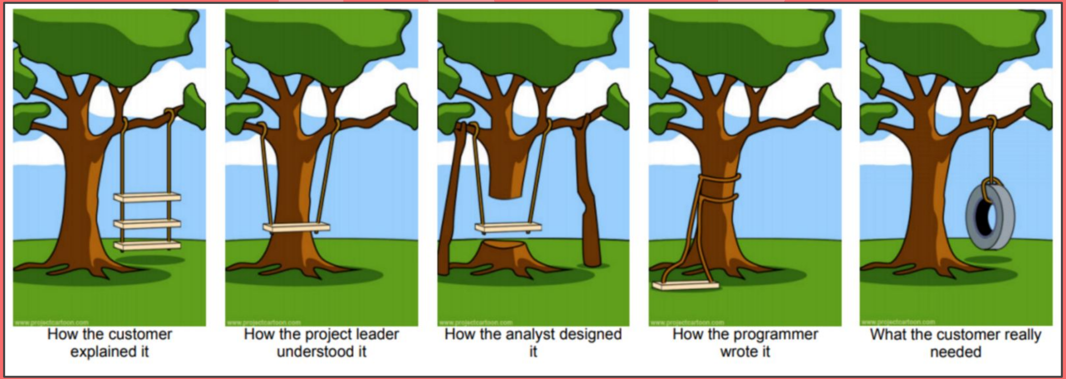
WILD CODE SCHOOL

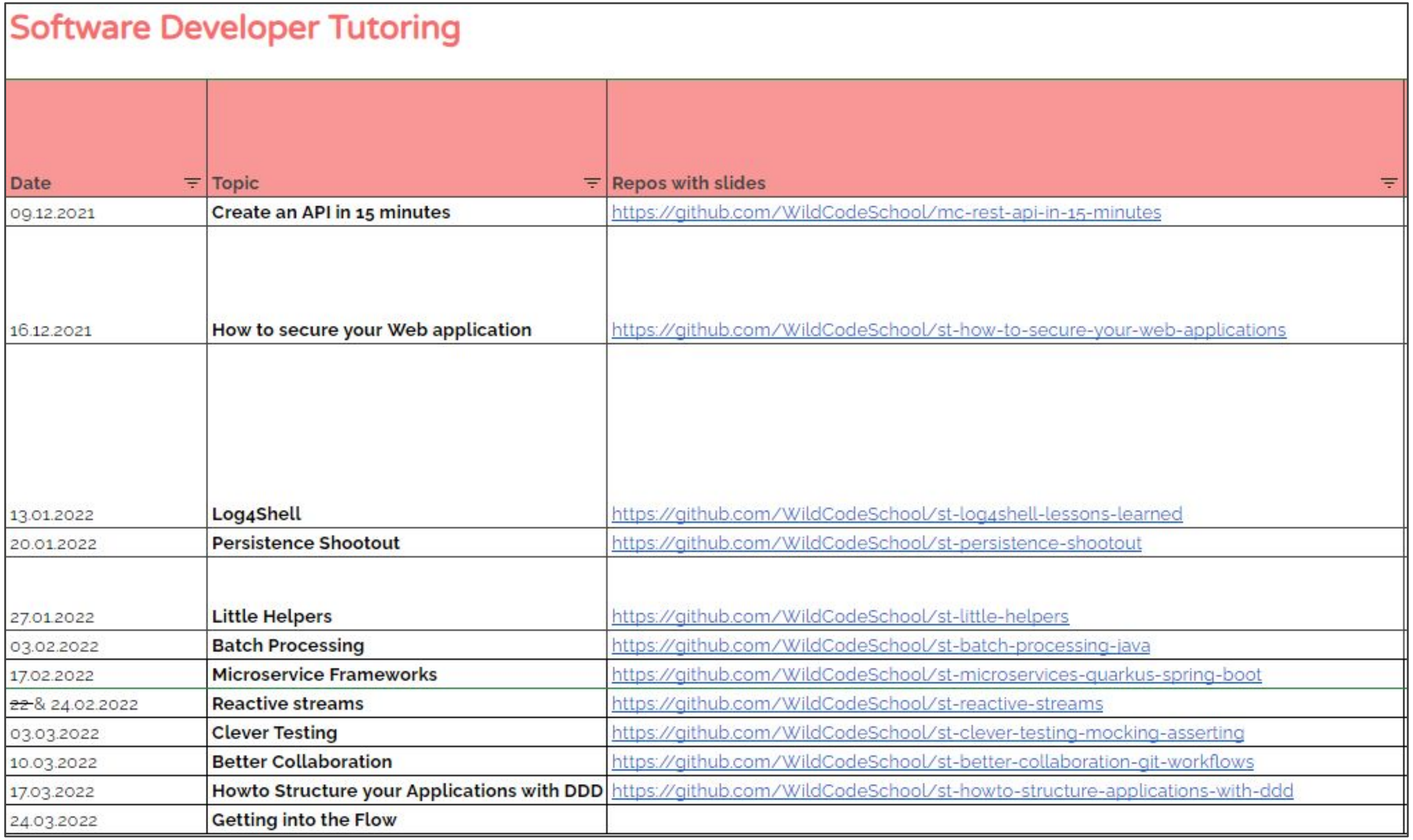
Structuring Applications with Domain-Driven Design


How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
What the customer really needed

WILD
CODE
SCHOOL

- What is Domain-Driven Design (DDD)?
- Strategic Patterns of DDD
- Tactical Patterns of DDD
- Using DDD in your Project
- Implementation Options for DDD

