Persistence

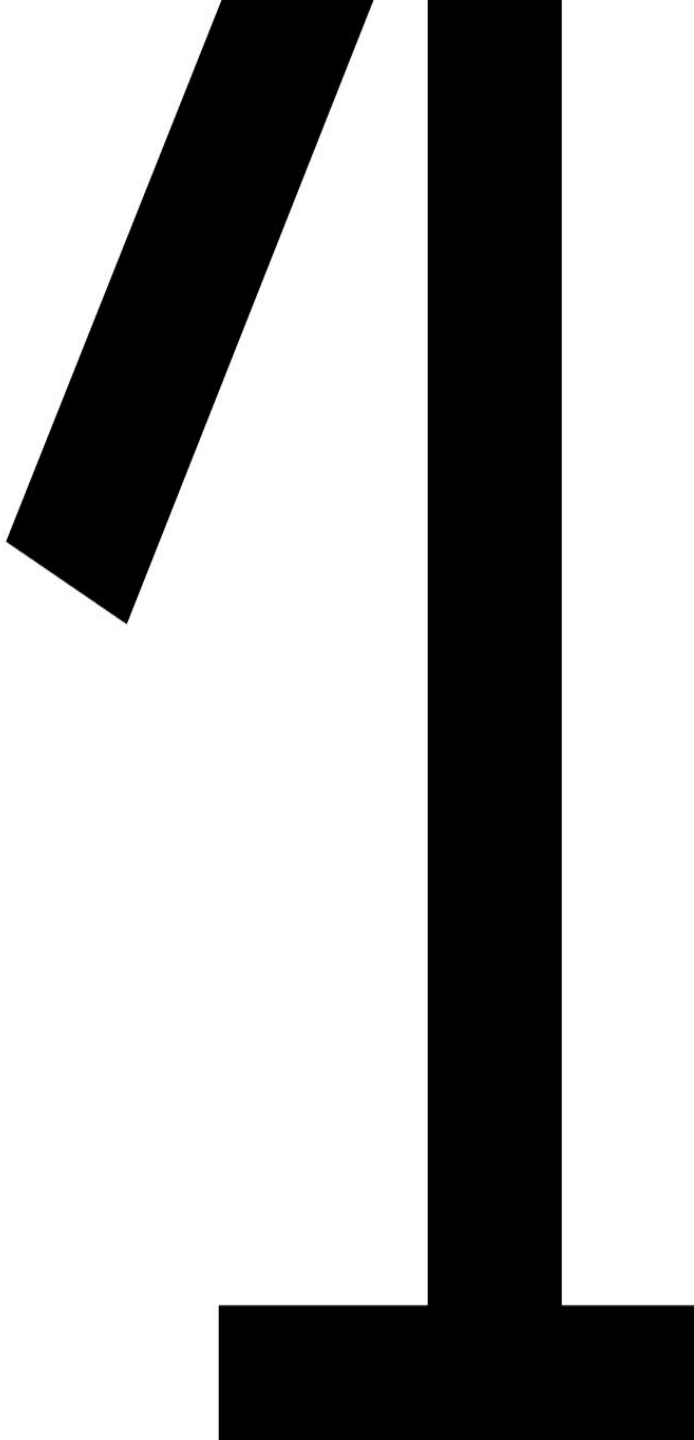
Programming 4

Karel de Grote Hogeschool

Agenda

1. Persistence stores
2. Repositories
3. Datamodels
4. Transactions
5. Object-Relational Mapping

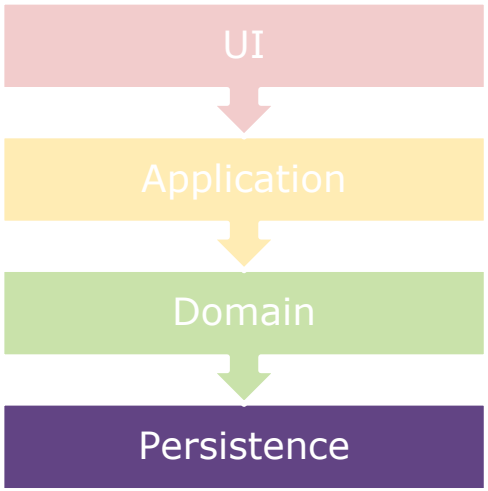
Persistence stores



Definition

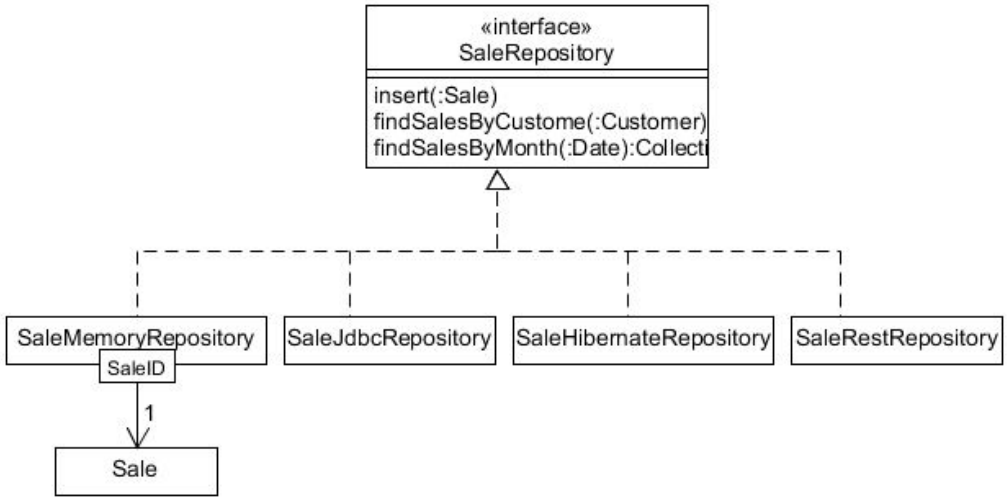
Persistence:
state that outlives
the process that created it.

lives longer than your process (in memory)



Persistence stores

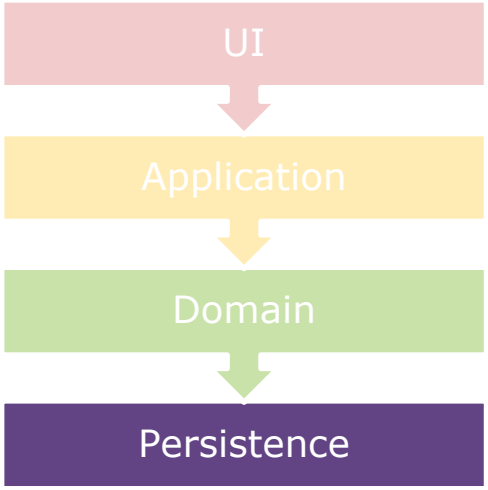
data access obj (repository) change
in between strategies to save time



- Structured data

- Databases

specialized PS



Karel de Grote Hogeschool

Structured data: Marshalling

serialization

- In memory data have any graph structure
  - Pointers in all directions
- Persistence requires conversion to a byte stream (marshalling) and back (unmarshalling)
  - Send to file (persistence)
    - Only one process can access the file at a time
  - Send to network (Remote communication)
- Storing data that can be read by another program
  - Not transactional

only one project can access a file at a time

Java serialization

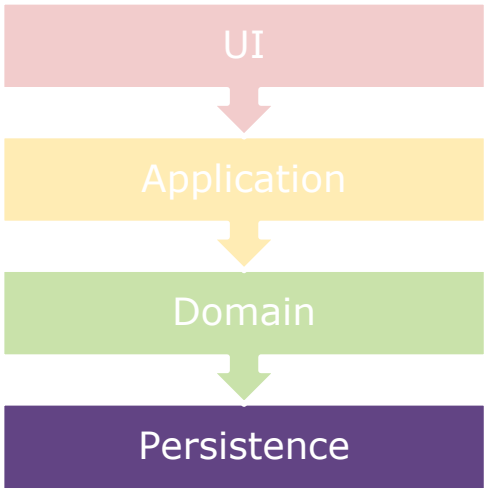
- Builtin API that automatically converts objects to bytestream
- Binary
- Supports graphs
- Java specific, OS/hardware independent
  - For deserialising you need to have all class code available in receiving application
- Usage
  - Save small amounts of data
  - Remote Method Invocation

java specific format

save class info but not the class itself

the other program needs to know the classes so that it can deserialize it

serialization is recursive: saves obj and all of their attributes



Java serialization

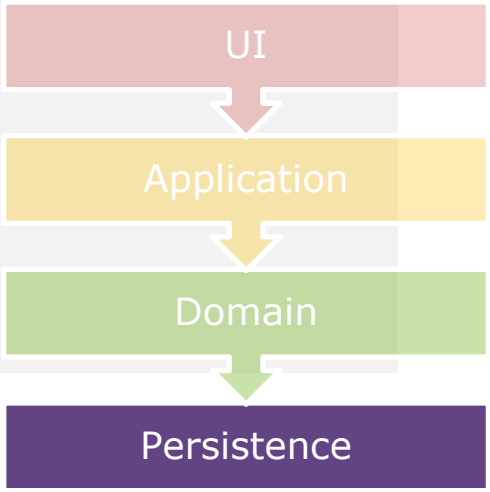
Serializable interface required
Marker (empty) interface

SerializeDemo.java

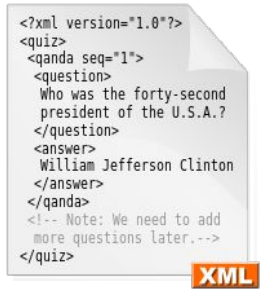
```
public class Student implements Serializable{
   private final int studentIdentity;
   private final String name;
   private final LocalDate birthDate;
   private transient String residence;
   //...
}
public class SerializeDemo {
   public static void main(String[] args) {
      List<Student> studentList = List.of(
        new Student(9999, "Charlotte Vermeulen", LocalDate.of(2000, 1, 24), "Antwerp"),
        new Student(666, "Donald Muylle", LocalDate.of(1952, 6, 14), "Roeselare"));
      try (ObjectOutputStream oos = new ObjectOutputStream(
        new FileOutputStream("data/students.ser"))) {
           oos.writeObject(studentList);
      }  catch (IOException e) {  e.printStackTrace();}
      // …
   } // end main
} // end SerializeDemo
```

if you don't implement the Serializable interface you'll not be able to serialize anything that wasn't specifically marked as serializable

Transient fields are not serialized



Marshalling to structured text



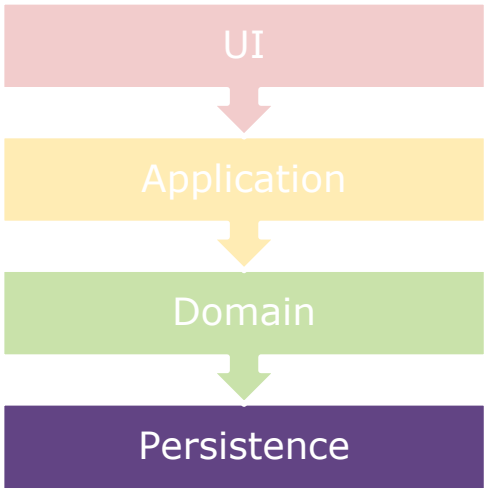
- **Saving limited amounts of language independent data**
- **CSV**
  - Not a strict format, requires additional conventions
- **json, xml, yaml...** ASCII formats
  - Tree structures: need a strategy to handle graphs
  - Used in webservices (REST)
  - Libraries: JAXB, Jackson, Moshi

{JSON}
JavaScript Object Notation

language and platform independent

they work in between diff languages

jason b: binding java objects to json



KdG
Karel de Grote Hogeschool

Databases

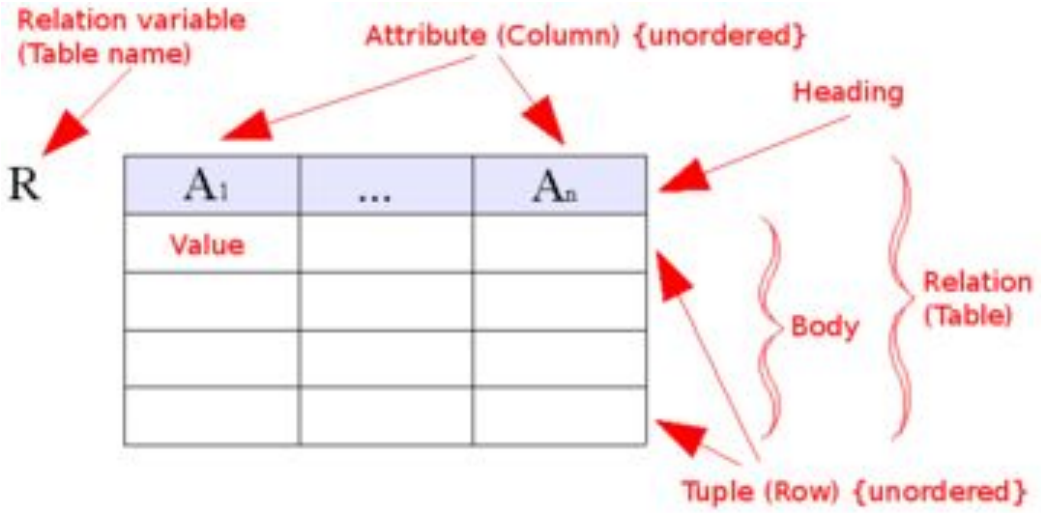
- Parallel access simultanyous access by multiple systems
  - Client-server
- Transactional can't save and retreat data in step
- large amounts of data

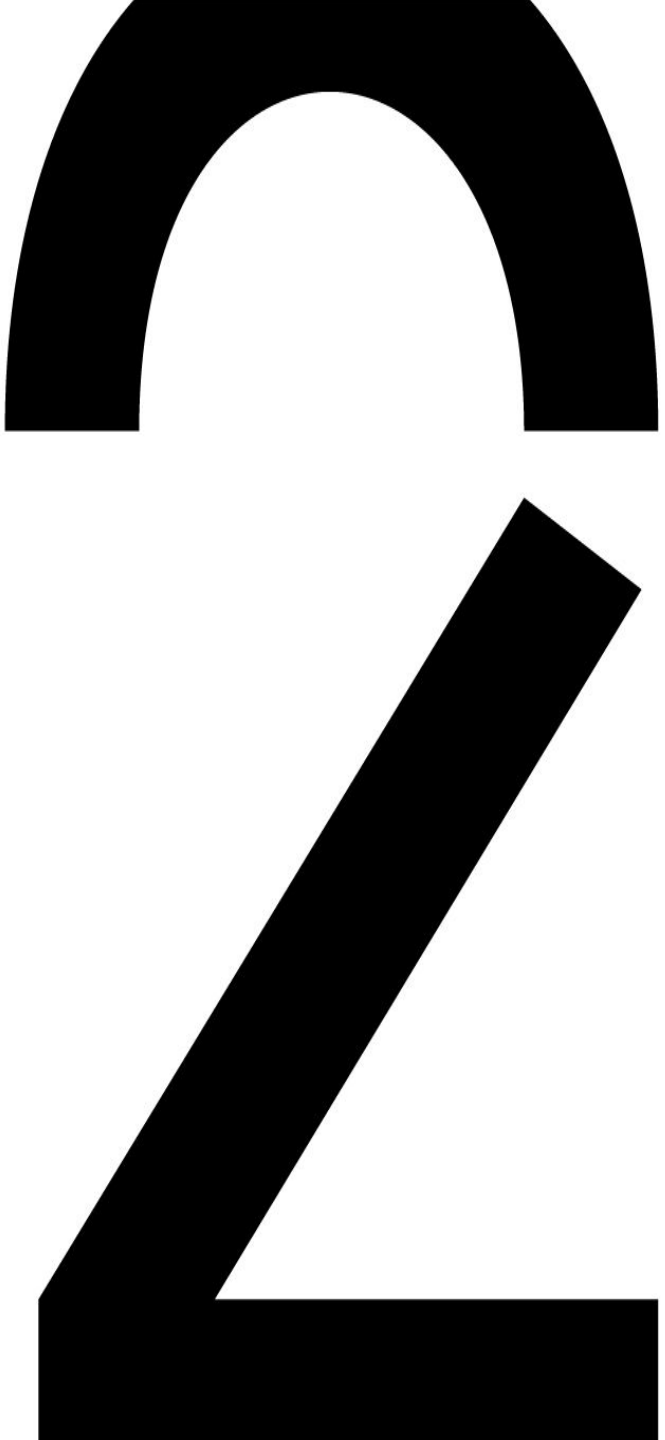
more persistent technologies

Relational databases

- Most common type of database
  other types gain importance(noSQL)
- Predefined datatypes and structure
  database schema with multiple tables
- Tables contain unique records (= tuples, rows)
  primary key
- Relations using keys
  foreign keys

json = open api(documenting what the json structure is for and if they're compliant for the schema)



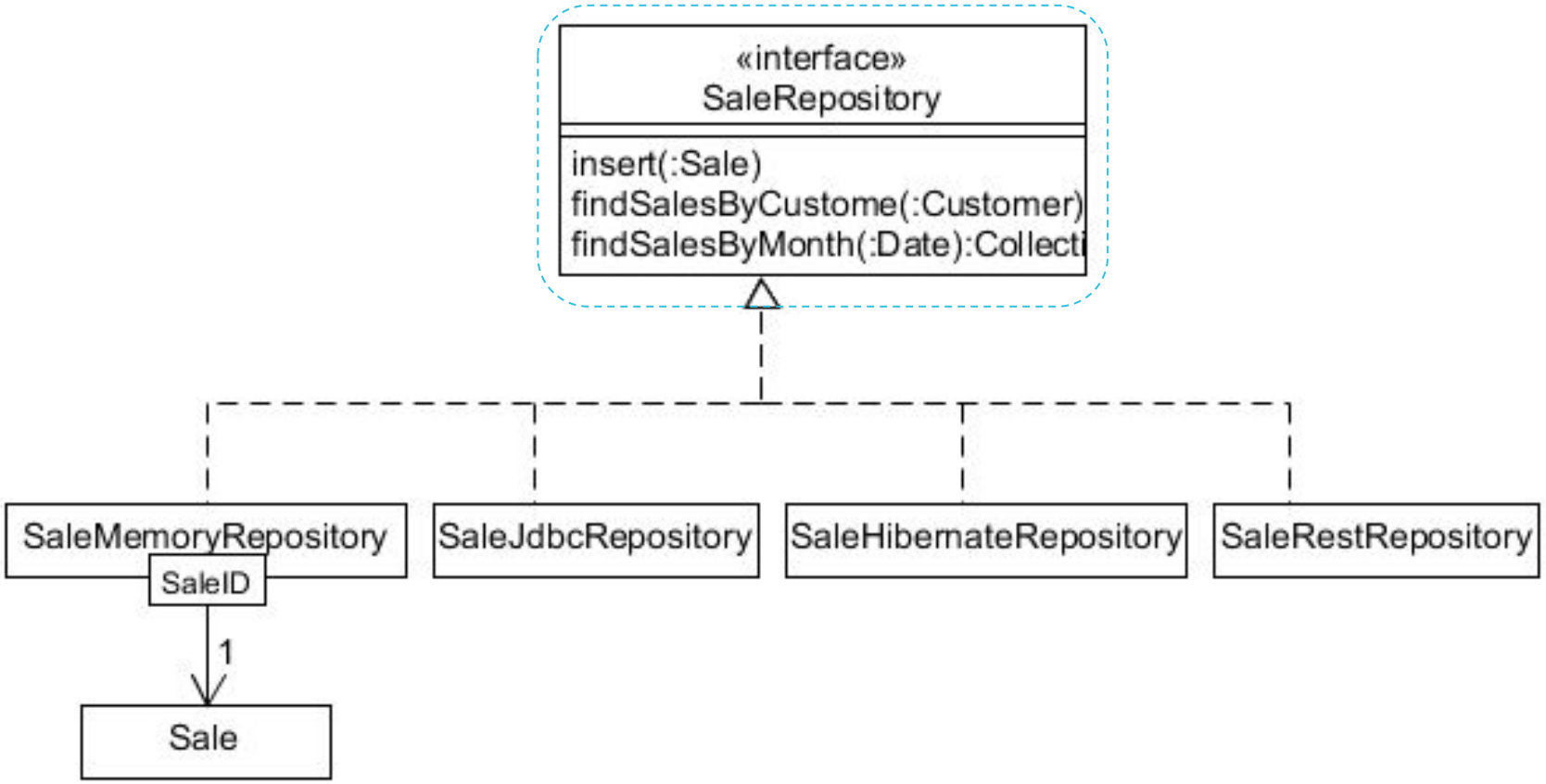


Repositories

KdG
Karel de Grote
Hogeschool

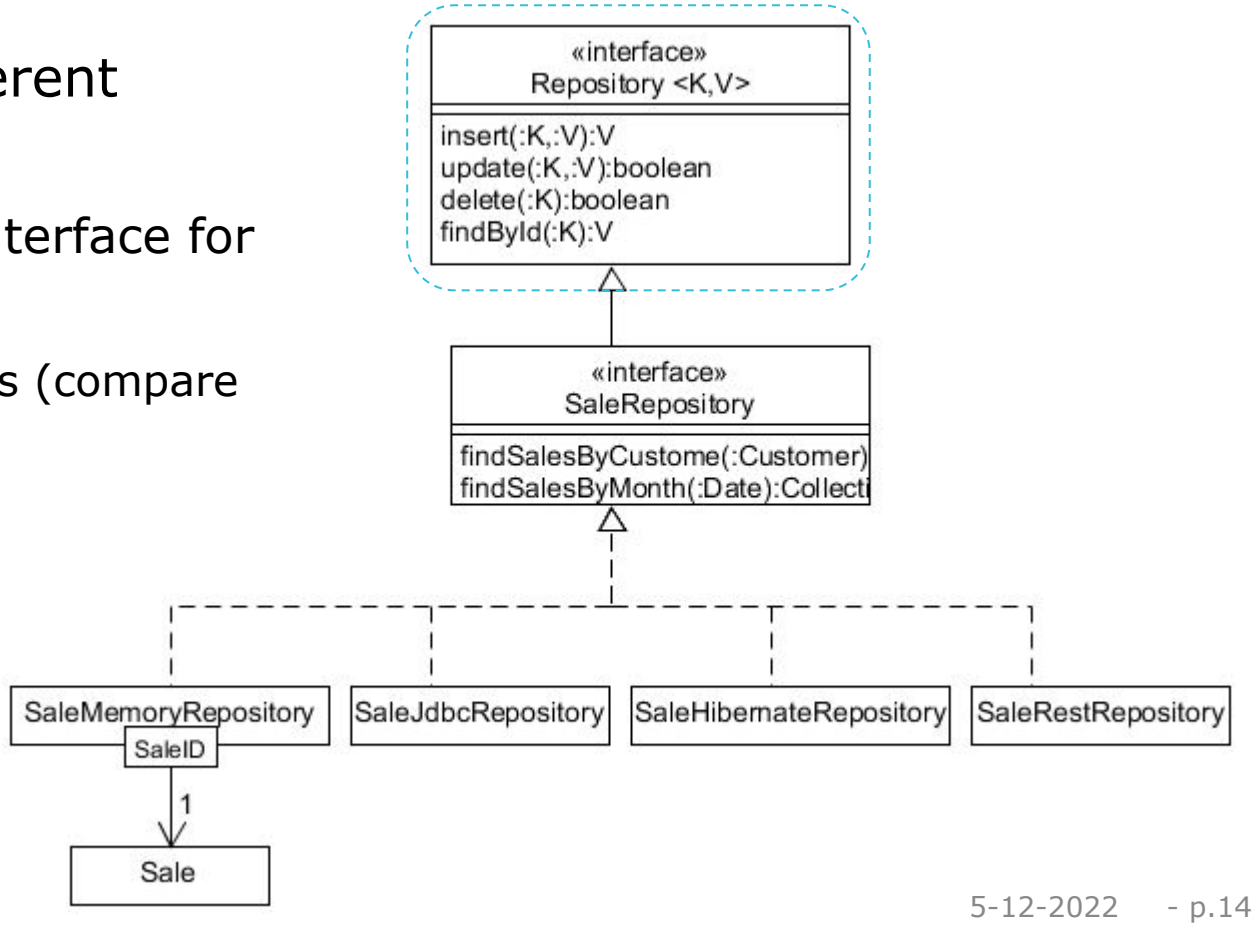
DDD: Repositories

- Repositories hide the differences between persistence technologies
  - Not always perfect: technology bleed



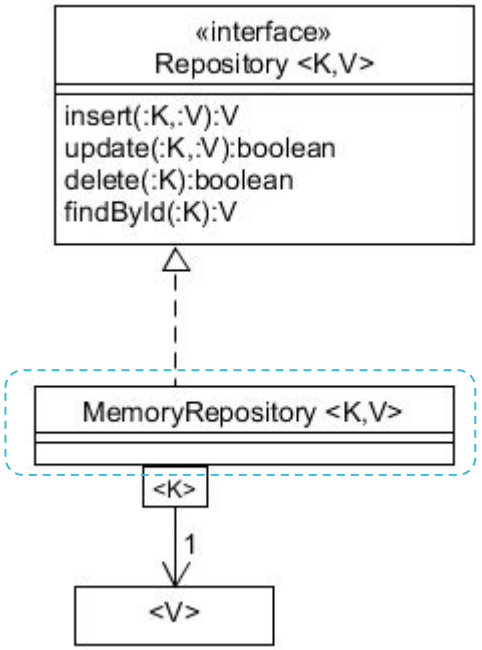
Generic Repository interface

- Different repositories do similar actions on one object (CRUD)
- Actions have different parameter types
  - Can you use an interface for this?
    - Yes, using generics (compare with Collections)



Generic Repository implementation

- Depending on the underlying technology a generic implementation can be feasible



Generic Repository implementation

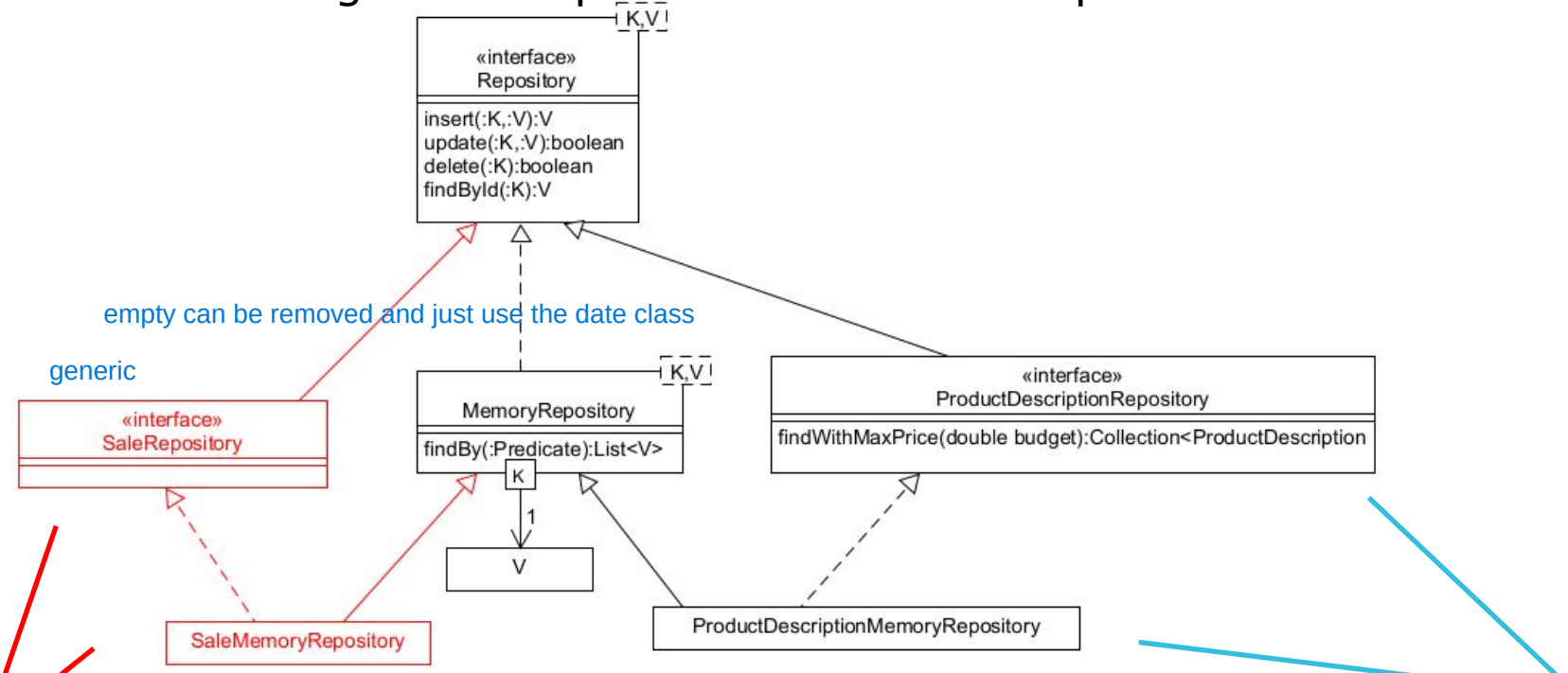
```
public class MemoryRepository<K,V> implements Repository<K, V> {
 protected Map<K, V> data =new ConcurrentHashMap<>();

 @Override
 public boolean update(K key, V value) {
  return  data.put(key, value)!=null;
 }

 @Override
 public V findById(K id) {
  return data.get(id);
 }

 // + Some methods specific to Memory Repository
 public List<V> findBy(Predicate<V> predicate) {
    return findStreamBy(predicate).collect(Collectors.toList());
 }

 private Stream<V> findStreamBy(Predicate<V> predicate) {
    return data.values().stream().filter(predicate);
 }
 //…
}
```

Use of generic repository implementation

```
public class SaleManager {
    private final Repository<Long,Sale> saleRepository =
    new MemoryRepository<>();
…
}
```

Specific Repository Implementation

- Read methods often return different object selections
  - Collection <V> findByXxx
  - Subclass the generic implementation to add specific methods



No extra methods,
MemoryRepository <long, Sale>
suffices

findWithMaxPrice is not part of
MemoryRepository
Specific subclass that adds this method.

Specific Repository Implementation

```
// ProductRepository is called ProductCatalog in larman
public interface ProductDescriptionRepository
    extends Repository<Long,ProductDescription> {
 Collection<ProductDescription> findWithMaxPrice(double budget);
}
```

```
public class ProductDescriptionMemoryRepository
 extends MemoryRepository <Long,ProductDescription>
    implements ProductDescriptionRepository {

  @Override
  public Collection<ProductDescription> findWithMaxPrice(Money budget) {
    return findBy(p -> p.getPrice().lowerThenOrEqual (budget));
  }

}
```

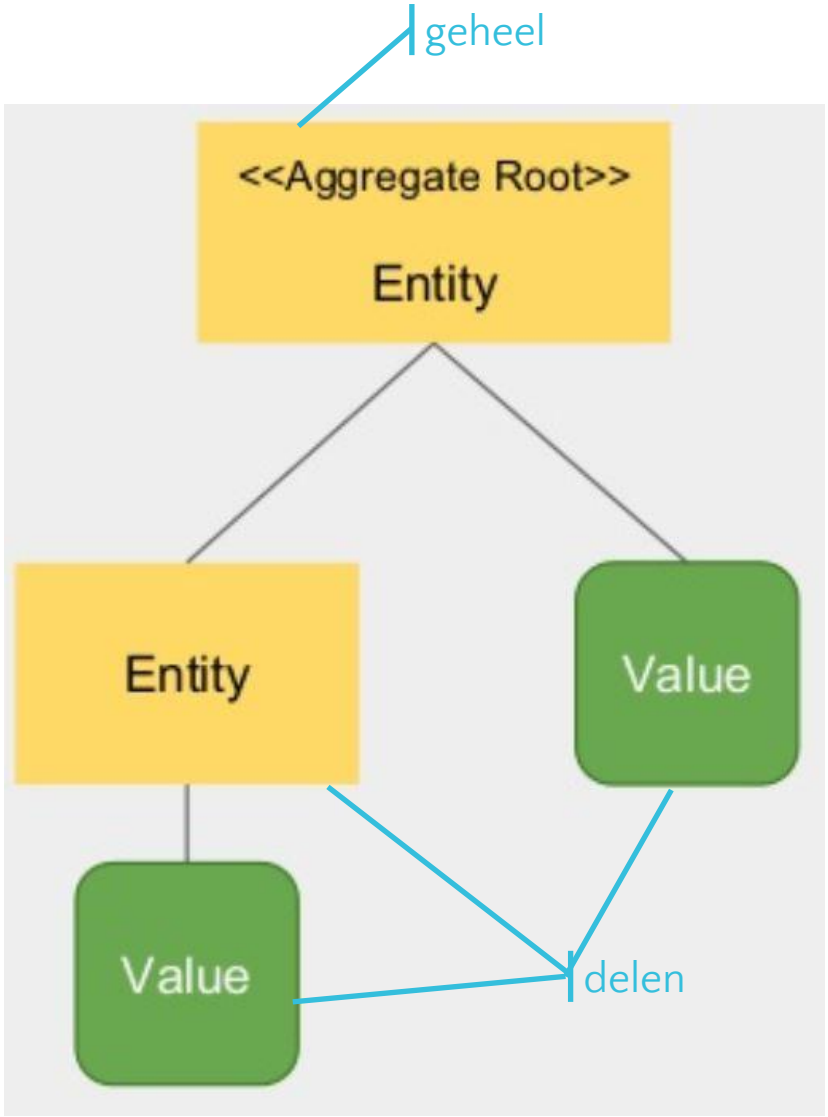
Karel de Grote Hogeschool

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

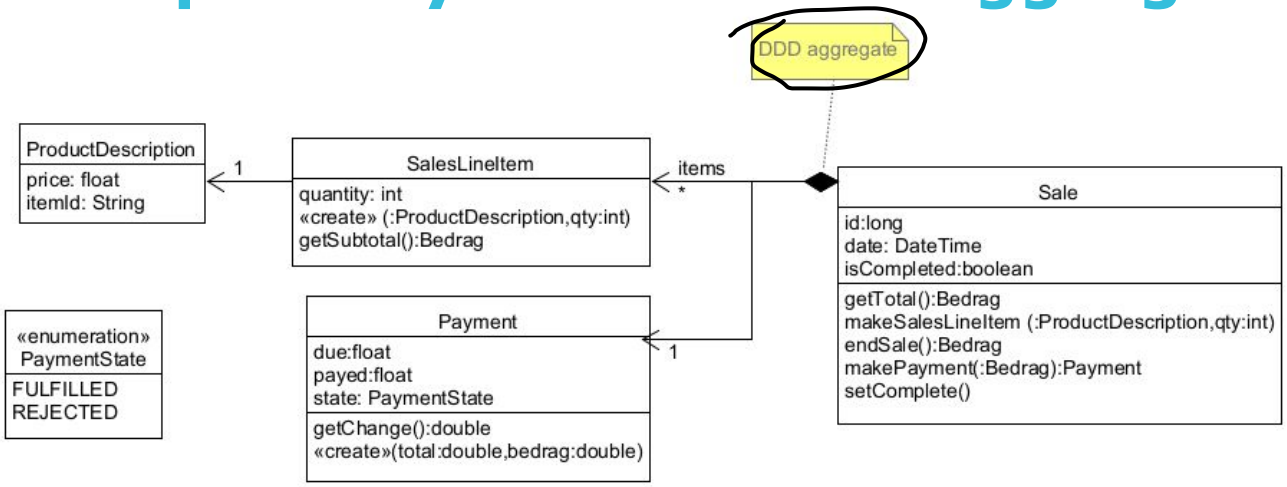
DDD aggregates

Refresher

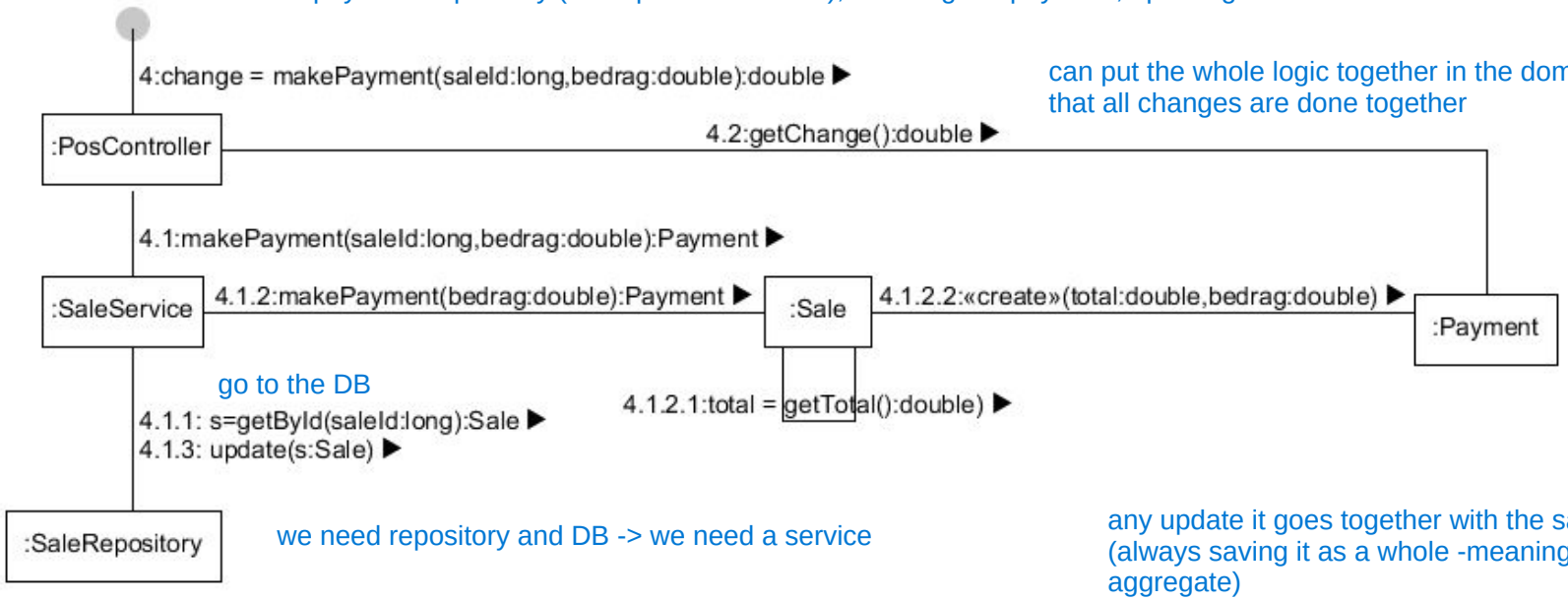
- All operations on the aggregate are always performed via the root
  - Business logic within aggergate implemented in domain, not in services
- Also for persistence => guideline: 1 repository per aggregate
  - All parts of the aggregate are retrieved and saved together



Example: Payment in Sale aggregate



if we take the payment out of the aggregate (remove rel with Sale) -> save the payment through a paymentRepository (diff repo than the rest), inserting the payment, updating the Sale to establish the relation

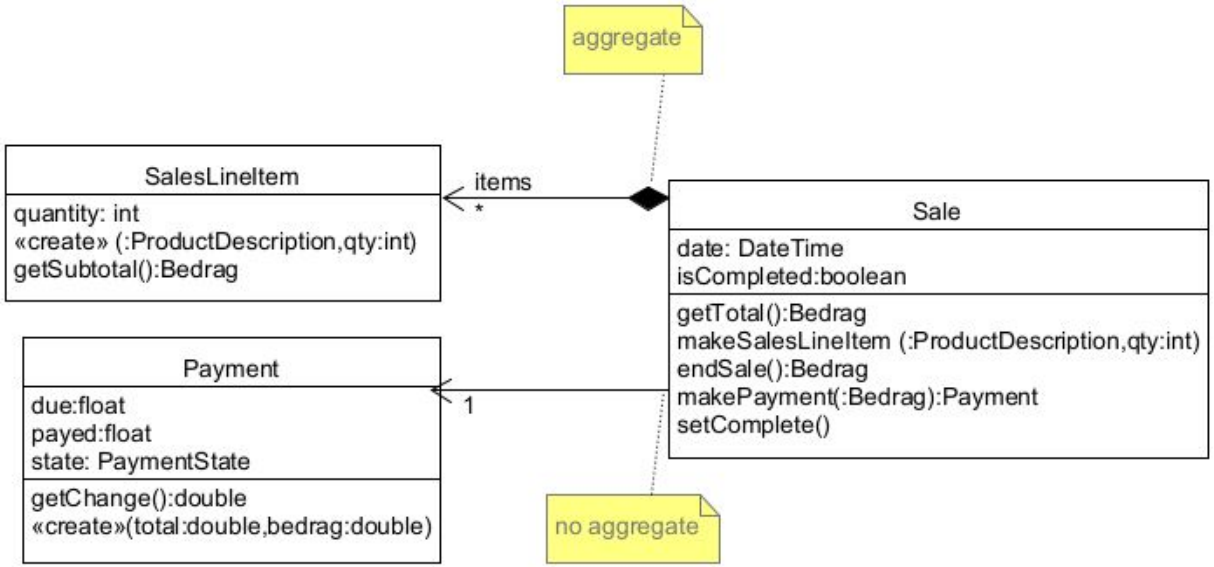


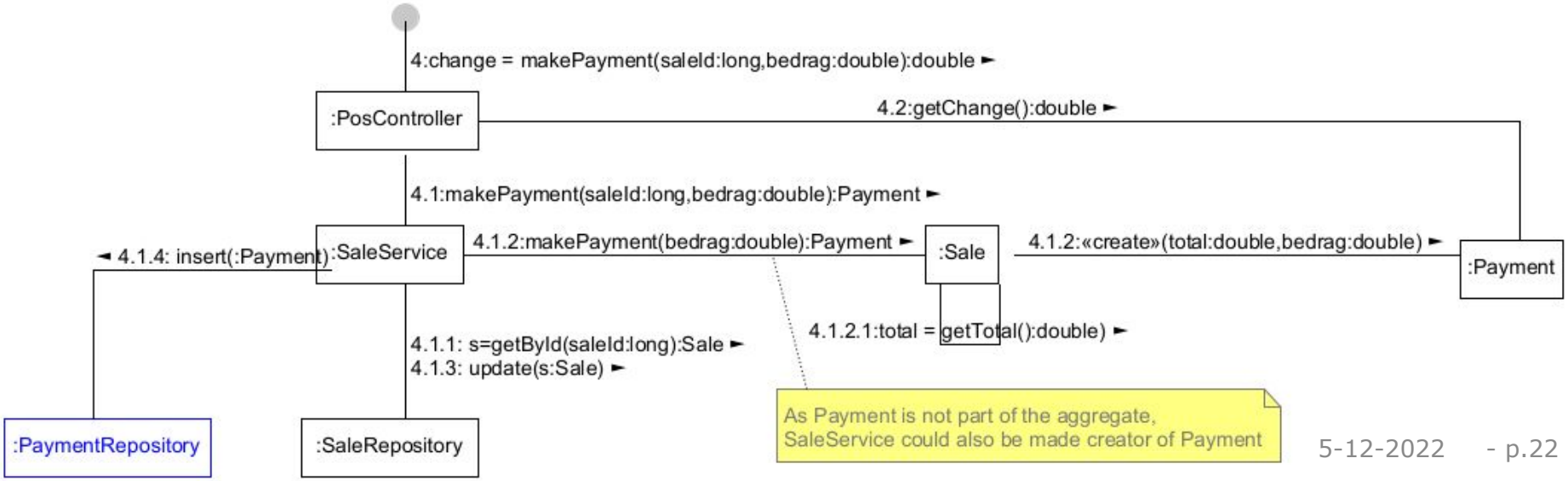
can put the whole logic together in the domain so that all changes are done together

we need repository and DB -> we need a service

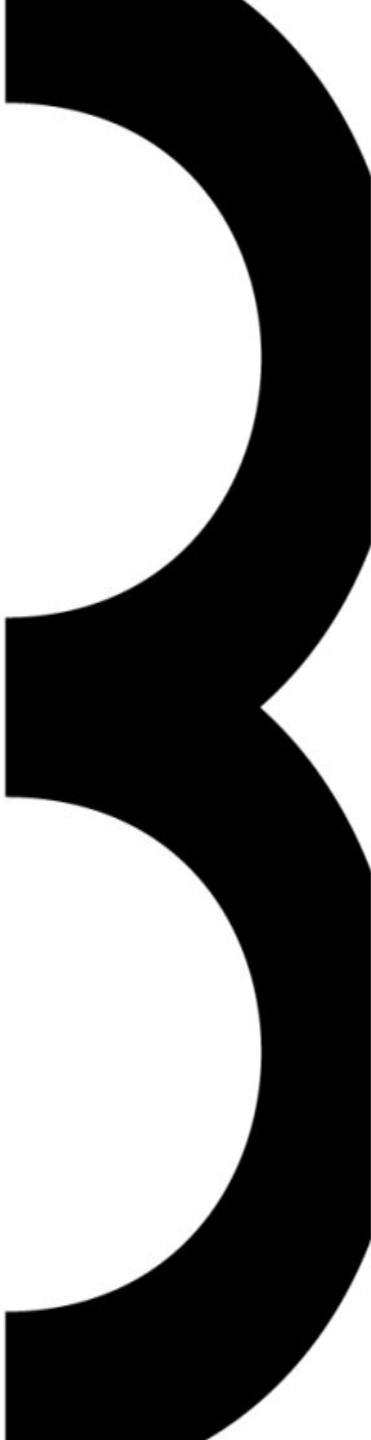
any update it goes together with the sale (always saving it as a whole -meaning of an aggregate)

Example: Payment not in Sale aggregate

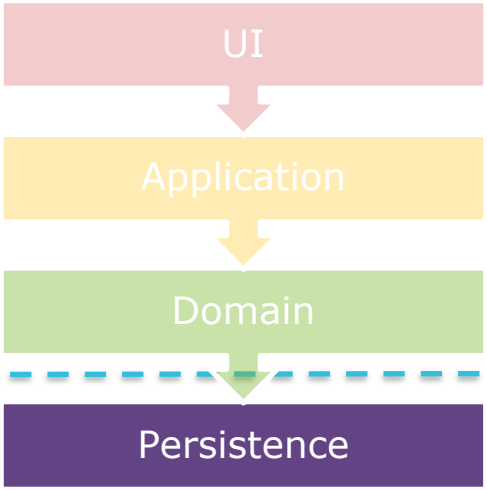




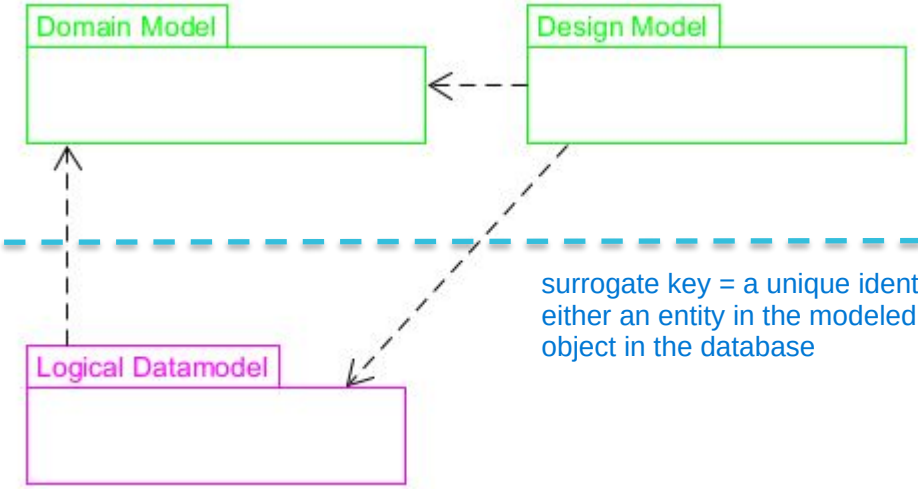
Datamodels



Designmodel and datamodel



composite key = a candidate key that consists of two or more attributes (table columns) that together uniquely identify an entity occurrence (table row)



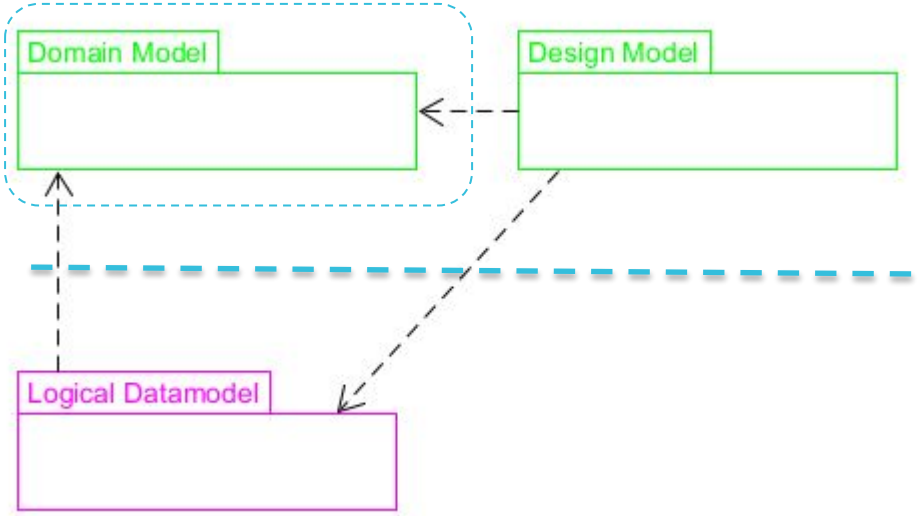
surrogate key = a unique identifier for either an entity in the modeled world or an object in the database

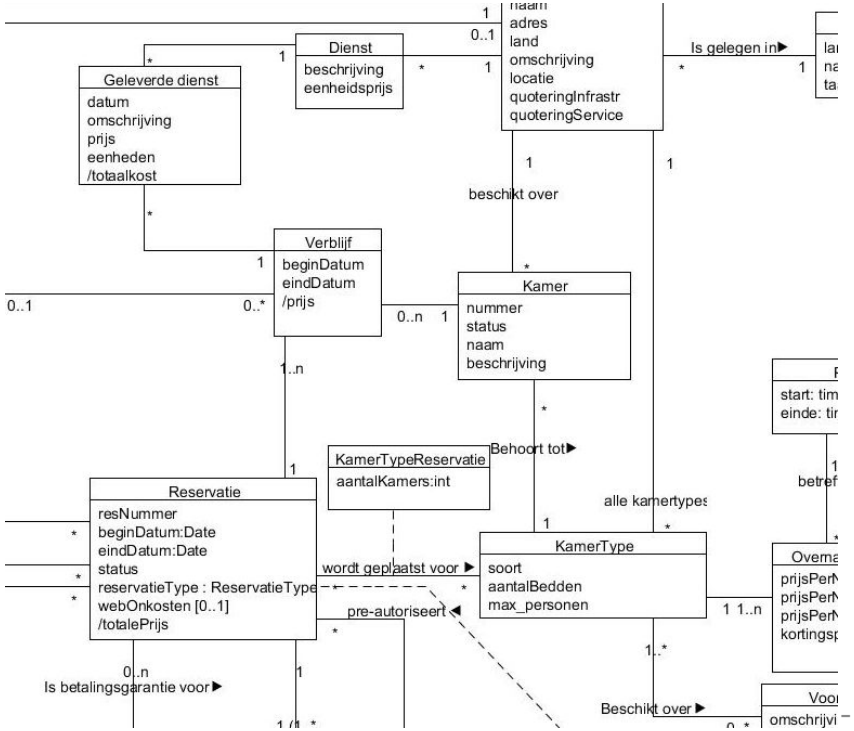
OO vs Relational:
class vs table
associations 1 to many (natural relation in relational, done through a List in OO)
1 to 1 (embedded in relational)
many to many - can't be done in the relational world - it is done using a 1 to many using an FK
attribute matches to a column
object maps to a row
behavior centric (partition is based on behavior) vs data centric (partition is based on data -that's why it's embedded)
associations are based on pointers (memory reference) associations are based on keys (based on FKs)
ID is the memory location vs ID is the PK
natural relation with a pointer = 1 to 1
pointer points in one direction (relations are unidirectional) vs bidirectional (can use the keys anywhere)
2 relations to make it bidirectional
memory vs persistent
methods vs there are none (closest thing to methods are stored procedures/trigger)

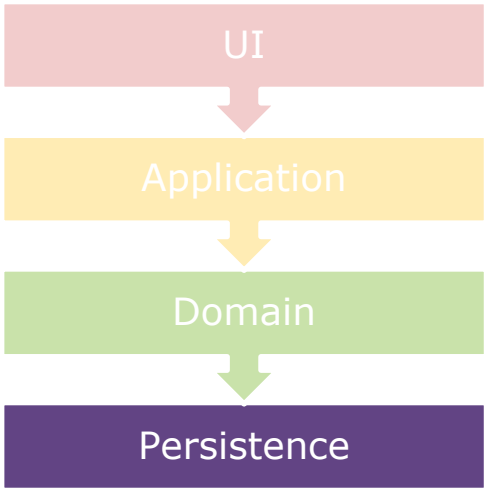
can't have hibernate generate the schema

Domain model

- Class diagram with conceptual classes
- Part of analysis

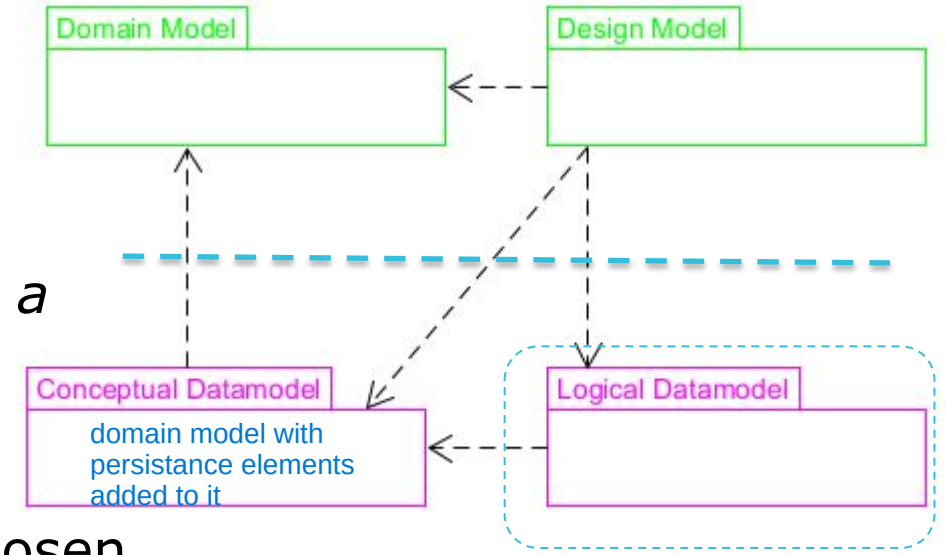




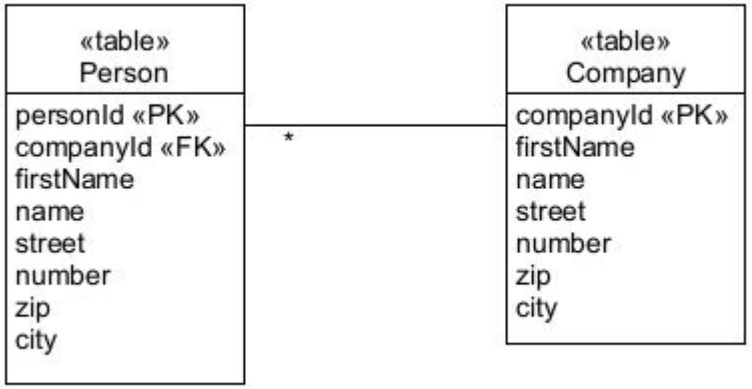


Karel de Grote Hogeschool

Logical *Data*model



- Only elements that are persisted
  (most classes, but not e.g. a pure behavioural class like SalePricingStrategy: does not need to be persisted)
- Storage technology was chosen
- Logical structure of datastructure
- Conventions
  - Use stereotypes for relational datamodel
    - **«table»**
    - **«PK»**
    - **«FK»**
  - Table names: plural



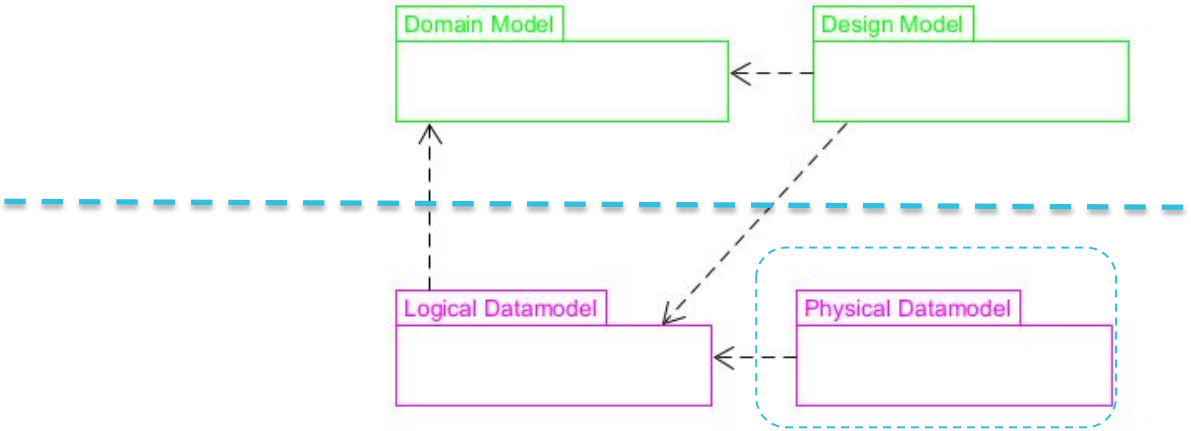
Note: sometimes people differentiate between **conceptual** datamodel (technology agnostic) and **logical** datamodel (technology specific)

Physical datamodel

- Physical structure

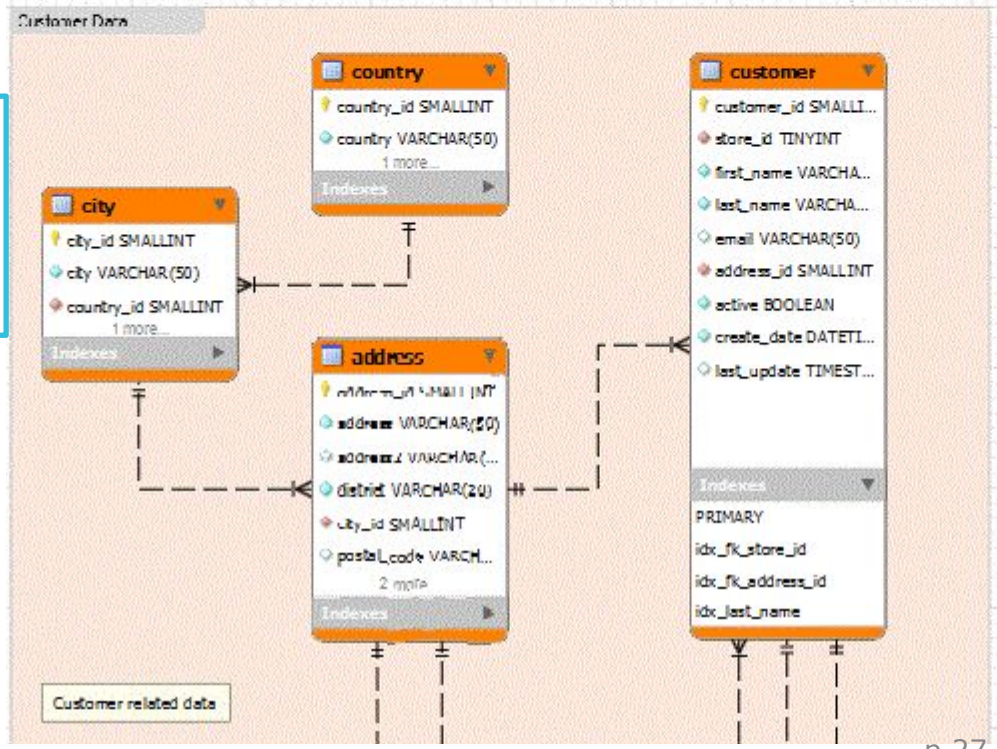
  database, product

  specific
- Exact data types
- Implementation aspects



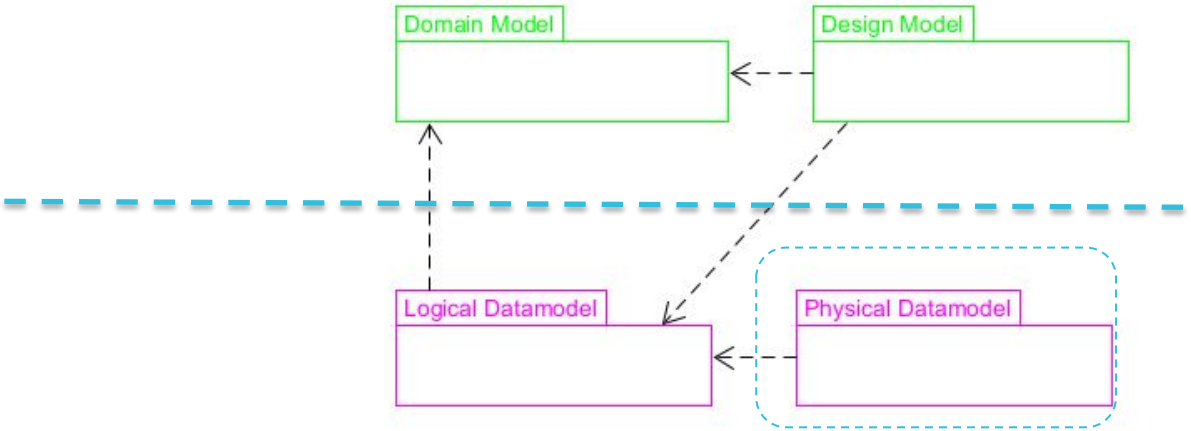
Often built using a database tool

e.g.. SQL Developer Data Modeler , IntelliJ, mySQL workbench…

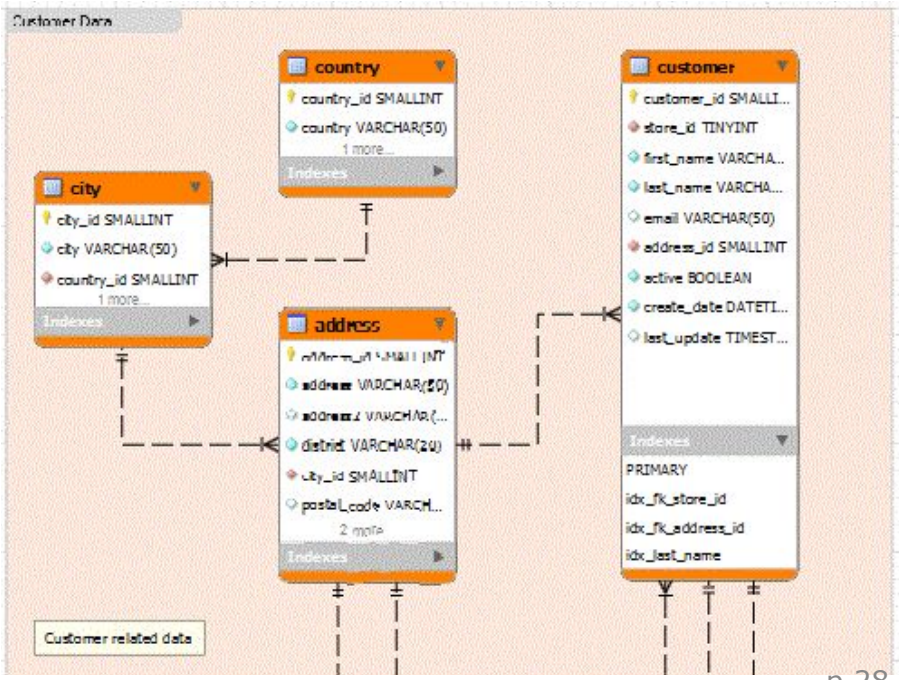


Karel de Grote Hogeschool

Physical datamodel



- Implementation aspects are often performance related
  - Define indexes on columns that are frequently searched
  - Denormalise
  - Product specific features:
    - Partitioning tables
    - Clustering tables
    - Special indexes: bitmap index



Karel de Grote Hogeschool

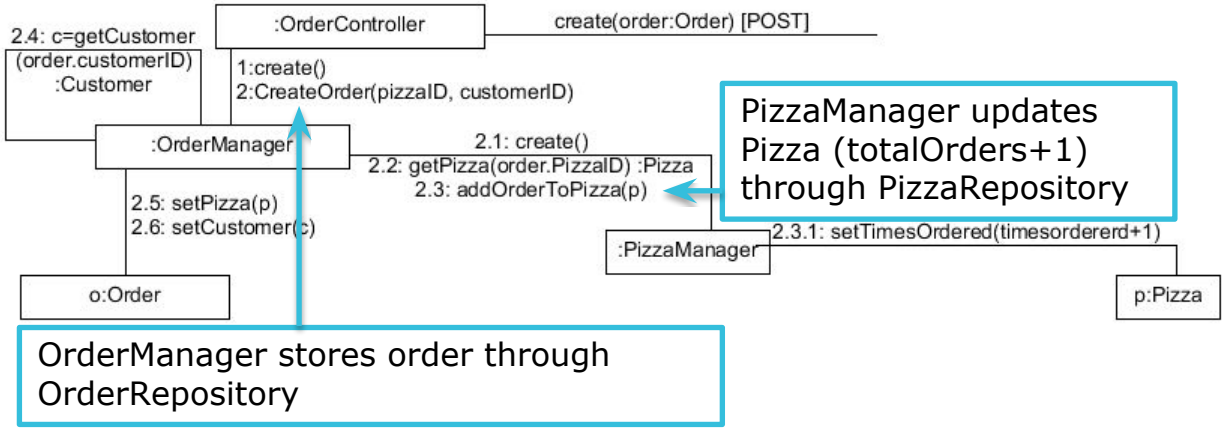
Transactions

4

KdG
Karel de Grote
Hogeschool

Transactions

- Transaction demarcation is a business decision, typically at the level of a technology agnostic service
- A service can invoke operations on multiple repositories
- Just like for databases, transactions should fail or succeed as a whole

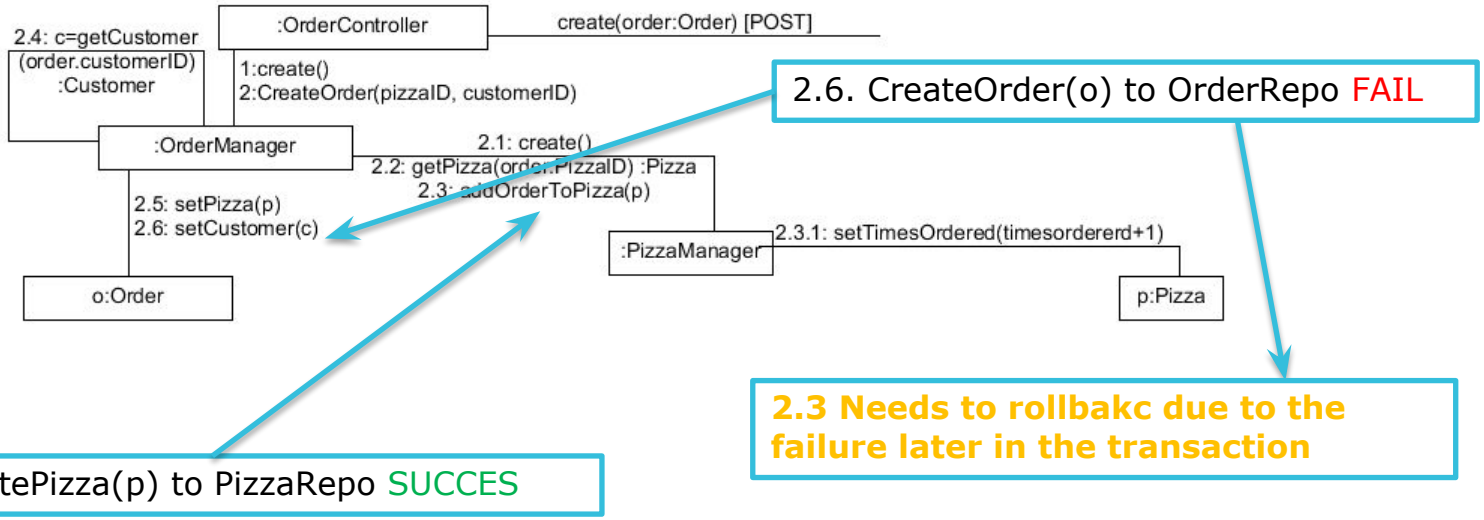


Transactions

- Transaction demarcation is a business decision, typically at the level of a technology agnostic service
- A service can invoke operations on multiple repositories.
- Just like for databases, transactions should fail or succeed as a whole

transaction ends when the method ends

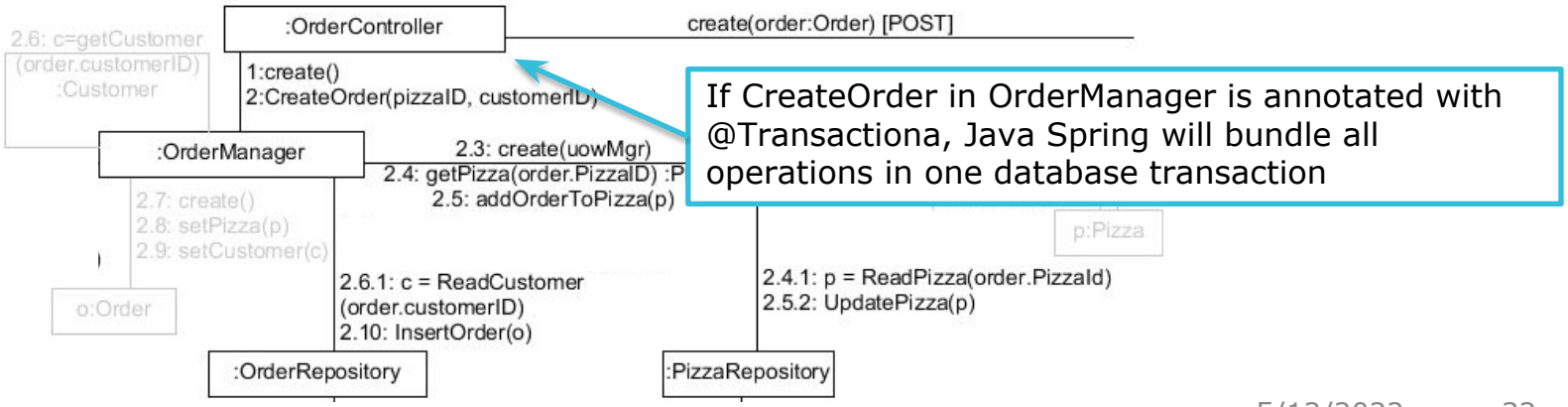
independent from DB



Transaction implementations

- Libraries offer solutions to implement transactions
- Java example
  - In Spring you can annotate a method with `@Transactional`
    - Before the first database modification Spring automatically starts a transaction
    - If all goes well the transaction is committed at the end of the method
    - If the method throws an exception the transaction is rolled back

can modify behaviors




Object-Relational
Mapping

Object

- Class
- Attribute
- Object
- Association
- Method

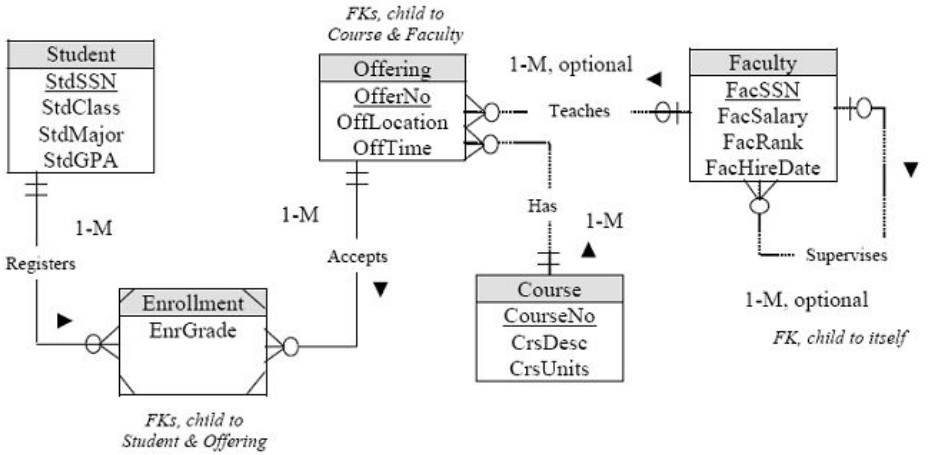
Relational

- Table
- Column
- Row
- Relation
- Stored Procedure

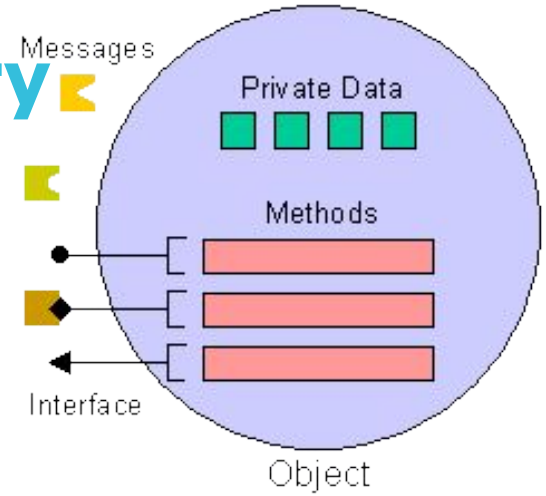
Many Objecyt Oriented concepts have a natural relational counterpart But there are differences as well…

OR gap: Persistent ⇔ Memory

- relational database is **persistent**
  - Entity associations use a key
    - = additional persistent attribute
  - A row contains one foreign key
  - A primary key can be referenced by many rows
  - Simplest relational association is a bidirectional 1-* association
  - *-* association: not supported. Split up in two 1-* associations.



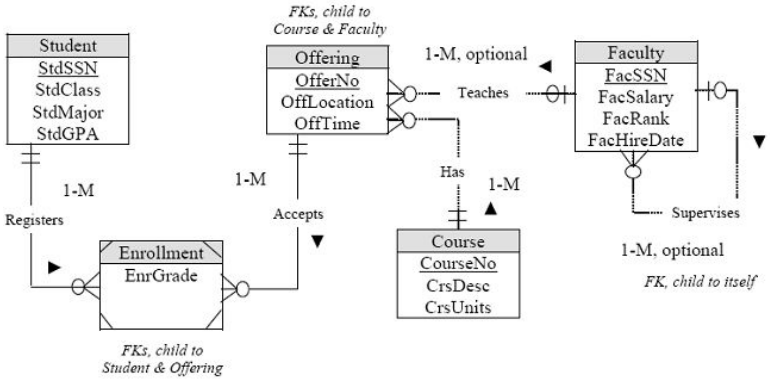
OR gap: Persistent ⇔ Memory



- A (Object Oriented) program is **transient**
  - Entity associations use a (memory) reference
  - References one thing
  - The same memory location can be referenced by many entities
  - Simplest memory relation is a unidirectional *->1 relation
  - -> * (… to Many) association: refer to a Collection
  - Bidirectional association: additional unidirectional association in the opposite direction
  - 1 -> (One to …) association: do not expose objects you want exclusive access to (or add extra logic)

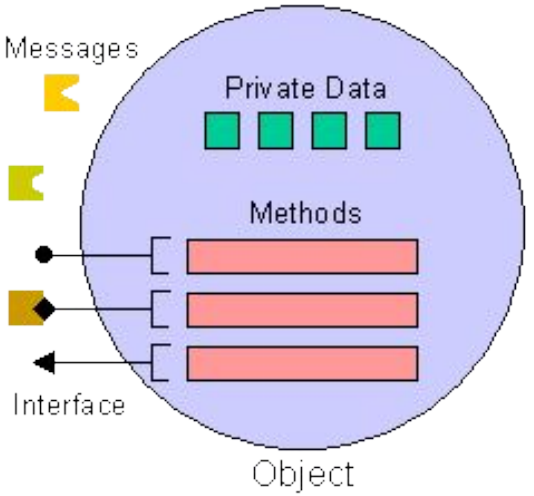
OR Gap: Data ⇔ Behaviour

- