Domain Model

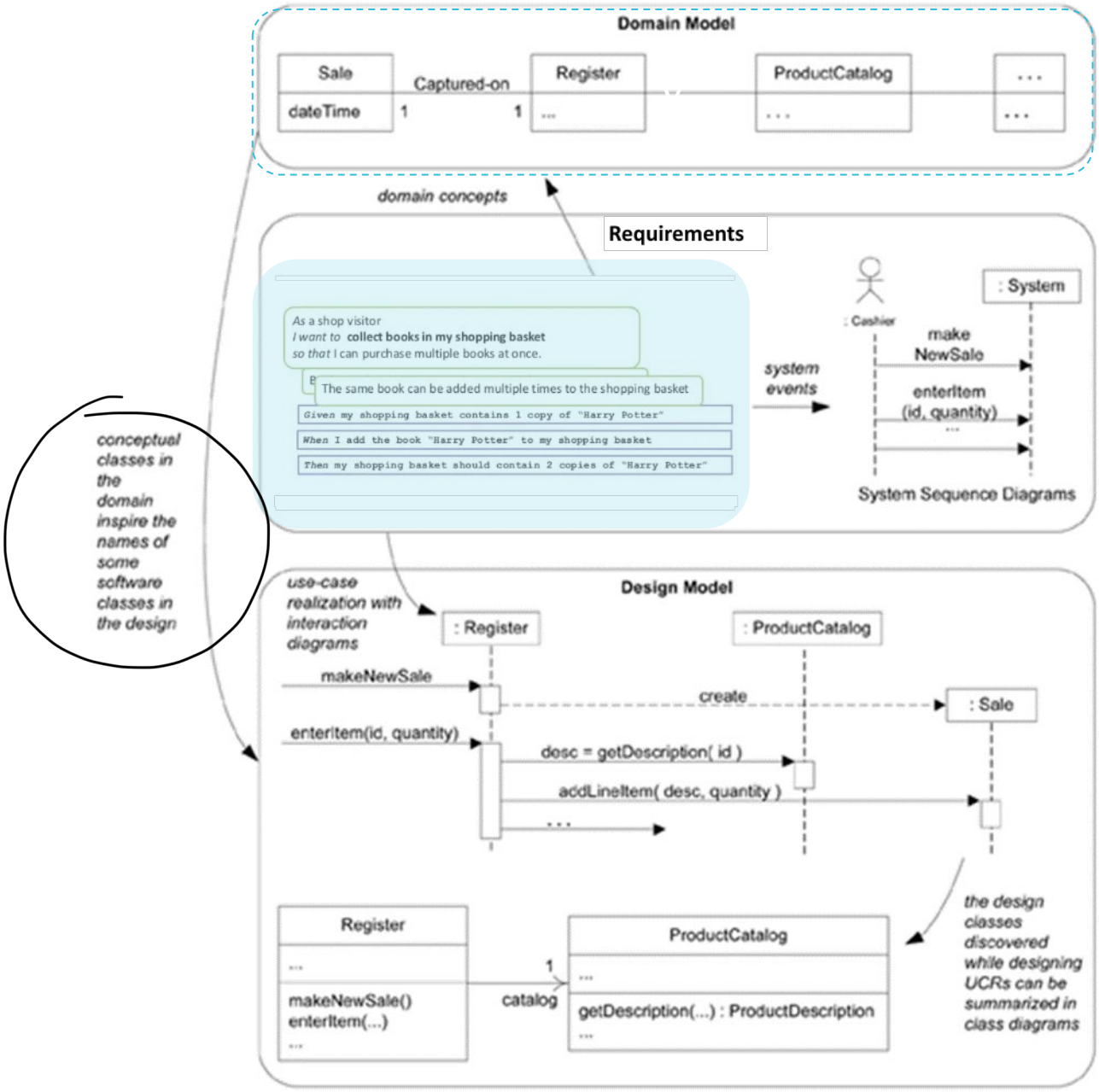
Programming 2.2

Analysis

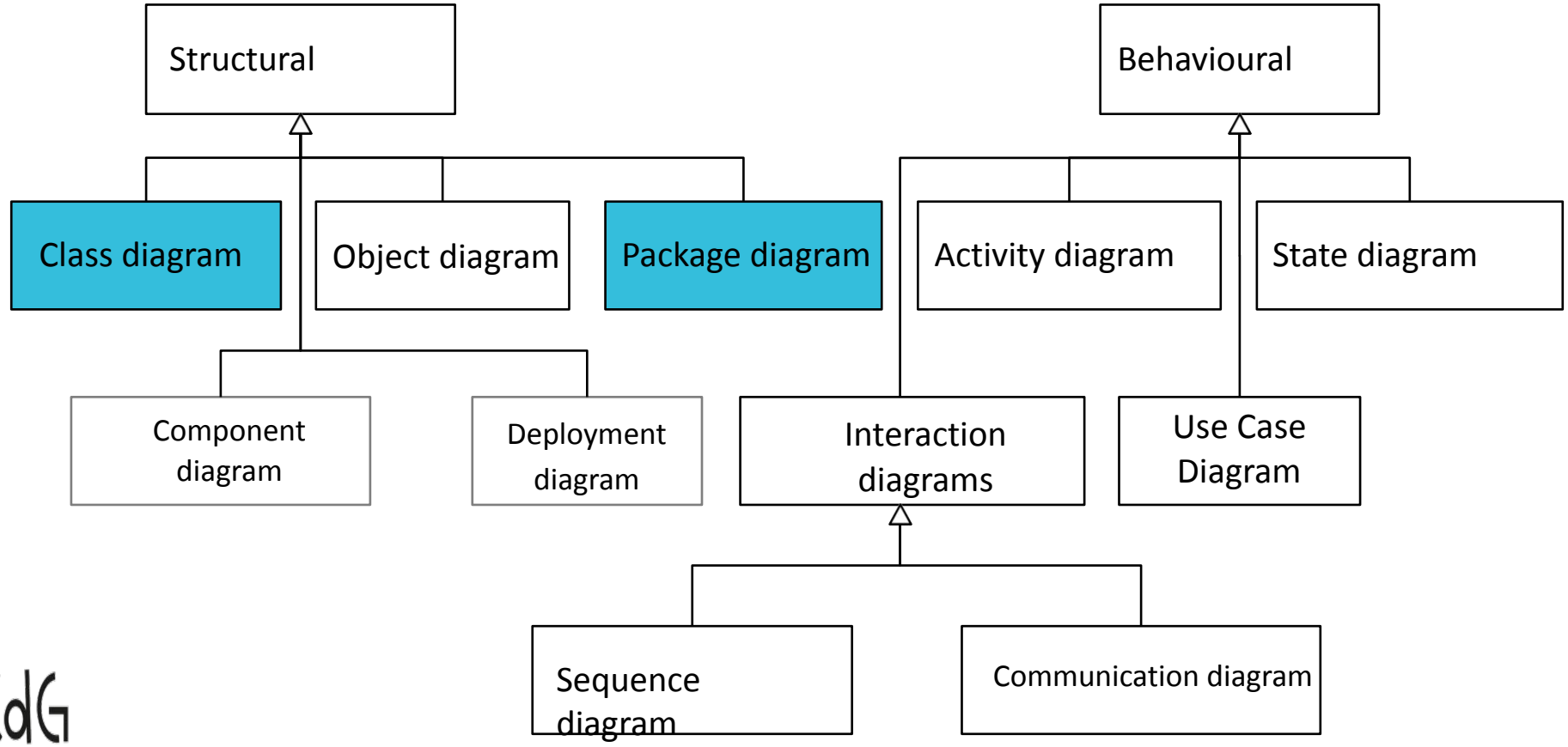
KdG Karel de Grote Hogeschool

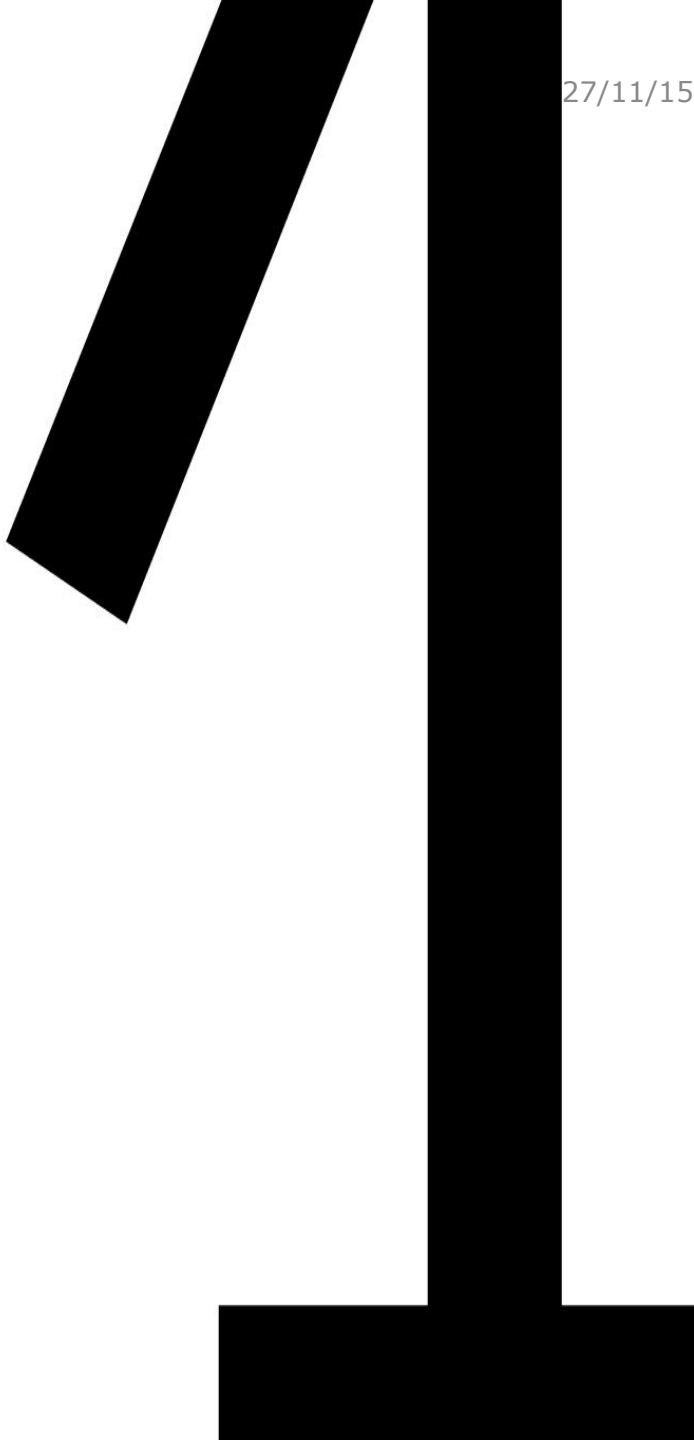
Agenda

1. Extending UML
2. Domain model (class diagram)
   - Entities and associations
   - Data types
   - inheritance
   - datastructures overview
   - guidelines
3. DDD: strategic design