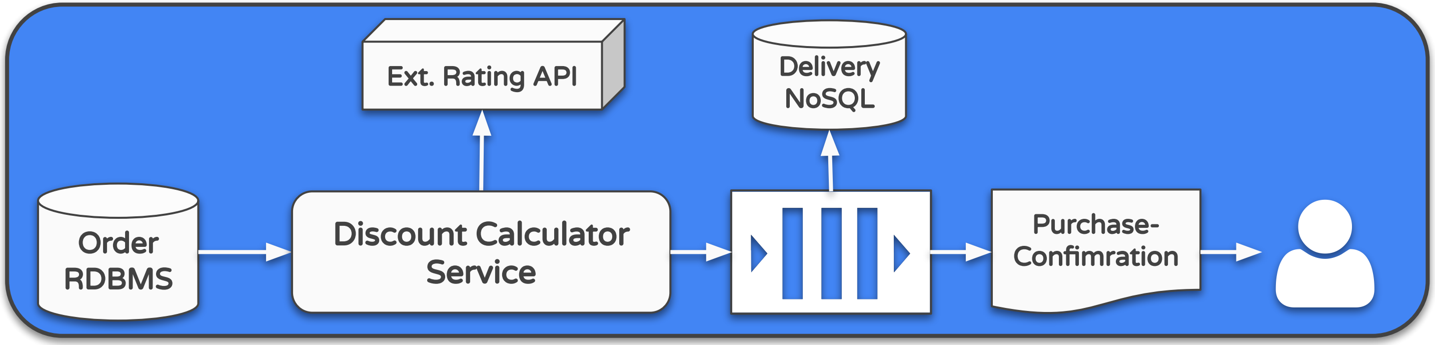
Tech People like Tech Talk…

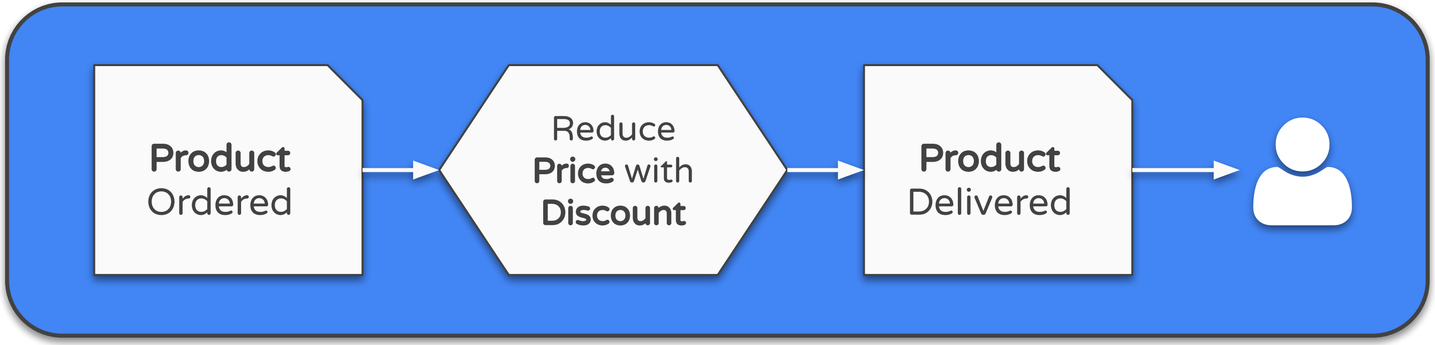


Software Developer Tutoring

Date	Topic	Repos with slides
09.12.2021	**Create an API in 15 minutes**	https://github.com/WildCodeSchool/mc-rest-api-in-15-minutes
16.12.2021	**How to secure your Web application**	https://github.com/WildCodeSchool/st-how-to-secure-your-web-applications
13.01.2022	**Log4Shell**	https://github.com/WildCodeSchool/st-log4shell-lessons-learned
20.01.2022	**Persistence Shootout**	https://github.com/WildCodeSchool/st-persistence-shootout
27.01.2022	**Little Helpers**	https://github.com/WildCodeSchool/st-little-helpers
03.02.2022	**Batch Processing**	https://github.com/WildCodeSchool/st-batch-processing-java
17.02.2022	**Microservice Frameworks**	https://github.com/WildCodeSchool/st-microservices-quarkus-spring-boot
~~22~~ & 24.02.2022	**Reactive streams**	https://github.com/WildCodeSchool/st-reactive-streams
03.03.2022	**Clever Testing**	https://github.com/WildCodeSchool/st-clever-testing-mocking-asserting
10.03.2022	**Better Collaboration**	https://github.com/WildCodeSchool/st-better-collaboration-git-workflows
17.03.2022	**Howto Structure your Applications with DDD**	https://github.com/WildCodeSchool/st-howto-structure-applications-with-ddd
24.03.2022	**Getting into the Flow**	

How does DDD help?





What is Domain-Driven Design (DDD)?

DDD is the **process** of **learning, refining, experimenting, and exploring** in the quest to **produce** an **effective model**.

It is often said that **working software** is simply **an artifact of learning**.

Placing the **project's primary focus on the core domain and domain logic**

The goal of a domain-driven design is an **alignment between the domain and the software**.

Ubiquitous Domain Language

A Ubiquitous Language **minimizes the cost of translation** and binds all expressions to the **code model** also known as the **true model**. A **shared language** also helps **collaborative exploration when modelling**, which can enable deep insights into the domain.

When modeling with stakeholders and domain experts, everyone should make a conscious effort to consistently **apply a shared language rich in domain-specific terminology**.

This language must be made explicit and be used when **describing the domain model and problem domain**.

WILD
CODE
SCHOOL

Strategic Patterns of Domain-Driven Design

Subdomains: Core, Supporting and Generic

Core

- ❖ Strategic investment in a single, well-defined domain model
- ❖ High value and priority
- ❖ *The company's secret sauce to distinguish it from competitors*

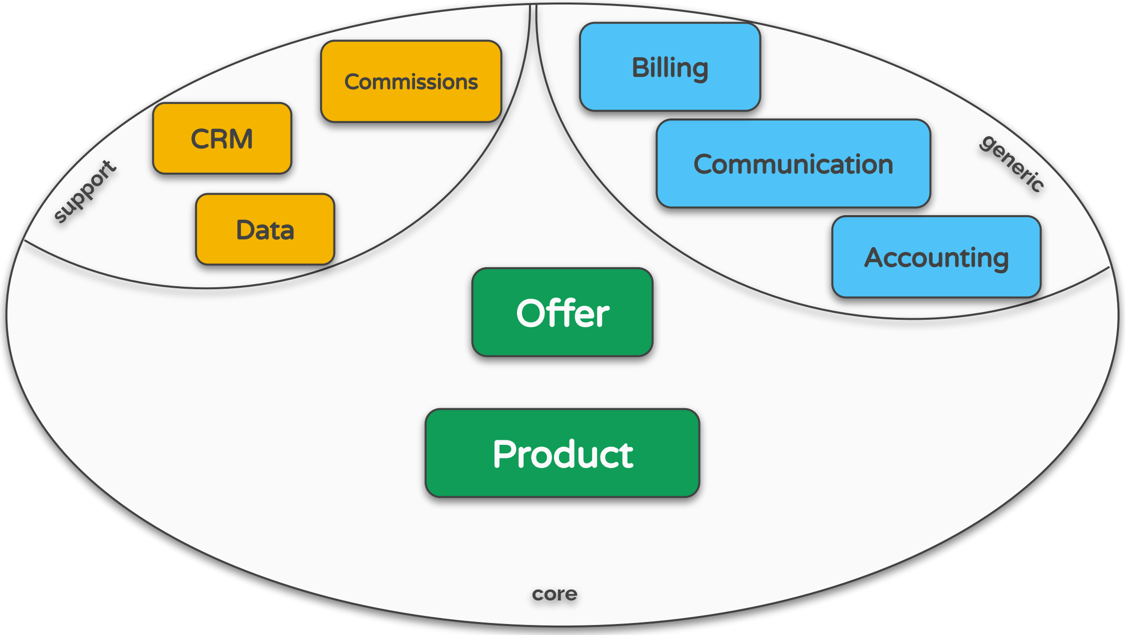
Supporting

- ❖ Custom development — no off-the-shelf solution
- ❖ Consider outsourcing development

Generic

- ❖ Purchase off-the-shelf solution
- ❖ Outsource development
- ❖ Examples: Accounting, CRM, Identity / Authentication

Subdomains (Telco Domain)



Bounded Context

- ➢ **Semantic** contextual **boundary** for a model
- ➢ Ubiquitous language is **consistent** within a bounded context
- ➢ Keep the **model strictly consistent** within these bounds
- ➢ **Separate** software **artifacts** for each bounded context

Subdomain and Bounded Context

Subdomains and **bounded contexts** are concepts that sometimes appear to be similar and can be confusing. However, both concepts can be easily understood by looking at the difference between a **domain** and **domain model**, which is probably easier to grasp.

The ***domain*** represents the **problem** to solve; the ***domain model*** is the model that implements the **solution** to the problem. Likewise, a ***subdomain*** is a segment of the problem domain, and a ***bounded context*** is a segment of the **solution**.

A subdomain in the problem space is mapped to a bounded context in the solution space.

https://www.microsoftpressstore.com/articles/article.aspx?p=2248811

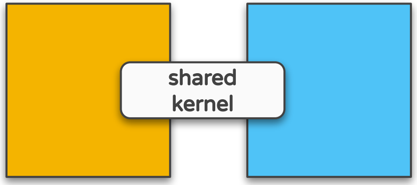
Context Integration

Define **relationship** and **translation** between bounded contexts (and ubiquitous languages)

Kinds of mappings

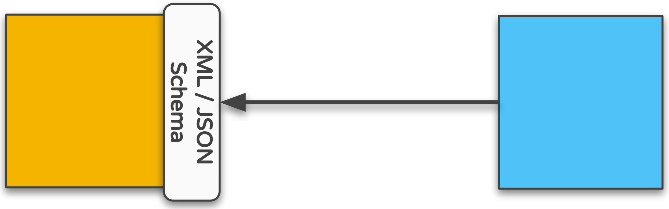
- ❖ Partnership
- ❖ Shared kernel
- ❖ Customer-supplier
- ❖ Conformist
- ❖ Anticorruption layer
- ❖ Open host service
- ❖ Published language
- ❖ Separate ways

Context Integration



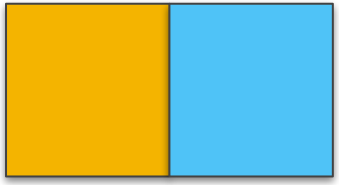
Shared Kernel

- ❖ simple if correct
- ❖ difficult to get right (versioning?)



