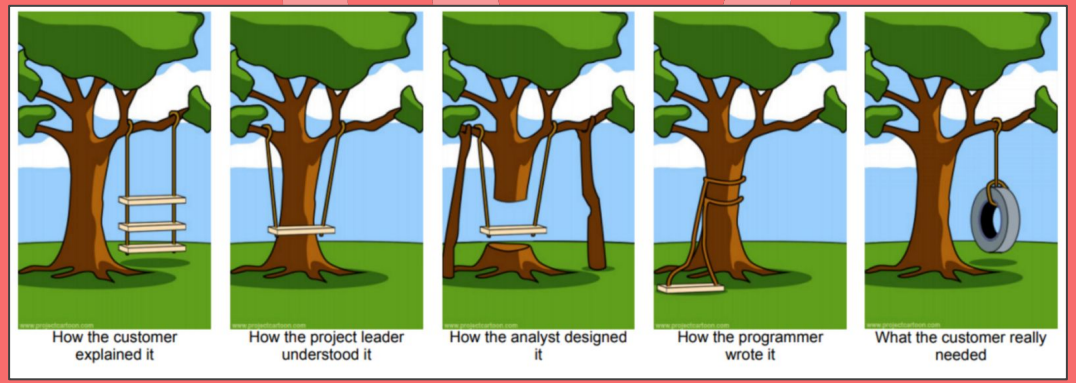
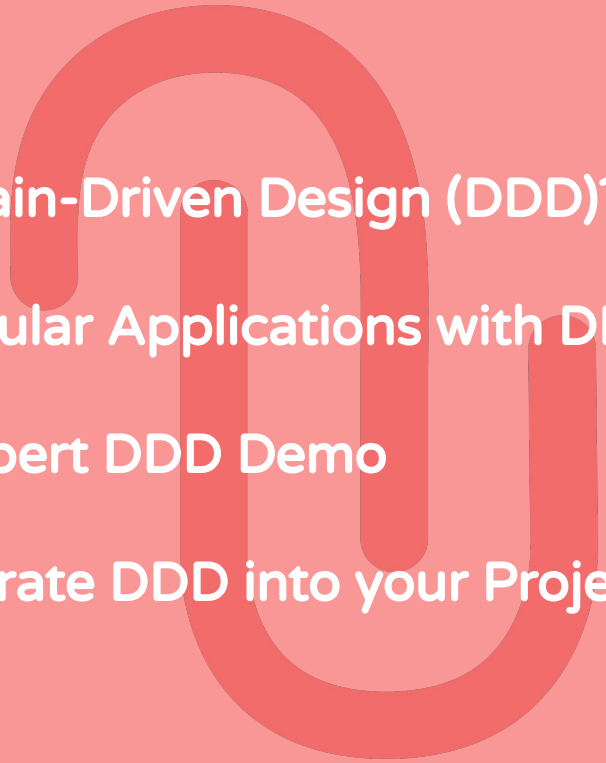
WILD CODE SCHOOL

Structuring Applications with Domain-Driven Design



WILD CODE SCHOOL



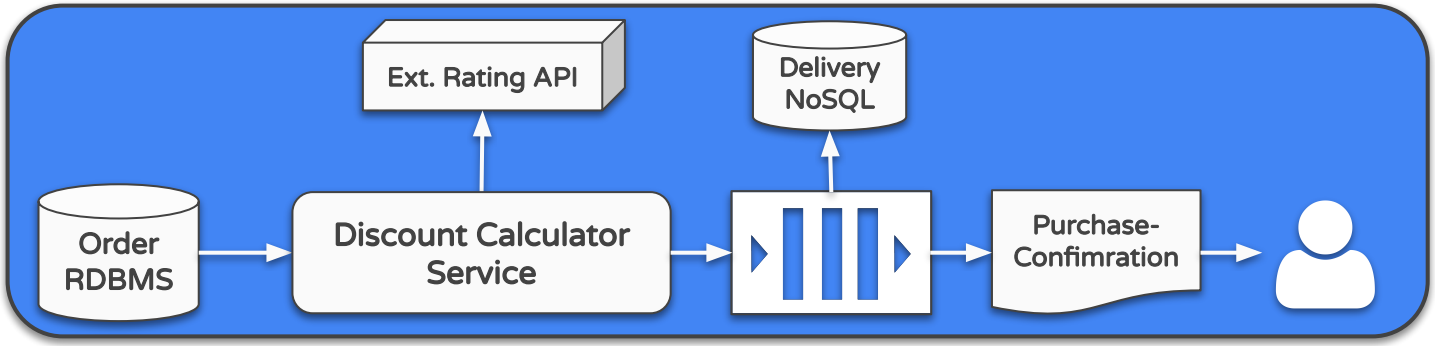
- What is Domain-Driven Design (DDD)?
- Building modular Applications with DDD
- The Tariff-Expert DDD Demo
- How-To integrate DDD into your Project

