Architecture

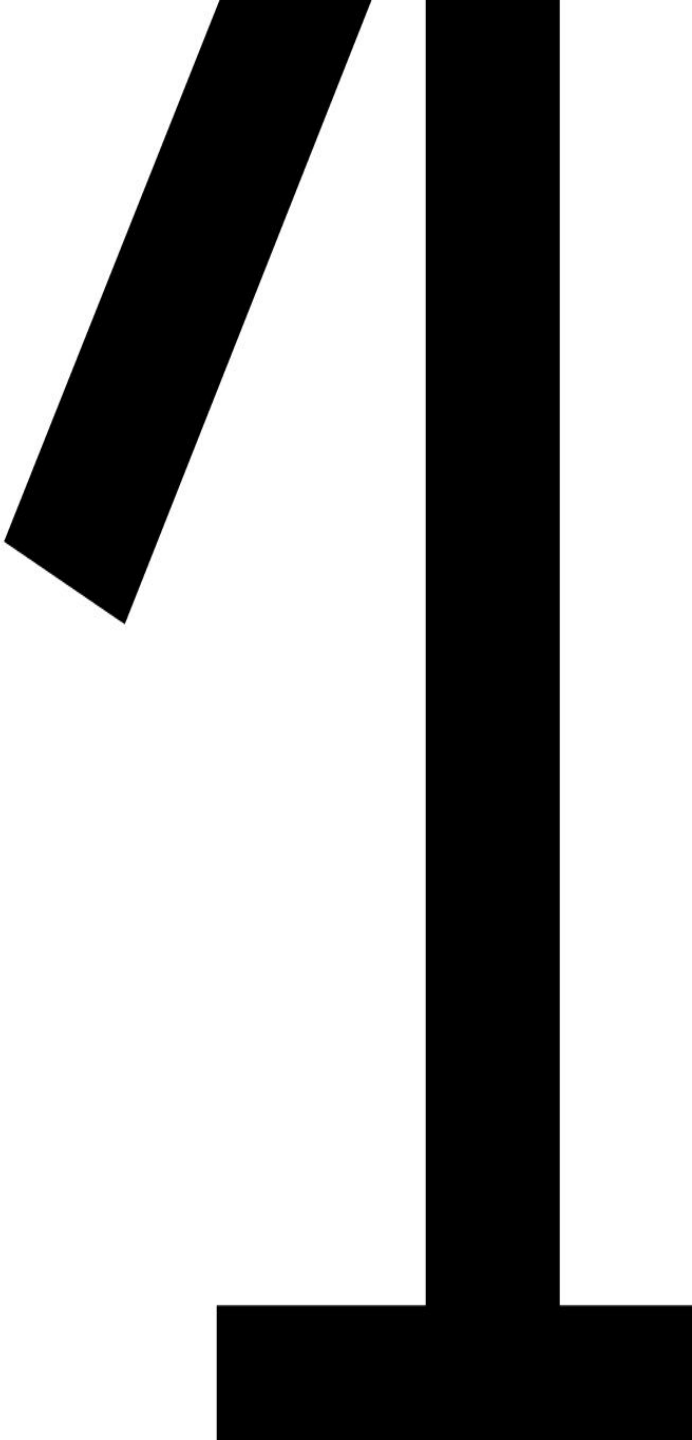
Programming 4

Karel de Grote Hogeschool

Agenda

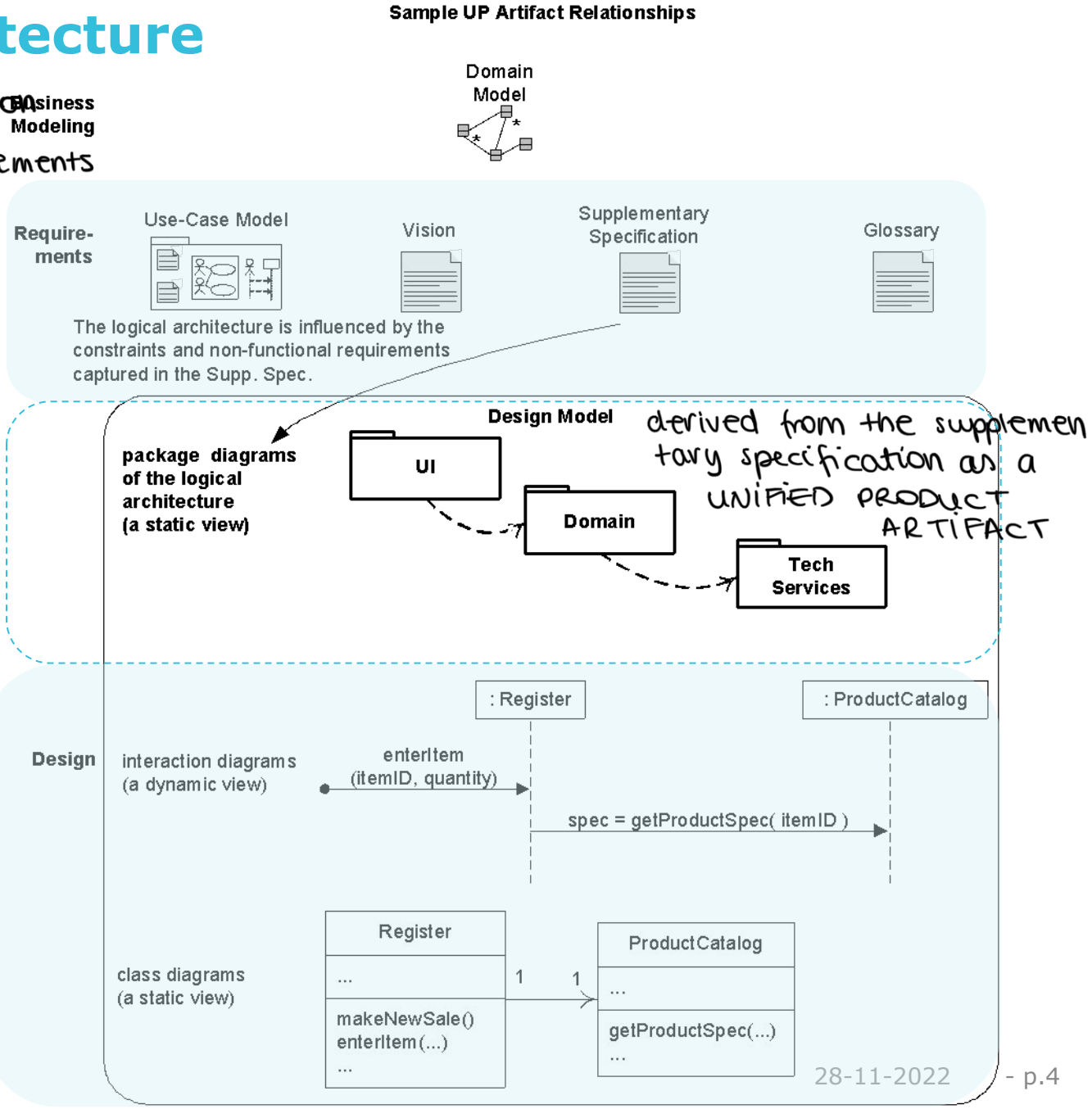
1. Introduction
2. Packages
3. layered architecture: basics
4. UML component and deployment diagrams
5. layered architecture: elaboration
6. POS business case architecture

Introduction



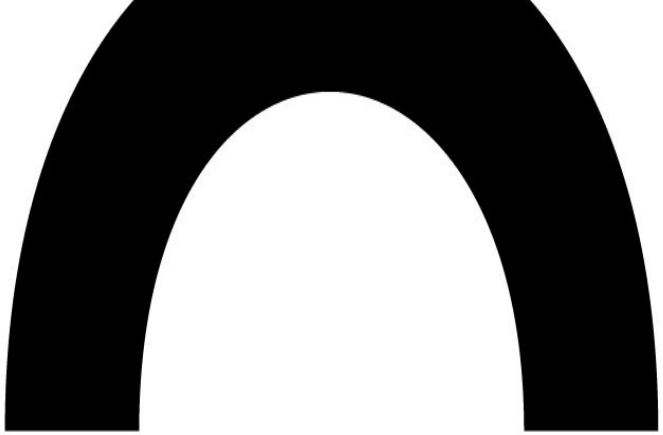
Logical architecture

major impact on our application
↳ non-functional are still requirements
↳ user friendly
- offer the most support



Architecture

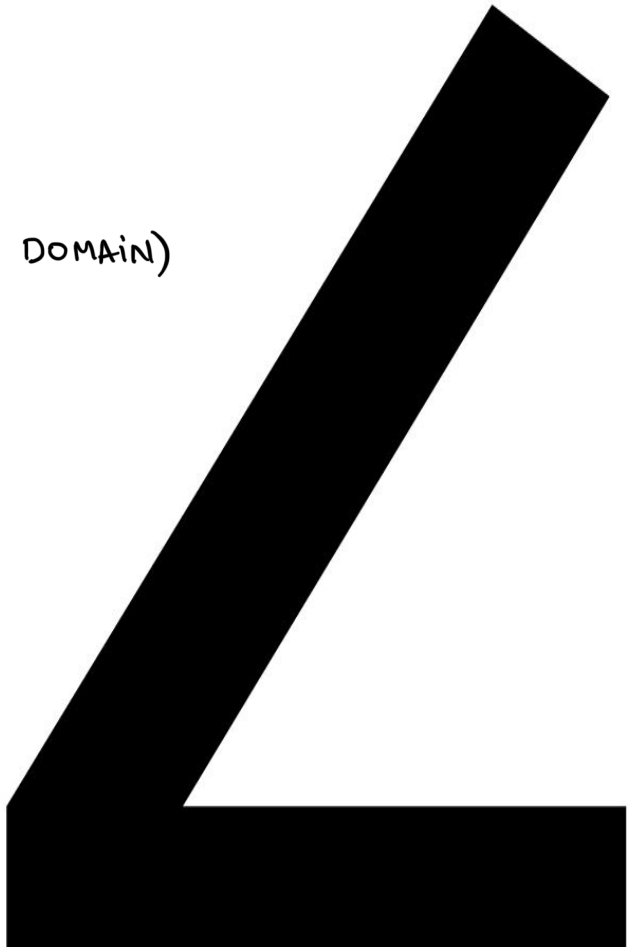
- We'll discuss some general principles
  - Implementation varies considerably depending on chosen architecture and technologies
- How to describe the architecture and high level structure of your project?



determine architecture

Packages (= high level structure within the DOMAIN)

= modules in DDD



Organising using packages

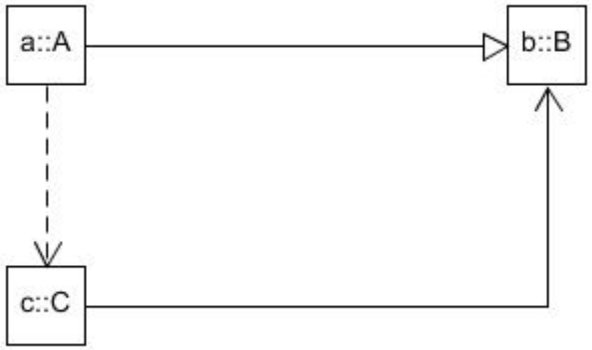
- Organises a program at a higher level than classes
  - Just like a class, a package should have high cohesion and low coupling

→ used like a class but at a higher level
  ↳ they have the same requirements

→ package level = organize them in a project
  ↳ as little dependency in between the groups

Packages: coupling and stability

- If a class A in package a depends on a class B in package b, then package a depends on package b

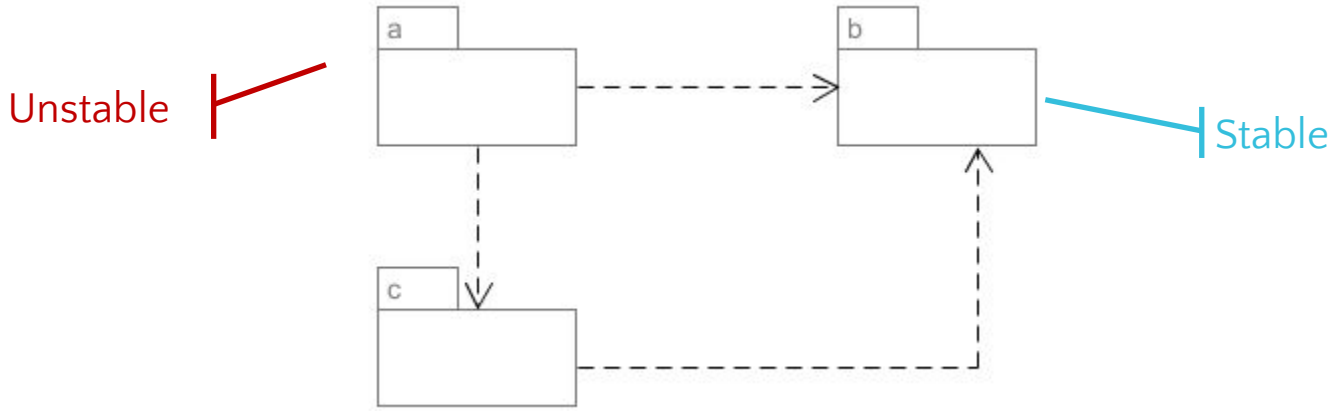


Discuss

Wat are the package dependenices in the example above?

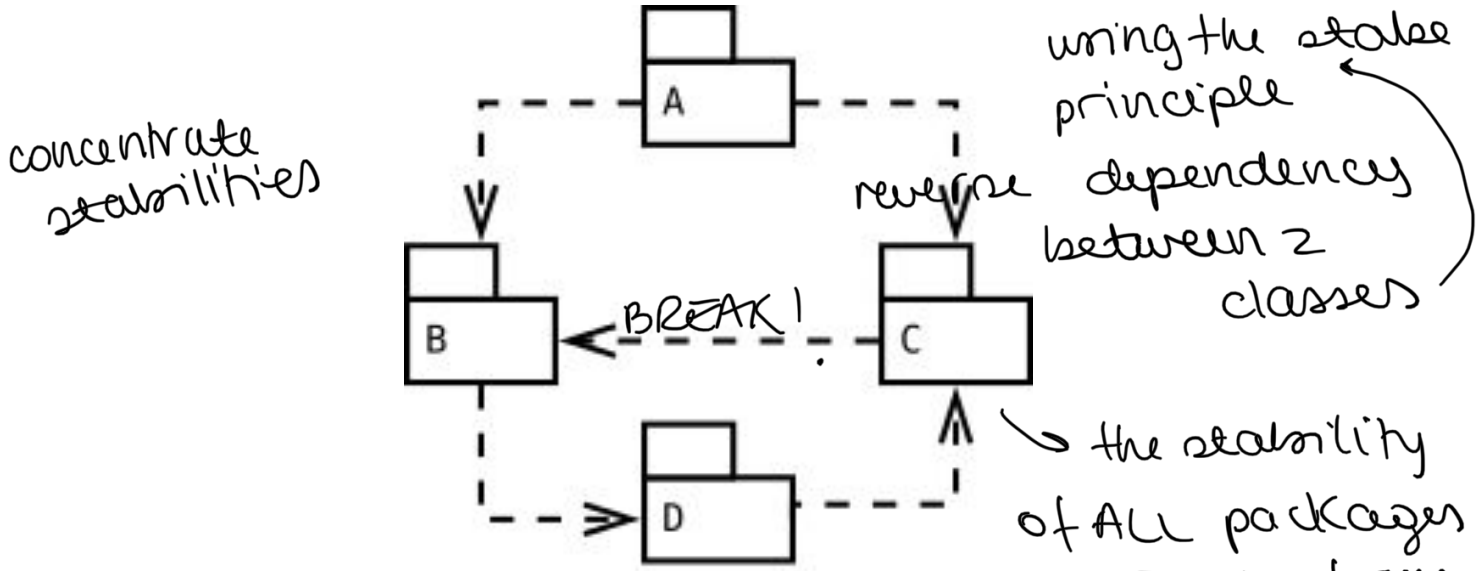
Packages: coupling and stability

- A package with only incoming dependencies is stable with respect to external changes
- A package with many outgoing dependencies is unstable with respect to external changes
  - have direction of dependencies in order of the most stable package
  - the package gets the stability of the package it's depending on



Packages: coupling and stability

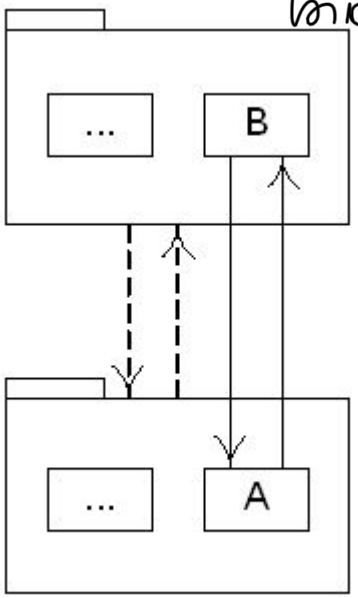
- Avoid cyclic dependencies (ADP: Acyclic Dependency Principle)
  - All packages would get the stability of the package with the lowest internal stability
  - Bidirectional dependency is the simplest cyclic dependency



Discussie How to solve the cyclic dependency?

Packages: coupling and stability

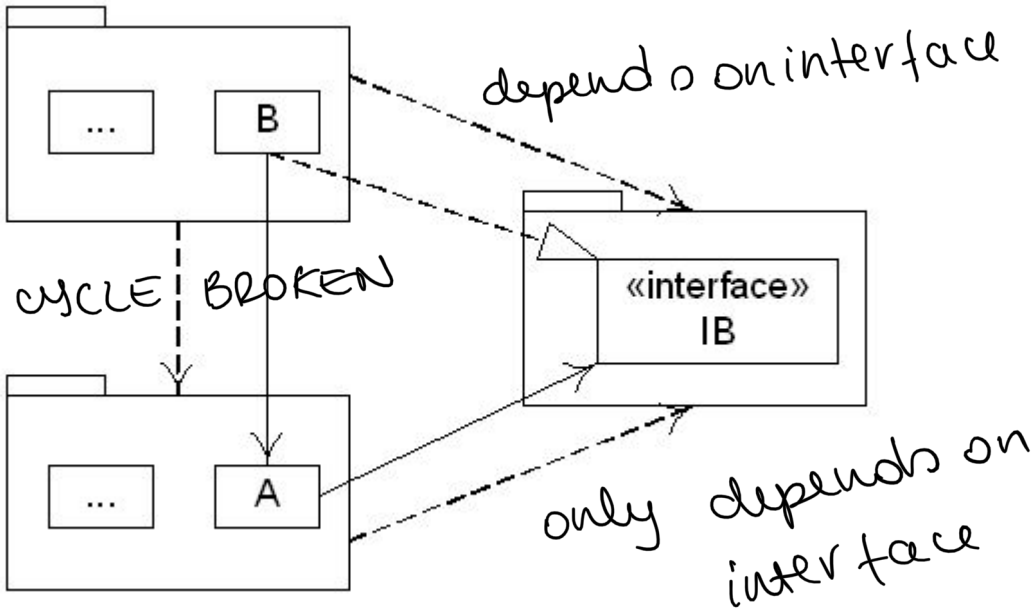
- Have dependencies point to packages with the highest internal stabilities
  - stable business data and logica
  - standard API's
  - General concepts
  - interfaces



Discussie How to solve the cyclic dependency?

Packages: coupling and stability

- Have dependencies point to packages with the highest internal stabilities
  - stable business data and logica
  - standard API's
  - General concepts
  - interfaces

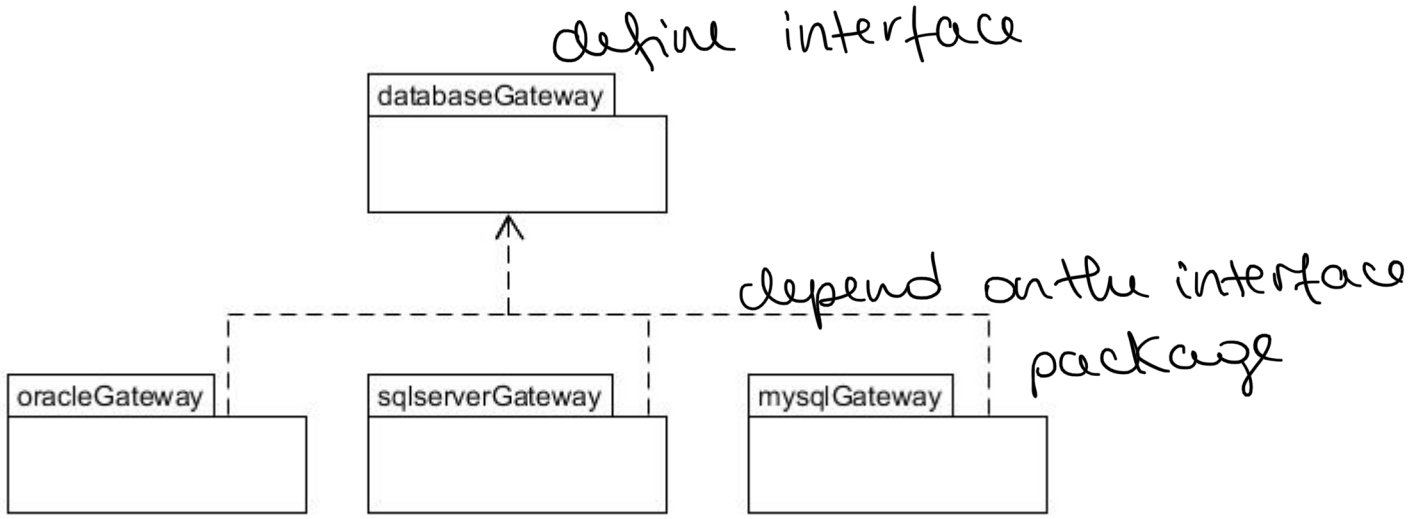


Discussie

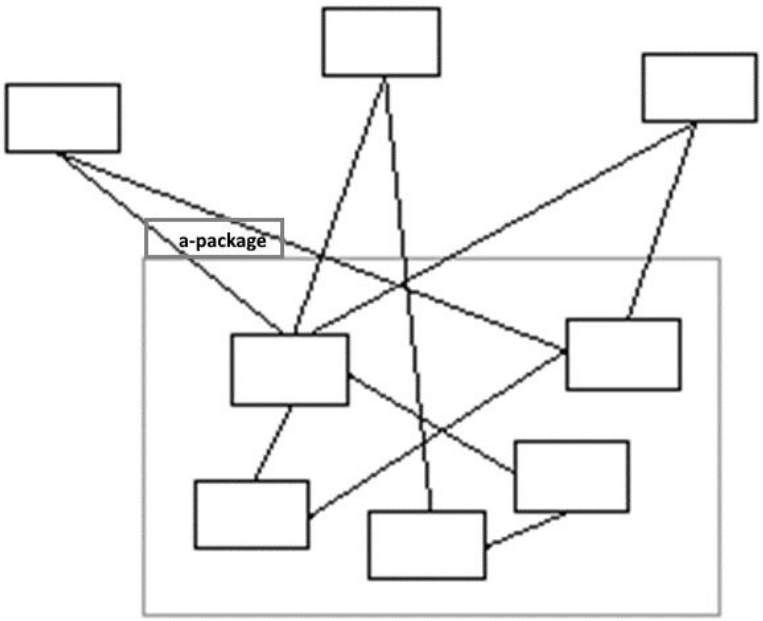
Dependency Inversion Principle

Packages: implementation strategies

- Implementation classes depend on interfaces and frameworks
  - Dependency Inversion Principle
- Put each strategy in its own package
- Package dependencies match class dependencies



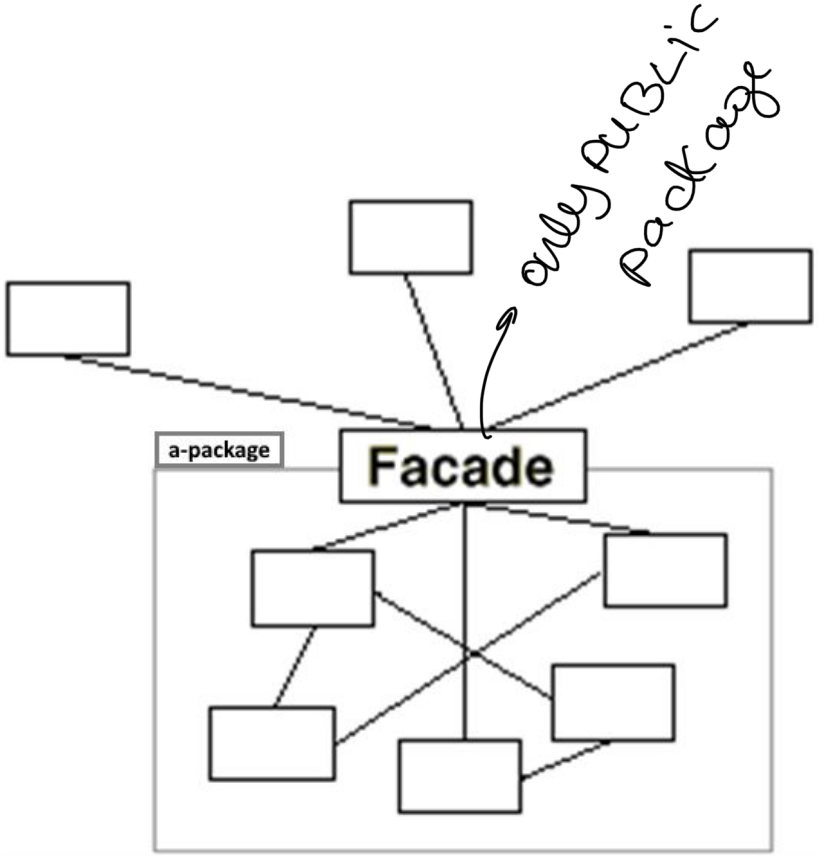
Problem: wat is a package is highly coupled with external classes?



coupling is TOO HIGH!
dependencies go to too many inner classes

fix using ADAPTOR (of bridge)

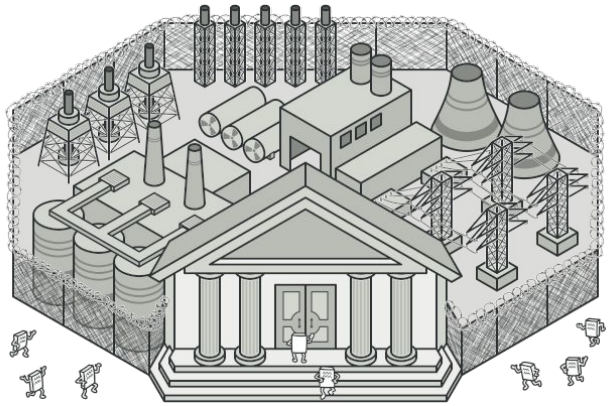
Oplossing: Facade (GoF design pattern)



- One class centralizing package access
- Classes in the package are not directly accessible anymore (encapsulation)

  ~ package access
- Comparable with interface that limits acces to a class

Facade (GoF desing patroon)



- Naam: **Facade**
- Family: **structural**
- Intent: facade = singleton

  "A single common interface for multiple internal interfaces in a subsystem"
- Context: A subsystem exposes a set of services in different classes. Define one central access point that removes coupling with internal classes and hides the internal structure
- Related patterns: *Adapter, Singleton*

packages should reflect your architecture

Basics

Layered architecture

KdG
University of Applied Sciences and Arts

Logical architecture

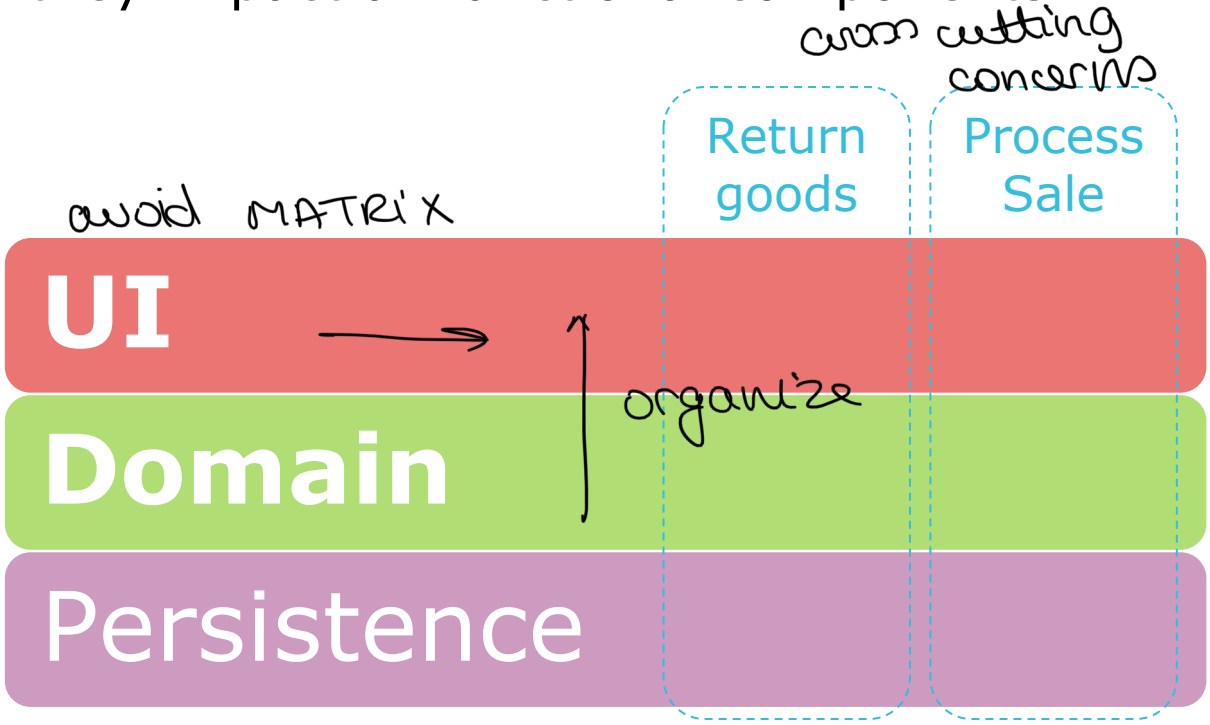
- An architecture is largely based on non functional requirements (-ilities)
- **DDD: layered architecture:** At a high level an architecture is often organised in layers
  - AKA horizontal components
- A minimum of 3 layers is common.

implement the use of all this layers → always go through these layers

deviding code

Logical architecture

- Each functional requirement (feature) uses all layers
  - Functional components are also called vertical components
  - Results in a matrix structure
- Horizontal components are also called cross cutting concerns, because they impact all functional components.
  - logging, security…



Logical architecture packages

- Use packages to group classes in architectural layers
- Packages have a tree structure. How to represent a matrix in such a tree?
  - The leftmost organisation is most common within a system

package = tree
organisation = table

most common one

- UI
  - return goods
  - process sale
- Domain
  - return goods
  - process sale
- Persistence
  - return goods
  - process sale

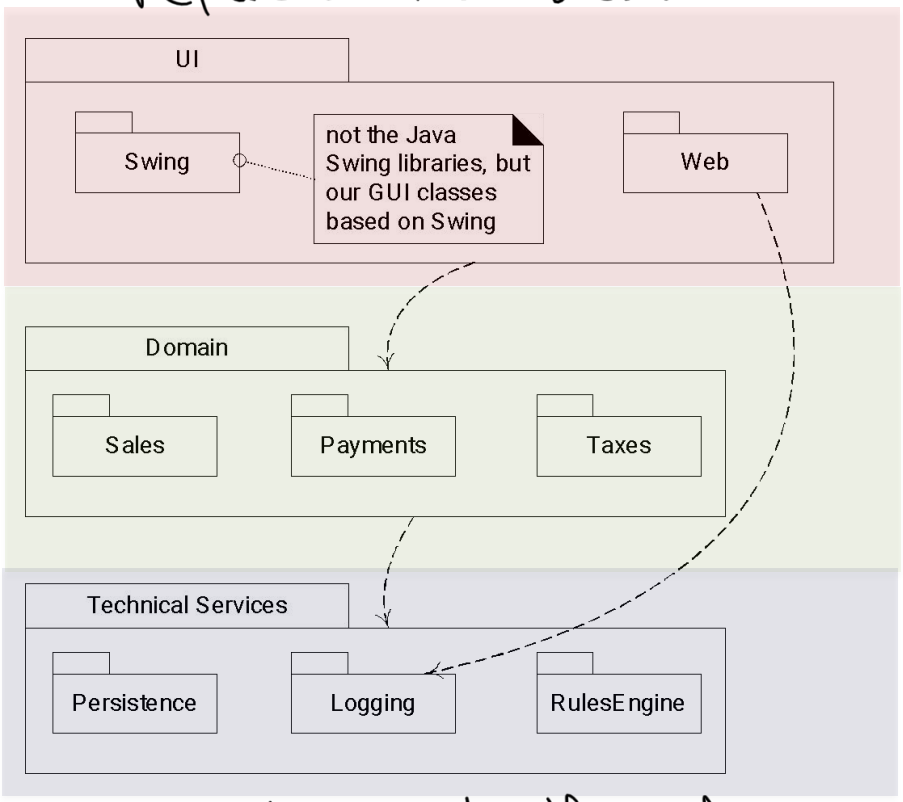
- Return goods
  - UI
  - Domain
  - Persistence
- Process sale
  - UI
  - Domain
  - Persistence

- Subsystem 1
  - UI
    - return goods
    - process sale
  - Domain
    - …
  - Persistence
    - …
- Subsystem 2
  - UI
  - Domain
  - Persistence

BOTH

logical architecture: layers

reflects the business



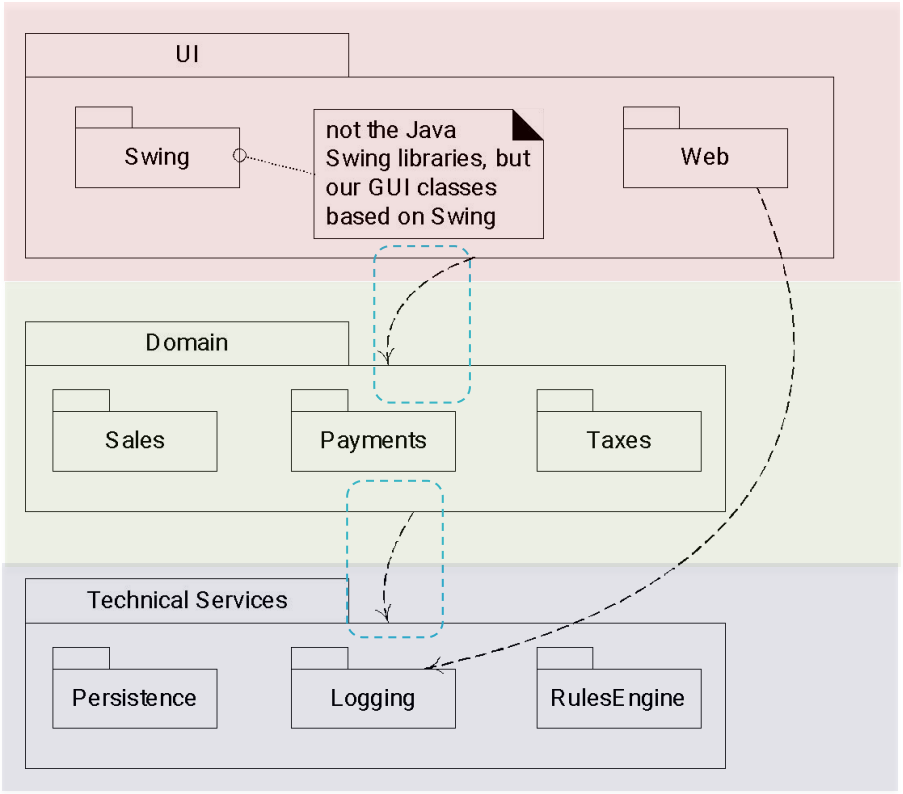
- UI (User Interface) or Presentation layer
  - Note: Swing is a predecessor of JavaFX
- Domain or business layer contains business logic and business rules
- infrastructure layer with technical services like persistence (database) of webservice calls
  - Can call external systems

don't want the changes in a package effect the other packages → put things that change together in the same package!

logical architecture: layers

PERSISTANCE!