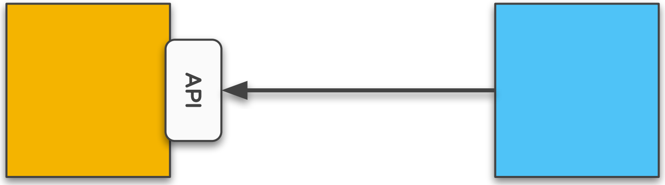
Published Language

- ❖ Well-documented information exchange language
- ❖ Enables simple consumption and translation by any number of consumers



Partnership

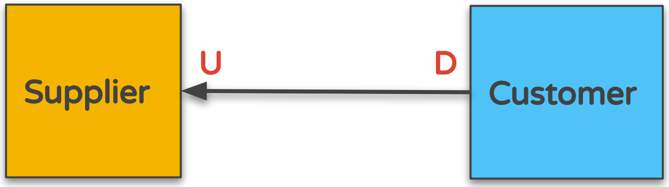
- ❖ succeed or fail as team
- ❖ communication overhead



Open Host Service

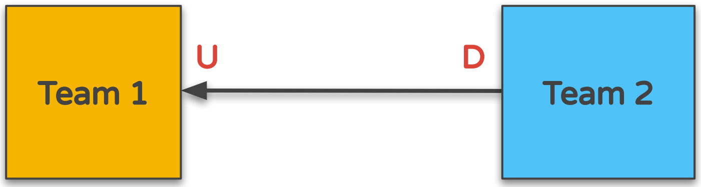
- ❖ interface or protocol that gives access to bounded context
- ❖ Well documented service API

Context Integration



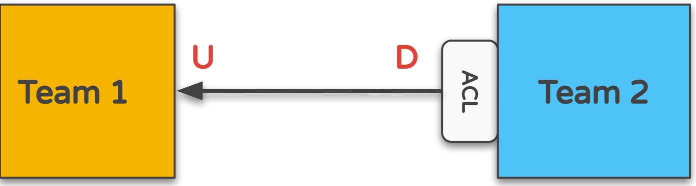
Customer-Supplier

- Supplier provides what the customer needs (but determines what & when)
- Typical relationship between teams witin an organisation



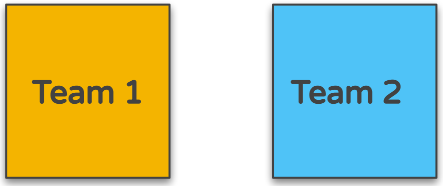
Conformist

- As customer-supplier, but no support for downstream team
- Downstream team conforms to upstreams ubiquitous language



Anticorruption Layer

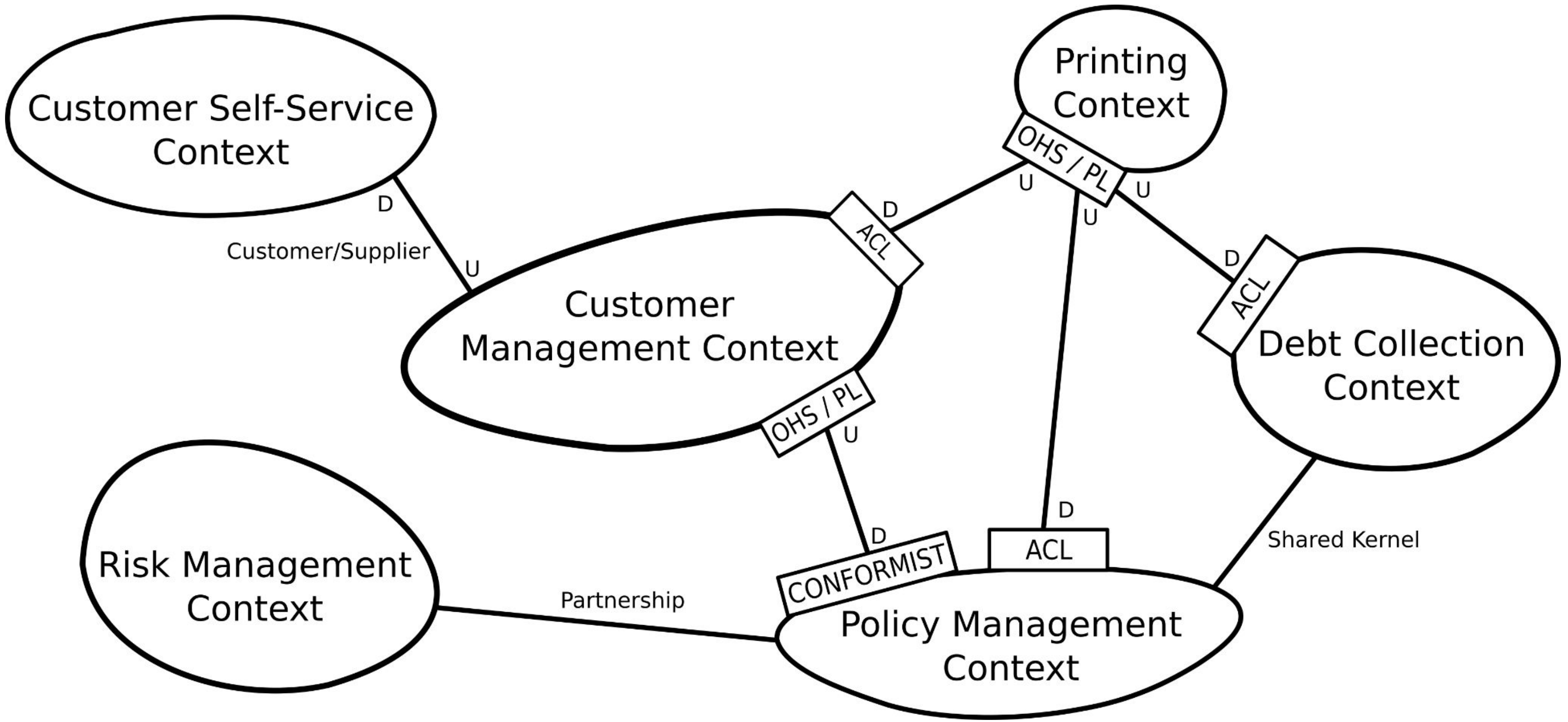
- Most defensive mapping relationship
- Downstream team creates a translation layer



Separate Ways

- simple if correct
- difficult to get right

Context Map (Sample)

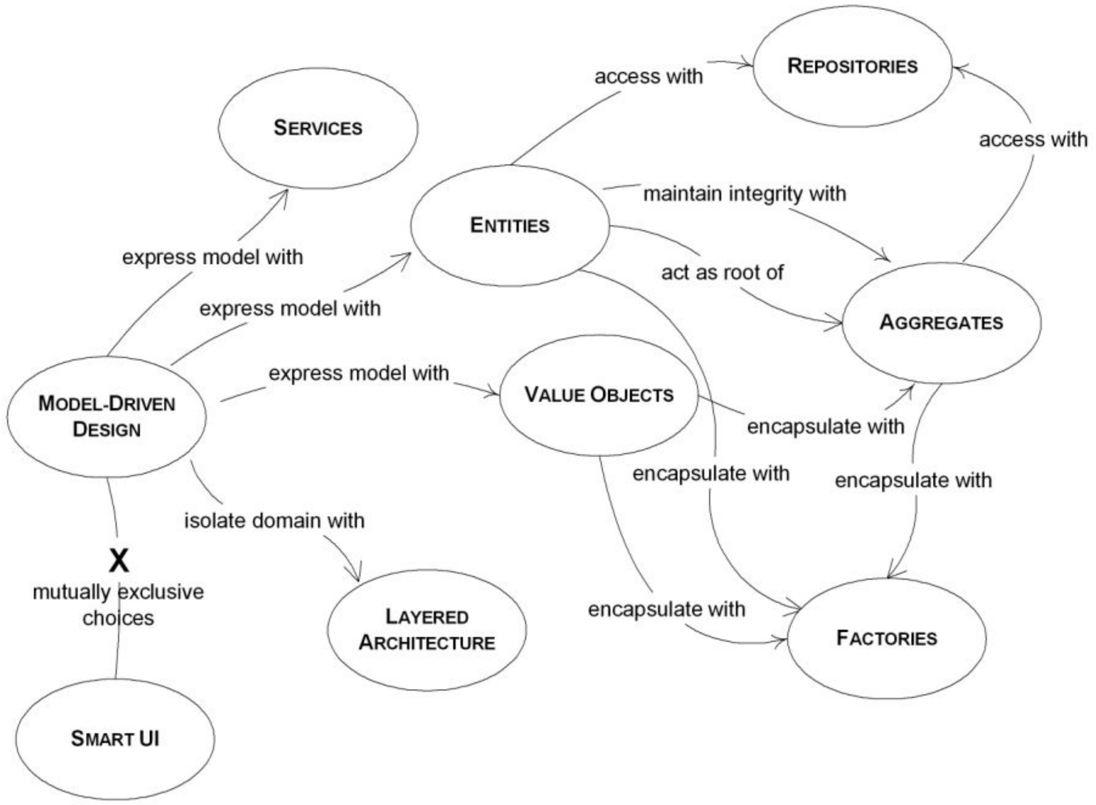


https://contextmapper.org/docs/examples/

WILD CODE SCHOOL

Tactical Patterns of Domain-Driven Design

Tactical Domain-Driven Design



http://www.infoq.com/minibooks/domain-driven-design-quickly

Entity

- Models an **individual thing**
- Has a **unique identity**
- Is **mutable** — its state changes over time
- Examples:
  - Tariff Option
  - Invoice
  - Customer

Value Object

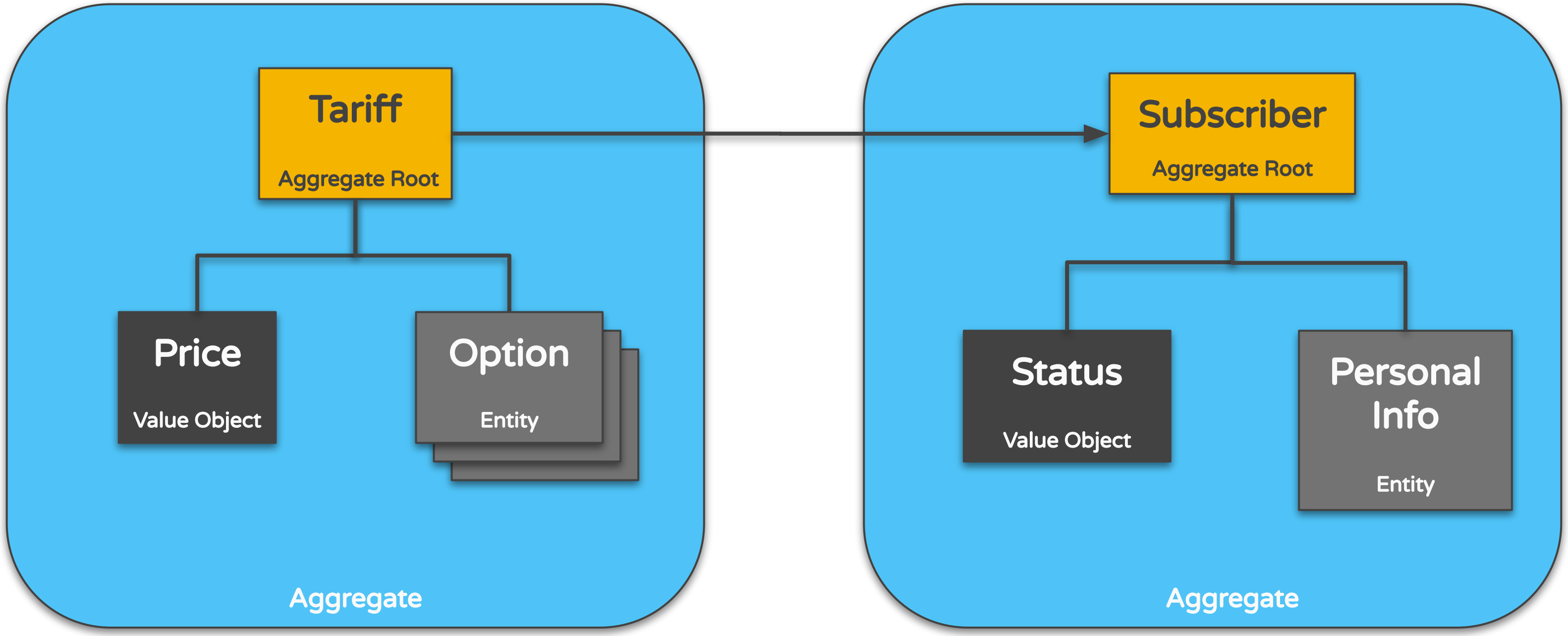
- Models just a **value**
- Doesn't have a unique identity
- Is **immutable**
- Equivalence is **determined** by **its attributes**
- Examples:
  - Address
  - Money
  - Discount Status

https://vaadin.com/learn/tutorials/ddd/ddd_and_hexagonal

Aggregate

- Composed of one or more **entities and value objects**
- Forms a **transactional consistency boundary**
- One entity is called the **aggregate root**:
  - **Owns** all other elements clustered **inside it**
  - **Access** to the aggregate **must go through** the **root entity**
- Examples:
  - Tariff
  - Customer
  - Invoice

Aggregates, Aggregate Roots, Entities & Value Objects



Aggregate

- Aggregate **enforces** transactional **consistency**
- Business **invariants must be protected** within the boundary
- Must be stored in a **whole and valid state**
- Allows **concurrent** transactions for different **aggregate instances**

Rules of Aggregate Design

- **Protect** business **invariants** inside aggregate boundaries
- Design small aggregates
- **Reference** other aggregates **by identity** only
- Update referenced aggregate using **eventual consistency**

Domain Event

- Record of some business-significant occurrence in a bounded context
- **Immutable facts**
- Named in the **past tense** using the **ubiquitous language**
- Can be used for **inter-service messaging**
- Examples:
  - TariffChanged
  - ProductDelivered
  - InvoicePaid

Service

- Contains **domain operations** that **don't belong** to an entity or value object
- Is **stateless**
- Examples:
  - TariffOptionAssignmentService
  - DiscountCalculationService
  - CurrencyConversionService

Repository

- **Store** domain objects (aggregates) into **persistence layer**
- **Retrieve** domain objects **from persistence layer**
- Examples:
    - CustomerRepository
    - TariffRespository

WILD CODE SCHOOL

Using Domain-Driven Design in your Project

Applying DDD: Event Storming

Event Storming is a flexible **workshop format** for **collaborative exploration** of complex **business domains**.

Event Storming helps to:

- ❖ **Evaluate** existing business and **discover** areas for improvements
- ❖ **Explore** the viability of a **new business model**
- ❖ **Envision new services** that help all stakeholders
- ❖ **Design** clean and maintainable **Event-Driven software**
- ❖ **Support** rapidly **evolving businesses**

Event Storming allows sophisticated **cross-discipline conversation** between stakeholders with different backgrounds, delivering a new type of collaboration **beyond silo and specialisation boundaries**.

Applying DDD: Domain Storytelling

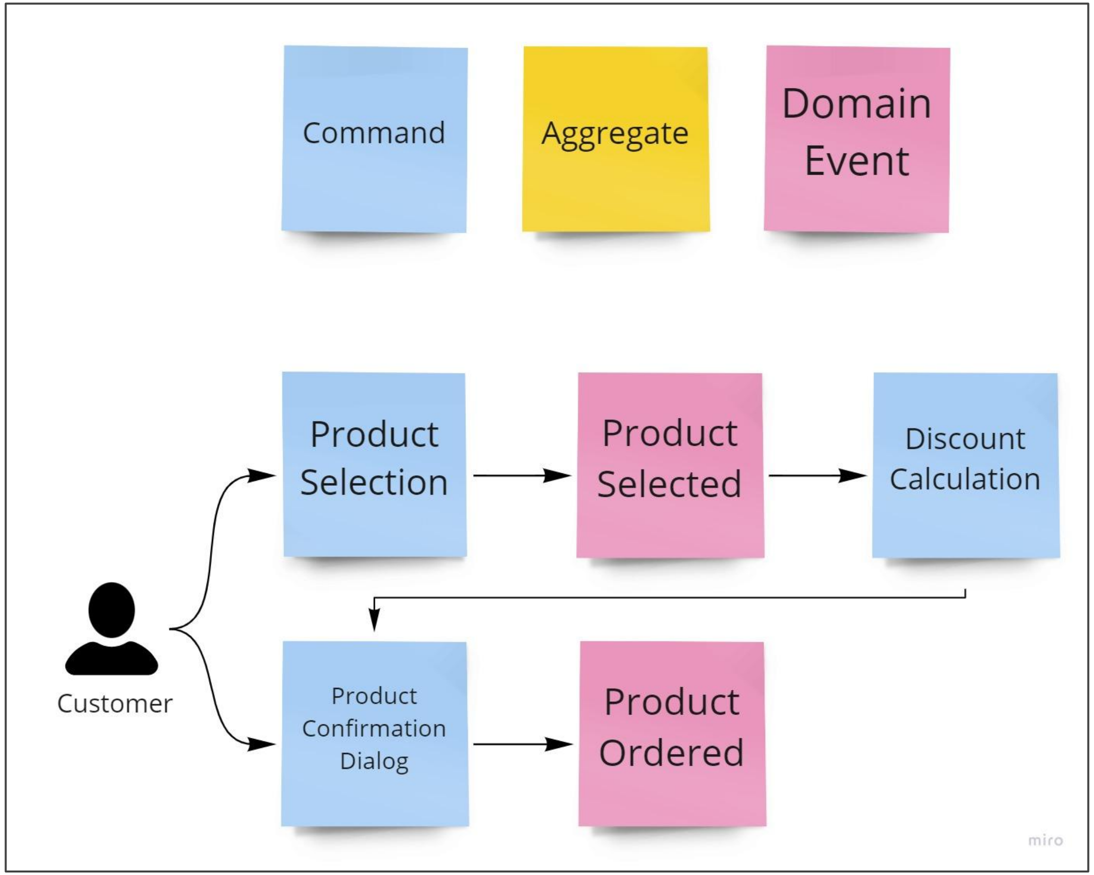
Domain Storytelling is a technique to **transform domain knowledge** into **effective** business **software**.

Domain Storytelling helps you to:

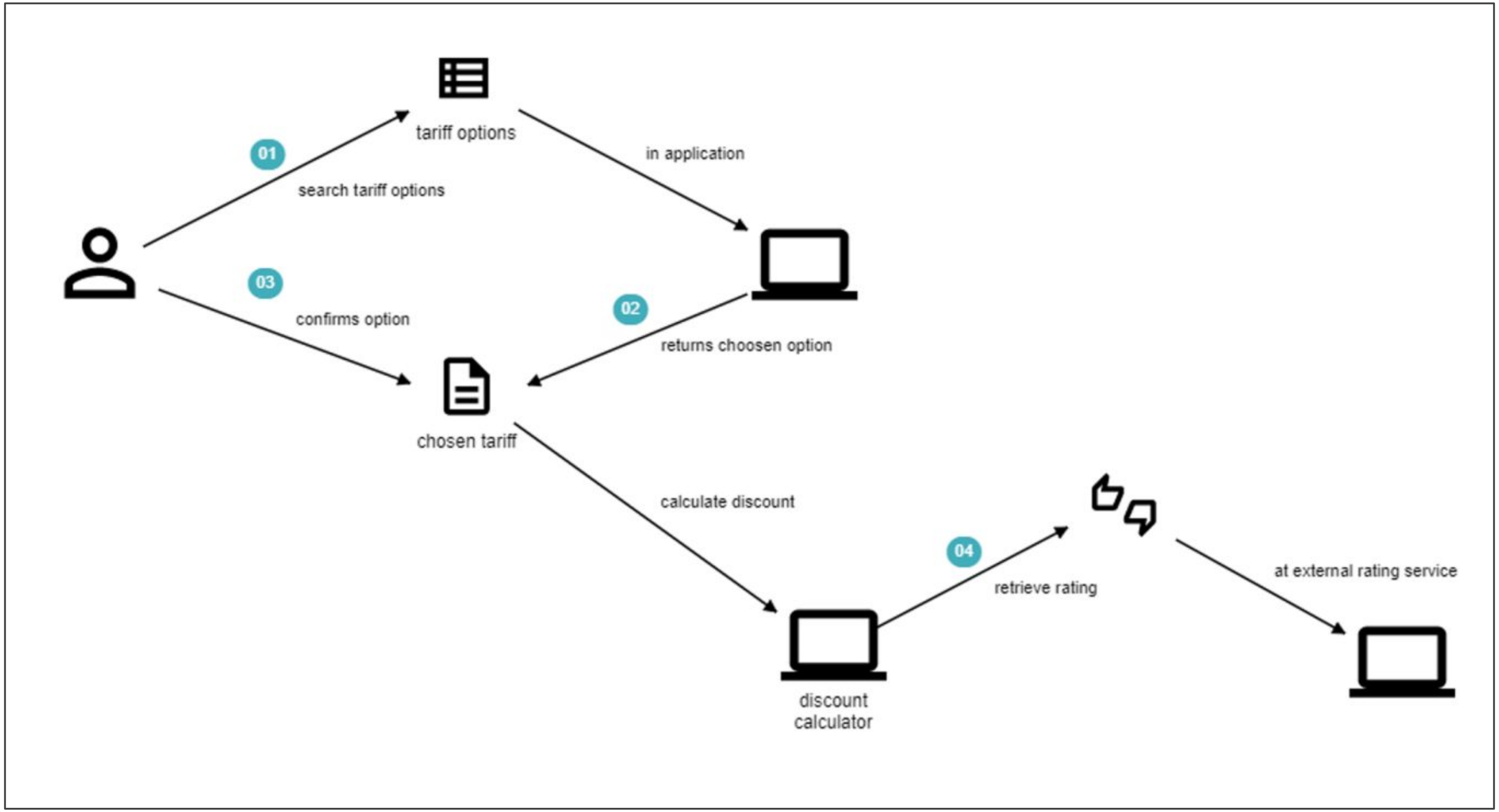
- ❖ Fully **align all project participants** and stakeholders, both technical and business-focused
- ❖ **Draw clear boundaries** to organize your domain, software, and teams
- ❖ **Transform domain knowledge** into **requirements**, embedded naturally into an agile process
- ❖ Gain better **visibility into your IT landscape** so you can consolidate or optimize it

Domain Storytelling brings together **domain experts** and **development teams**. The domain experts can assess **immediately** whether there is **correct shared knowledge** with the development team.

