains or sub-domains (problem and solution areas)
- Each domain or subdomain has a bounded context
- On this basis, a context map is created
- Contexts can still have dependencies (relationships) to each other.

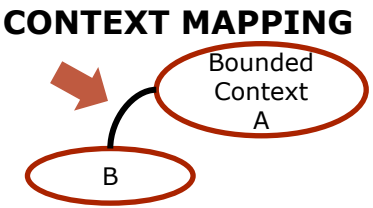


Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

> A context map is the overlap between project management and software design
>
> Eric Evans

Context Mapping

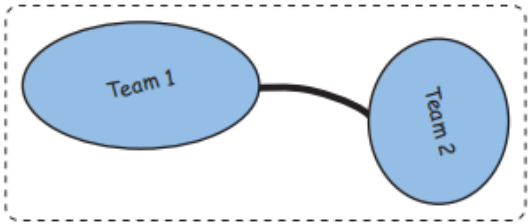
- Context Mapping is the strategy of how a context is integrated into another
- Context mapping brings organizational aspects to the fore and describes applicable patterns.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Context Mapping Strategies (1)

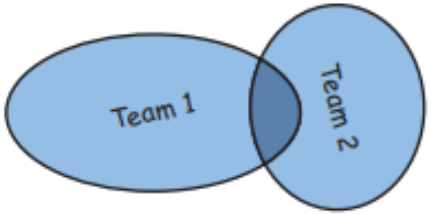
- Partnership
  - Common basic conditions such as infrastructure, deployment, release planning
  - Strong dependency of the teams - teams can only succeed together
  - High need for communication
  - Each team has its isolated bounded context.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Context Mapping Strategies (2)

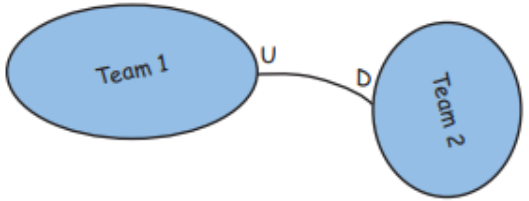
- Shared Kernel
  - Each team has its own bounded context
  - A subset of the contexts are identical in both domains (same domain model and business area)
  - Responsible person for build, deployment and maintenance of the common part necessary
  - High need for communication.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Context Mapping Strategies (3)

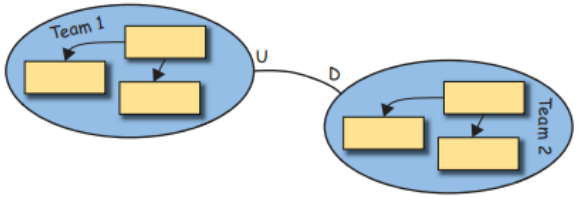
- Customer-Supplier
  - One team or bounded context is leading (supplier)
  - Customer and supplier plan together
  - The final decision lies with the supplier
  - Increased need for communication.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Context Mapping Strategies (4)

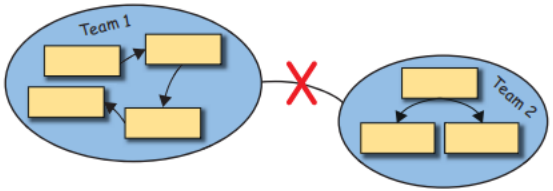
- Conformist
  - A team becomes a conformist and accepts the bounded context as defined by the leading team
  - Many teams become conformists especially when they have to integrate into a very large, complex model that is already well established
    - Example: If you want to become an Amazon partner, you will have to conform to the Amazon model in order to integrate.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Context Mapping Strategies (5)

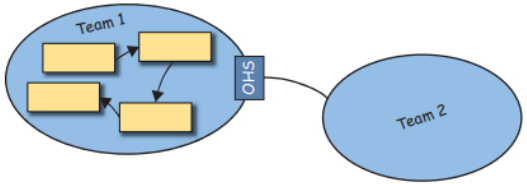
- Separate Ways
  - If no integration benefits can be achieved, teams should work independently
  - High flexibility.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Context Mapping Strategies (6)

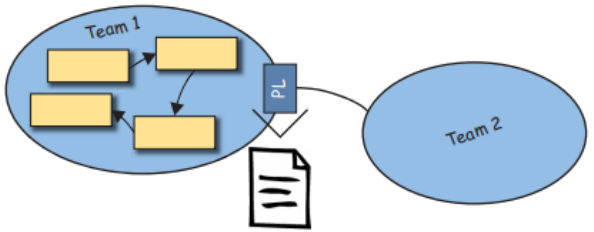
- Open Host Service
  - Create Service API for existing bounded contexts to simplify access
  - A means for the integration of non-adaptable legacy systems
  - Enables DDD to be used in a legacy environment and facilitates mastering the architectural impact.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

Context Mapping Strategies (7)

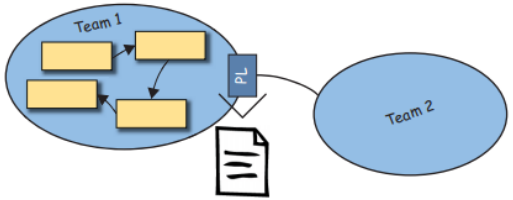
- Published Language
  - Publish and well documented language for a bounded context
  - Provision in common data formats such as JSON, XML or protobuf
  - Combination with the Open Host Service pattern very suitable / common.



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

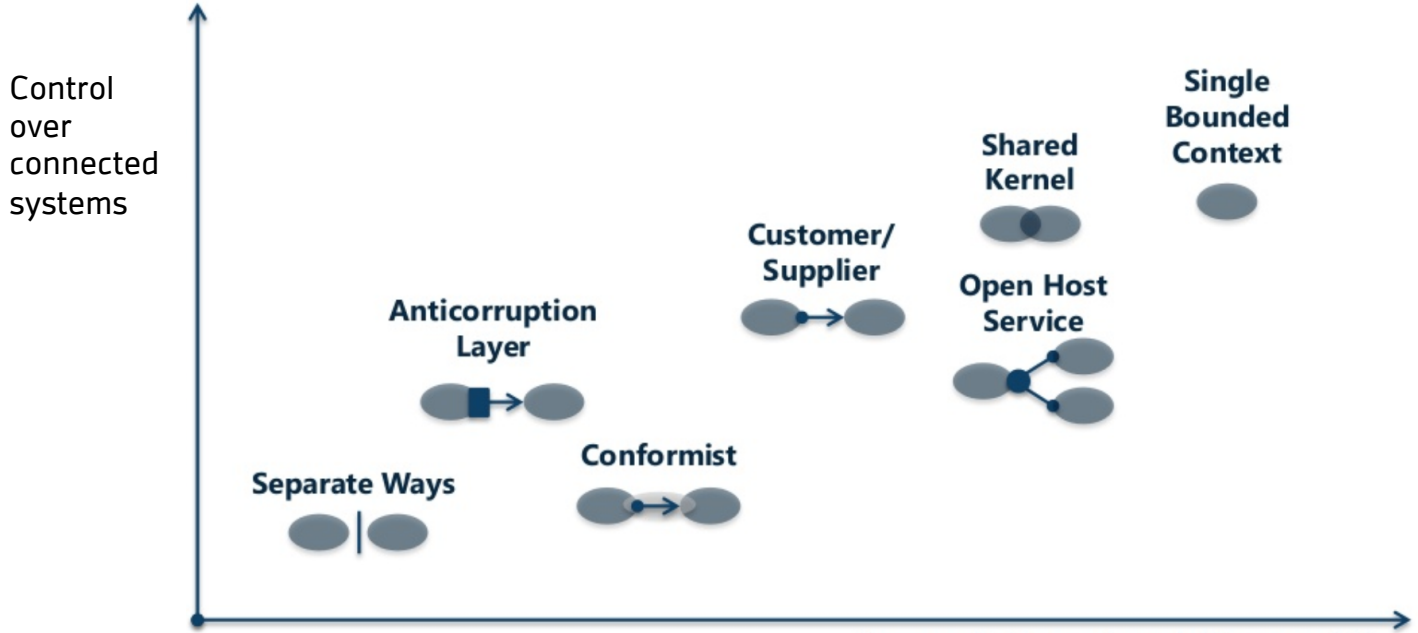
Context Mapping Strategies (8)

- Anticorruption Layer
  - Defensive integration of a bounded context
  - Helps to keep the own context stable in case of changes in the integrated context
  - Adaptation of an external context, e.g. "classical" with the Adapter Pattern of the Gang of Four
  - Increased flexibility
  - Separate component in the application architecture e.g. based on the port and adapter architecture style (Onion / Hexagonal Architecture).



Source: Vernon, Vaughn: Domain-Driven Design kompakt. Leitfaden. 1. Aufl. Heidelberg

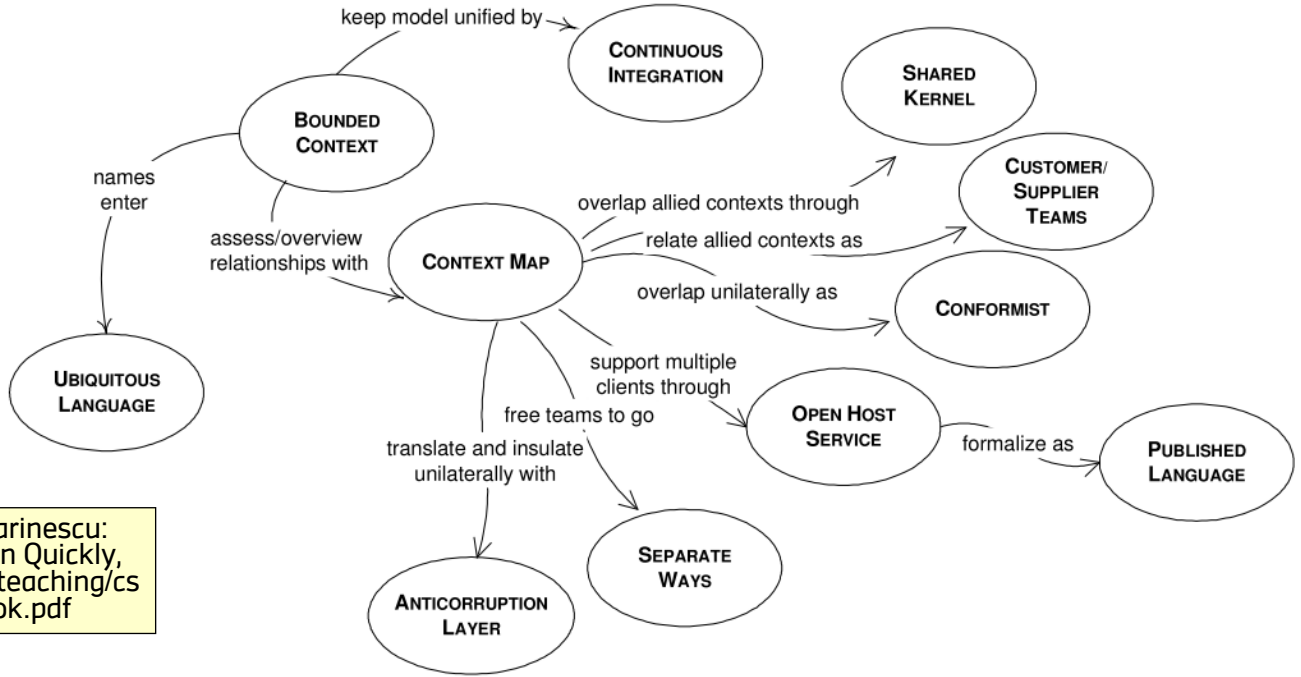
Context Mapping Strategies - Classification



Ute May: Domain Driven Design – Strategic Design
https://de.slideshare.net/iksgmbh/domain-driven-design-strategic-design-75913405

Context Mapping Strategies - Summary

- Basically, reduce coupling / enforce cohesion



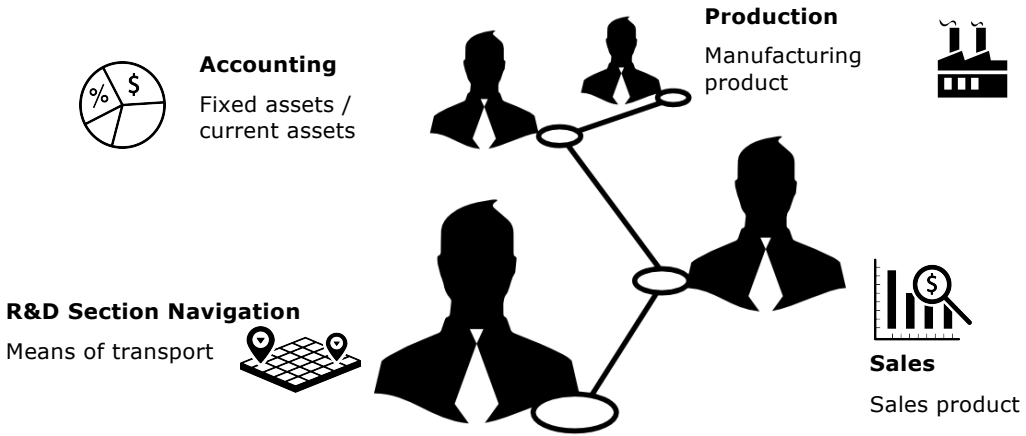
Abel Avram, Floyd Marinescu: Domain-Driven Design Quickly, http://sosa.ucsd.edu/teaching/cse294/fall2007/dddbook.pdf

Implementation possibilities of Context Mapping

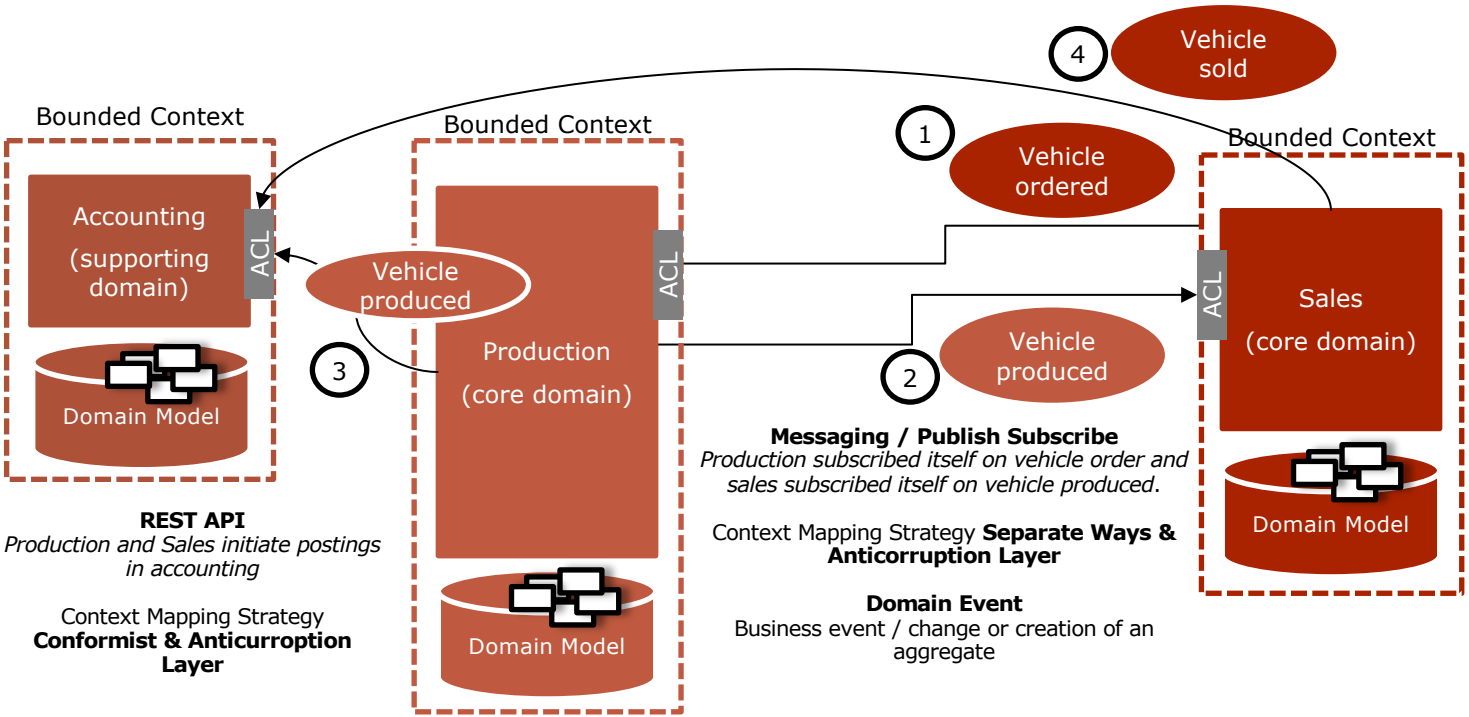
- Recommended
  - RESTful API
  - WebServices SOAP
  - Messaging with domain events
- Possible
  - Integration through libraries (e.g. JAR file, node modules) in an automated environment
    - Artefact Repository (Artefactory) / Dependency Management Tool (maven, gradle)
- Antipattern
  - Database Integration
    - In this case it is essential to use the Anticorruption Layer, Open Host Service and Published Language.

Bounded Context and Context Mapping using the example of a vehicle

- Example Automotive Group
- Each (large) unit has a different perspective on the business object vehicle



A bounded context helps to find the granularity of a microservice

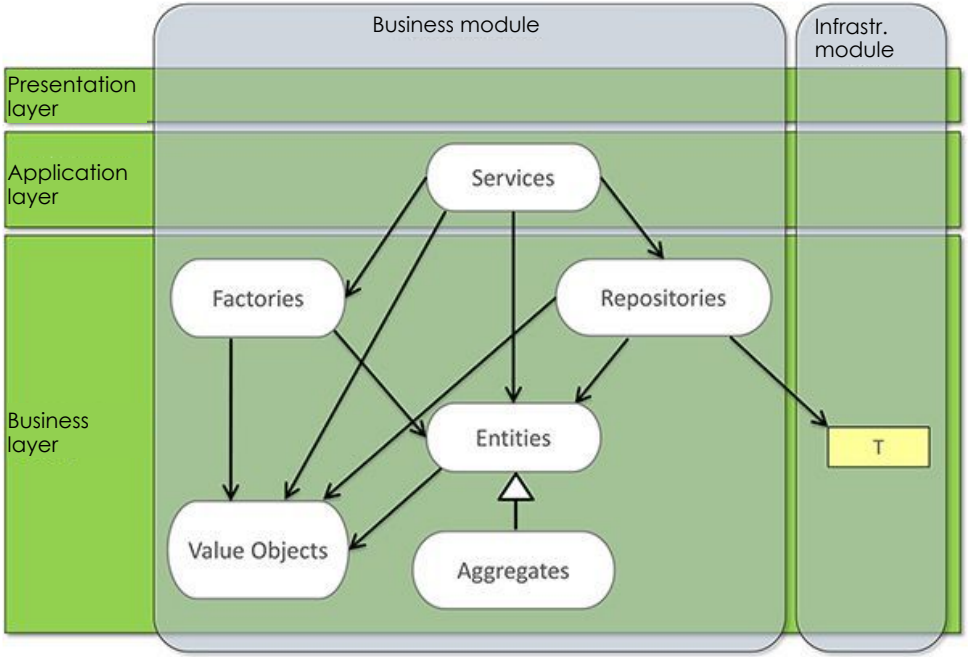


Overview

- Introduction
- DDD Strategic Design
- DDD Tactical Design / Pattern language.

Domain Driven Design - Tactical Design

- The tactical design of DDD is dedicated to the internal structures of a bounded context. A pattern language is used for this purpose!



Source: Lilienthal, Carola: Langlebige Software-Architekturen. Technische Schulden analysieren, begrenzen und abbauen. 1. Aufl. Heidelberg

Pattern languages promote modularity and the comprehensibility of software systems (1)

- A pattern language…
  - defines a **set of sample elements and rules at class level**
  - defines **class types** and **allowed relationships** between them
  - promotes comprehensibility, maintainability and structural quality (especially modularity)
  - promotes **conceptual integrity** as an elementary component for durable systems
  - refines **architectural styles** and patterns.

Pattern languages promote modularity and the comprehensibility of software systems (2)

- Why is it recommended to use a sample language? Imagine...
  - System with over 250,000 lines of code
  - Layered architecture / Microservices
  - Numerous technical modules
  - Several thousands of classes distributed on layers / services and modules
  - Development team distributed at several locations

- Architectural styles give no rule on **how to arrange** thousands of classes! Conceptual integrity at the class level can only be achieved through pattern languages!

Conceptual integrity

- Similar application of solution patterns and principles for building module- and class-structures, so that the same concept ideas are reflected in all structures
  - "An important point is above all the preservation of conceptual integrity: if possible, you should **break down all parts according to similar aspects** and apply (and best document) this concept consistently." [Gharbi 2018]
  - "Software architects pursue conceptual integrity (also called consistency): All software design should **follow a consistent style**. In particular, **similar tasks in systems should be solved similarly**. This facilitates understanding and long-term development." [Strong 2015]
  - "Lack of conceptual integrity: **Identical problems are solved differently** within a system, there are several different, sometimes contradictory approaches to solving them". [Strong 2015]

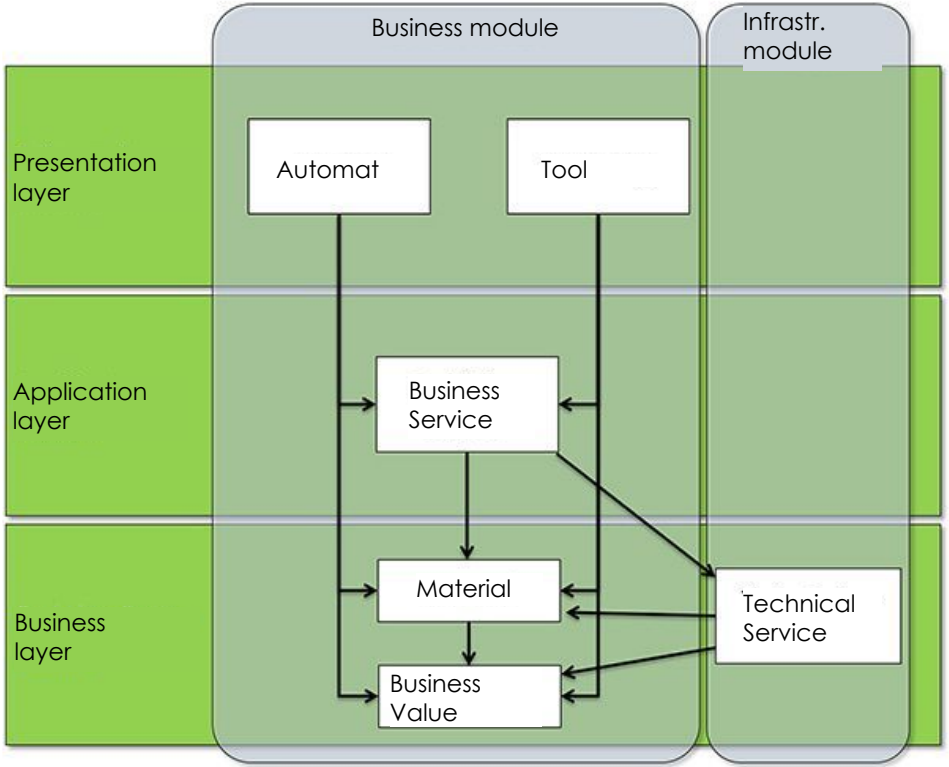
Rules of a Pattern Language

- Element rules
  - Definition of an element type (module & interface)
  - Example: Services are stateless classes
- Bidding rules
  - Rules for relationships between elements
  - Example: Services work on business objects and change their status
- Prohibition rules
  - Prohibited relationships between element types
  - Example: Business objects may not use services.

Domain Driven Design Pattern Language

- Entity
  - Persistent core objects of the business domain with identity
  - State is mapped with Value Objects
- Aggregates
  - Root entity that encapsulates entities and Value Objects
- Value Object
  - No own identity
  - May consist of other Value Objects, but not of entities
- Service
  - Logic, procedures and processes
  - Results are represented by Entities and Value Objects
- Repository
  - Encapsulation of technical details of the infrastructure
  - Procurement of object references for Entities
- Factory
  - Creation of Aggregates, Entities and Value Objects
  - Not necessarily the factory of the GoF.

Tool Automat Material Pattern Language



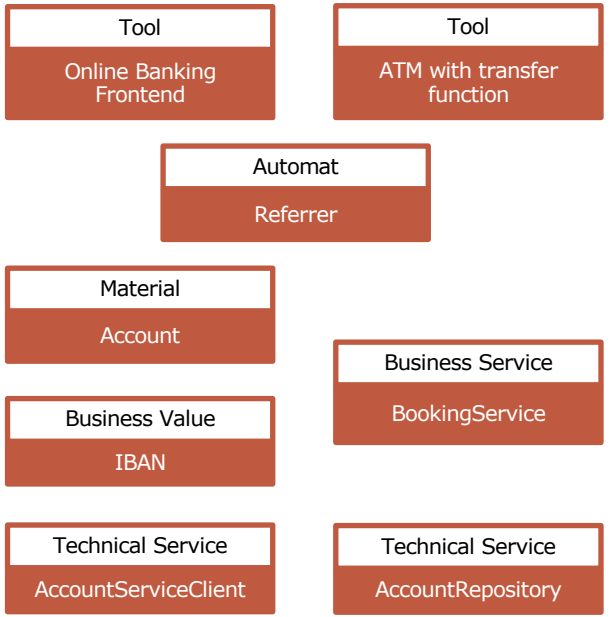
Source: Lilienthal, Carola: Langlebige Software-Architekturen. Technische Schulden analysieren, begrenzen und abbauen. 1. Aufl. Heidelberg

Tool Automat Material Pattern Language

- Automat
  - Recurring routine activities
  - Automated sequence of work steps
- Tool
  - Means to carry out a professional activity
  - Frontend, Device, etc.
- Business Value
  - User-defined values of the domain
  - No own identity
  - Value Object from DDD
- Business Service
  - Technical knowledge of a domain that cannot be represented in materials
  - Bundling of matching materials
  - Processes and services for tools and automats
  - Service in DDD.

Tool Automat Material Pattern Language

- Material
  - Objects and concepts that are part of a work result
  - Embody the domain's technical concept with identity and ensure technical consistency
  - Entity in DDD
- Technical Service
  - Capsules technical interfaces
  - Procure materials and business values from the database or from other systems
  - Repository in DDD.



Rules of pattern languages (examples)

Rule	Domain Driven Design	Tool Automat Material
Element rules	Value Objects are immutable	Business Values are immutable
	Services are stateless	Business Services are stateless
Bidding rules	Services have the exclusive task of changing the status of Aggregates/Entities or triggering their persistence in Repositories, while the programmatic creation of Aggregates and Entities is the task of Factories	Tools process Materials
	An aggregate forms the bracket for a Transaction, Domain Event and Service	Automates run without the need for further intervention
Prohibition rules	The persistence of data and access to persistent data is done exclusively by Repositories	Technical Services are reserved for exclusive access to infrastructure, such as databases with persistent data or file systems
	Factories must not be visible to the outside world. Use of Factories may only be realized through Services	Business Values have no identity.

Model languages bring business to the fore

- Packages divided according to **technical aspects reduce clarity** and distribute classes that belong together technically across the entire application
  - For medium to large systems, a functional structure is preferable
  - For small systems, technical structuring may be easier
- **Business structuring has a more advantageous effect on changeability** and expandability than technical structuring
  - Technical and business structuring is usually combined

- Microservices are a **modularization concept that focuses on the business structuring** of the application into individual services.

Conclusion

- DDD acts as a **stabilising factor** in complex software projects
- The ubiquitous language ensures a **common domain understanding**
- The domain model is the **immediate basis** for development and **remains closely linked to the code**
- Bounded Contexts help to **define delimited system parts** and their relationships to each other.

> For most software projects, the primary focus should be on the domain and the domain logic
>
> Complex domain designs should be based on a model
>
> Eric Evans

Literature

- Evans, Eric
  - Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, ISBN-13: 978-0321125217
- Vernon, Vaughn
  - Domain-Driven Design Distilled. Addison-Wesley, ISBN-13: 978-0134434421