Domain Model
Sale
dateTime
Captured-on
1
1
Register
...
ProductCatalog
...
...
...
domain concepts
Requirements
As a shop visitor
I want to collect books in my shopping basket
so that I can purchase multiple books at once.
The same book can be added multiple times to the shopping basket
Given my shopping basket contains 1 copy of "Harry Potter"
When I add the book "Harry Potter" to my shopping basket
Then my shopping basket should contain 2 copies of "Harry Potter"
system events
: Cashier
: System
make NewSale
enterItem (id, quantity)
...
System Sequence Diagrams
conceptual classes in the domain inspire the names of some software classes in the design
use-case realization with interaction diagrams
Design Model
: Register
: ProductCatalog
: Sale
makeNewSale
create
enterItem(id, quantity)
desc = getDescription(id)
addLineItem(desc, quantity)
...
Register
...
makeNewSale()
enterItem(...)
...
1
catalog
ProductCatalog
...
getDescription(...) : ProductDescription
...
the design classes discovered while designing UCRs can be summarized in class diagrams

UML diagrams



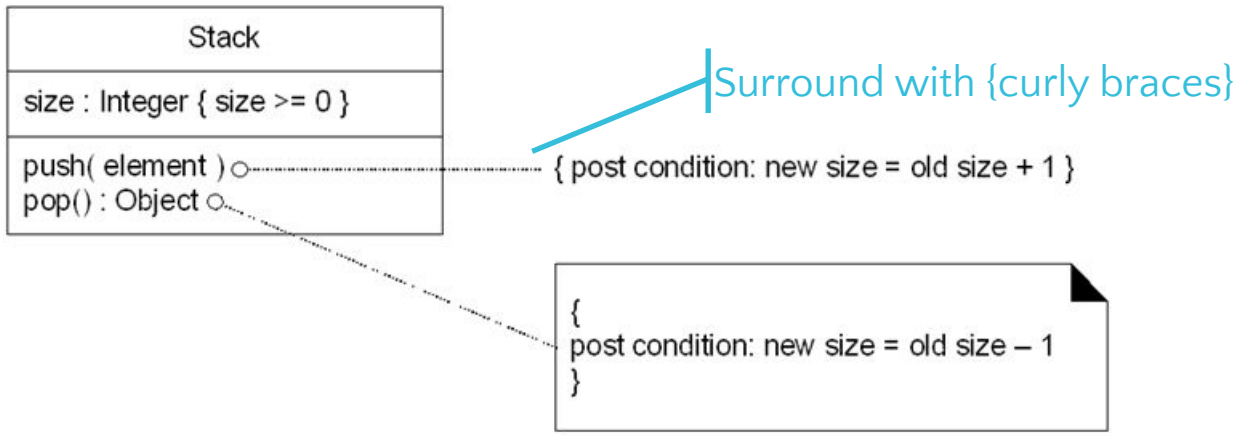


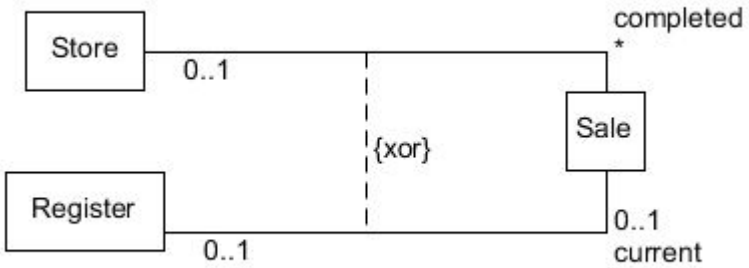
Extending UML: constraints and stereotypes

KdG
Karel de Grote Hogeschool

{constraint}: restriction on a UML element

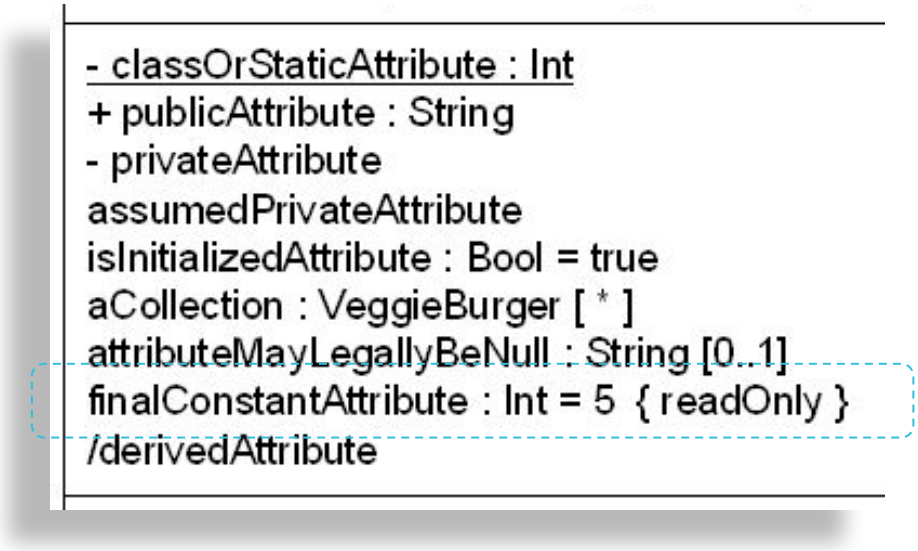
- Constraints indicate an additional restriction on a UML element by specifying a locgial condition, property or a restricted set of values





xor: exclusive or.
A sale is either linked with a Store OR a register

{constraint}


- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute


Money
amount:double
valuta {DOLLAR,EURO}

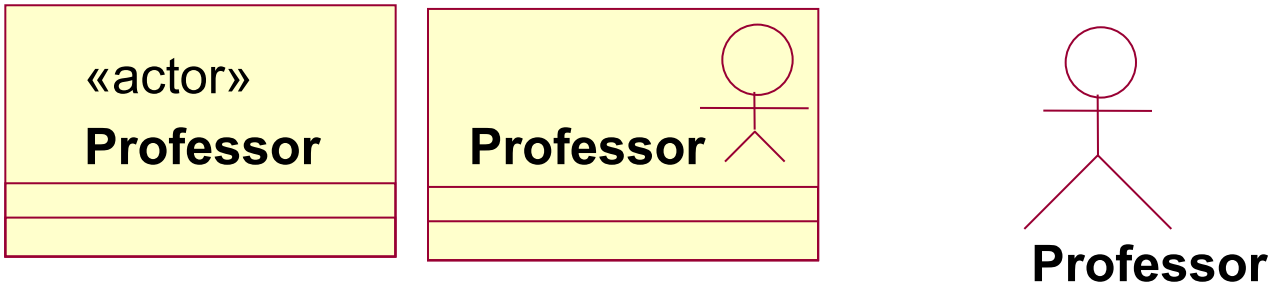
Stereotypes

- You can add new elements to the UML language by adding stereotypes: add a name between «guillemets» on an existing UML element
- Stereotypes can contain
  - extra constraints
  - extra attributes (name/value pairs)
- Some stereotypes are predefined as UML keywords: «enumeration», «actor», «include», «dataType» …

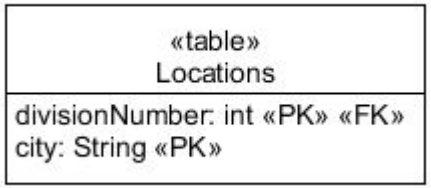
Stereotypes

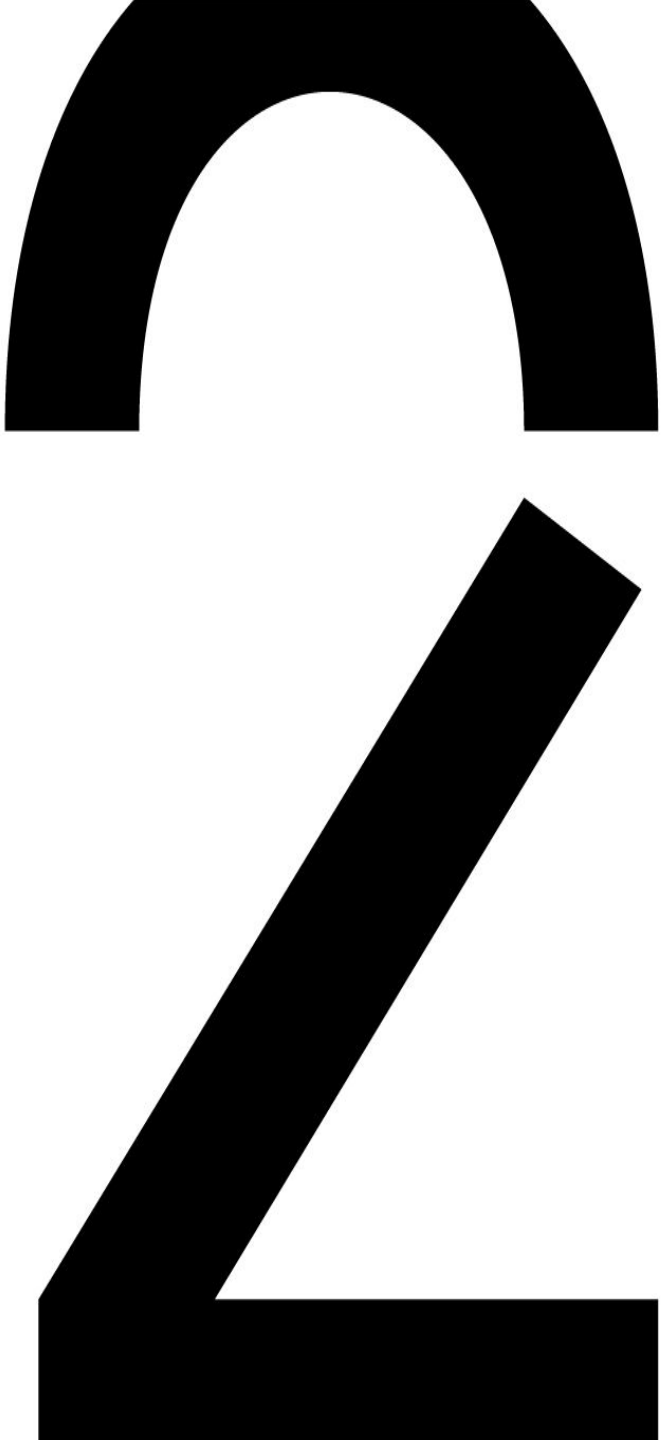
- Stereotypes can have their own graphical representation (icon)
- 3 options for showing stereotypes

«name» decoration icon



- You can add your own stereotypes (just add the «name»)





Domain model

KdG
Karel de Grote
Hogeschool

Entities and associations

KdG Karel de Grote Hogeschool

Domain

- Conceptual perspective of objects in the real world
  - Business concepts in the problem domain
  - no IT concepts (cloud, scanner…)
  - Technology neutral (independent of programming language, database…)
- Guideline: work like a mapmaker
  - Use existing names in the territory
  - Exclude what is not relevant
  - Work at the level of detail needed
  - Do not add things that are not there
- Domain Driven Design (DDD) designs software based on your domain model
  - The domain model evolves as your design advances