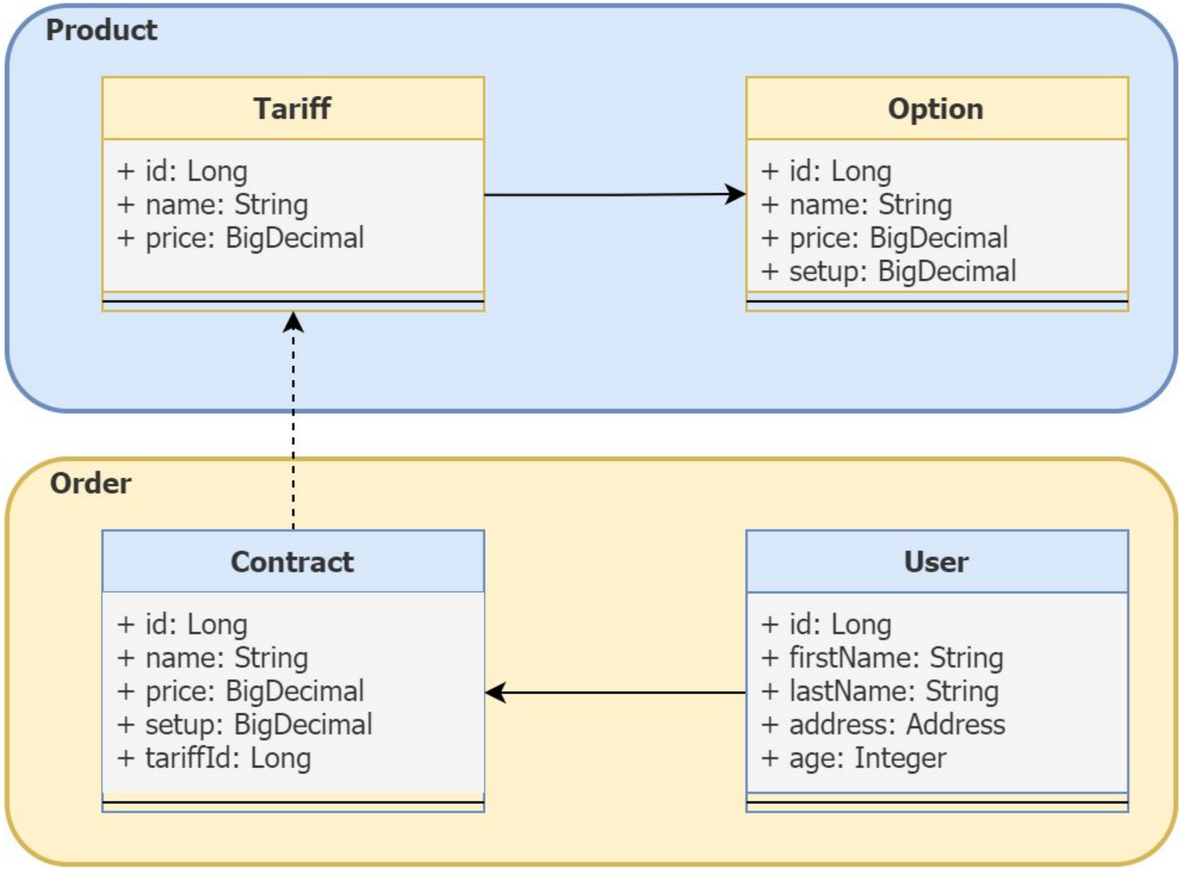
Applying DDD: Event Storming

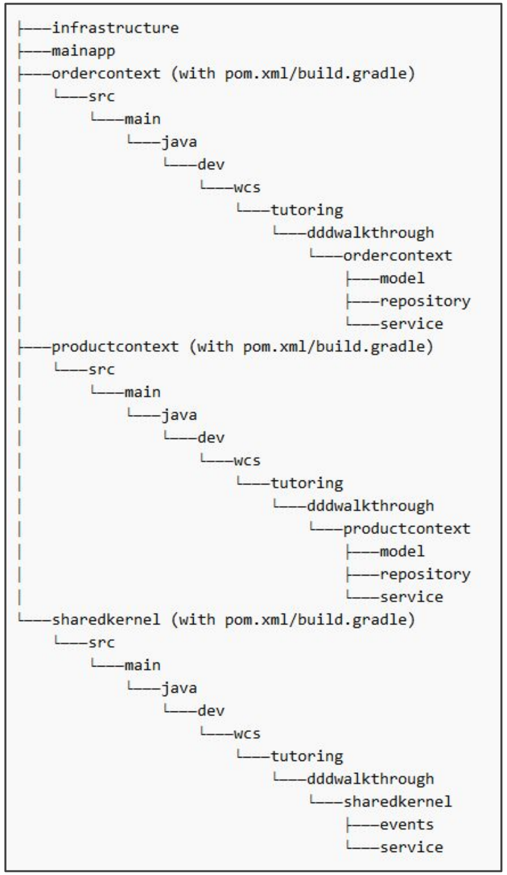


Applying DDD: Domain Storytelling



Applying DDD: DS / ES -> Domain Model



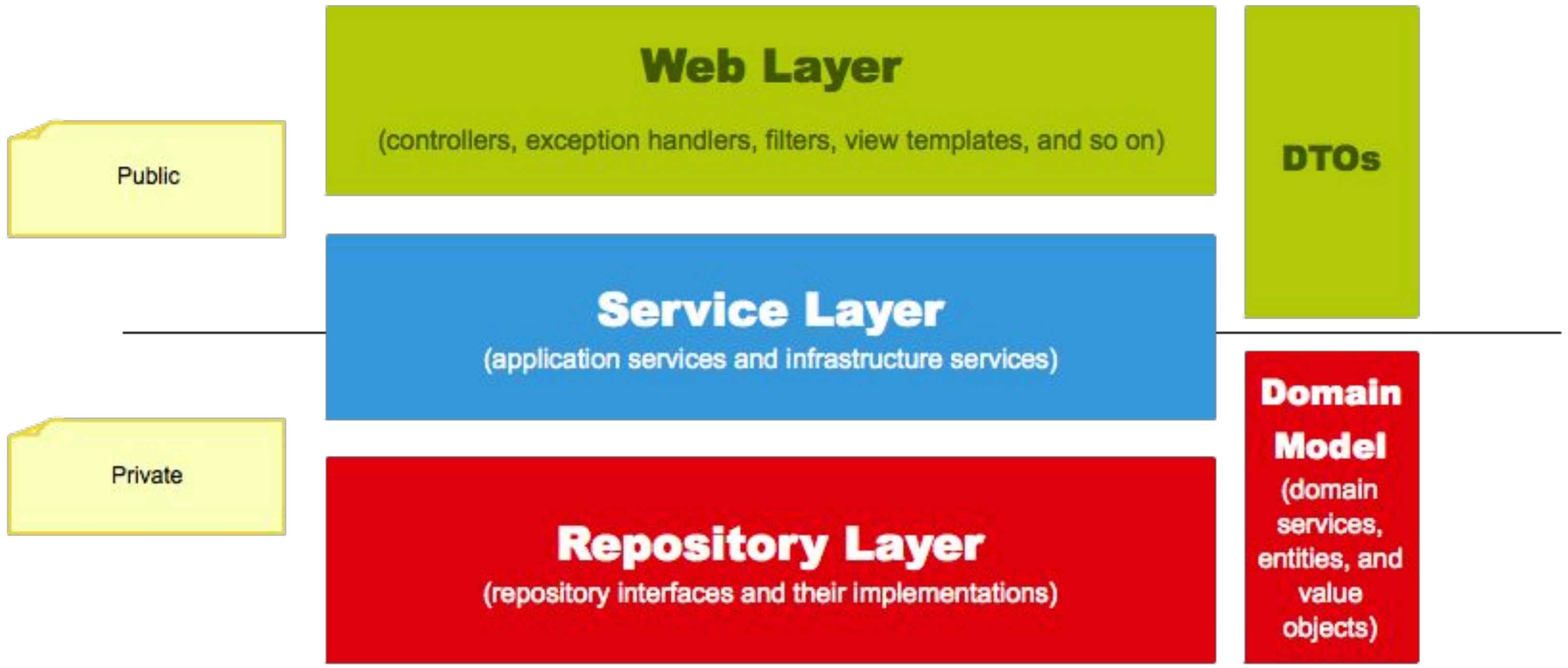


```
├───infrastructure
├───mainapp
├───ordercontext (with pom.xml/build.gradle)
│   └───src
│       └───main
│           └───java
│               └───dev
│                   └───wcs
│                       └───tutoring
│                           └───dddwalkthrough
│                               └───ordercontext
│                                   ├───model
│                                   ├───repository
│                                   └───service
├───productcontext (with pom.xml/build.gradle)
│   └───src
│       └───main
│           └───java
│               └───dev
│                   └───wcs
│                       └───tutoring
│                           └───dddwalkthrough
│                               └───productcontext
│                                   ├───model
│                                   ├───repository
│                                   └───service
└───sharedkernel (with pom.xml/build.gradle)
    └───src
        └───main
            └───java
                └───dev
                    └───wcs
                        └───tutoring
                            └───dddwalkthrough
                                └───sharedkernel
                                    ├───events
                                    └───service
```

WILD CODE SCHOOL

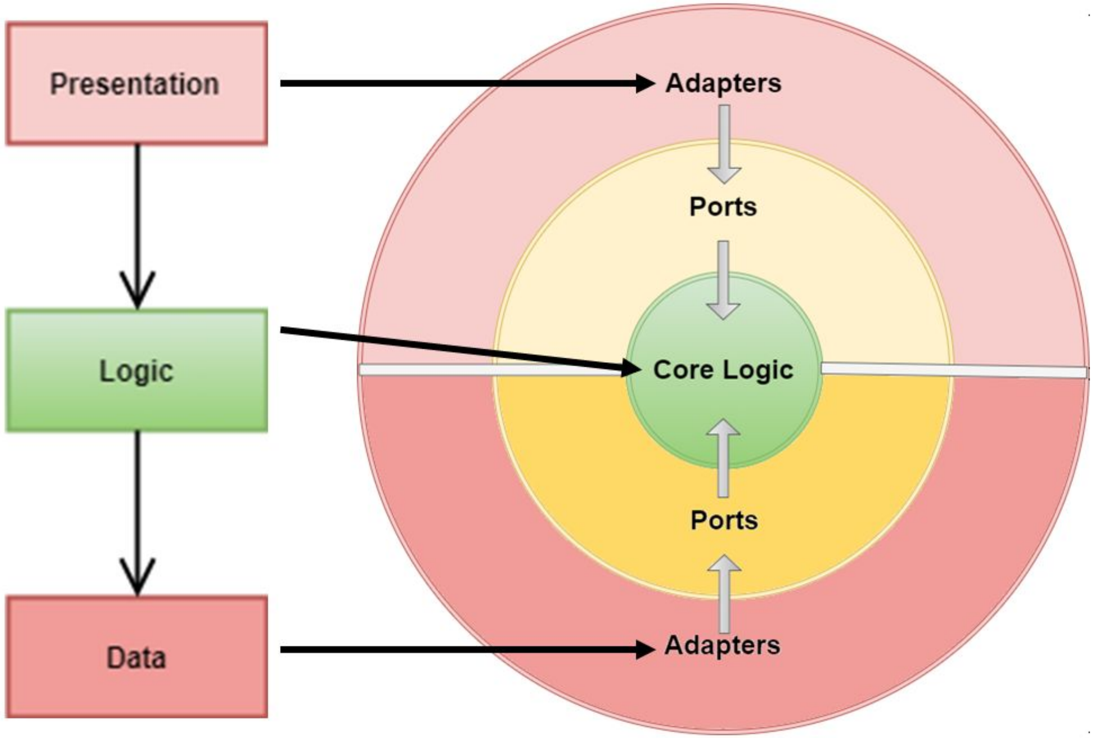
Implementation Options for Domain-Driven Design

Traditional Layered Architecture


Web Layer
(controllers, exception handlers, filters, view templates, and so on)
DTOs
Public
Service Layer
(application services and infrastructure services)
Private
Repository Layer
(repository interfaces and their implementations)
Domain Model
(domain services, entities, and value objects)

https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/

Hexagonal (Ports & Adapters) vs Layered Architecture

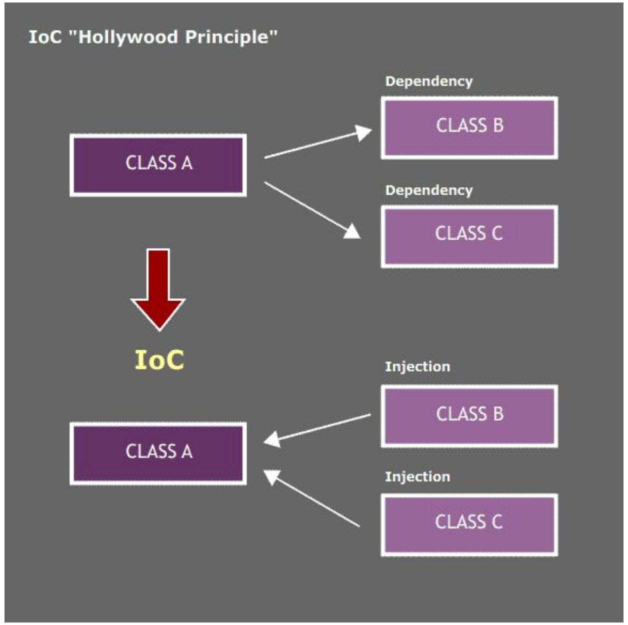


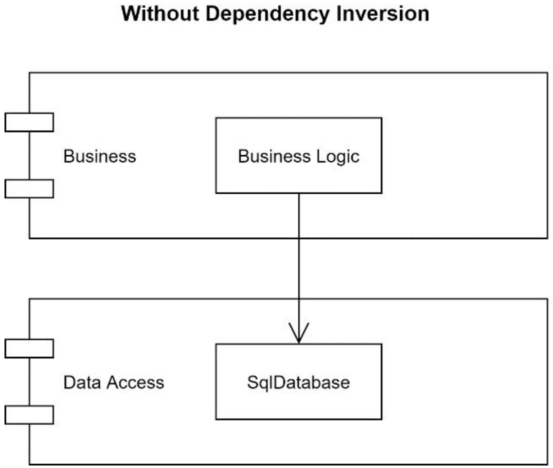
https://www.maibornwolff.de/blog/ddd-architekturen-im-vergleich

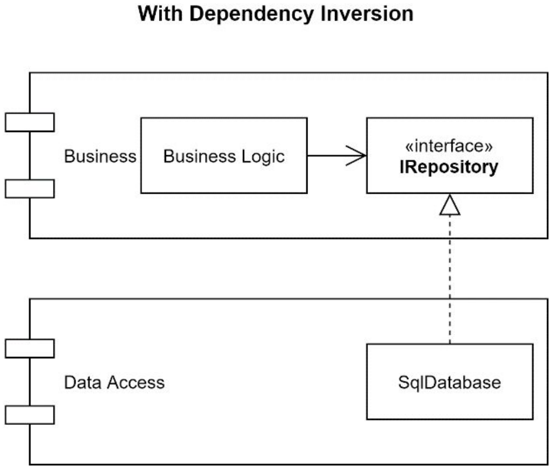
WILD CODE SCHOOL

Implementing DDD in Modern Development Environments

Inversion of Control / Dependency Injection (IoC/DI)



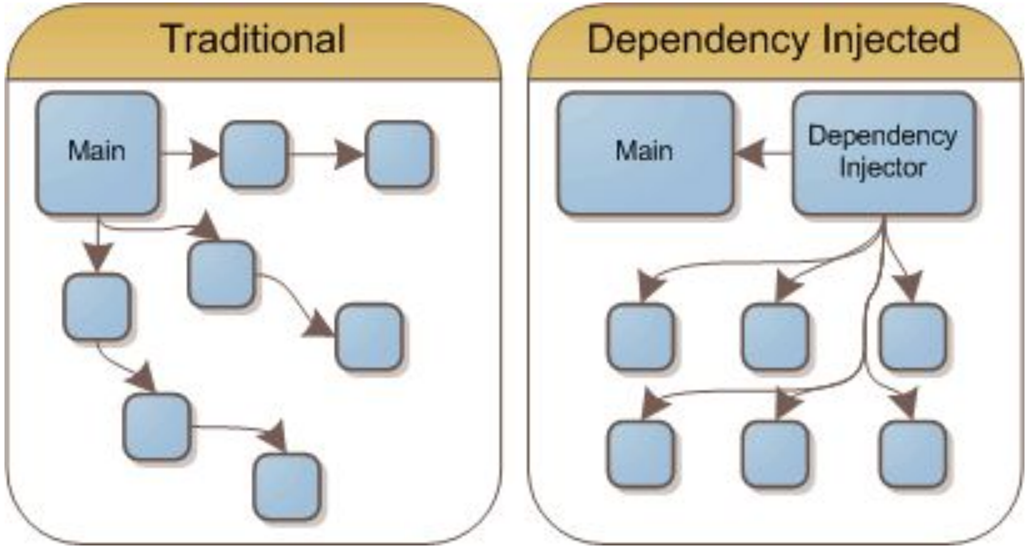




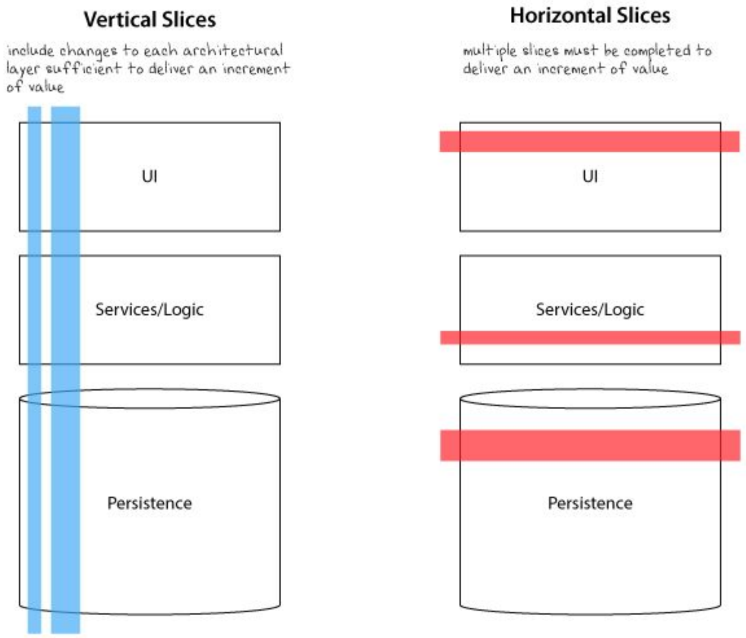
https://medium.com/analytics-vidhya/dependency-injection-concept-of-spring-framework-d9c3688005f8

Dependency Injection (DI)

In Spring, Jakarta EE (CDI) or Quarkus (CDI), control inversion is implemented by **injecting dependencies**.



Modelling: Layers vs Slices

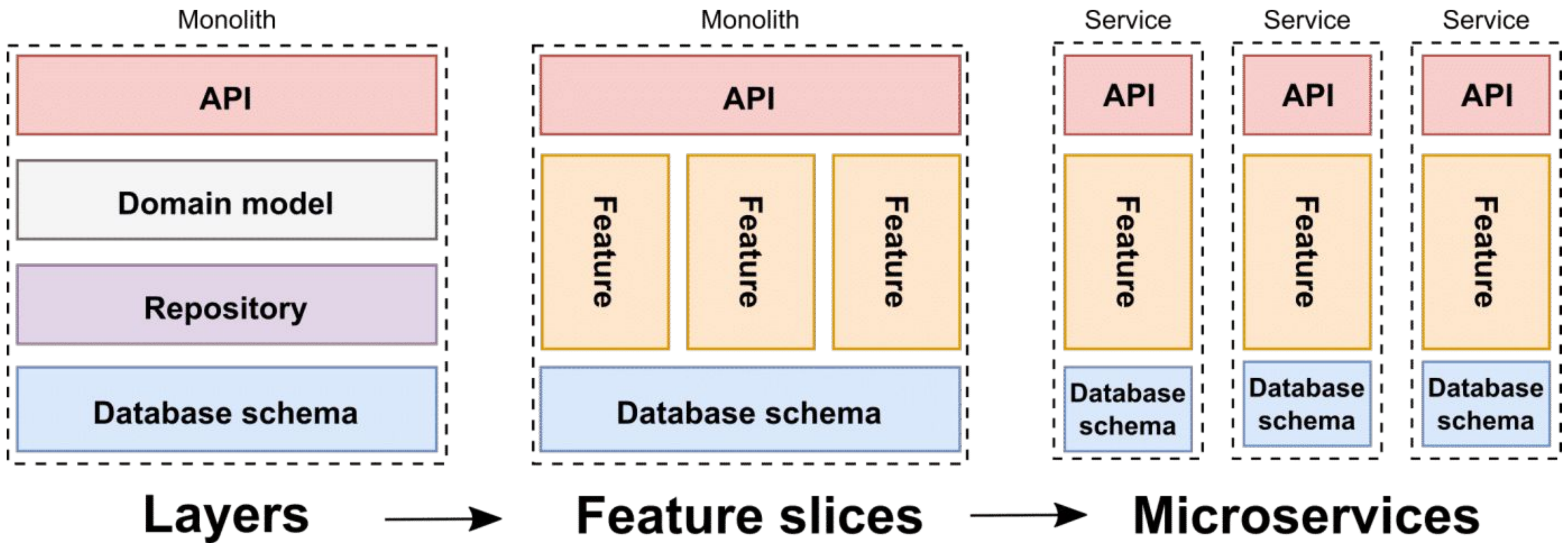

Vertical Slices
include changes to each architectural layer sufficient to deliver an increment of value
UI
Services/Logic
Persistence
Horizontal Slices
multiple slices must be completed to deliver an increment of value
UI
Services/Logic
Persistence

```
+- example (Layer)
        |
        +- controller
        |  +- ProductController.java
        |  +- OrderController.java
        |  +- CustomerController.java
        |
        +- dao
        |   +- ProductRepository.java
        |   +- OrderRepository.java
        |   +- CustomerRepository.java
```

```
+- example (Slices)
        |
        +- product
        |   +- ProductController.java
        |   +- ProductService.java
        |   +- ProductRepository.java
        |
        +- order
            +- OrderController.java
            +- OrderService.java
            +- OrderRepository.java
```

https://www.humanizingwork.com/vertical-slices-and-scale/

Modelling: Layers vs Slices (Monolith vs Microservices)

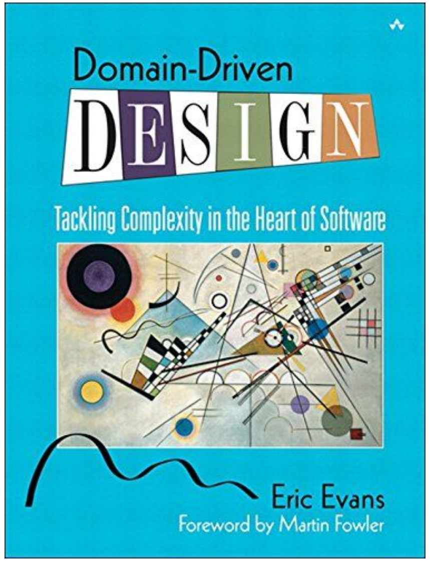


https://www.betterask.erni/news-room/slices-vs-layers/

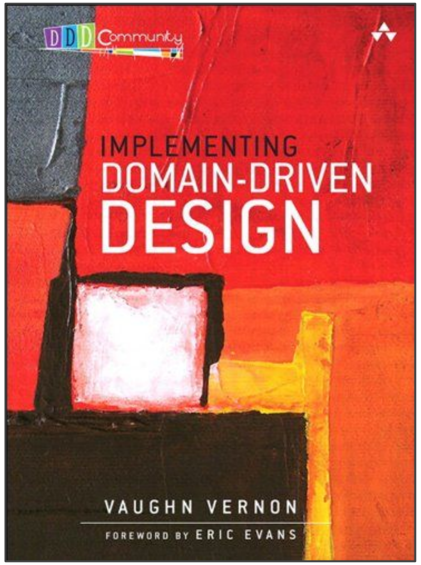
WILD CODE SCHOOL

More Information to Domain-Driven Design

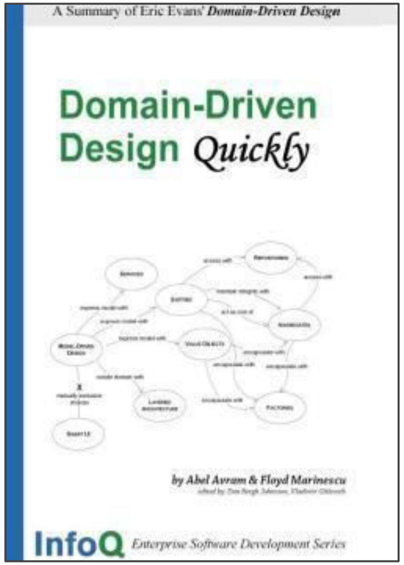
Reference Books on DDD


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

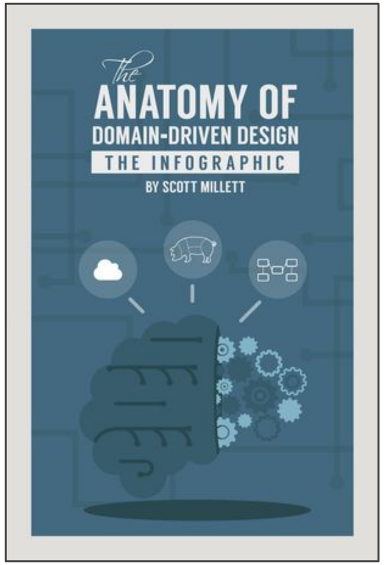
Reference Book inventing DDD



More Practical, but complete Book


A Summary of Eric Evans' Domain-Driven Design
Domain-Driven
Design Quickly
by Abel Avram & Floyd Marinescu
InfoQ Enterprise Software Development Series

Glossary and Distilled DDD (free)



Overview of DDD-Concepts

Links and other information

Learn DDD

- **Detailed DDD Introduction:** https://vaadin.com/learn/tutorials/ddd/strategic_domain_driven_design
- **Traps in DDD with Java:** http://scabl.blogspot.com/p/advancing-enterprise-ddd.html
- **xMolecules/jMolecules:** https://github.com/xmolecules/jmolecules

Apply DDD

- **Domain Storytelling:** https://domainstorytelling.org/
- **Event Storming:** https://www.eventstorming.com/
- **WPS Modeler:** https://egon.io/
- **Context Mapper with C4:** https://structurizr.com/
- **The Perfect Greenfield:** https://github.com/buschmais/The-Perfect-Greenfield
- **Comparison Domain Storytelling & Event Storming (German):** https://www.innoq.com/de/blog/vergleich-event-storming-und-domain-storytelling/