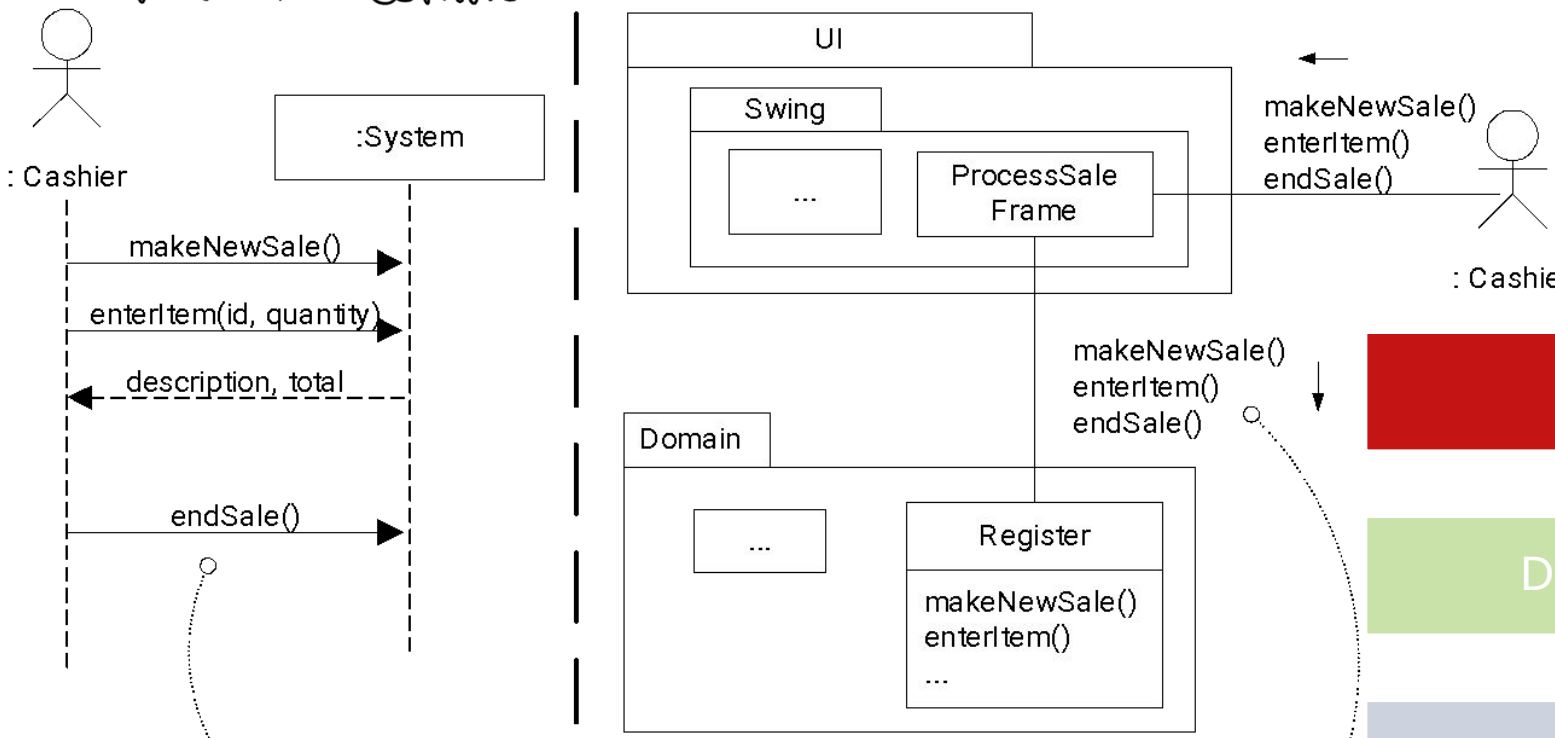
Don't skip layers

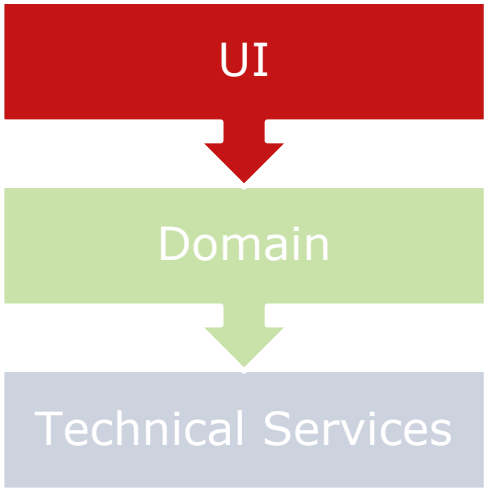


- Higher layers depend on lower layers
  - They call code in lower layers
  - Do not call higher layers from lower layers

UI layer

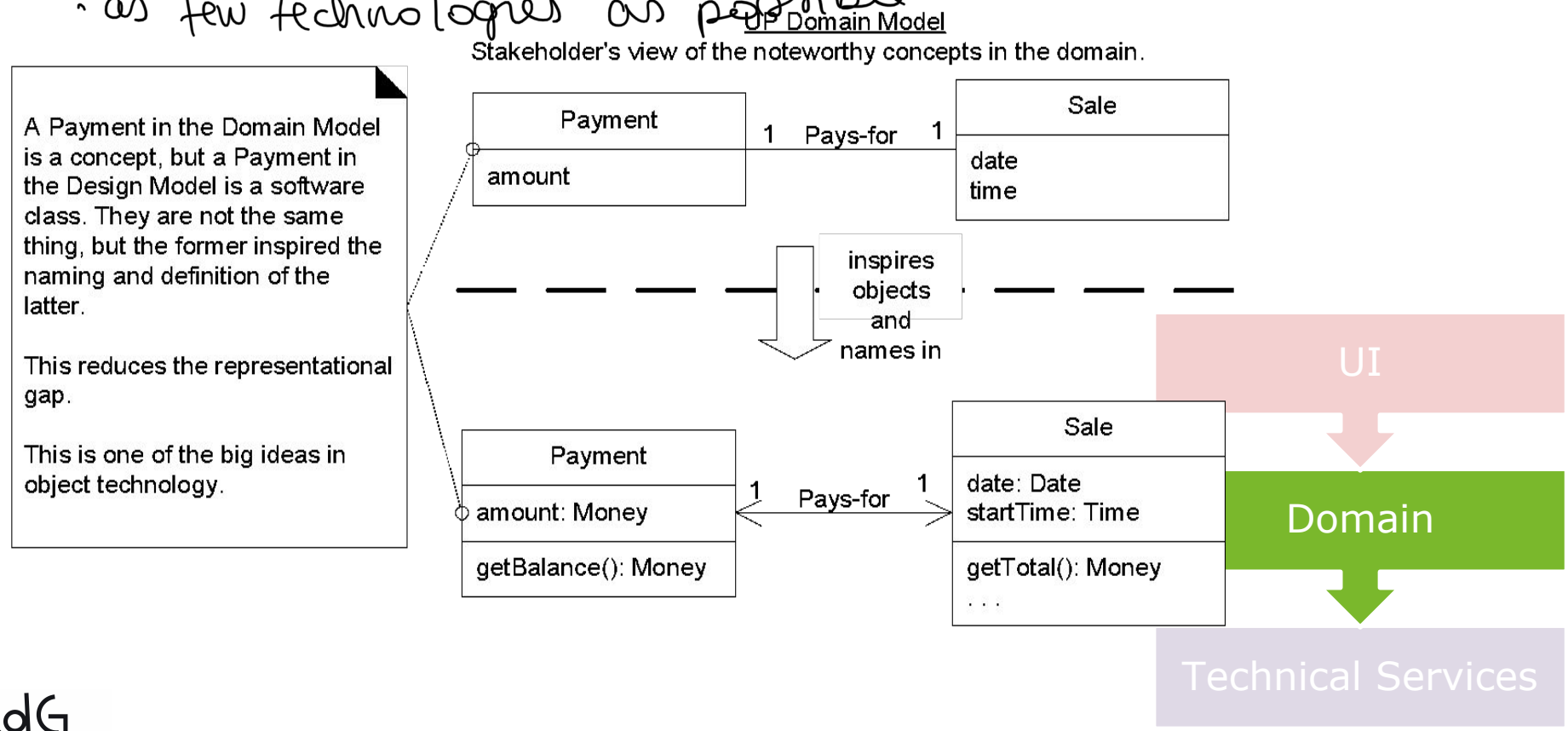
- Actor uses UI (presentation layer), UI calls domain layer
- If the actor is an application, interface (API) layer is a more generic name
  - how it communicates with the Domain (controller)





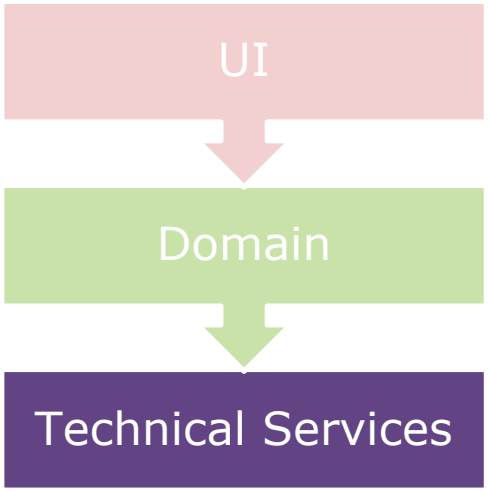
Domain layer

- Focus of this course (and of Domain Driven Design)
- Conceptual classes from domain model and business logic
  - data + methods
  - as few technologies as possible



Technical Services

- layer accessing back end technologies like persistence, security… Communicates with other systems (database webservice, directory…)
- AKA persistence layer (the term we'll use), infrastructure layer

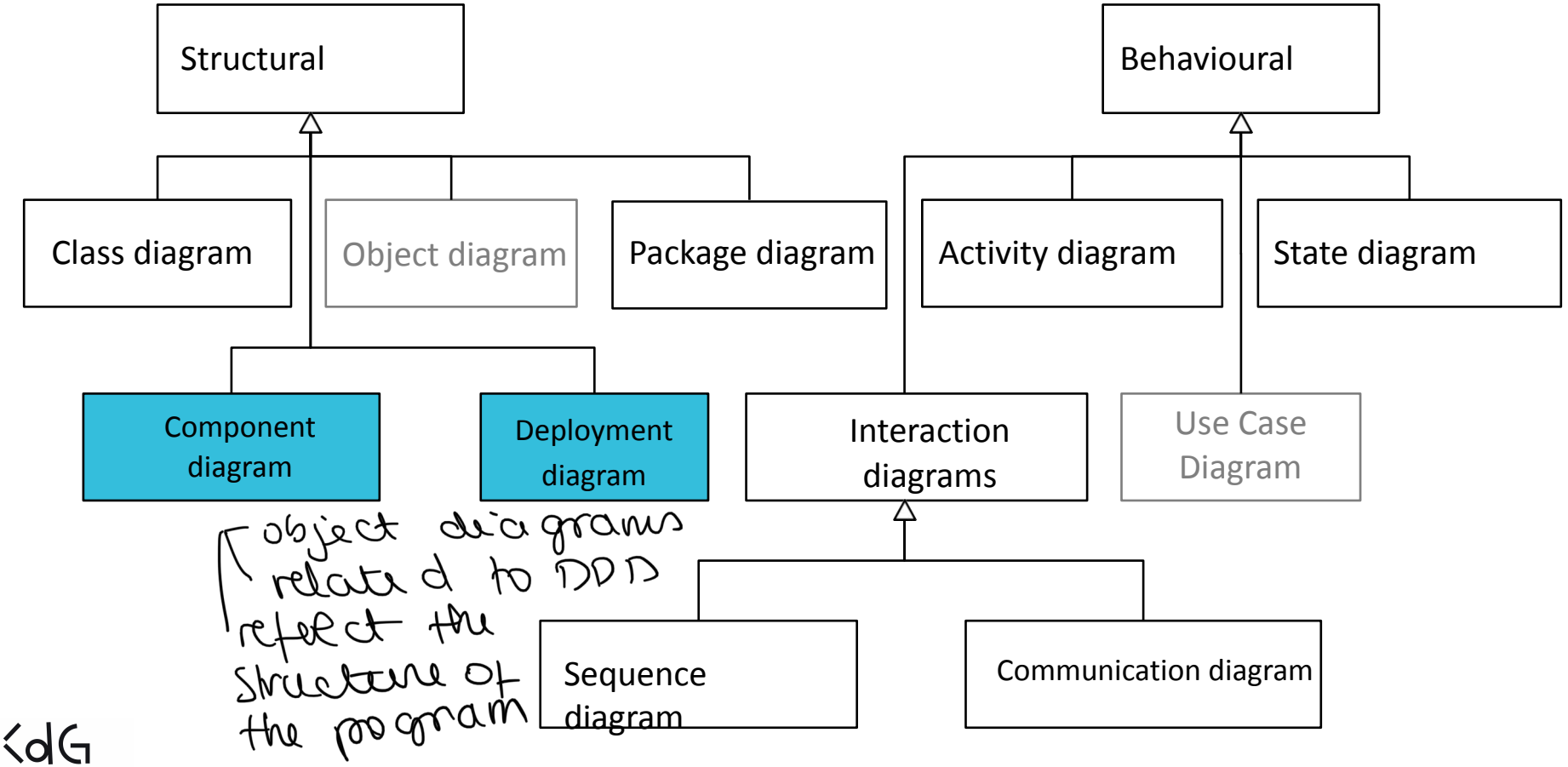


UML component and deployment diagrams

4

KdG
University of Applied Sciences and Arts

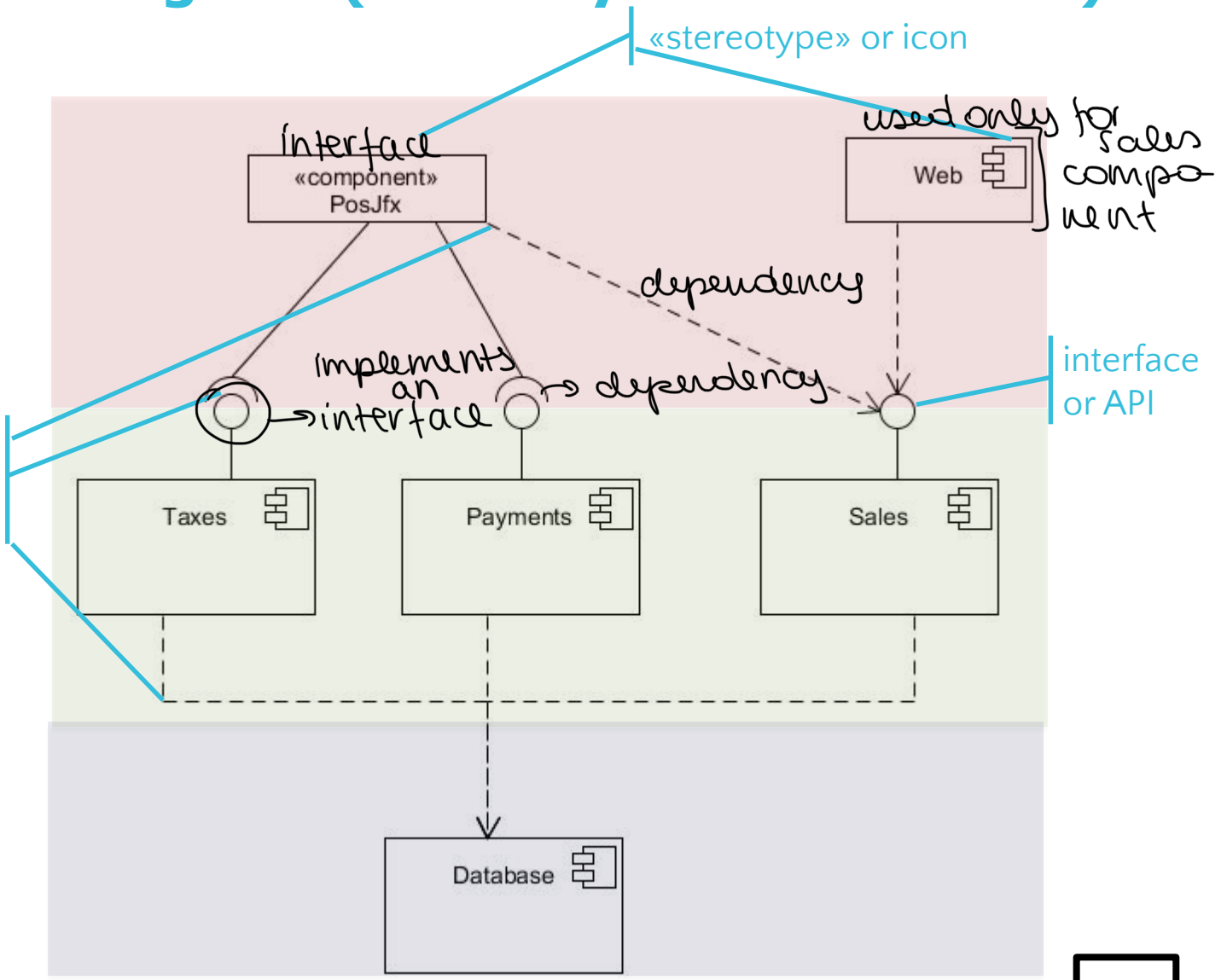
UML diagrams



Component diagram

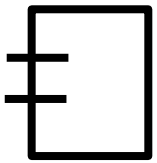
- Components are software modules (DDD Modules)
  - AKA libraries
  - A module is always installed in its entirety
    - modular system: you can choose which modules you install (example: since Java 9, Java is modular)
  - Will typically match a high level package
  - Has a software interface (API)
    - Can sometimes be replaced by another module implementing that API

Component diagram (for 3 layer architecture)



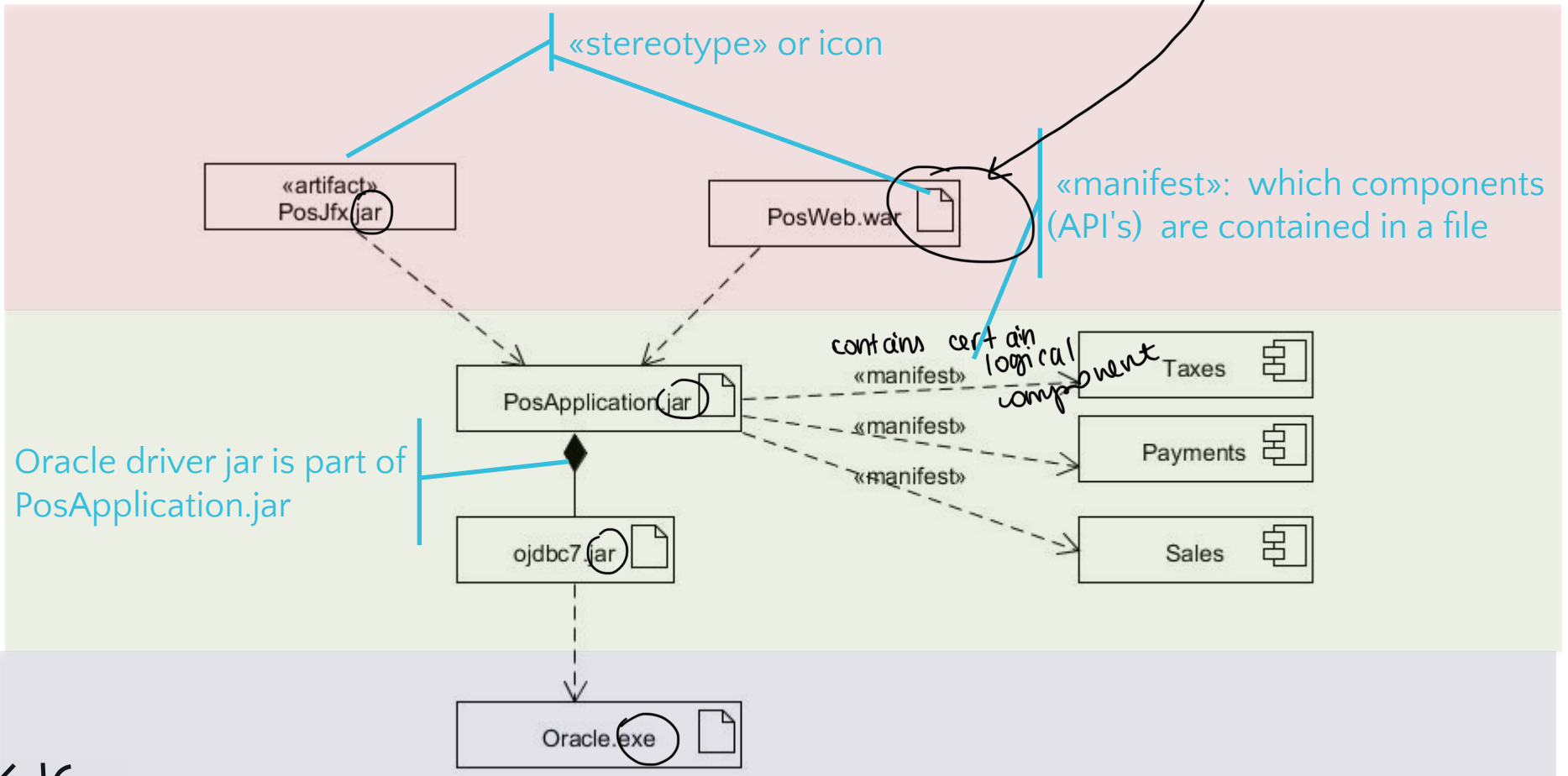
Uses interface
(3 representations)

Simpler component representation when drawing by hand:



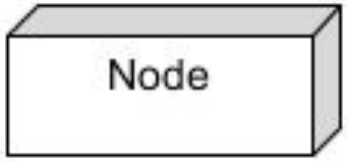
Physical architecture: artifacts

- objects / files made on the computer
- represent software files (executables, libraries…)



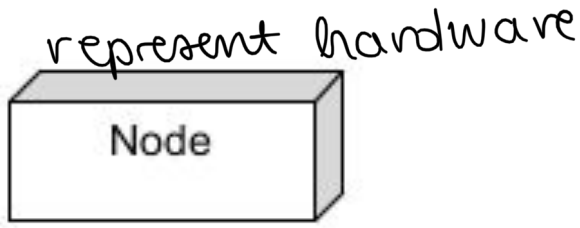
Installation: deployment diagram

- Environment in which the software runs
  - hardware device (physical)
    - PC
    - Server
    - cloud device
    - …
  - execution environment (software)
    - Operating system
    - browser
    - virtual machine
    - …

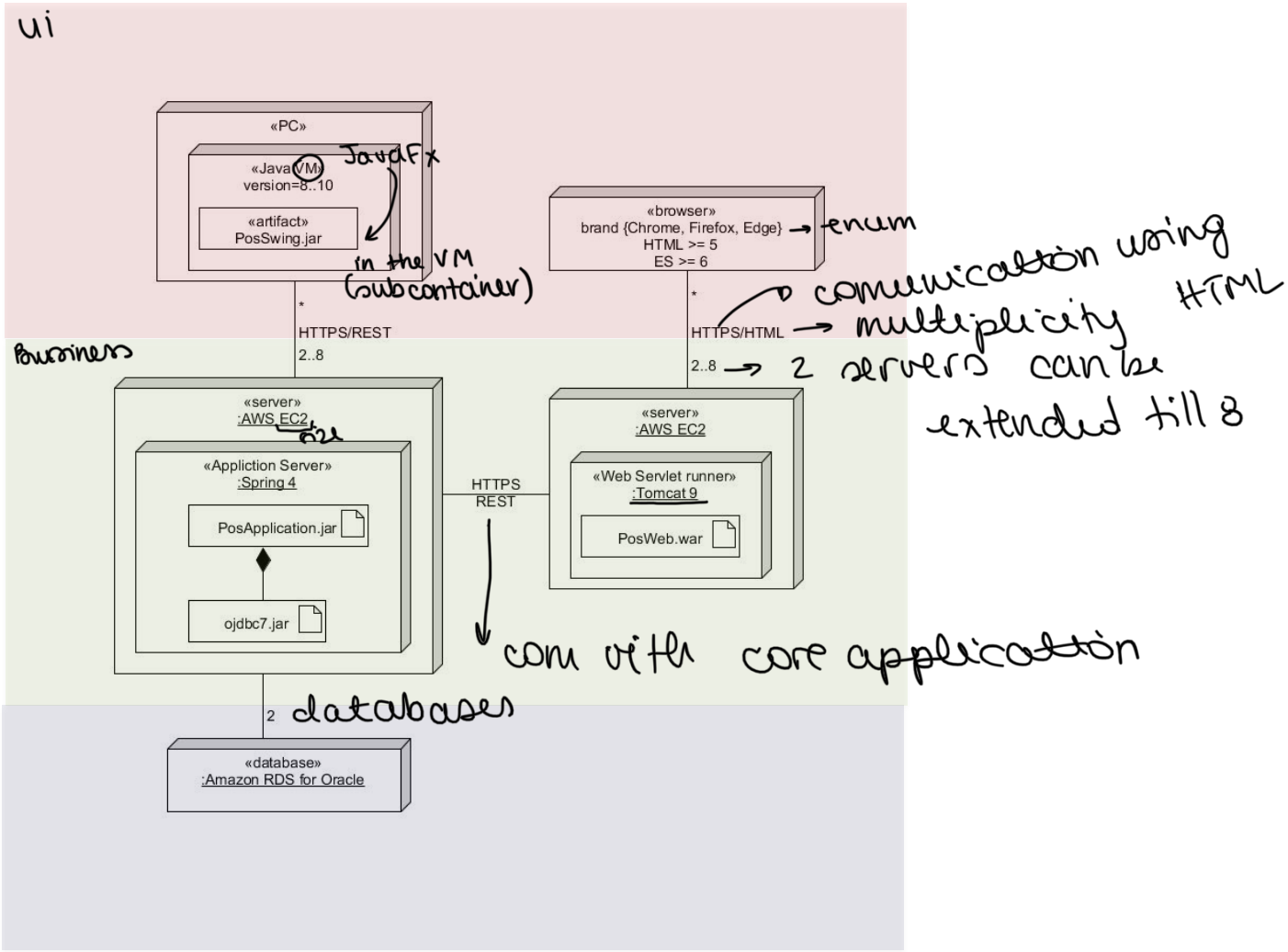


Installation: deployment diagram

- UML objects with hardware and software specifications
- Associations with
  - multiplicities
  - Communication specifications (protocols, data formats…)
- Physical software artifacts are often integrated in the deployment diagrams

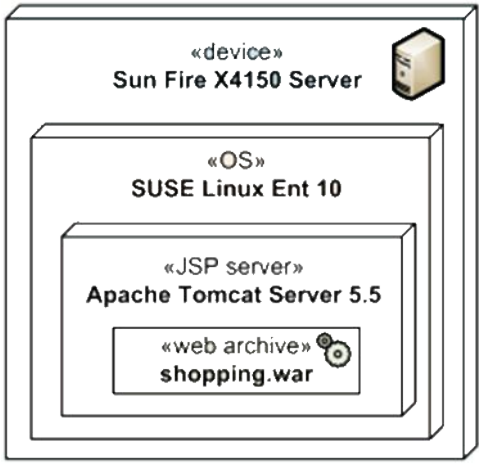


Installation: deployment diagram

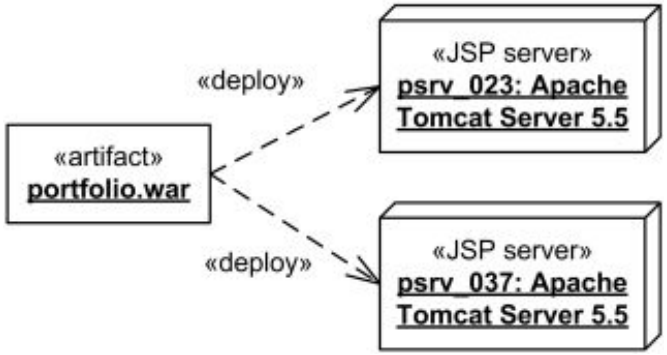


Installation: deployment diagram

- With stereotype icons



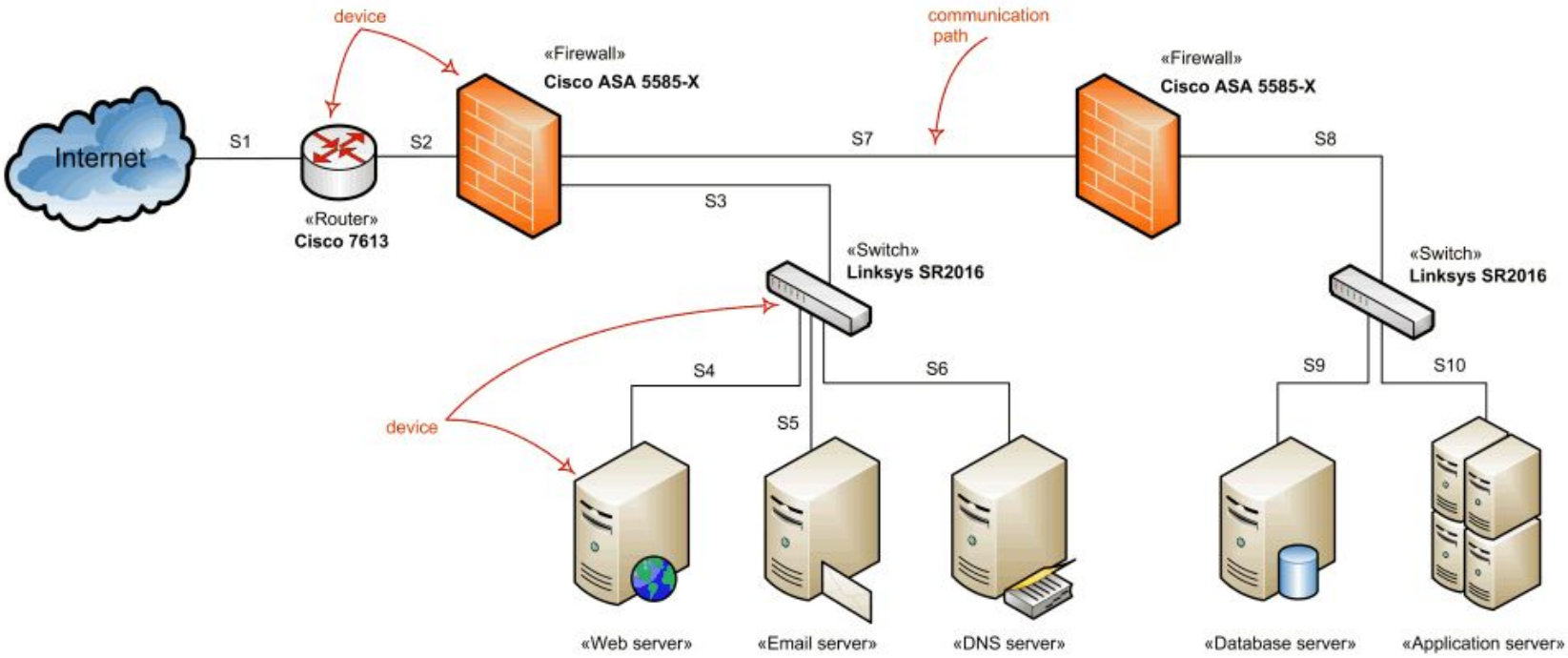
- You can also show artifact deployment using a `«deploy»` dependency



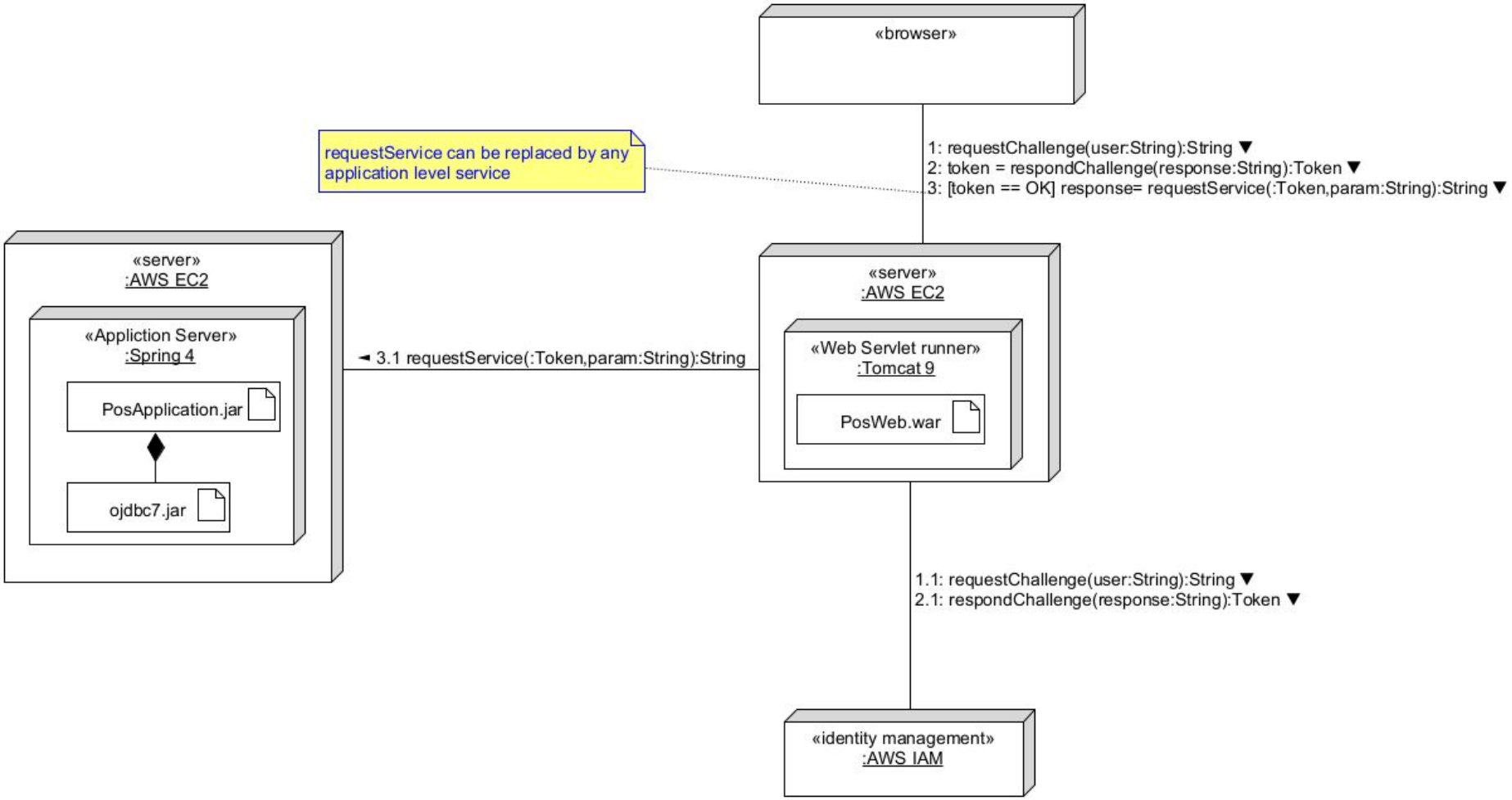
Installation: netweork

Verdieping

- UML is not often used for Network architecture (MS Visio…)



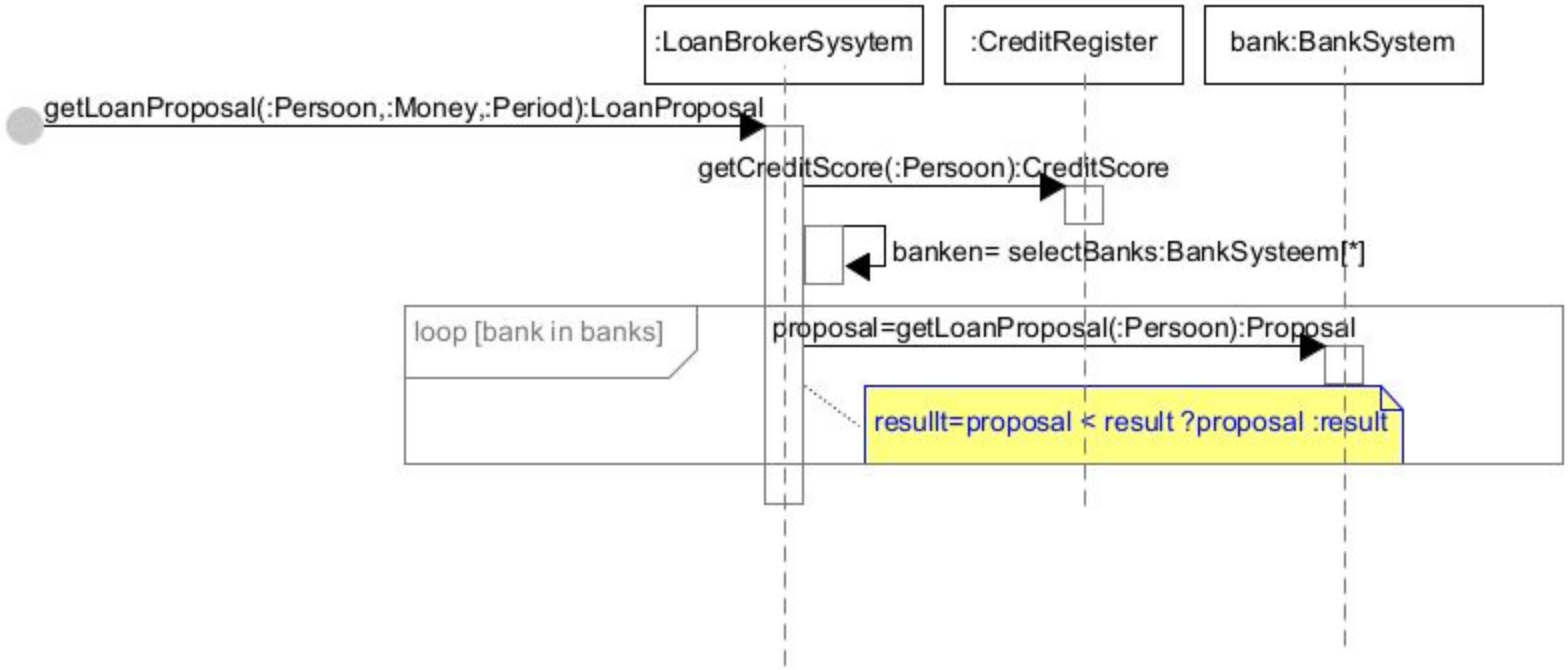
Architectural interaction diagram: example



Architectural interaction diagram

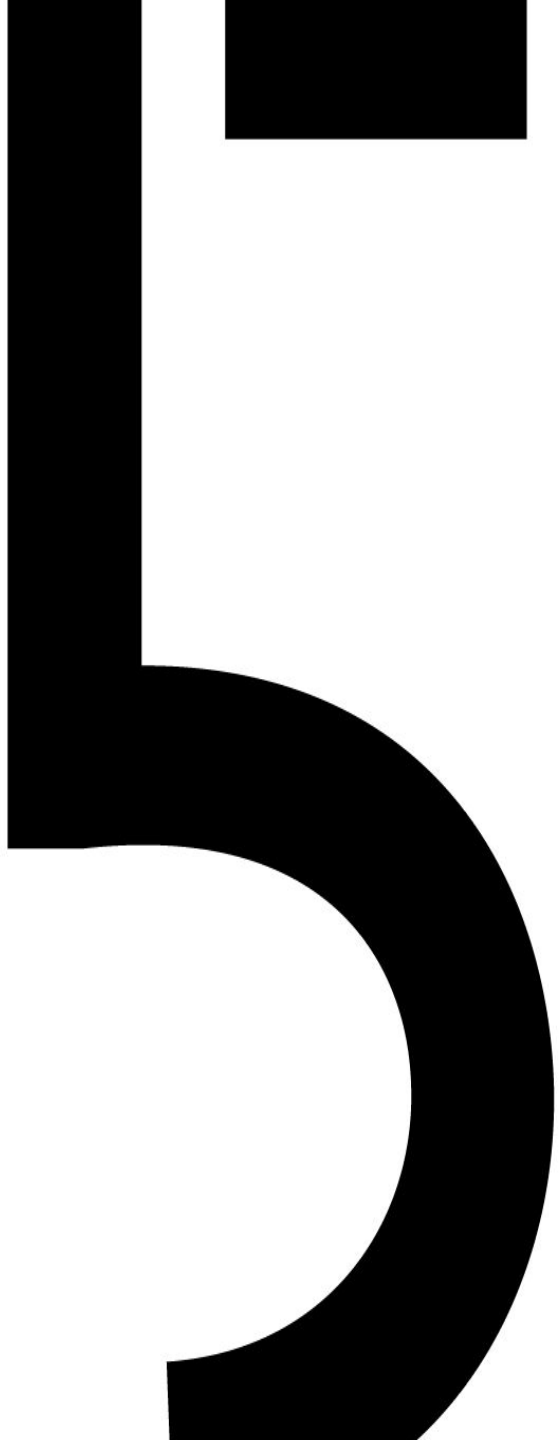
- You can use architectural elements in interaction diagrams
  - components, nodes…
  - Can contain generic calls, databank requests… to show typical interaction sequences in the chosen architecture
- Use to explain the high level structure and interactions in your system
- When elaborating a service (class level) it is not necessary to repeat interactions that are shared by every service

Architectural interaction diagram: example



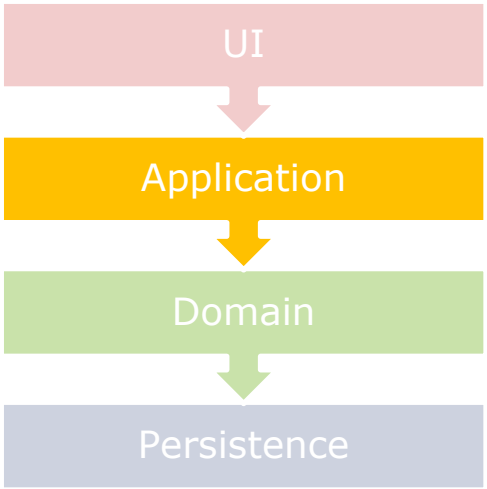
More detail…

layers architecture



Application layer

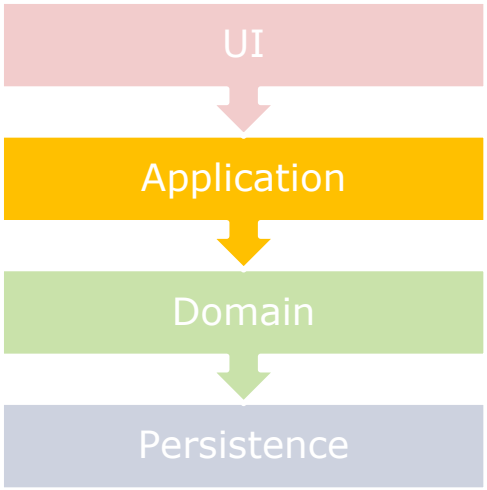
- Access point for user interactions from user stories
  - Applications logic related to UI structure (e.g. screen navigation)
  - Convert UI data
  - Communication protocol (e.g. REST…)
  - calls business logic in domain
- Controller is the key element in this layer (discussed in GRASP)
- Often not treated as a separate layer



Application layer

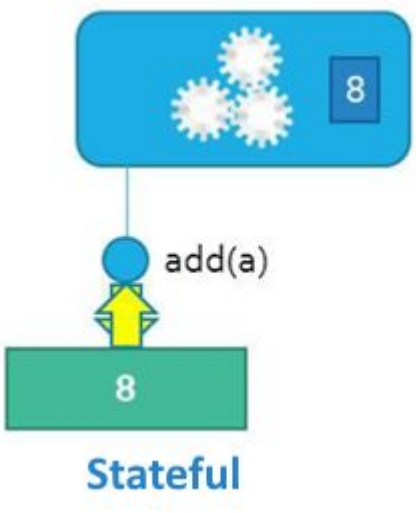
Refresher

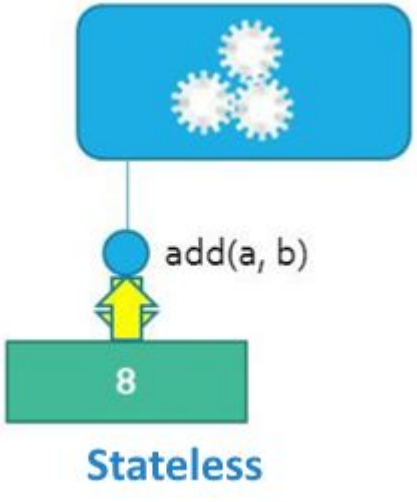
- **Divide responsibility over multiple controllers if cohesion would become too low**
  - **Facade controller:**
    - Handles all actions for small system or subsystem
    - Front controller: proxy controller that delegates to other controllers
  - **Route controller:** in web frameworks (.Net MVC, Express.js, Spring MVC…) you often have a controller per URL pad
  - **User story controller:** handles all actions for a user story or epic

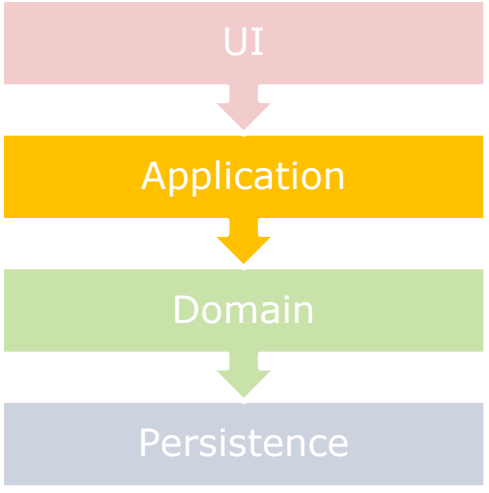


Application layer: stateful of stateless?

- Stateful: controller stores information of a user interaction as a precondition for the next interaction
  - One controller object per user session (session controller)
  - session information is stored in the controller (POS example: Sale, User…)

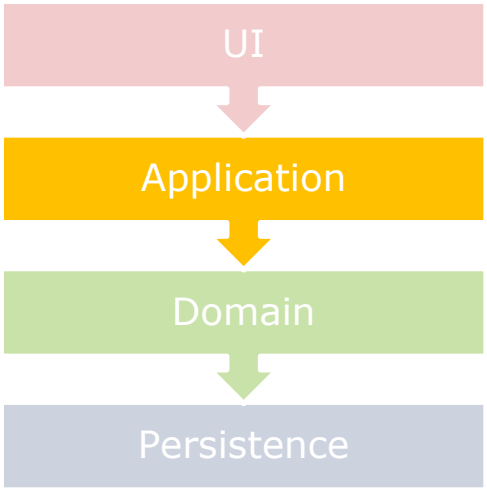






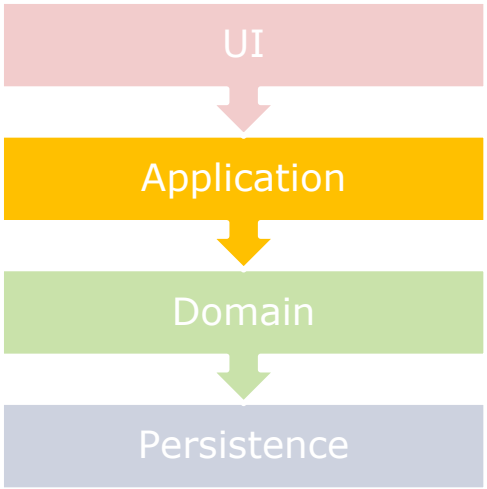
Application layer: stateful of stateless?

- Stateless: controller attributes do not change between calls
  - Session data are saved elsewhere
    - user interface: keeps session information and always sends it to controller (example: saleId, salesLineItems…)
    - database
  - Simplifies controller object lifecycle
    - a controller object can be shared between sessions
    - Calls in a session can go over different controller objects
    - Easier for load balancing



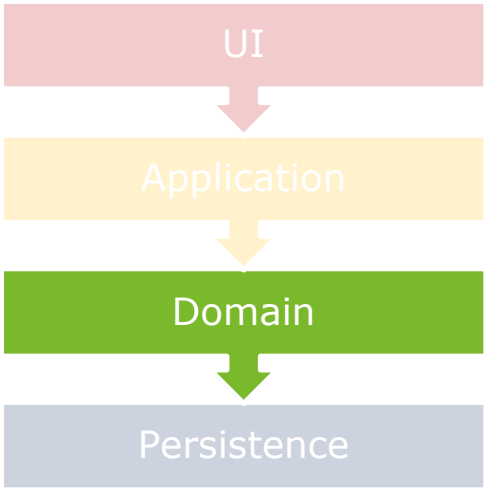
Application layer

- Controller attributes
  - Have visibility on domain layer
  - Can keep session information in a stateful controller
- Controller methods
  - Provide controlled access (interface) from the external world (user interface…) to the domain layer
  - There is a method for each external interaction (scenario step)
  - Do not contain business logic. They rather translate external requests (user interface) to business service calls in de domain layer
    - Can handle communication specifics (protocol controller) and data formats when interacting with UI or external systems



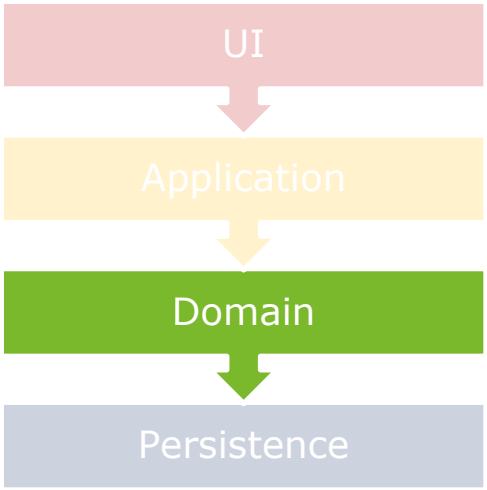
Domain layer: DDD Service

- Accessed by Application layer
- Exposes business logic in **DDD Service** facade classes
  - AKA manager, business service
  - technology neutral: works with domain objects



Domain layer: DDD Service

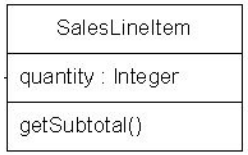
- Behavioural class.
  - Stateless
  - Calls methods on domain objects or on other services
  - Coordinates, delegates to domain objects
- Guideline: A service per high level business concept. Do not span packages.
  - Examples: StoreService, ProductService
- Can communicate with persistence layer
  - Retrieve and store domain objects
  - pure domain service: a service that does not access a technical (persistence) layers

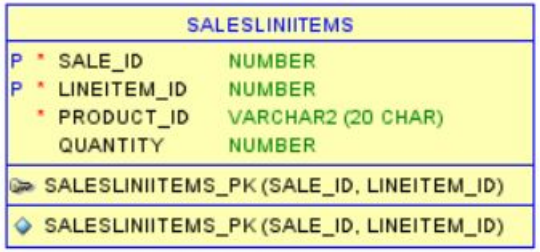


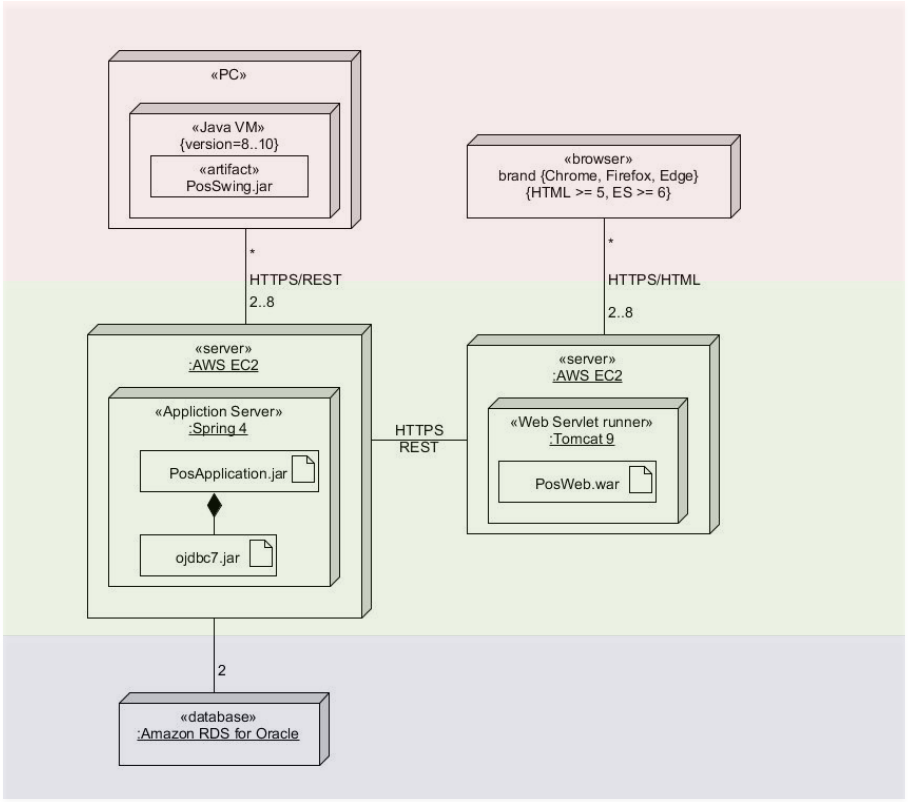
Domain layer

- Domain objects (data: entities, Value Objects…) are transported over the layers
  - Parameters of return values of methods
  - format can differ from layer to layer (OO in domain layer)

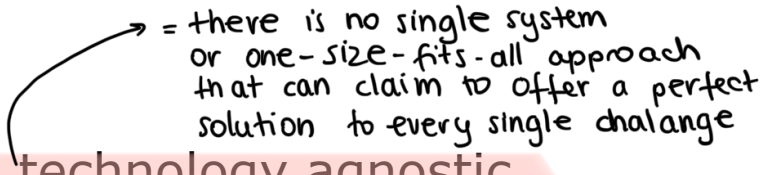
SalesLineItem Window





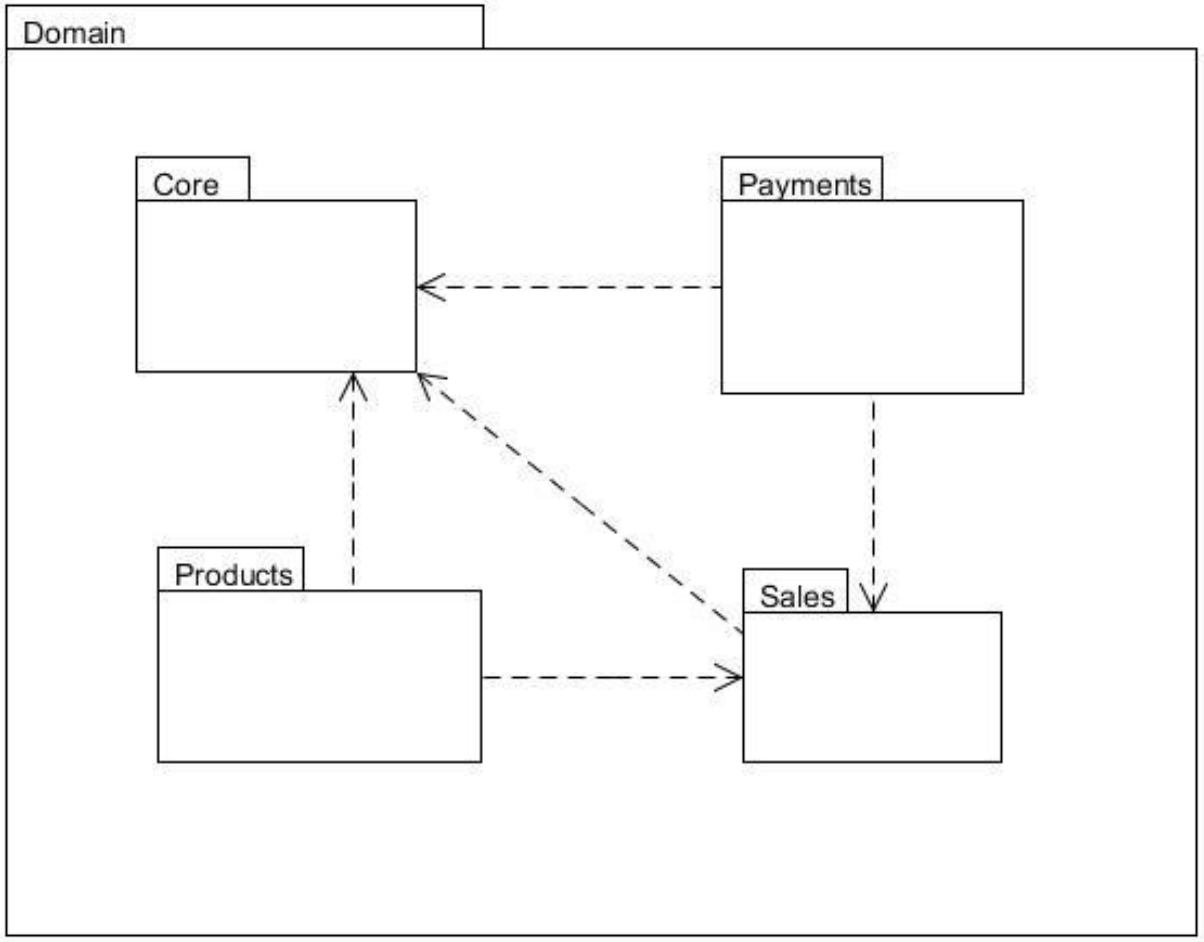


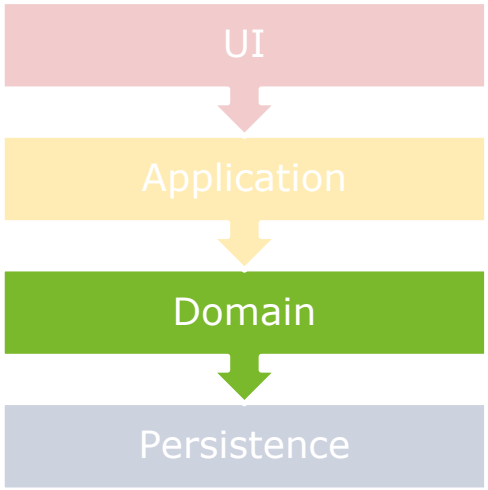
Domain layer: DTO



- Domain objects are intended to be technology agnostic.
- Transferring them to other layers can still be problematic
  - Mismatch of class design with data needed in other layers
    - Too much data transferred
    - Privacy concerns
  - Technology bleed
    - Dependency on libraries from other layers
    - Annotations galore
    - Other layer imposes a particular data organisation
- Data Transfer Object Pattern (DTO): If a domain object is not fit for communication with another layer, the data can be copied to a Data Transfer Object: a Value Object that is tailored for this job.
- Libraries for easy creation of DTO's: Mapstruct, ModelMapper…

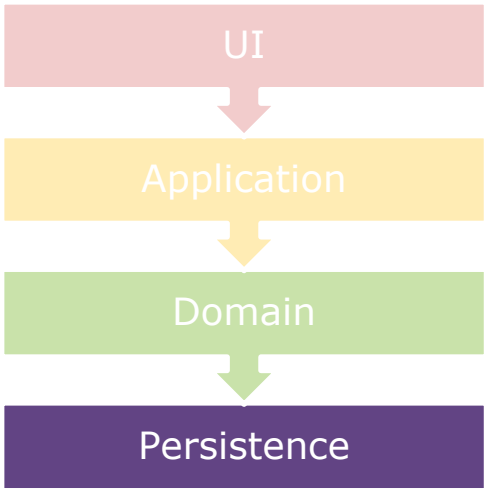
POS Domain package dependencies





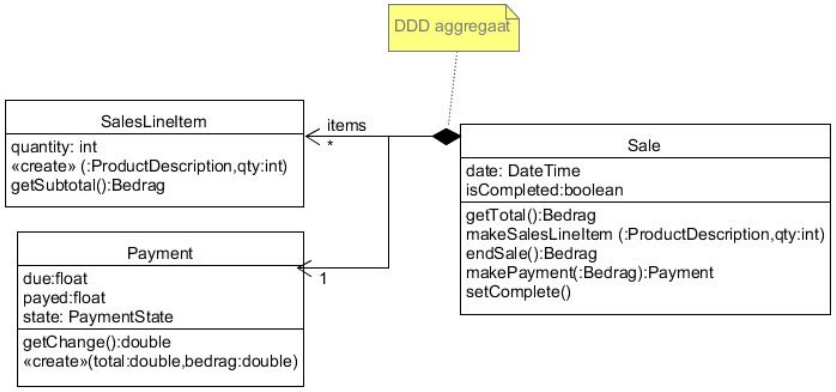
Persistence

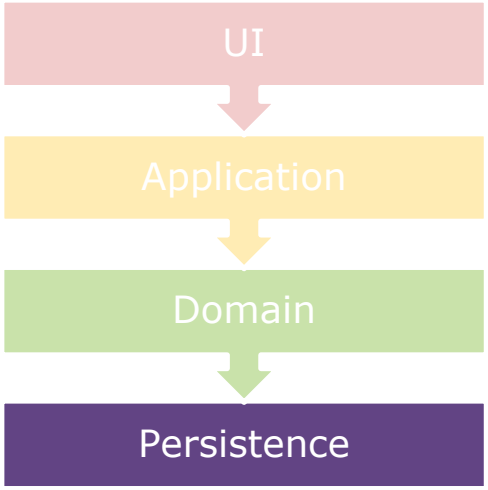
- Our discussion of the technical services layer, focuses on persistence: storing domain data
  - Other technical services: REST, authentication system, message bus…
- Often designed using Data Access Object (DAO) pattern
  - AKA: **DDD Repository**, Catalog
  - **Create/Read/Update/Destroy operations**
  - Examples: SaleDao, SaleCatalog, SaleRepo



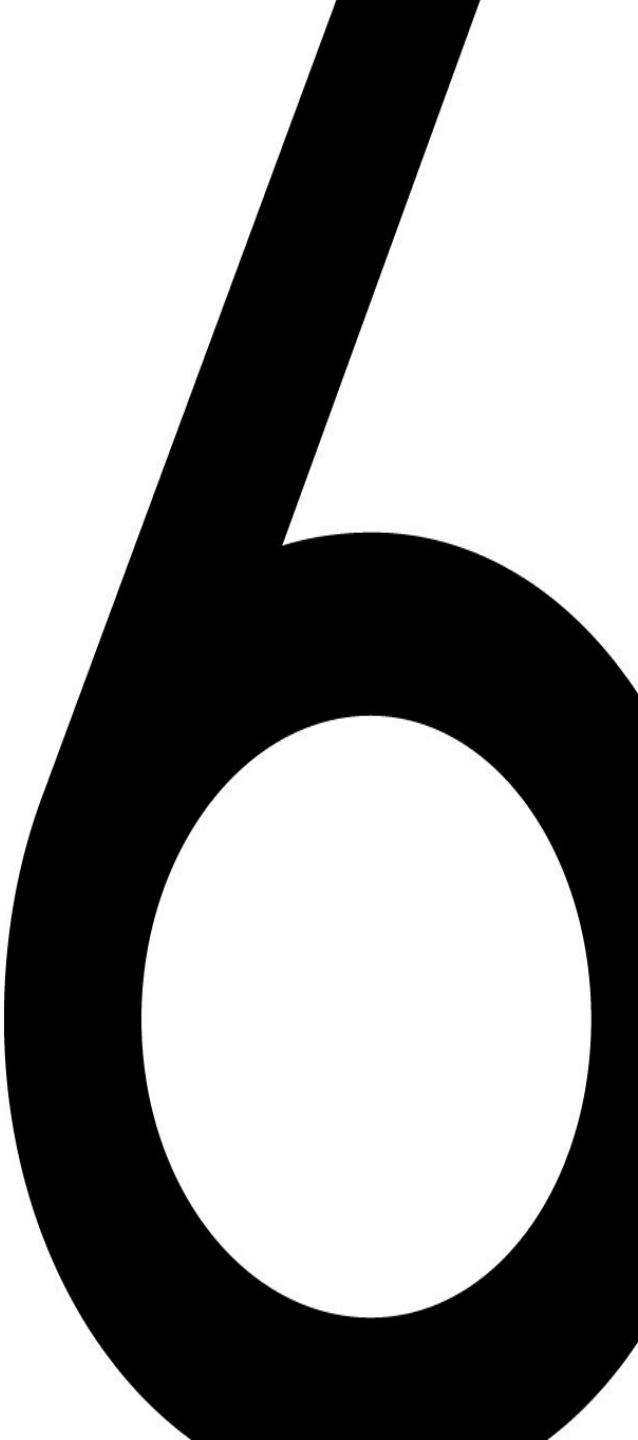
Persistence

- Guideline: use 1 Repository per **DDD aggregate** → = a cluster of domain objects that can be treated as a single unit
  - Aggregates are saved and retrieved as a unit by a repository
  - Reminder: internal logic for an aggregate and its components resides entirely within the aggregate
  - Logic that coordinates between multiple aggerates/entities resides within a service
    - The service will do multiple calls to repositories (or other services) to save changes to all collaboration entities





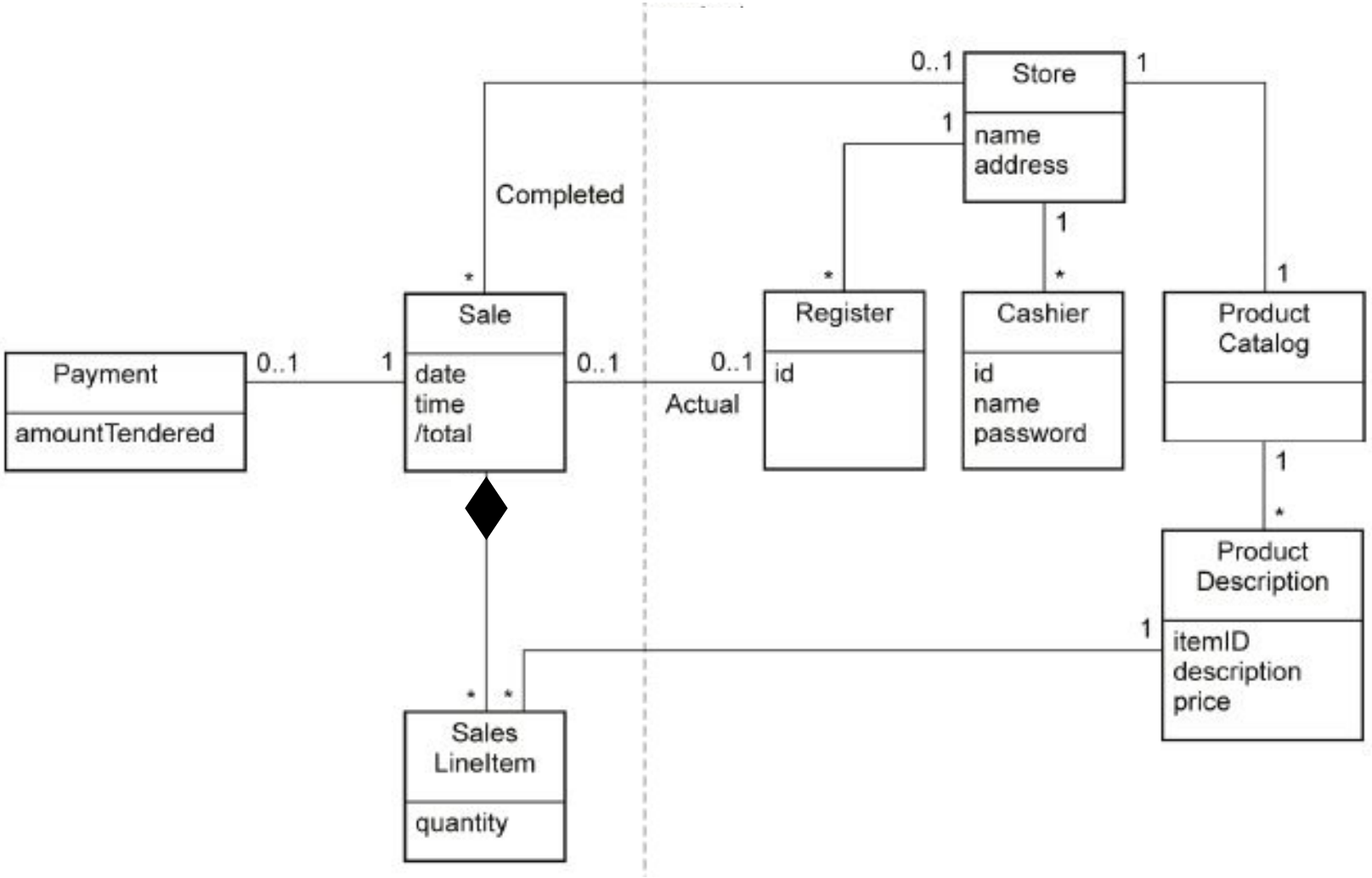
POS business case architecture



POS business case architecture

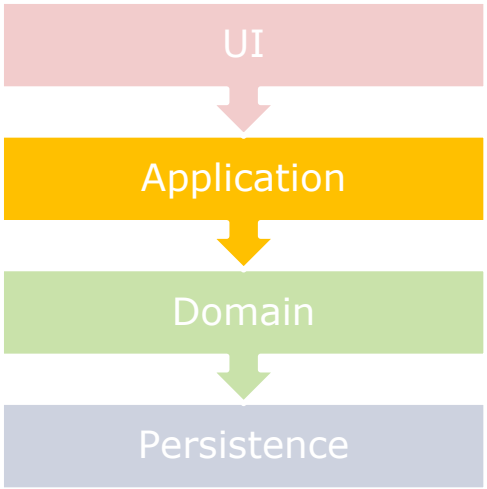
- Example of an architecture POS business case
  - This is just an example: other architectural choices can be made
  - Focus on domain layer
  - Does not include (user) interface layer
  - Persistence layer is handled in more detail in a separate module

POS domain


Store
name
address
Completed
Sale
date
time
/total
Payment
amountTendered
Register
id
Actual
Cashier
id
name
password
Product Catalog
Product Description
itemID
description
price
Sales LineItem
quantity
0..1
1
*

Application layer

- Controller is a **design** pattern.
  - Design and architectural patterns (Controller, Service, Repository…) are not present in the analysis. They are added during design.
    - alternative for Register: POSController



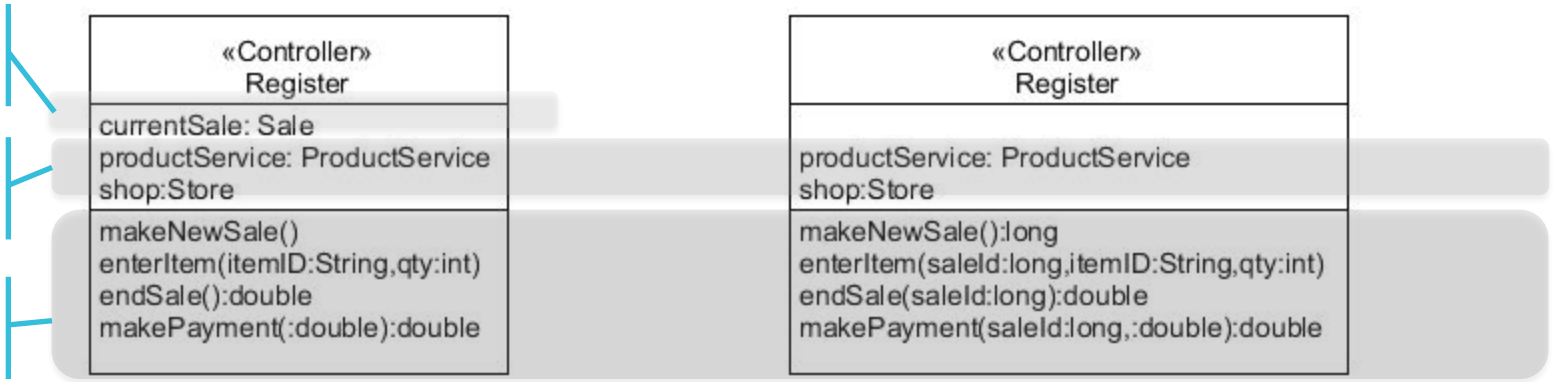
Stateful controller

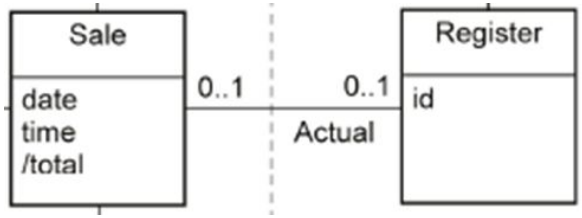
Stateless Controller

Session context

Domain services

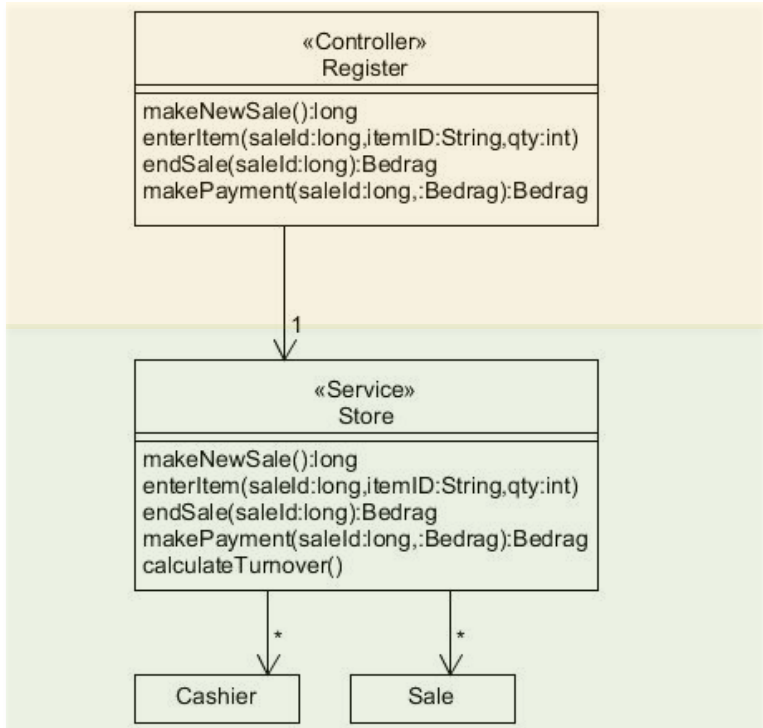
Scenario steps





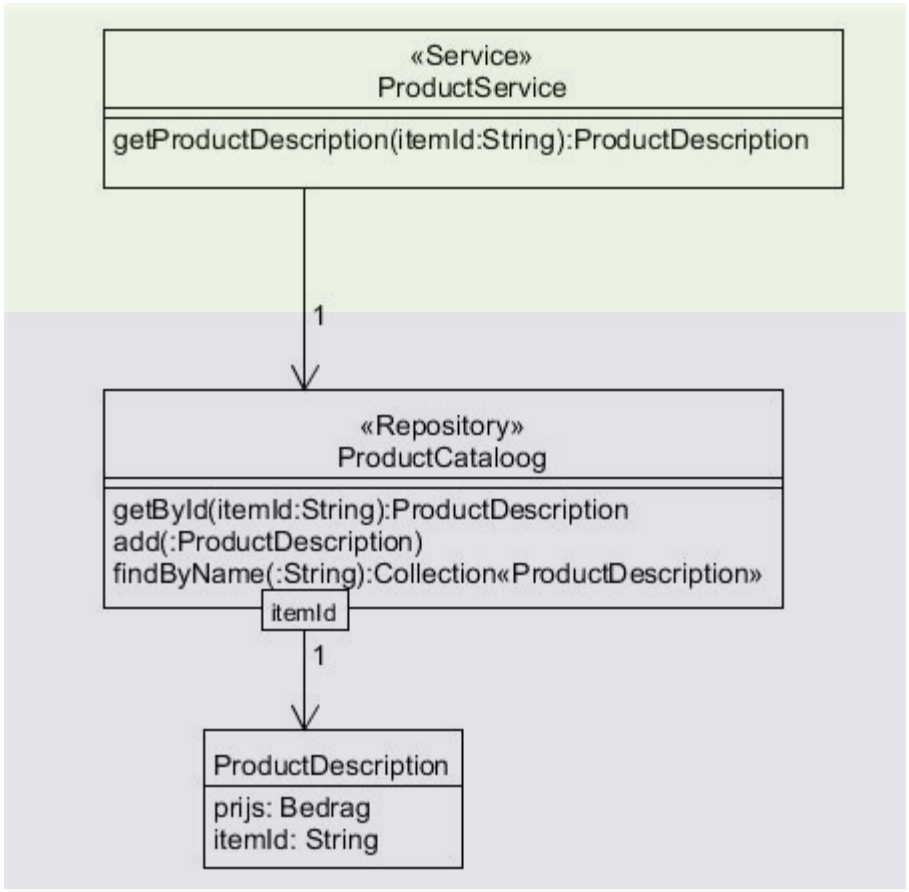
Domain layer: [DDD:Service]

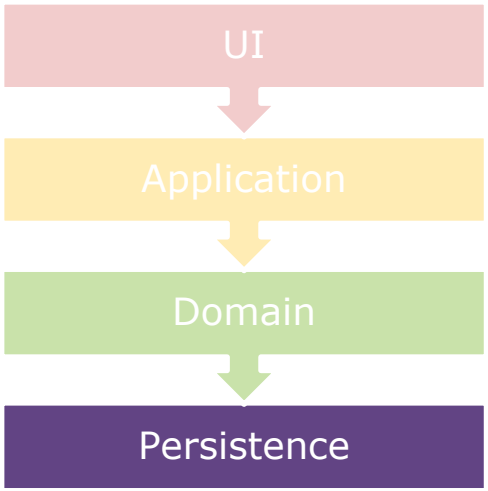
- The service contains business logic that coordinates between multiple objects
  - coordinates: do not put all code in the service. Use GRASP to assign responsibilities.
  - Delegates to other services and domain objects (information expert pattern)
    - Sale
    - Cashier
    - ProductDescription
    - …
- De service delegates retrieval and storage of domain objects to the persistence layer



Persistence

- Repository: implements CRUD operations
- A service can access multiple repositories





Persistence

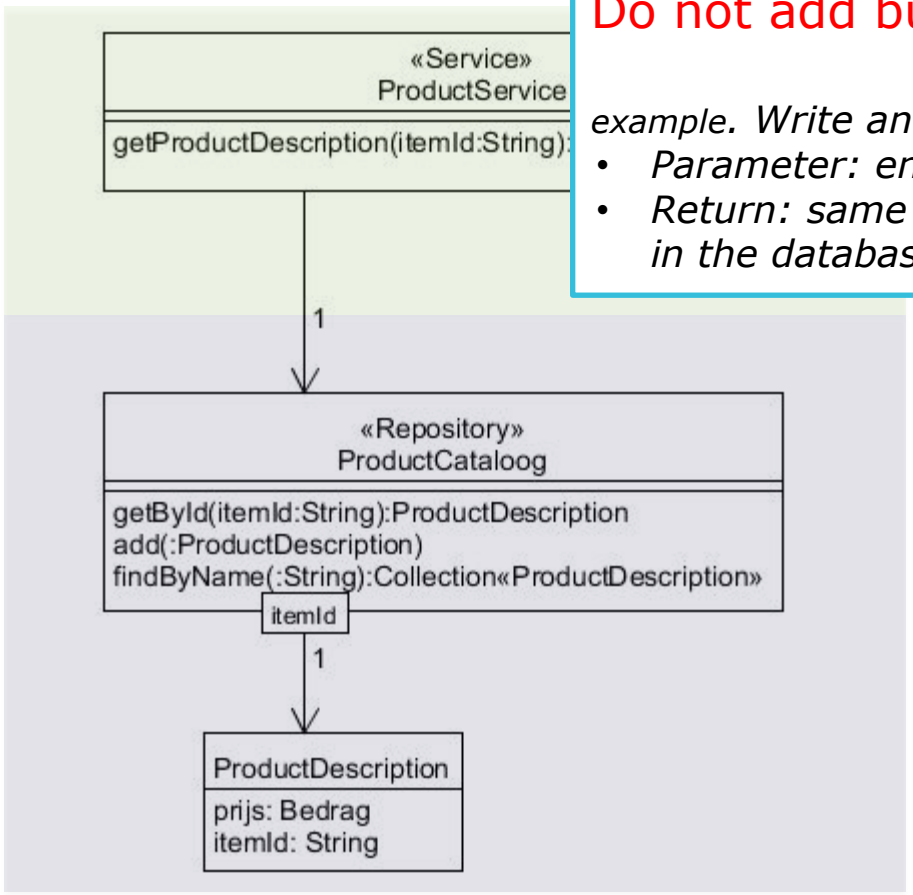
Methods in different repositories are similar

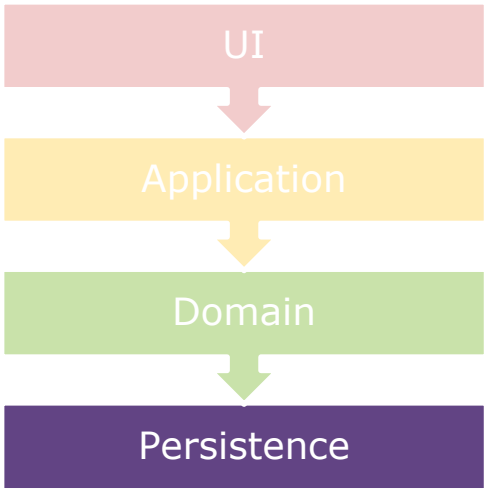
- Create/add
- findAll/Read
- findFiltered
- Update
- Delete/remove

Do not add business logic

example. Write an entity to the database

- *Parameter: entity instance*
- *Return: same entity including id assigned in the database*

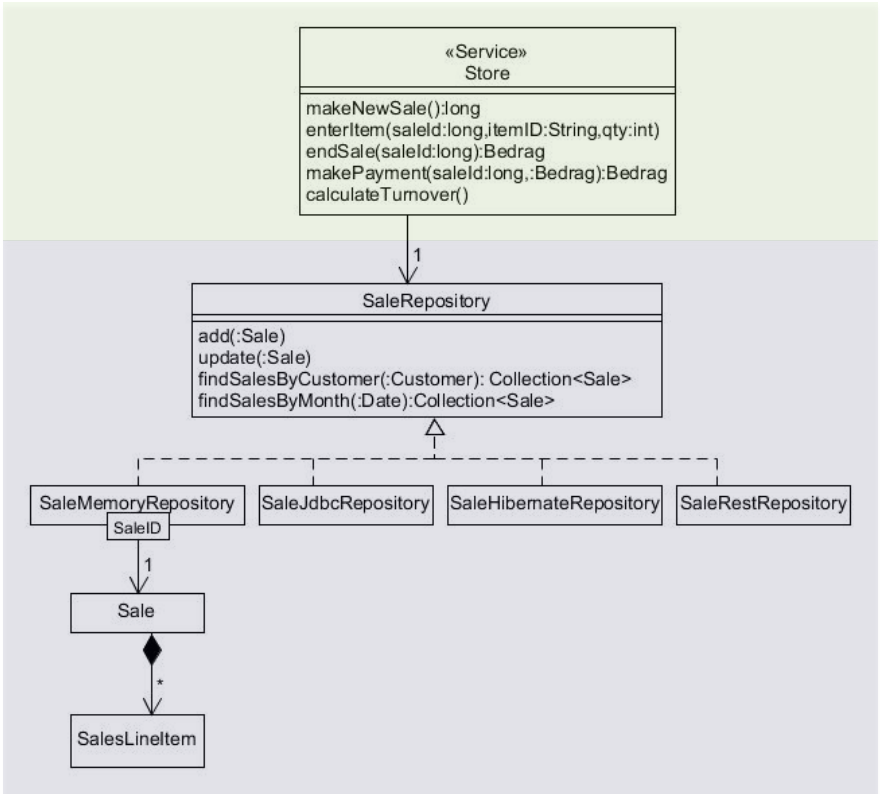


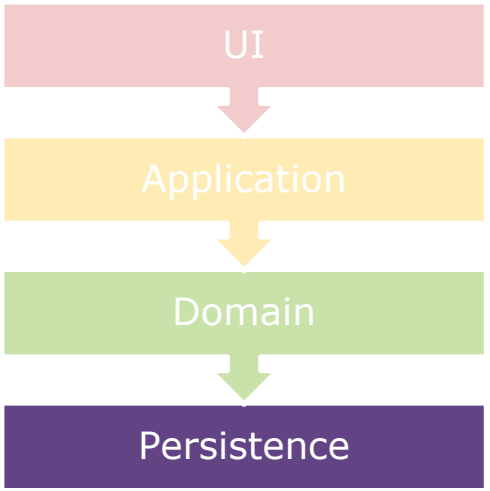


Persistence

- Repository implements technology specific aspects
  - Database, file, webservice, memory…
  - POS case uses Collections (memory)
  - Interface, called by service layer is technology neutral

= choosing the most appropiate technology ade quite to some1's needs are requirements



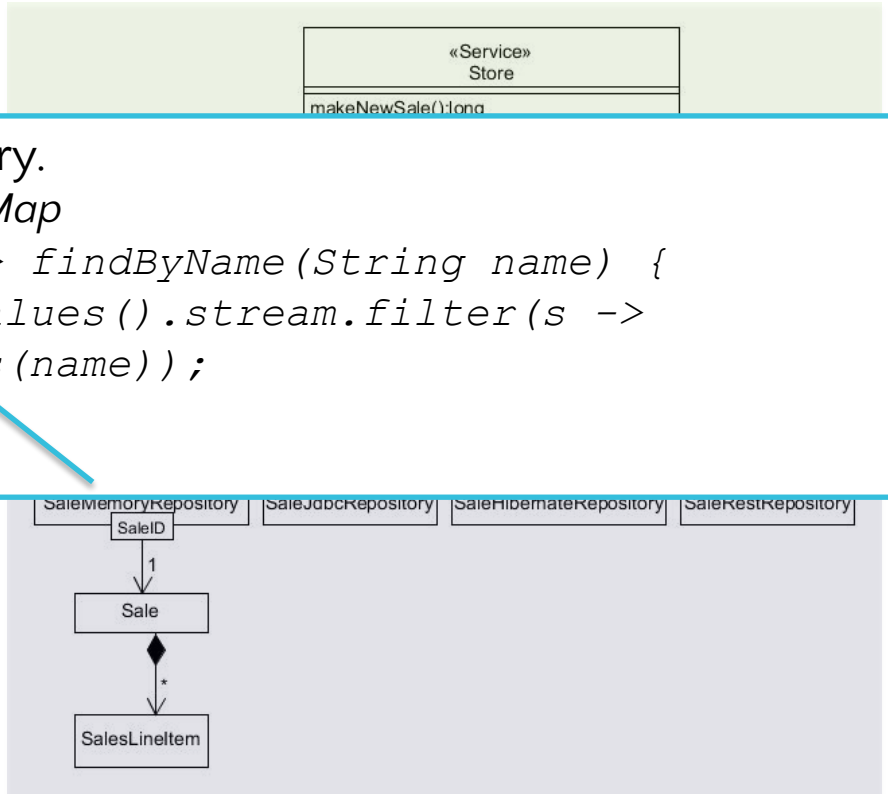


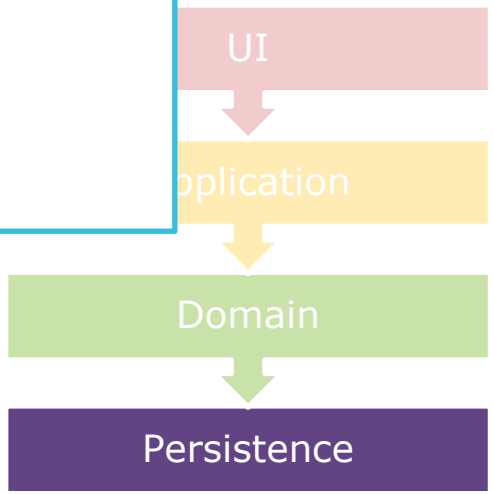
Persistence

- Repository implements technology specific aspects
  - Database, file, webservice, memory…
  - POS case uses Collections (memory)
  - Interface, called by service layer is technology neutral

Stores data in memory.
Repository contains a Map

```
public List<Sale> findByName(String name) {
  return sales.values().stream.filter(s ->
    s.name.equals(name));
}
```



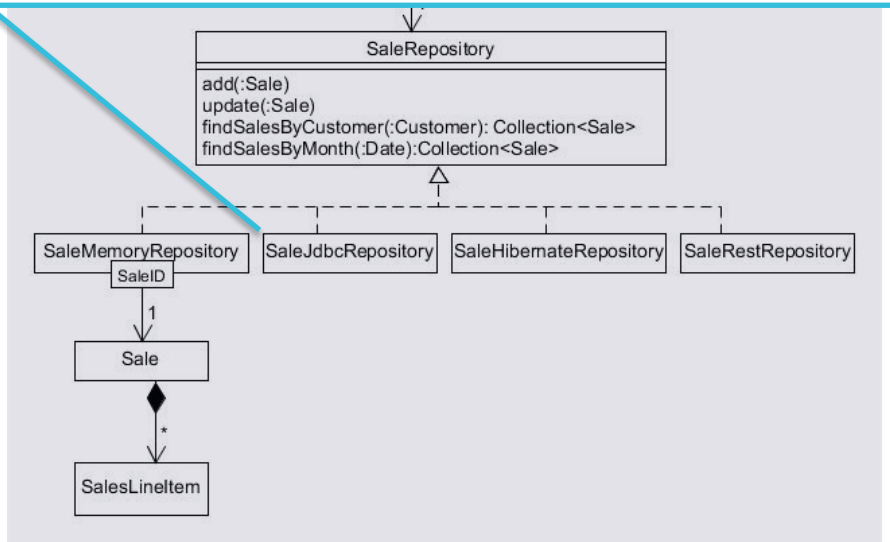


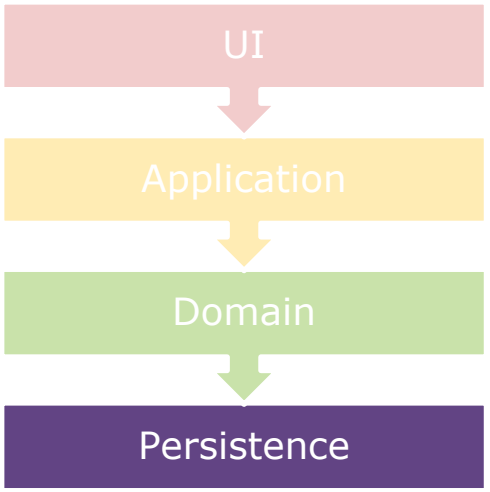
Persistence

- Repository implements technology specific aspects

Reads and writes data using embedded SQL (JDBC)

```
public List<Sale> findByName(String name) {
  String sql = "SELECT * FROM Sales WHERE name = ?";
    PreparedStatement ps = c.prepareStatement(
      "SELECT * FROM Sales WHERE name = ?");
    ps.setString(1, "John Smith");
    ResultSet rs = ps.executeQuery();
    …
}
```





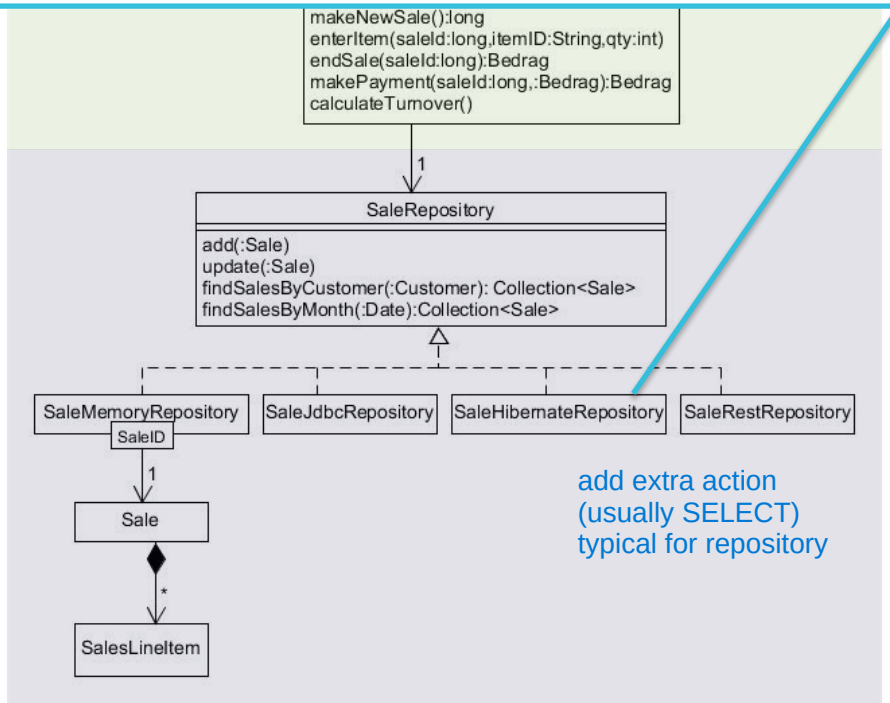
Persis

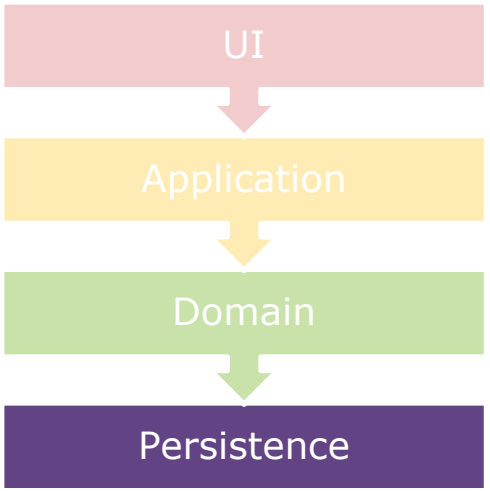
Reads and writes data using Object Relational Mapping(ORM) framework:

- The framework will generate the database SQL for your

```
// JPA example
public Collection<Sale> findByName(String name) {
   return entityManagar.createQuery(
      "SELECT s FROM Sales s WHERE s.name = :name")
   .setParameter("name",name)
   .getResultList();
}
```

- Reposi
  - Datab
  - POS c
  - Interf

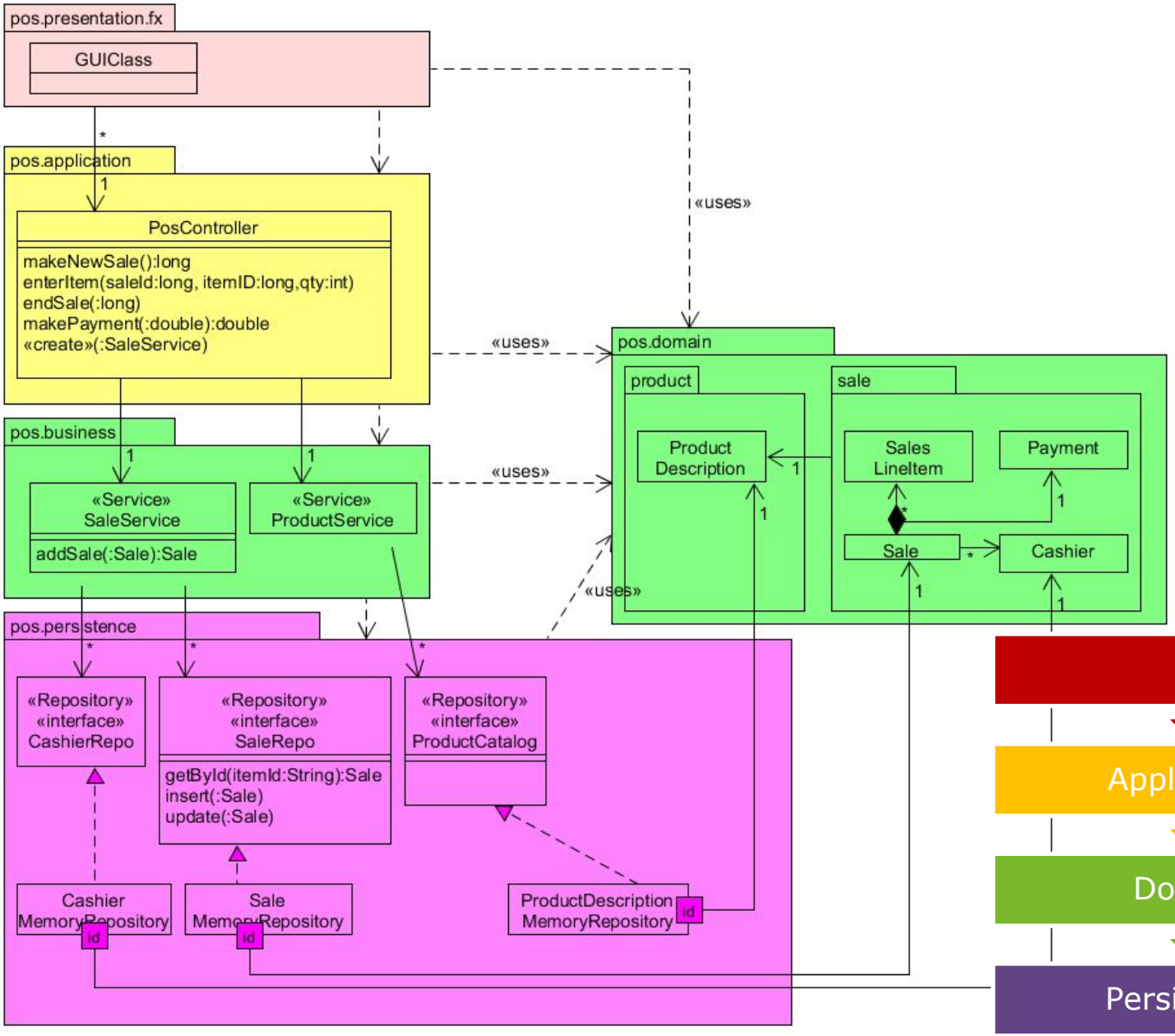




POS => 3 layer architecture

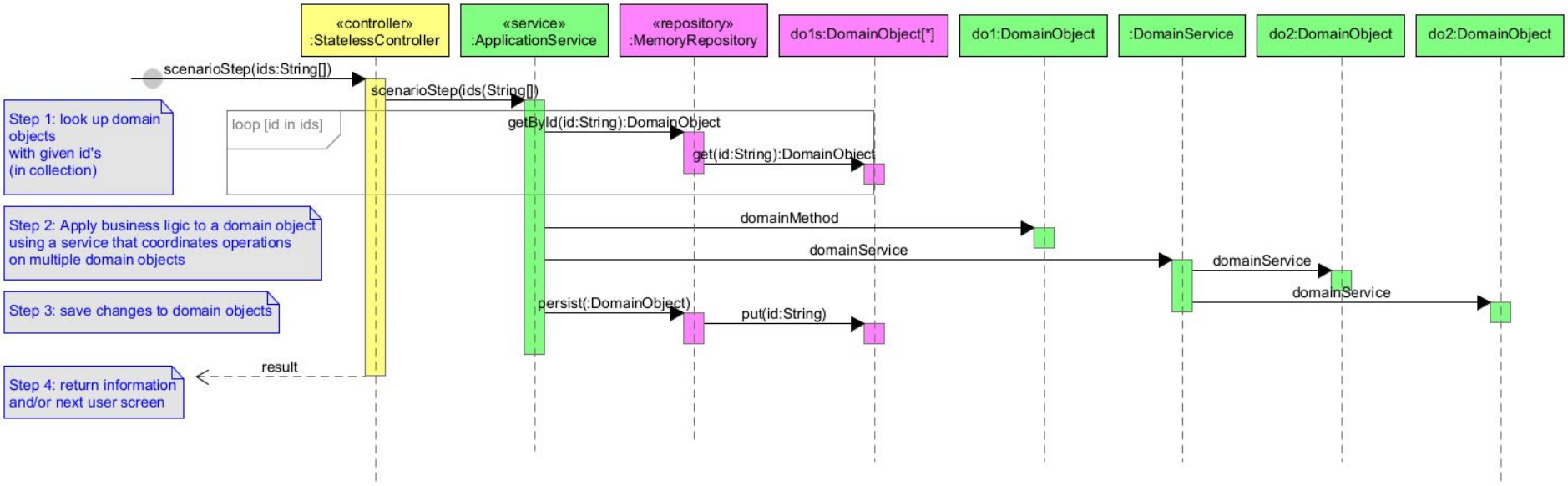
- Add classes to fit the POS case in a 3-layer architecture
  - Add a high level package for each layer

POS 3-layer model



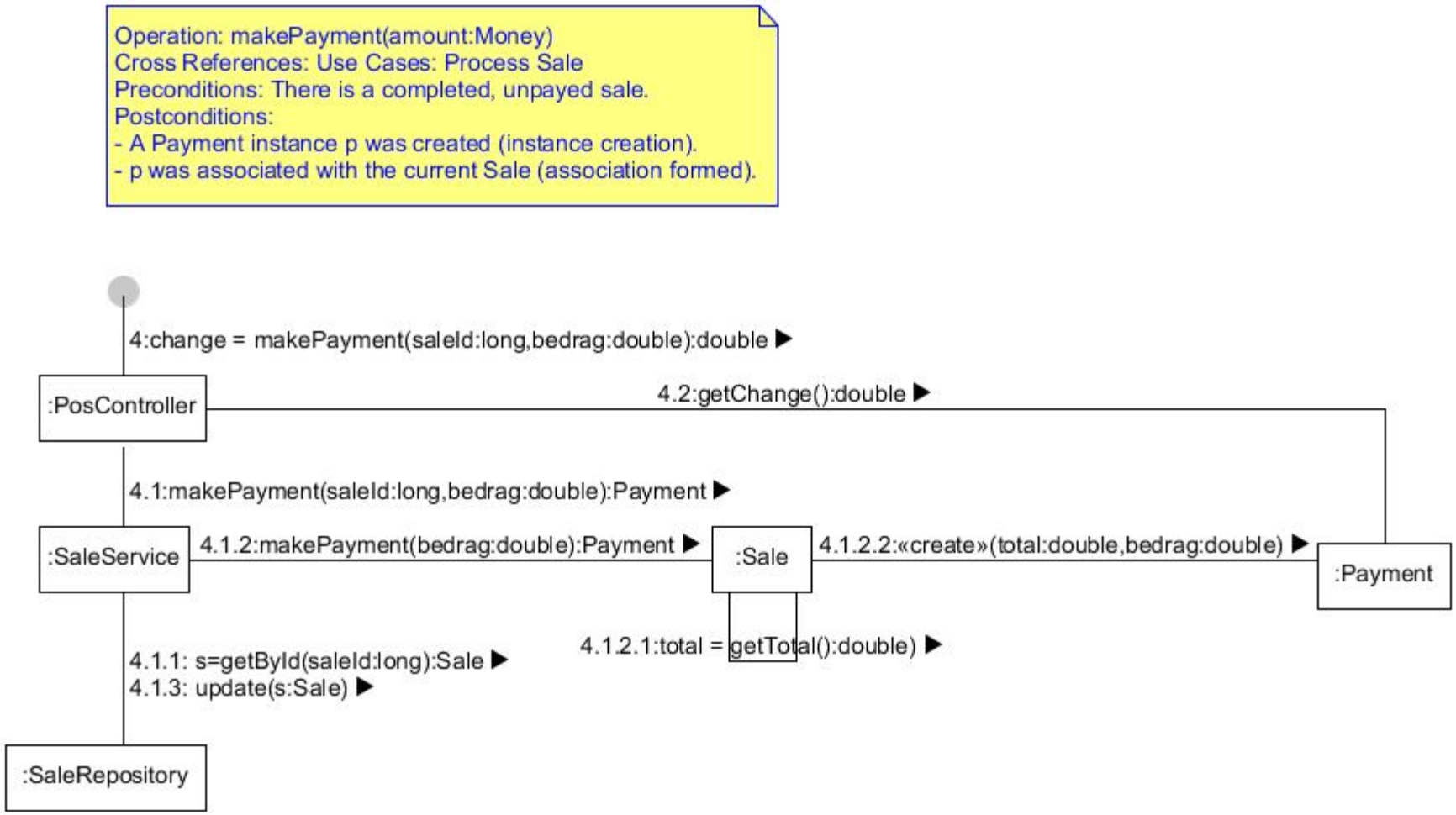
Architectural Interaction diagram for 3-layer model

- Explains the working of your logical architecture with a generic interaction diagram (generic classes aService, repository1…)
- Afterwards you can limit interaction diagrams for a services to what is really specific for that service

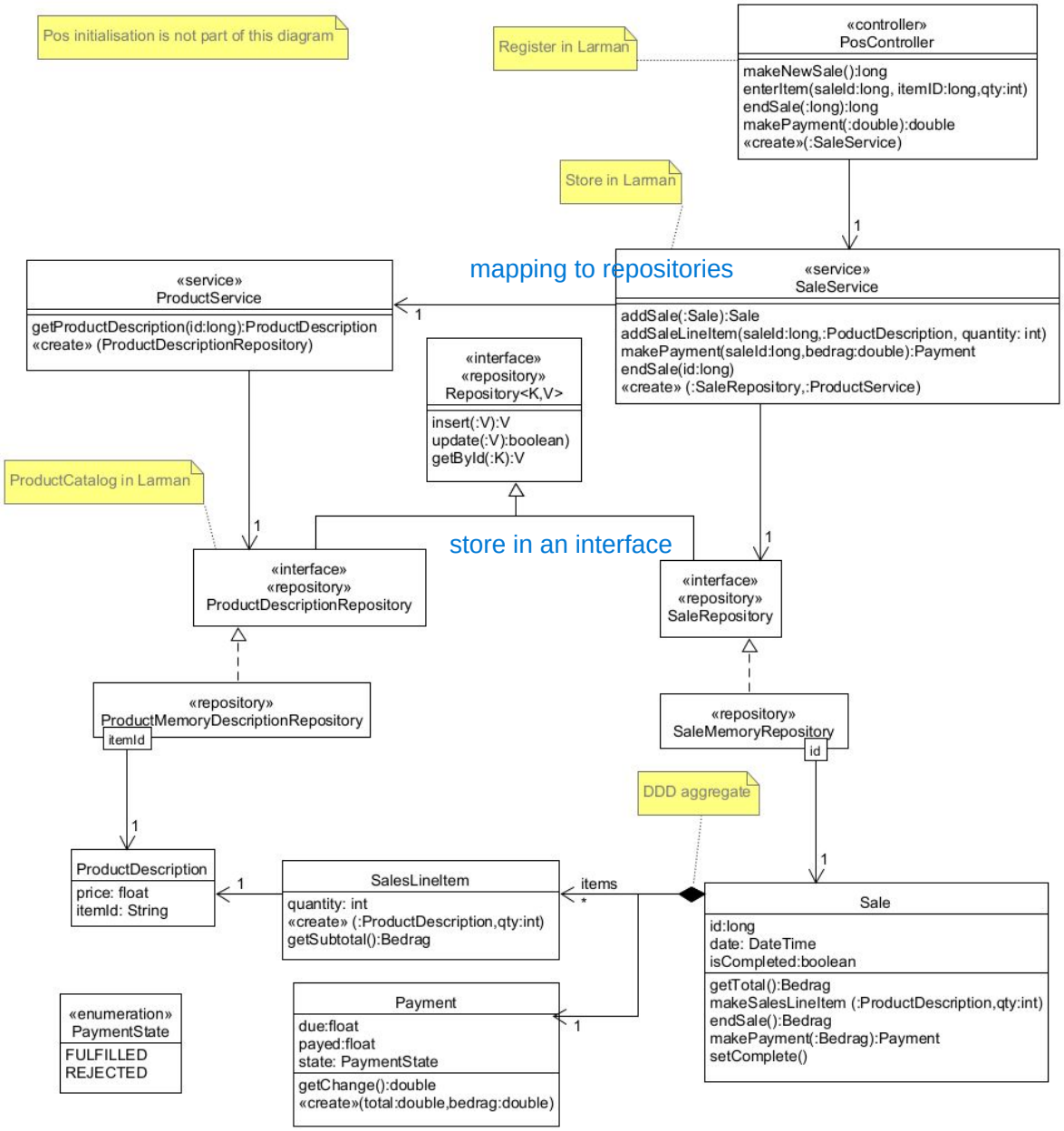


Elaboration *process sale* in architecture

- On Canvas you can find diagrams and code for the process sale user stories using the given architecture



Detailed design class diagram *process sale*



DDD: partial pattern overview

Overzicht

1. Introduction
2. Packages
3. layered architecture: basics
4. UML component and deployment diagrams
5. layered architecture: elaboration
6. POS business case architecture