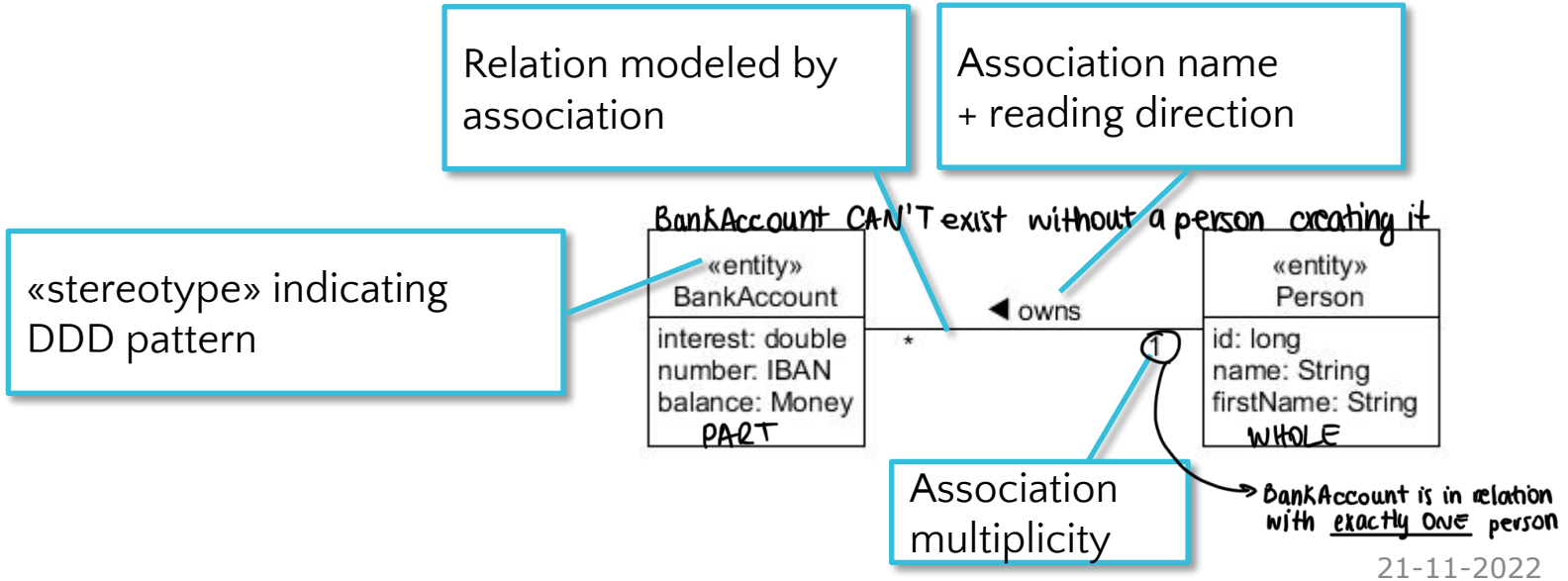
Karel de Grote Hogeschool

Entities

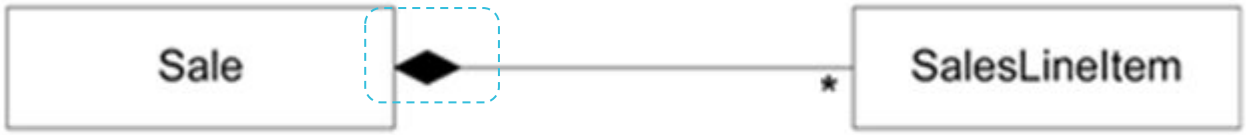
- In DDD, domain concepts are called entities.
- Implemented as Classes
  - Instances are reference Objects
- Have an explicit identity
  - Often one attribute or a subset of attributes
    - Example: BankAccount has an IBAN number
    - In Java equals() and hashcode() are based on identifying attributes
- Have a lifecycle in which the state (data in some attributes) can change. Changing the state does not change the identity.
    - Example: even if you change the balance, you are still handling the same Bankaccount

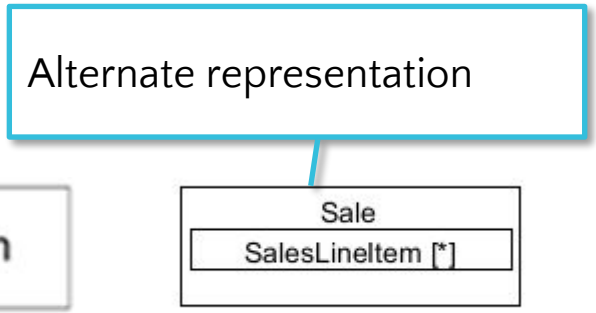
Associations

- Relations between entities are modelled as UML associations
  - implemented as attributes (Person will typically be an attribute of BankAccount)
  - Attributes contain the **state** of an object
- Entities are typically stored by the system and can be referred to by multiple other entities
  - It often makes sense to search for a stored entity

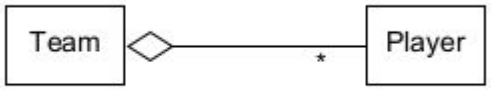


Composite (DDD:aggregate)



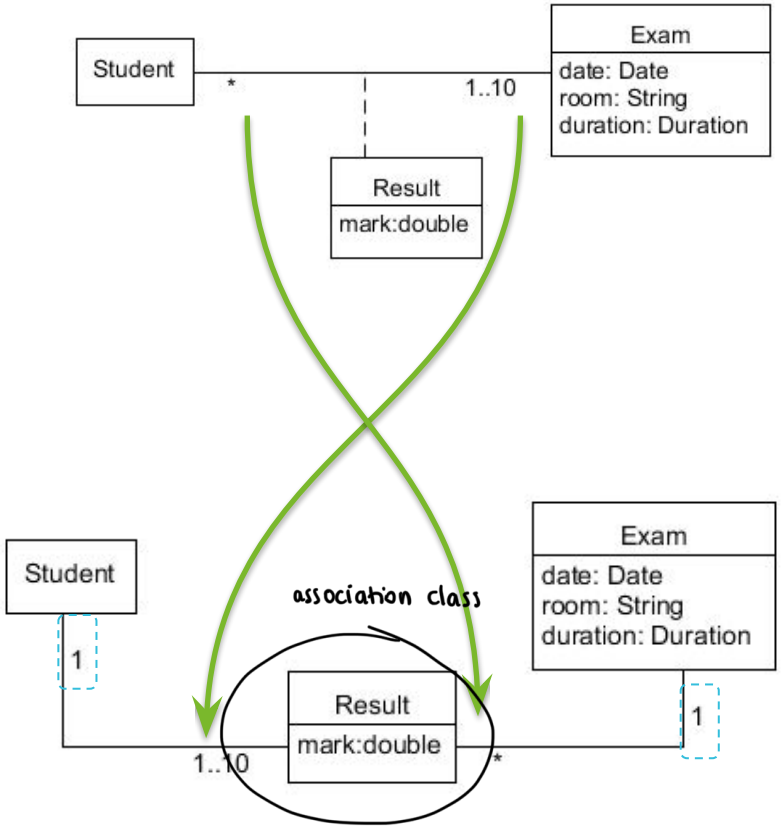


- Strong form of association: "owns or contains" relation
- whole (Sale) / part (SalesLineItem) relationship
- The whole is responsible for creating the parts
  - Parts cannot exist without the whole
  - Part object is member of exactly one whole object
- In DDD this is called an aggregate
  - All operations on the aggregate are always performed through the whole (in DDD called aggregate root)
  - Unfortunatly UML uses the term aggregate (open diamond <>) for a weaker form of composition, but this UML notation is not often used



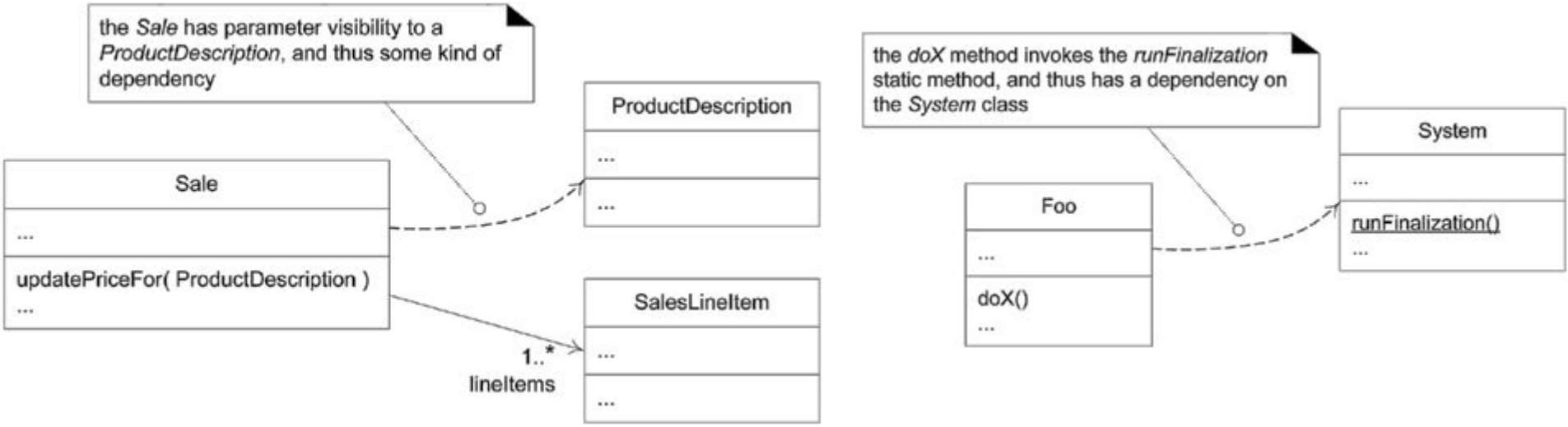
Association class

- If an association has attributes, you can put them in an association class
  - The association class contains data on a relation between 1 instance at each side (there can be a Result for each student-exam combination)
  - Often implemented as
    - Is this exactly the same?



Dependencies

- A dashed arrow (with open head -->) indicates that one UML element depends on another
  - If the other element changes this impacts the first
  - Can be used in all UML diagrams
- An association implies a dependency (no need to draw one explicitly)



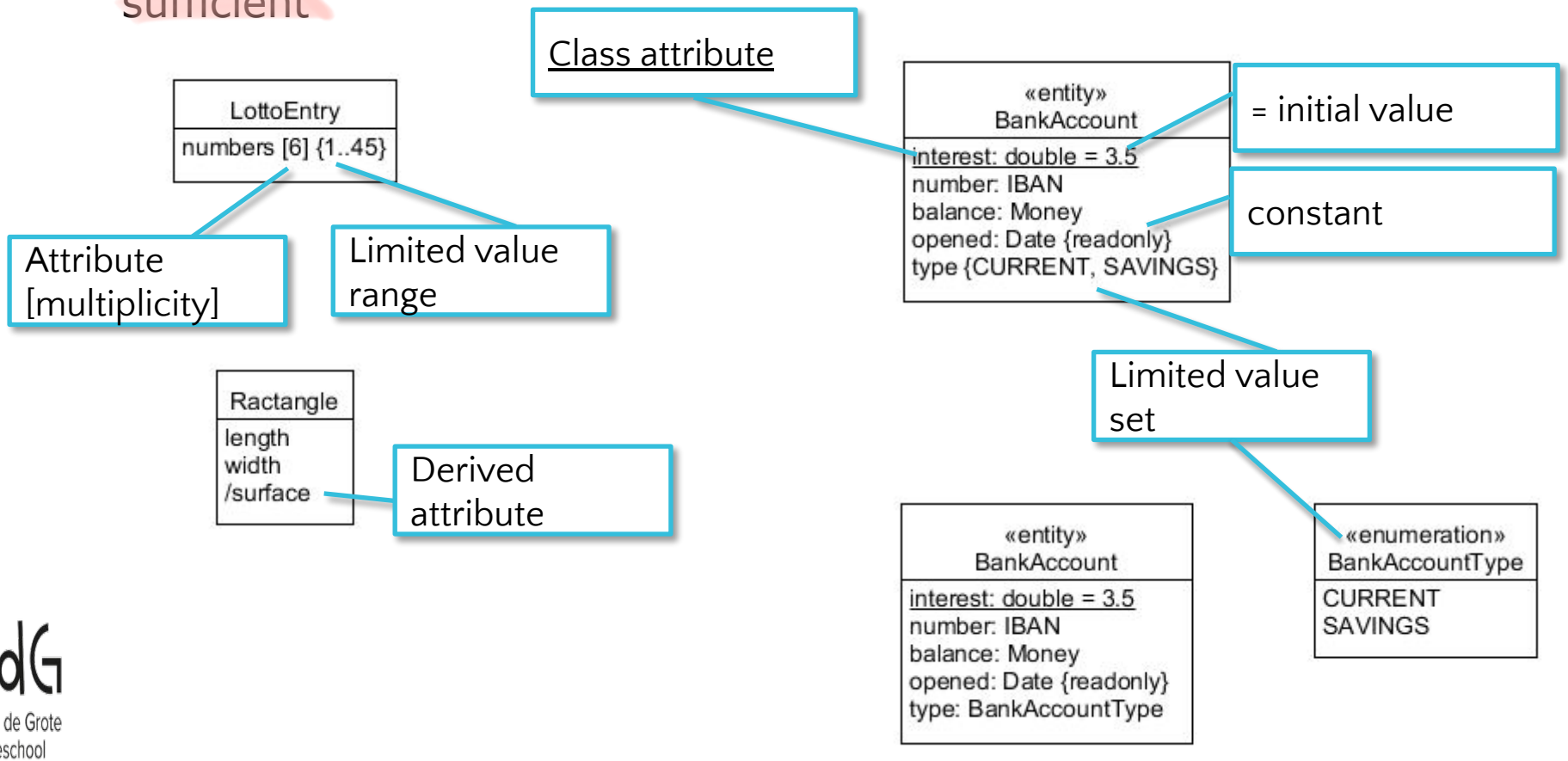
Data types

KdG Karel de Grote Hogeschool

Primitive datatypes

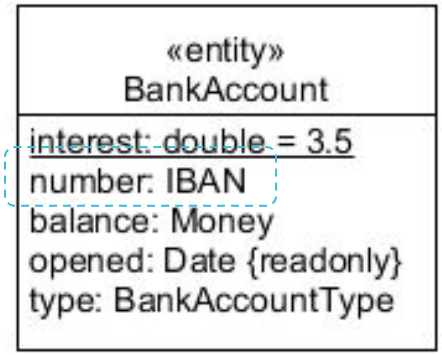
Refresher

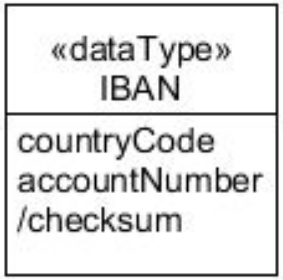
- Simple values like person name
- Modelled as UML attributes
  - During analysis, a semantic datatype (number, date, text) is sufficient



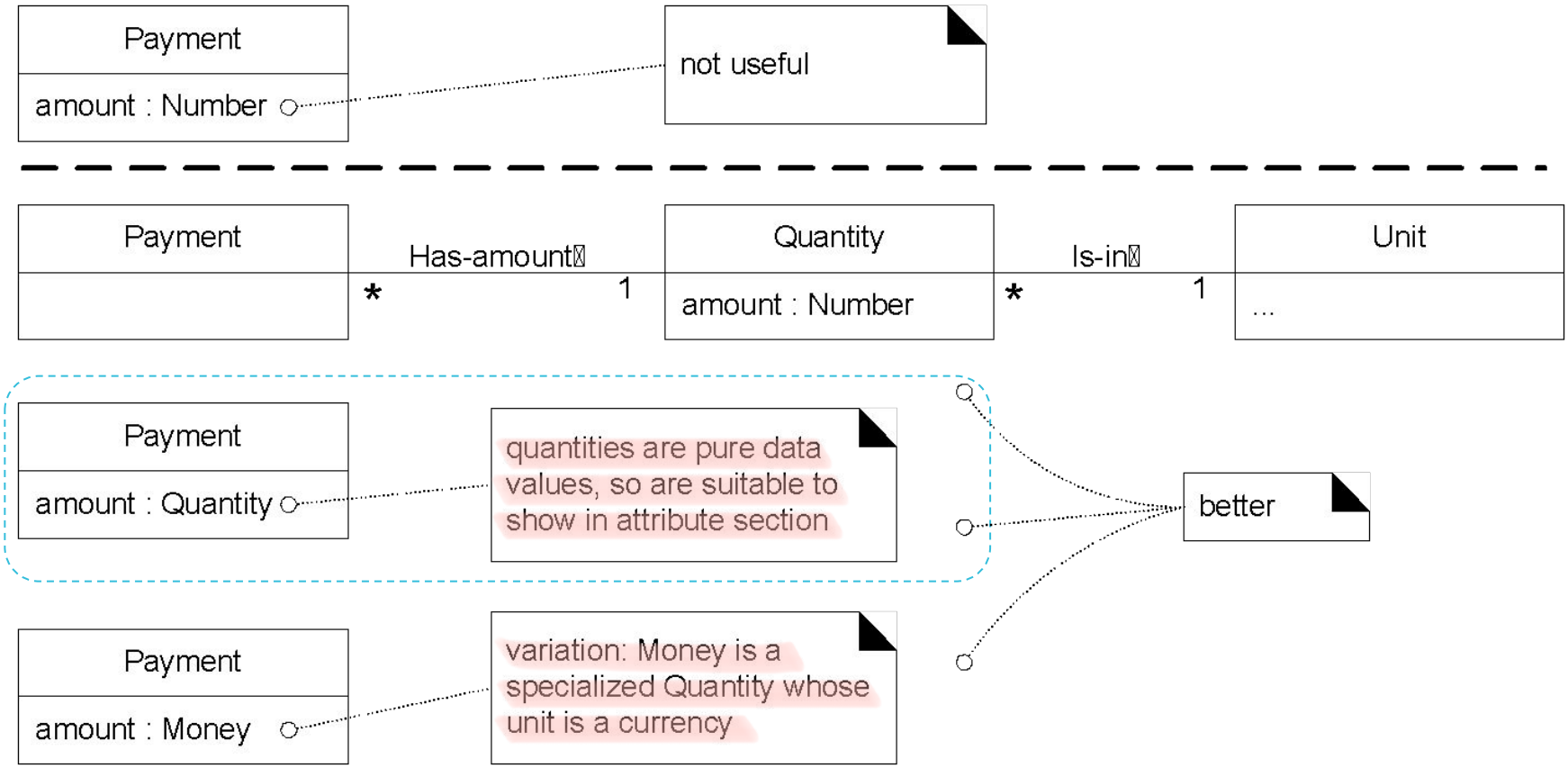
Value objects

- DDD Value objects are instances of complex datatypes (or datatype classes): values with multiple fields (example: Date)
  - «dataType» UML stereotype
- In UML classes refer to datatypes as attributes
  - Drawing an association to the Date class (like we would do for an entity) from every class that uses it, would mess up our diagrams
  - Classes also refer to enumerations as attributes (see previous slide)





Value Objects



Value Objects

- Are not entities, but rather describe entities.
  - Example: a Date is a characteristic of a Payment
- Do not have a unique ID
  - primitive (int) and complex (Date) datatypes do not have an identity
    - Are saved, together with their entity and not shared with others
      - You do search or reuse them (you just make a new Date)

Value Objects

- Completely defined by their values
  - Any attribute you change, results in another value (any field you change in a date, gives a different date)
  - A well designed datatype class is immutable.
    - All attributen are set by the constructor
      - attribute **{readonly}** // in Java: `private final`
    - No setter

Value Objects

Exercise

Java's Date class is not immutable

```
# calculate someone's birthday in 2022
Date dateOfBirth = person.getDateofBirth();
dateOfBirth.setYear(2020);
```

deprecated → replaced with LocalDate because it is IMUTABLE

Value Objects

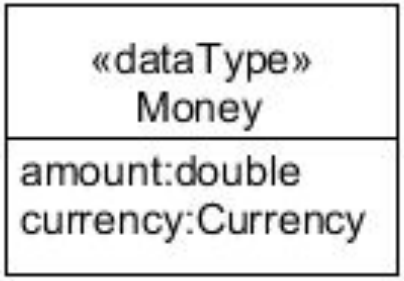
- Date methods like **void** setYear(…) have been deprecated
- Replaced by LocalDate with methods like

```
LocalDate          withYear(int year)
                   Returns a copy of this LocalDate with the year altered.
```

  - **LocalDate** is immutable. Changing something is accomplished by first making a new copy of the date.

Value Object: Write the Money class

Exercise



Value Object: Write the Money class

Exercise

```java
package be.kdg.se2e.immutable;

import java.util.Currency;
import java.util.Objects;

//final class so there can be no mutable subclasses
public final class Money { Complexity is 10 : OK
  //final attributes can only be set by constructor
  private final double amount;
  private final Currency currency;

  public Money(double amount, Currency currency) {
    this.amount = amount;
    this.currency = currency;
  }

  // no setters
  public double getAmount() {
    return amount;
  }

  public Currency getCurrency() {
    return currency;
  }

  @Override
  public boolean equals(Object o) { Complexity is 9 : OK
    if (this == o) return true;
    if (!(o instanceof Money)) return false;
    Money that = (Money) o;
    return Double.compare(that.amount, amount) == 0
      && currency.equals(that.currency);
  }

  @Override
  public int hashCode() {
    return Objects.hash(amount, currency);
  }
}
```

Value Object: Money record

Exercise

```
package be.kdg.se2e.record;

import java.util.Currency;

public record Money(double amount, Currency
currency) {}
```

- Java 16 `record`
  - immutable class with all boiler plate code implicitly present
  - Money record and money class are equivalent
    - Instead of `getAttibute()` use `atttibute()`

Syllabus

Jenkov Record tutorial

Example

https://gitlab.com/kdg-ti/acs-programming-2.2/m05_domain_money

modelling actors

Limit to what the system needs

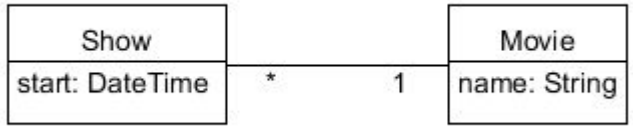
- Only model associations to an actor in a user story if your program needs to track this information. Example:

  As a planner

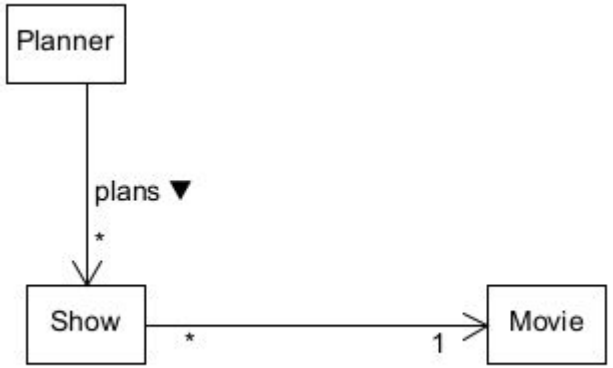
  I want to schedule movies

  So that I can align it with demand and best times for its audience

1. No need to track who managed a movie



2.System needs to know who manages each movie



Exercise 1-3

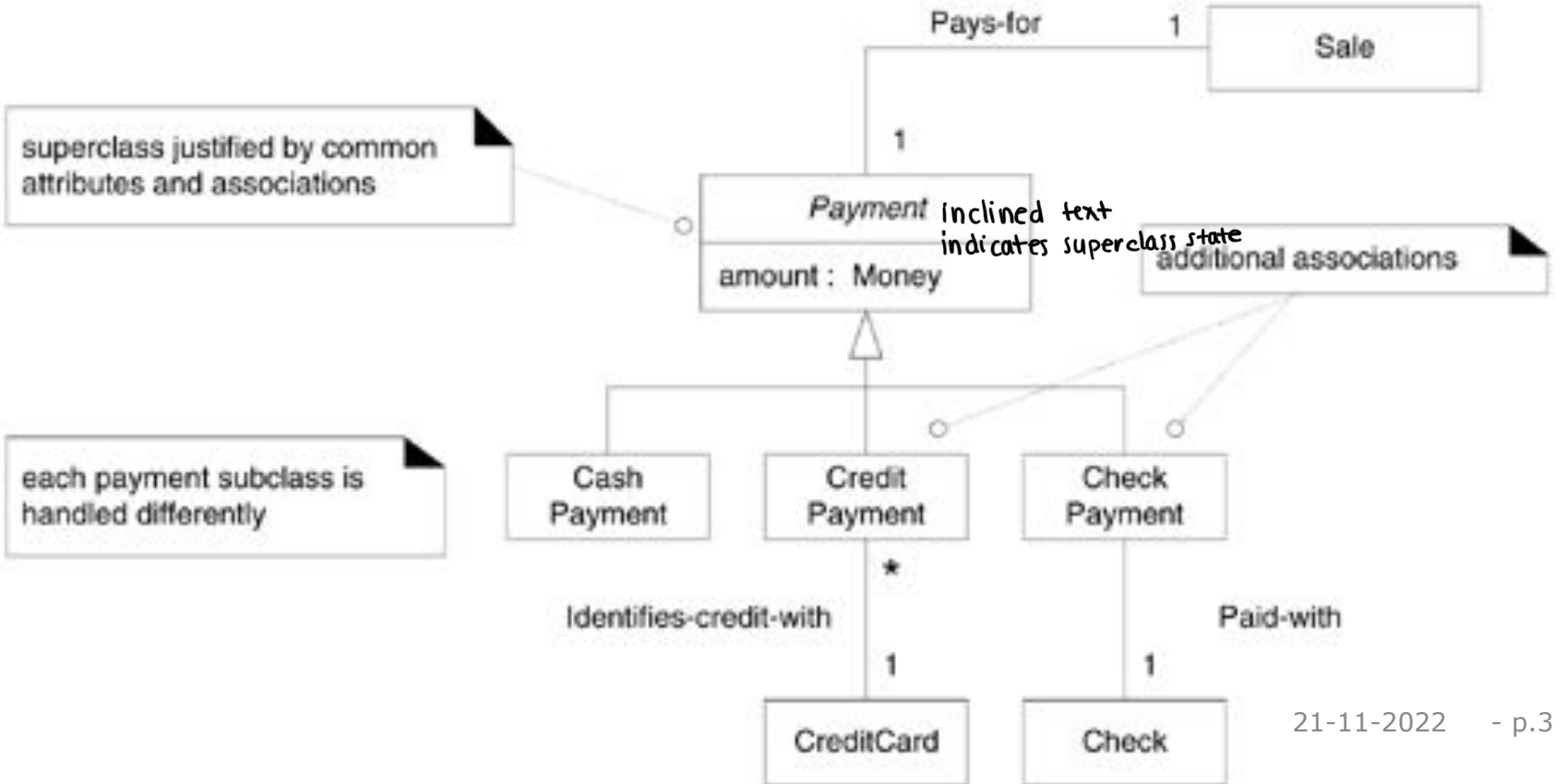
Inheritance

KdG Karel de Grote Hogeschool

Specialisation/Generalisation

Refresher

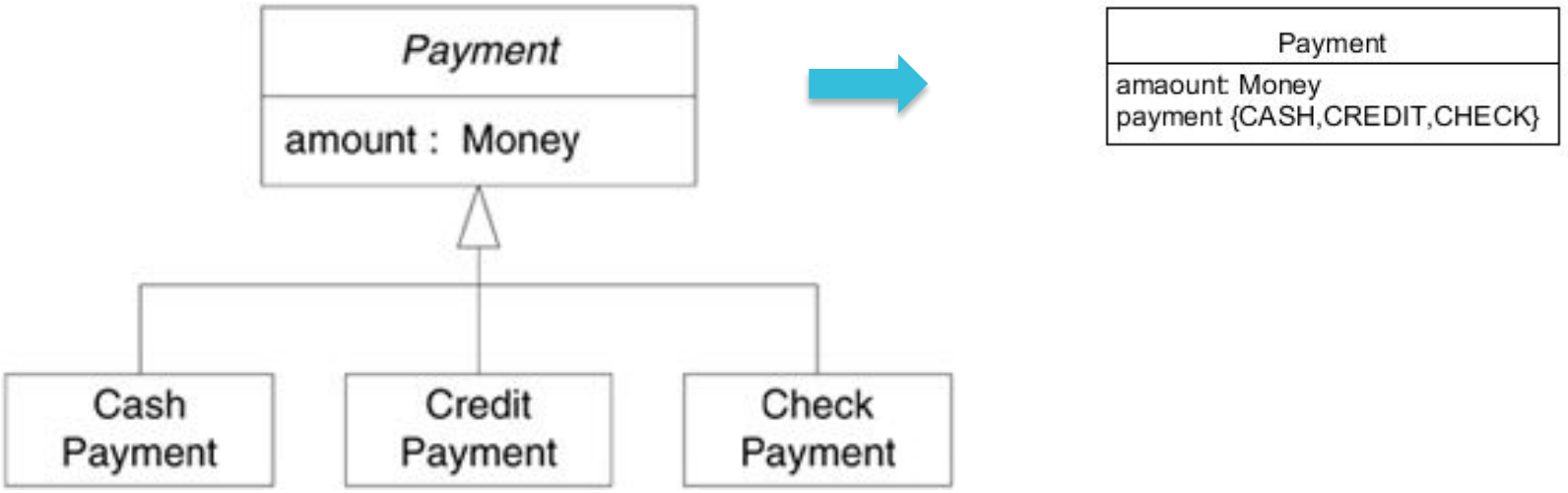
- Specialisation: subclass has all state (attributes and associations) and behaviour of the superclass + extra state/behaviour of its own: is a kind of relation
- Superclass is a generalisation of the subclass



Specialisation/Generalisation

Refresher

- Useful if subclasses have additional state and/or behaviour
  - If not modelling this with an enumeration is simpler



Datastructures overview

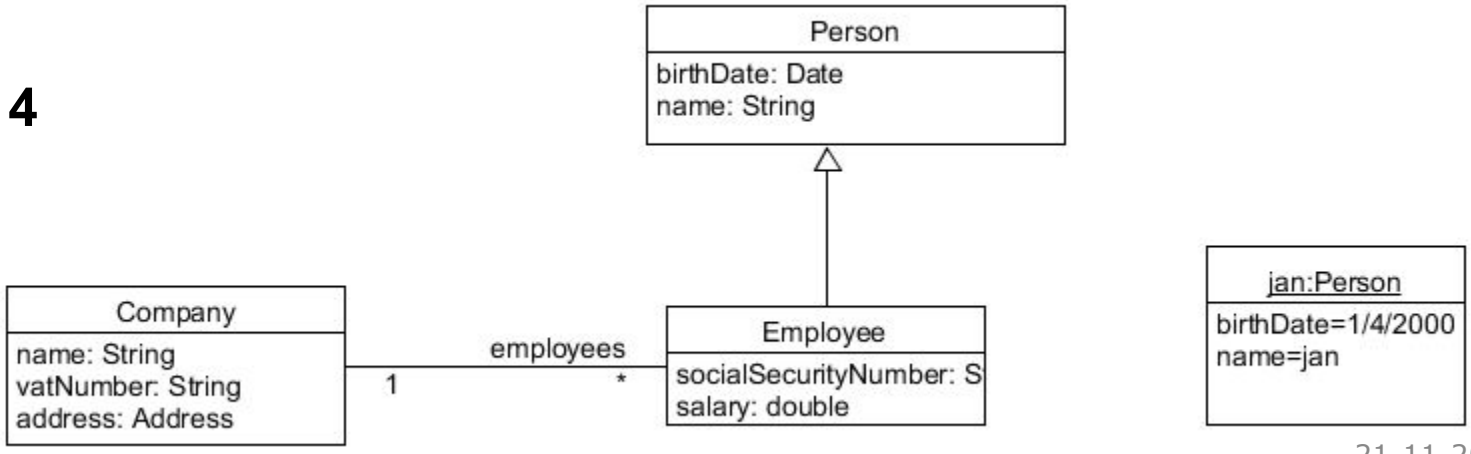
Domain model

Karel de Grote Hogeschool

Overview data structures

structure	inverse	example
Object (instance)	Class	jan is an instance of Person
Record (Relation)	Attribute	Company • name • address • VAT number • employees
Verzameling (Collection)	Element	employees: Collection <Employee>
Generalisation	Specialisation	An Employee is a (kind of) Person

Exercise 4

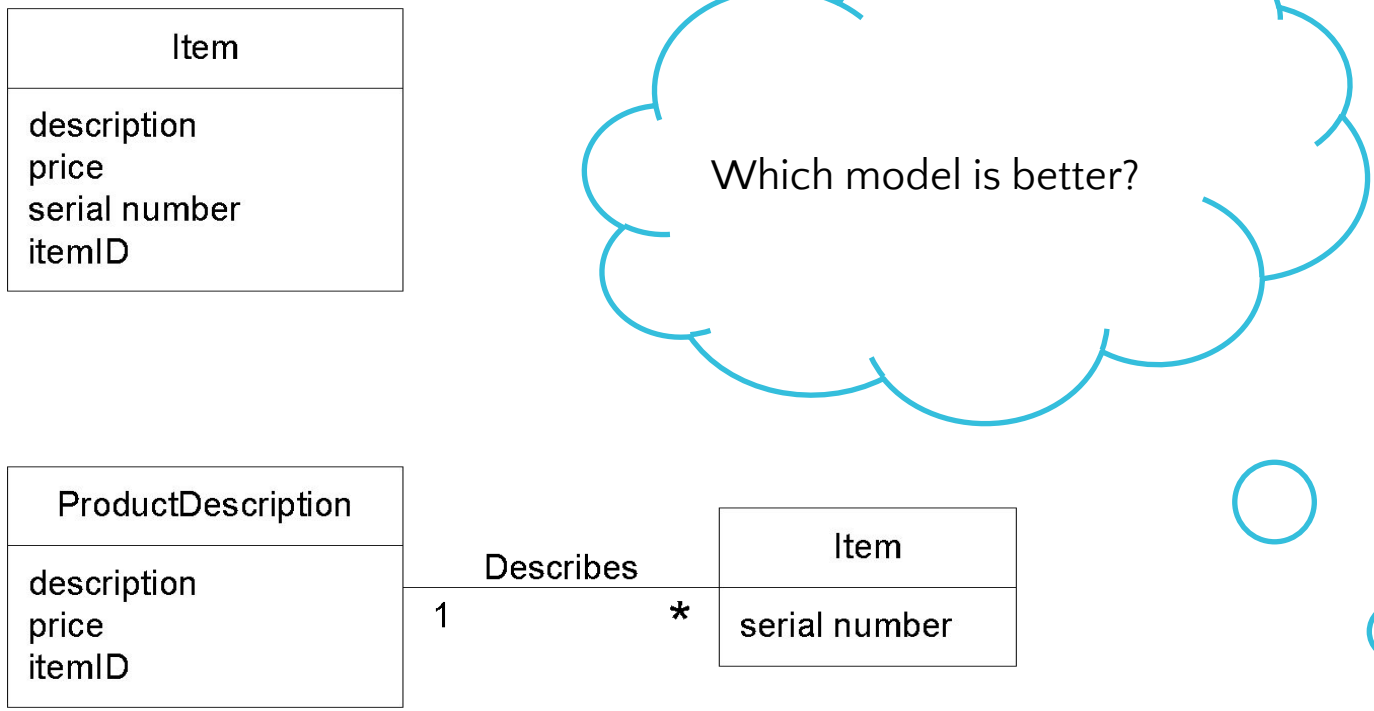


Domain model guidelines

KdG Karel de Grote Hogeschool

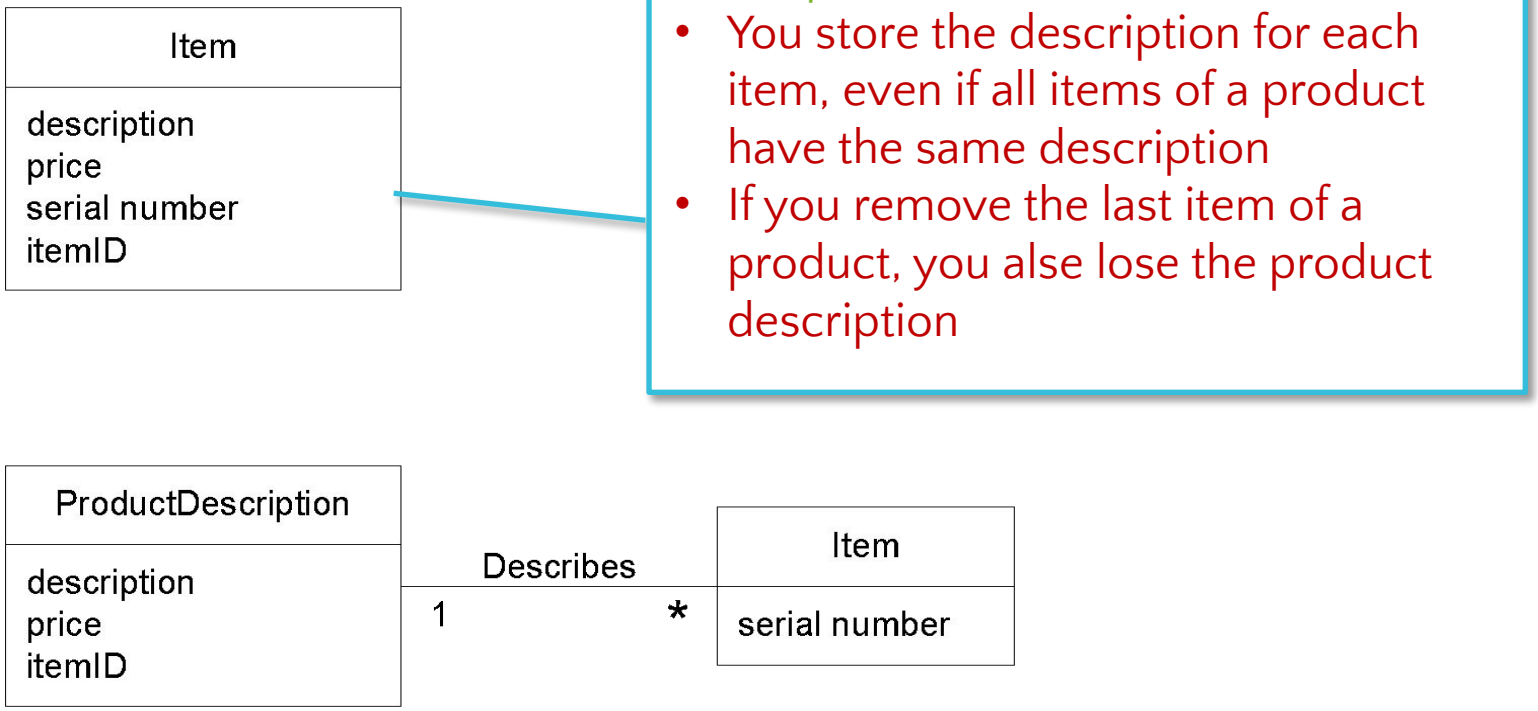
Item-Descriptor pattern

Refresher



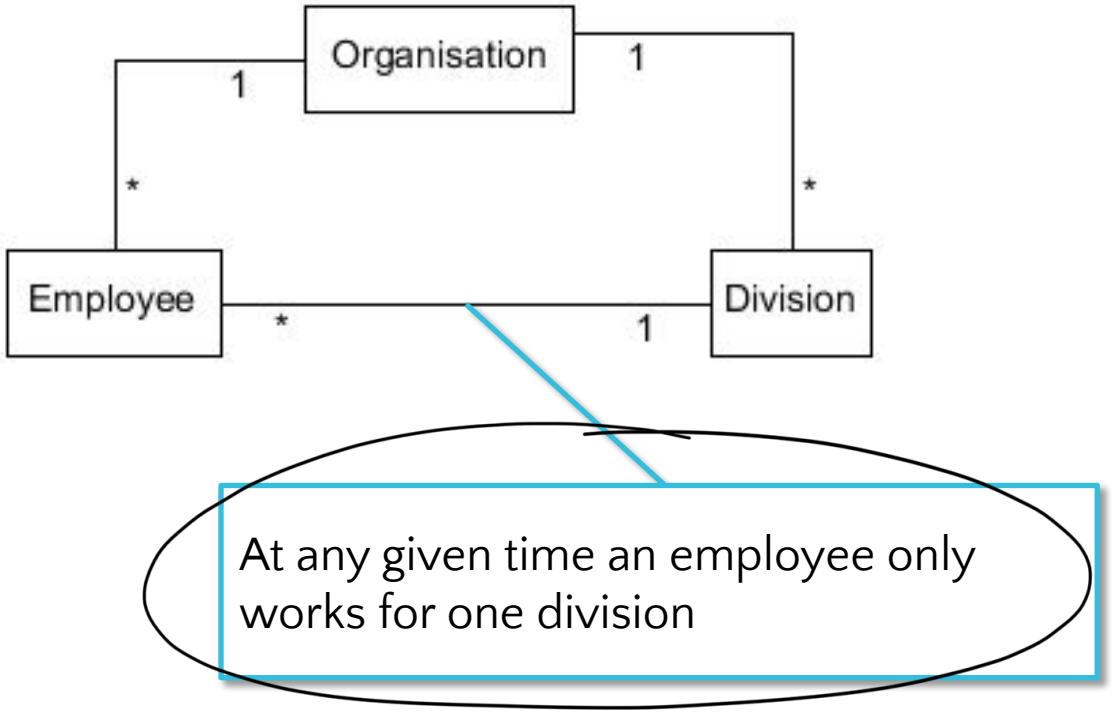
Description pattern

Refresher



modelling history

- Multiplicities normally show relations at one point in time
  - Not over a period of time

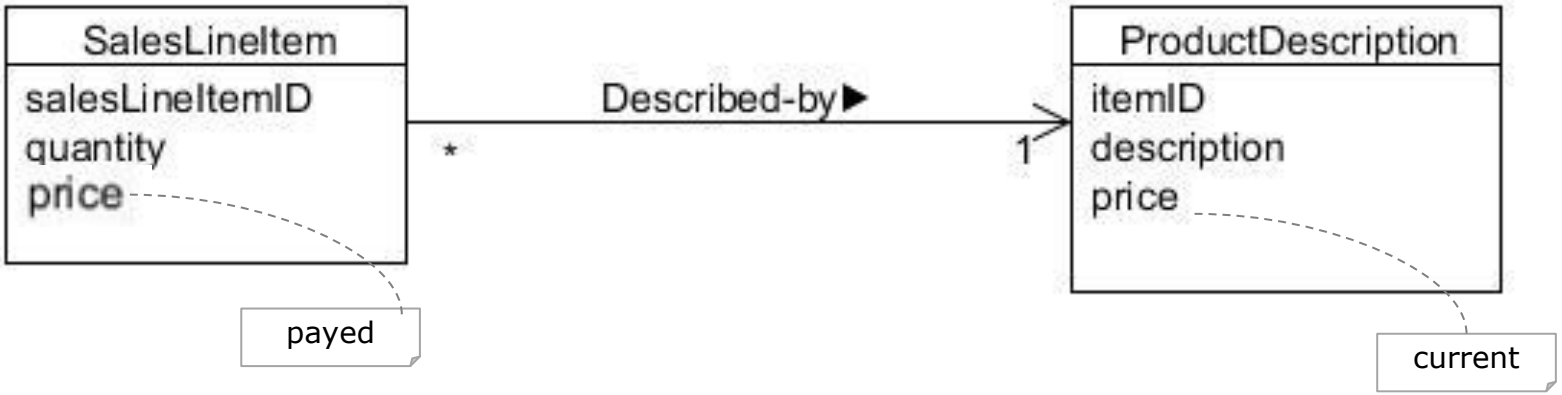


modelling history: price

Is this model OK?
Do you always know at which price a product is sold?

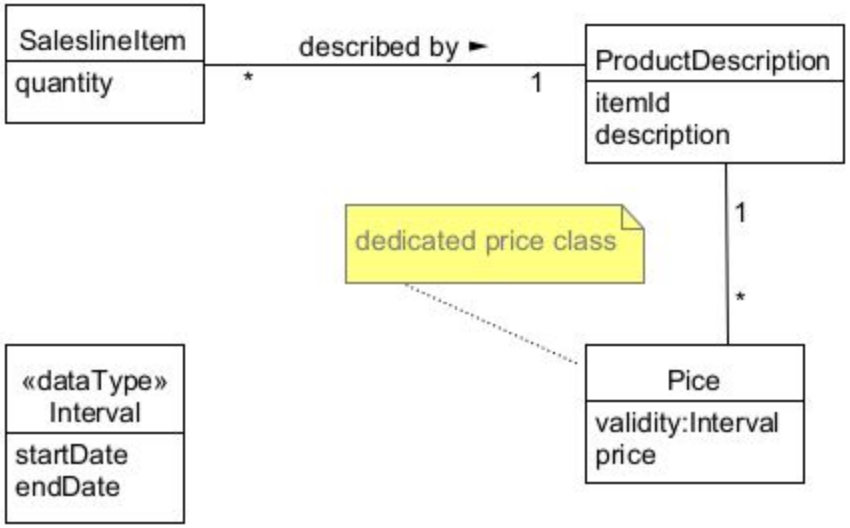
modelling history: price

- Solution 1: model history (duplicate current price)



modelling history: price

- Solution 2: model history (time interval)