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**	

Ubiquitous Domain Language



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**.

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

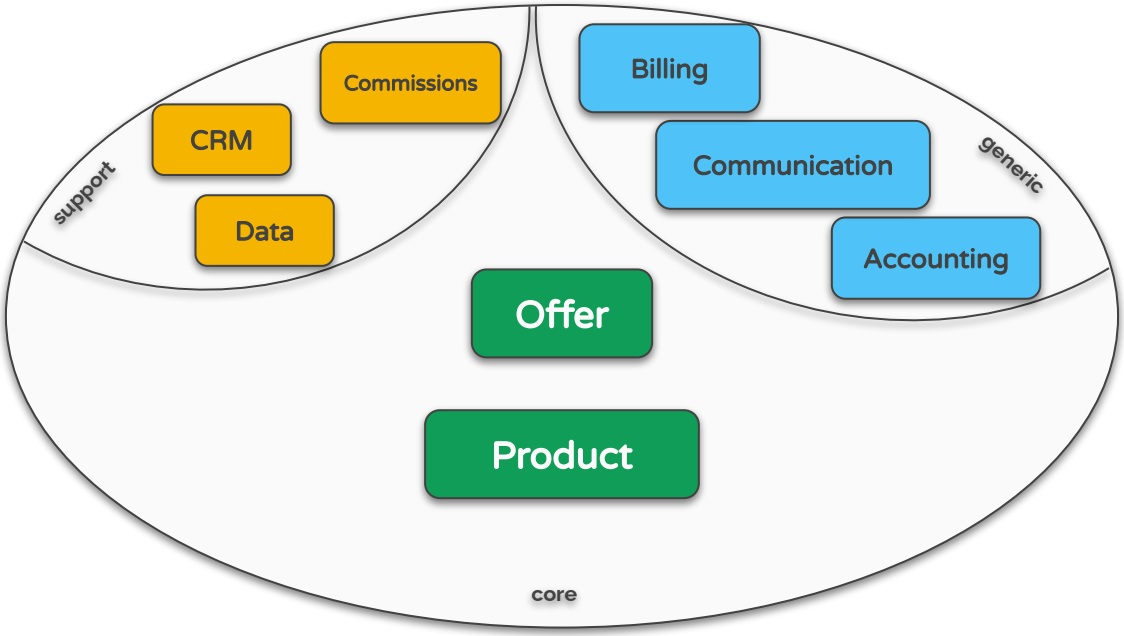
Supporting

- 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