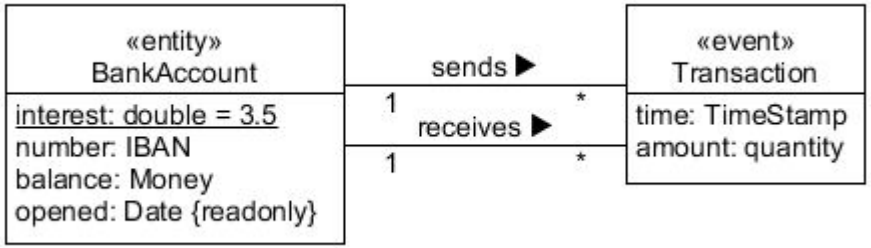


Exercise

Compare solutions (1) and (2) (pro/cons)

Modelling history

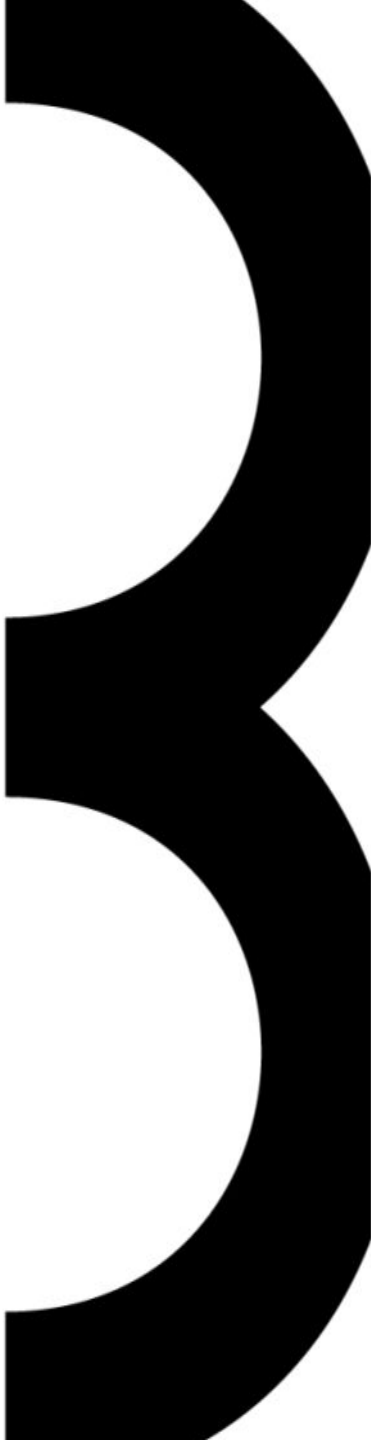
- Solution 3: DDD Domain events: log events triggering a change



- Similar to logging history
- information on changes (when, who, why, what…)
  - immutable
  - May have an identity (transactionNumber)
- Name can be past participle of the action, e.g. MoneyTranfered
- Can be used in design
  - Implement side effects in unrelated entity or external system (EventHandler, publish-subscribe, Event sourcing)

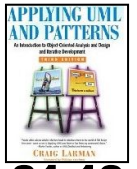
Compare solutions (2) and (3) (pro/cons)

DDD: Strategic design

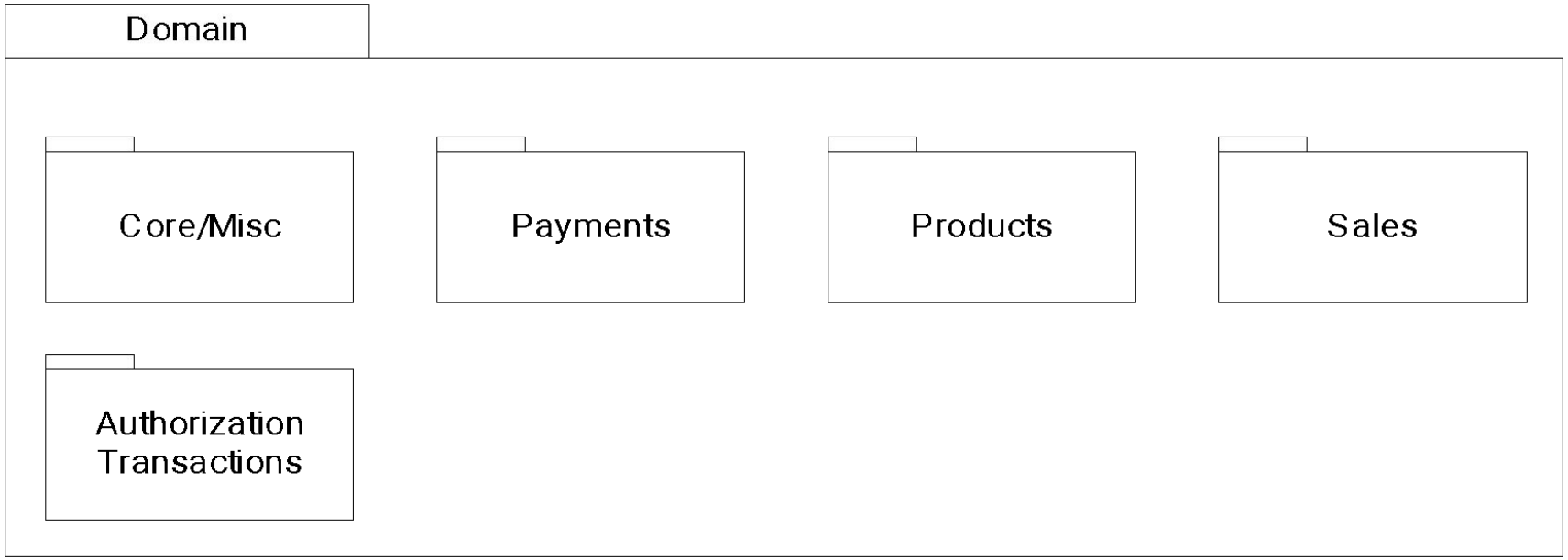


KdG
Karel de Grote Hogeschool

Packages (DDD: Modules)



- Structure a large domain in packages containing strongly related concepts
  - package names are part of the DDD ubiquitous language
- UML package diagram
  - Hierarchical tabbed folder representation



Strategic design

- Domain Driven Design aims to manage large business domains (and large projects)
  - DDD Strategic design groups patterns for high level design (business modelling)
- **DDD domain**: business processes we want to automate
- Is split up into **DDD subdomains**
  - Each subdomain is assigned to a project team
    - Team size is limited in agile processes
    - Small projects have a higher success rate

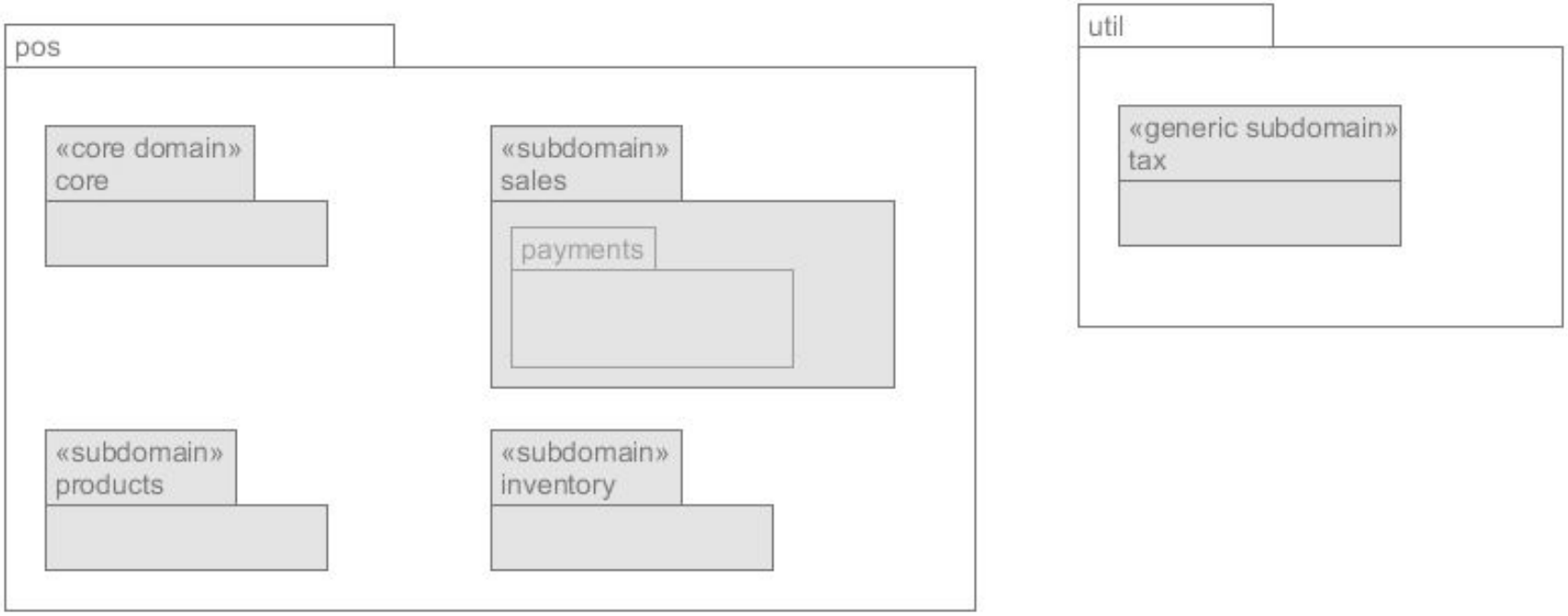
DDD Subdomain

- A domain is plit up in subdomains based on
  - Coupling and cohesion of packages
    - Cohesion: find one or more packages with a common purpose or subject
    - Coupling: put packages with hgh coupling in the same subdomain
  - Size: the subdomain must be small enough to be done by one team
  - onderlinge afhankelijkheden van packages
- Each subdomain is automated in a separte project
  - One subdomain cannot be handled by multiple projects (teams). Split up into smaller subdomains if needed
  - If subdomains are too small for a project, a team can get a project consisting of multiple subdomain

DDD antipattern: Big ball of mud

- Big Ball of mud: a domain that is NOT split up into manageable subdomains with high cohesion and low coupling
- Related to: spaghetti code

Example: Pos subdomains



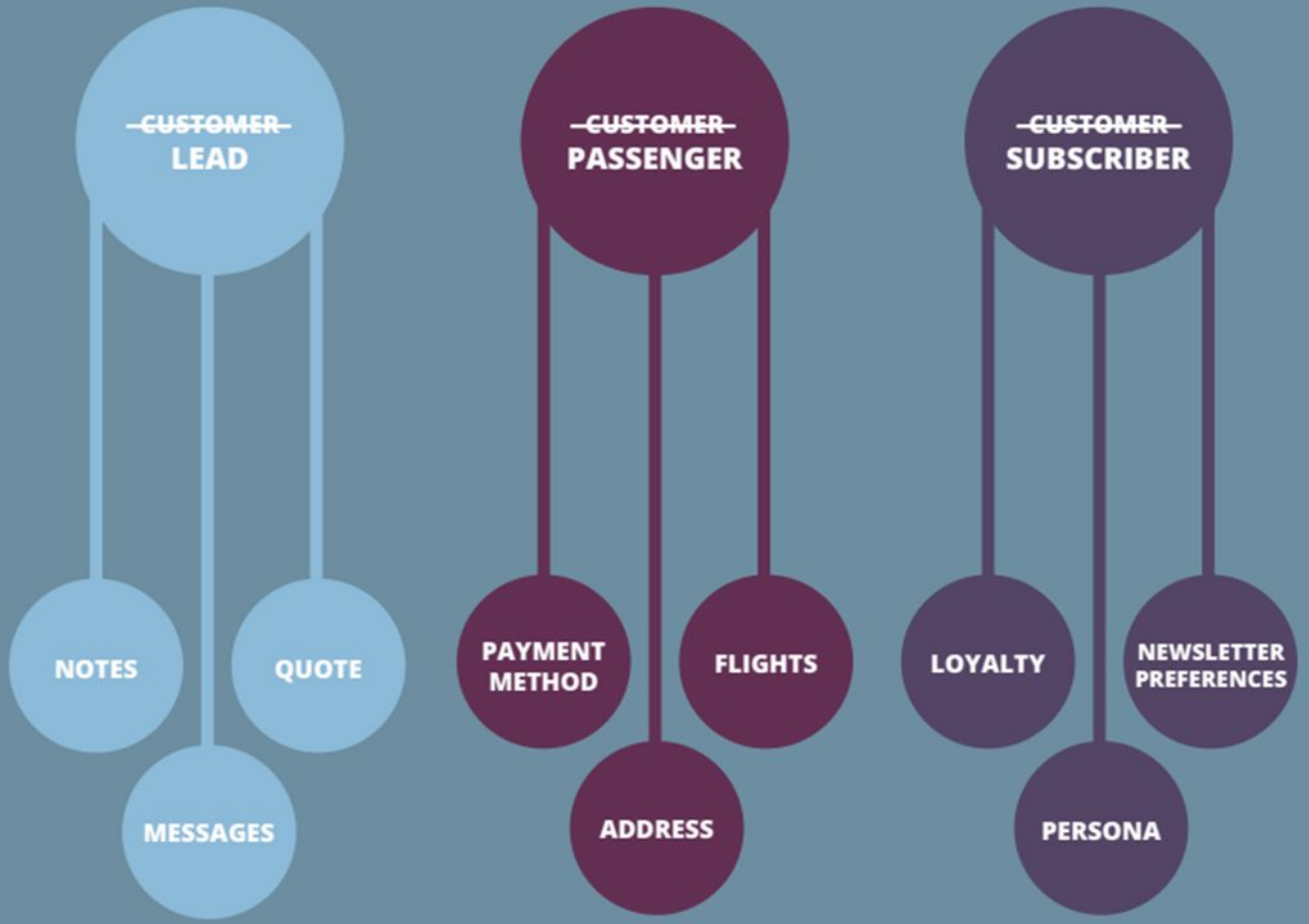
- POS is subdivided into (small) subdomains
  - core subdomain (shop management): contain core (most important) business logic
  - Each subdomain can be implemented in a different (microservices) project

DDD Bounded Context

- A project has a bounded context
  - Done by a separate team
  - Consists of one (or more) subdomains
  - Produces a system or subsystem
- Consequene:
  - **DDD ubiquitous language** is per project **team**

⇒ each **bounded context** has its own **ubiquitous language**

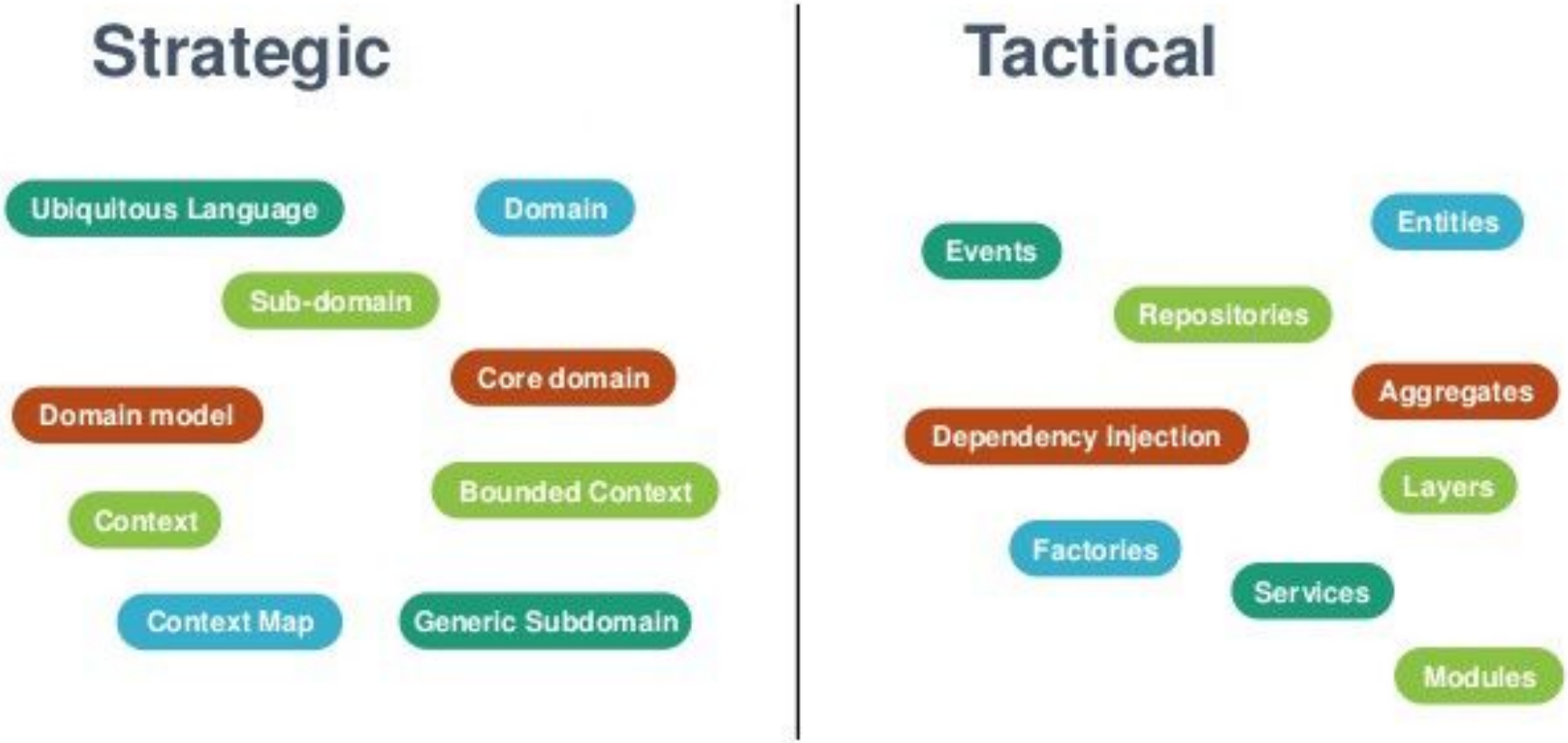
Use Specific Terminology In Each Bounded Context.



DDD: tactical design

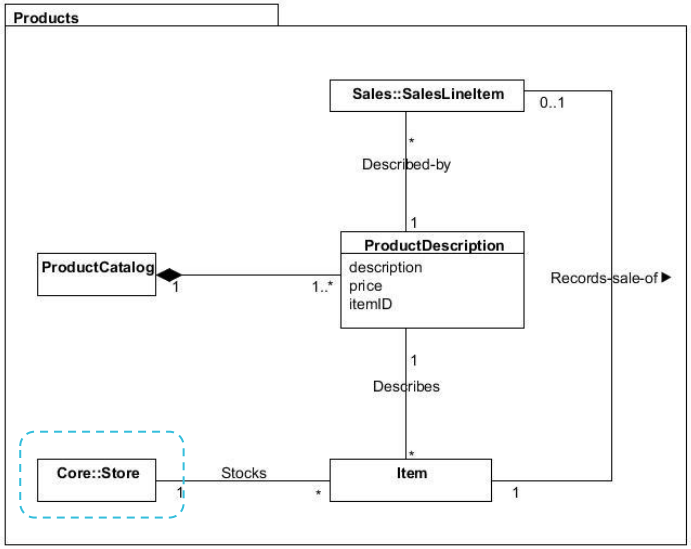
- DDD tactical design
  - applied within a bounded context
  - groups patterns for detailed design (software design)

DDD Strategic/tactical: some examples



Bounded Context: consequences

- Wat if we use concepts from another bounded context?
  - Example: we are using a Store from another bounded context
    - We can give the concept another name (e.g. Shop)
    - We may avaluate which attributes are important for us and adapt them



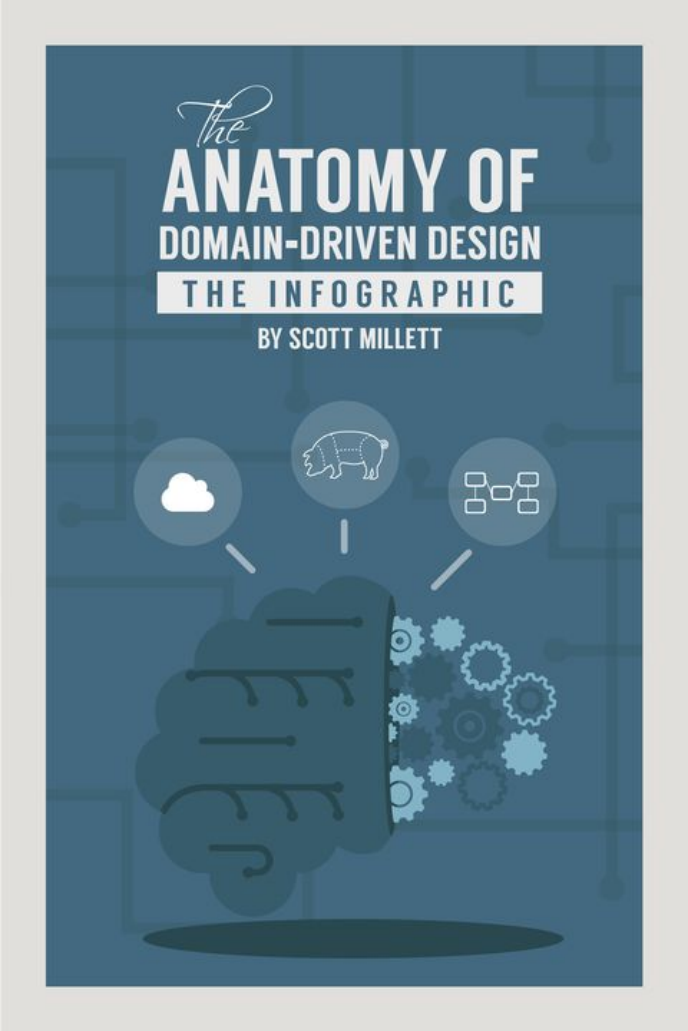
Bounded Context: consequences

- Within a bounded context: reuse code (DRY: Don't Repeat Yourself)
  - **DDD Continuous Integration:** share your work often and use the ubiquitous language
- We do **not** strive to reuse code from other subdomains
  - Exception: generic subdomains can be integrated in multiple projects (utility libraries…)

Bounded Context: consequences

- Even if the ubiquitous language can differ accross bounded contexts, you need to keep data consistent
  - example: Even is you use a store in the one context and a shop in the other one, their names and addresses must by kept in sync
    - Can be stored differently in both context (eg separte field for postbox or not)
- **DDD context mapping**: patterns to communicate between bounded contexts and keep data consistent

Intro DDD



https://leanpub.com/theanatomyofdomain-drivendesign

Summary

1. Extending UML
2. Domain model (class diagram)
   - Entities and associations
   - Data types
   - inheritance
   - datastructures overview
   - guidelines
3. DDD: strategic design