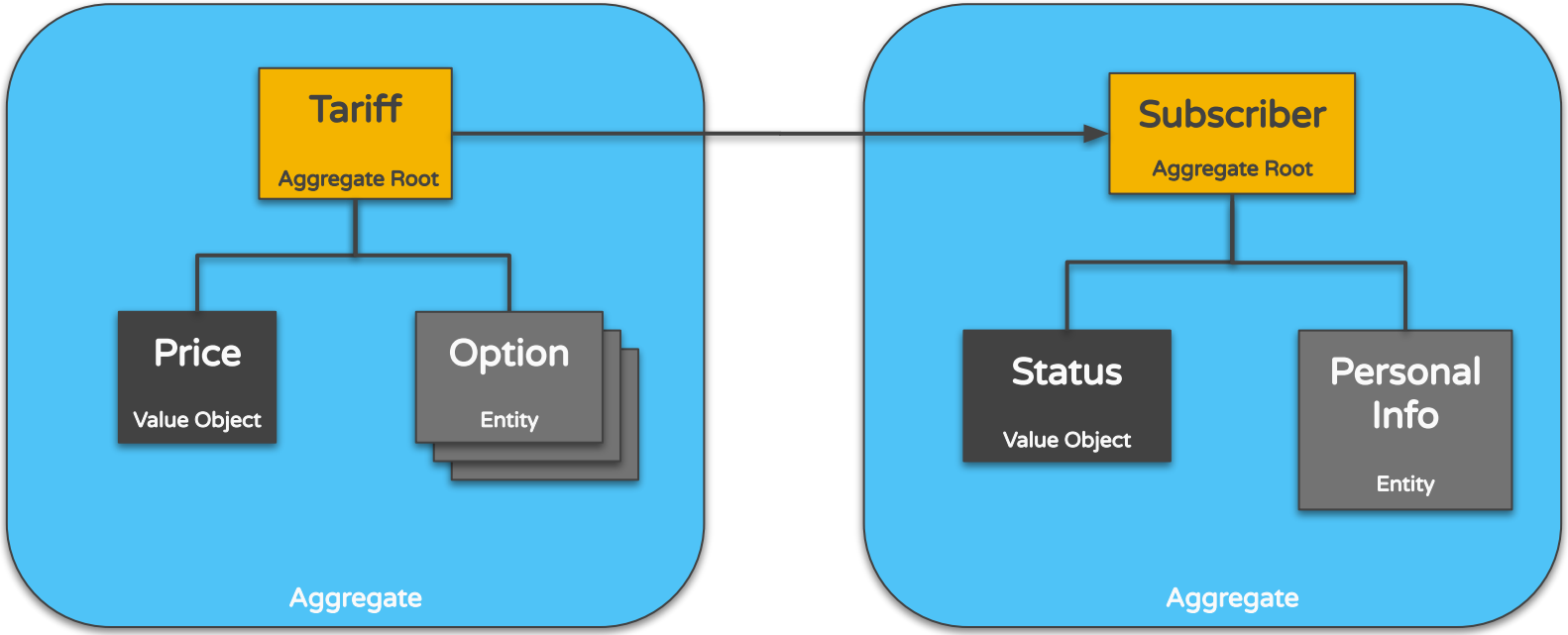
Context Map

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

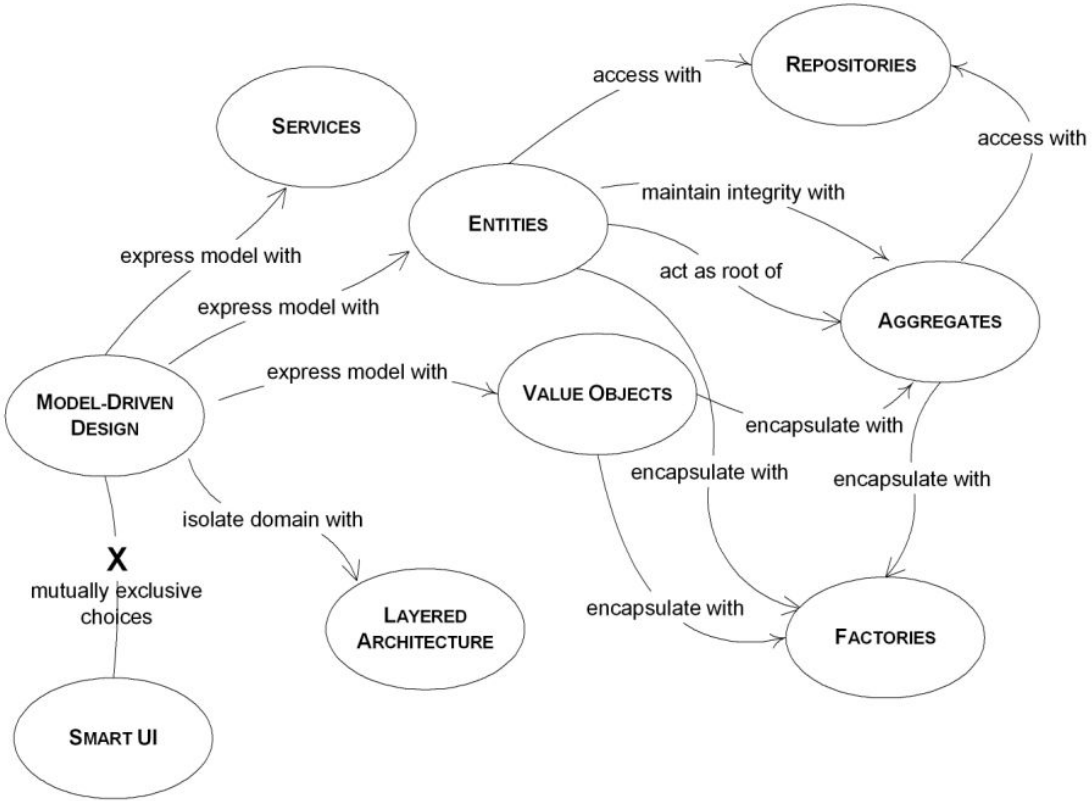
Kinds of mappings

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

Aggregates, Aggregate Roots, Entities & Value Objects



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

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

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

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

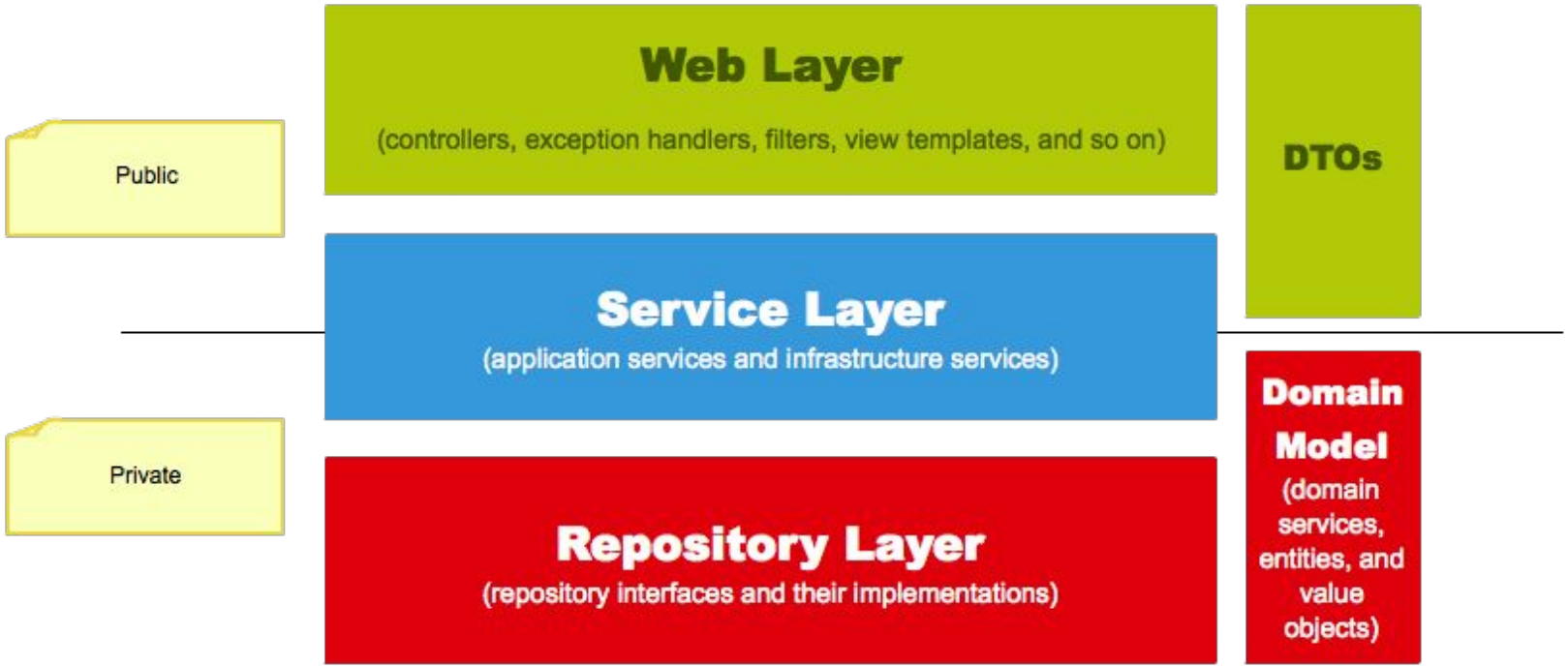
Service

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

Repository

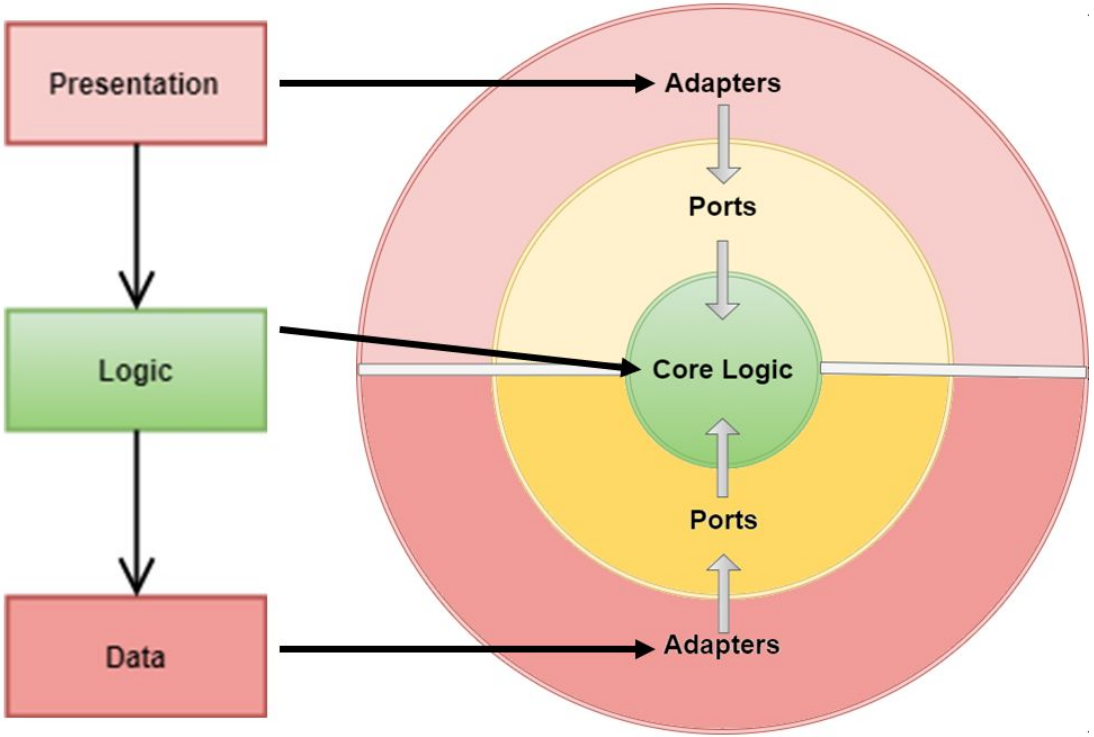
- **Store** domain objects (aggregates) into **persistence layer**
- **Retrieve** domain objects **from persistence layer**
- Examples:
  - CustomerRepository
  - TariffRespository

Traditional Layered Architecture



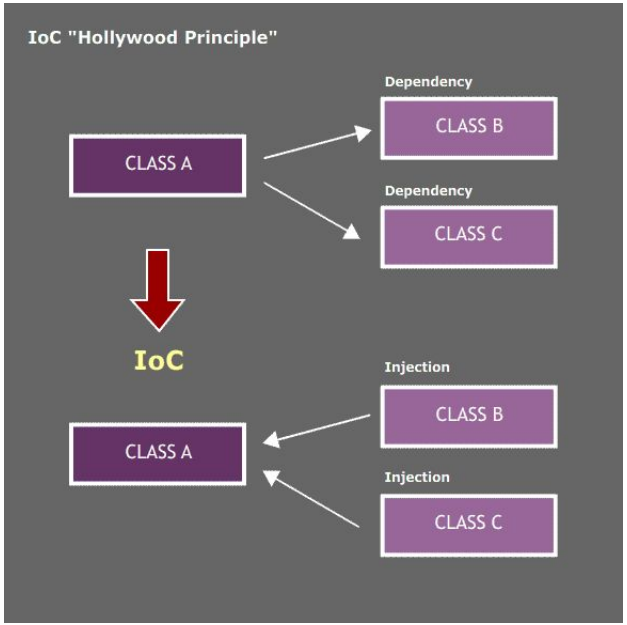
https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/

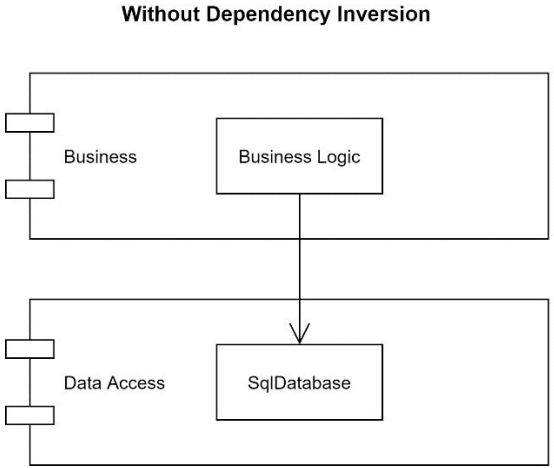
Hexagonal (Ports & Adapters) vs Layered Architecture

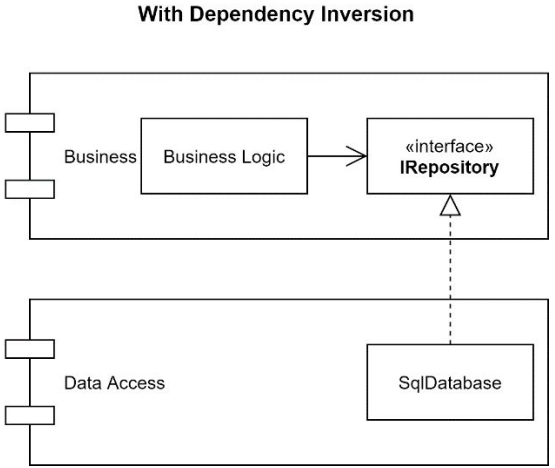


https://www.maibornwolff.de/blog/ddd-architekturen-im-vergleich

Inversion of Control / Dependency Injection (IoC/DI)



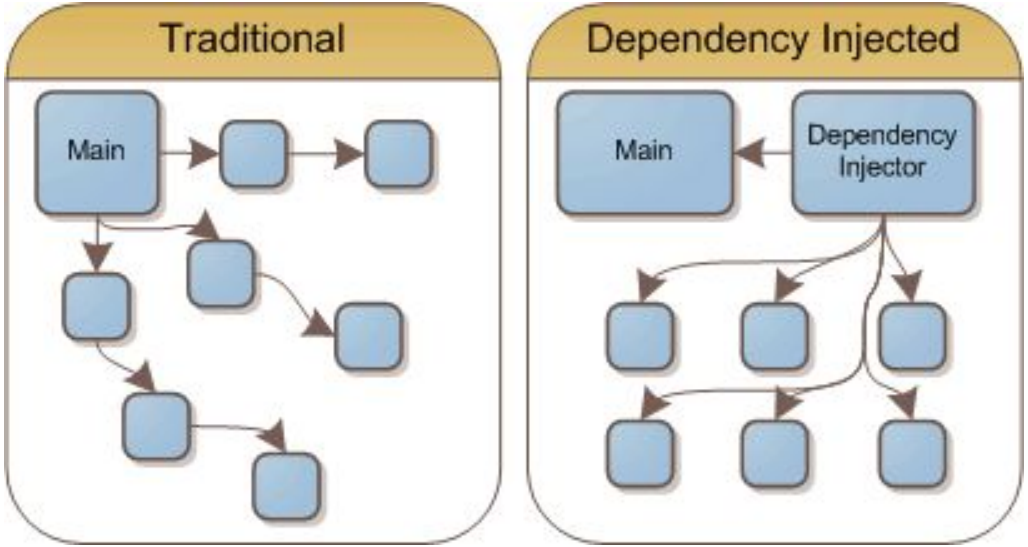




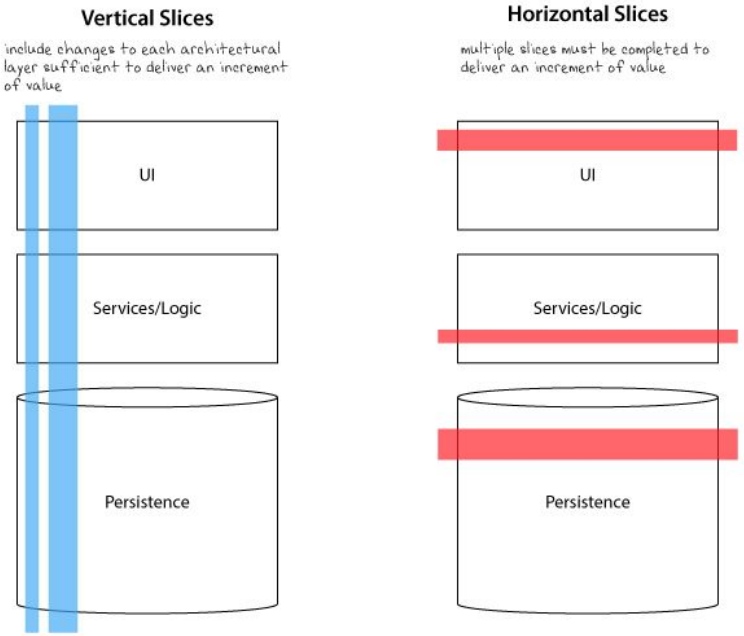
https://medium.com/analytics-vidhya/dependency-injection-concept-of-spring-framework-d9c3688005f8

Dependency Injection (DI)

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



Modelling: Layers vs Slices

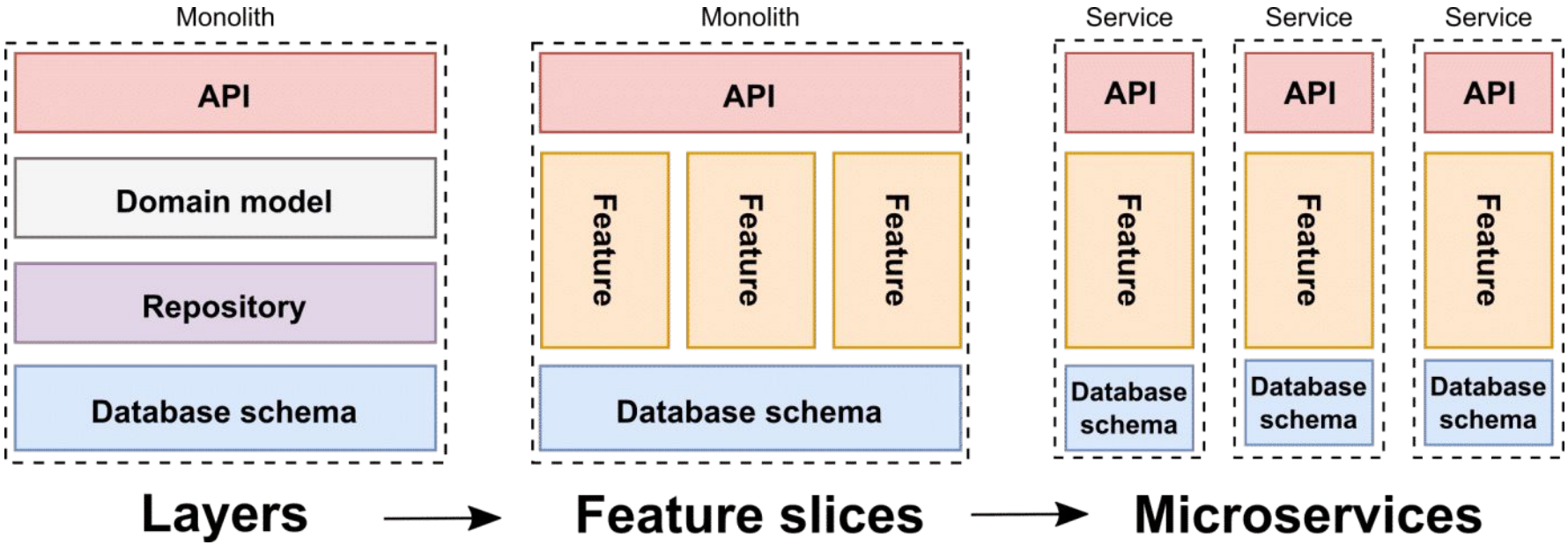



```
+- example (Layer)
        |
        +- controller
        |  +- ProductController.java
        |  +- OrderController.java
        |  +- CustomerController.java
        |
        +- dao
        |   +- ProductRepository.java
        |   +- OrderRepository.java
        |   +- CustomerRepository.java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

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

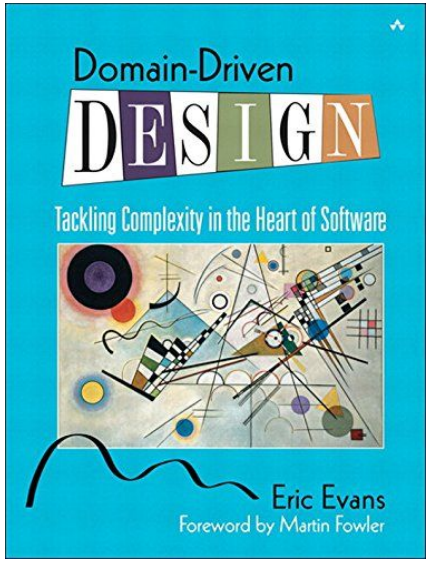
https://www.humanizingwork.com/vertical-slices-and-scale/

Modelling: Layers vs Slices (Monolith vs Microservices)

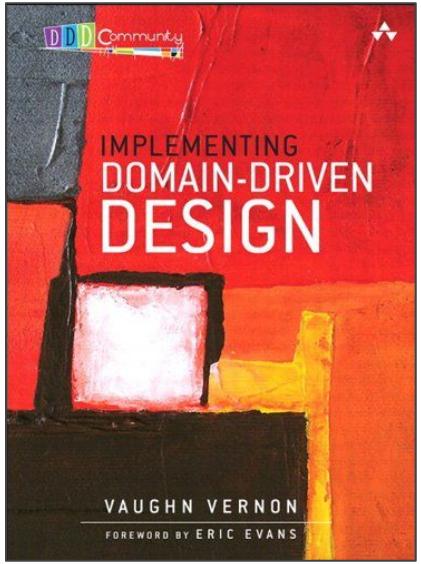


https://www.betterask.erni/news-room/slices-vs-layers/

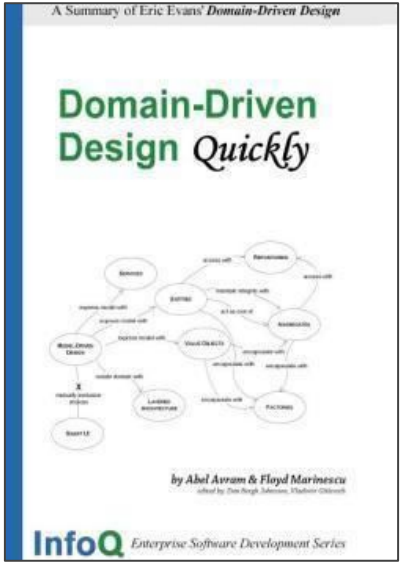
Reference Books on DDD



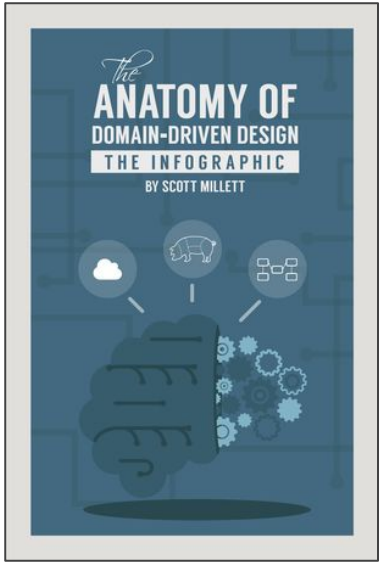
Reference Book inventing DDD



More Practical, but complete Book



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/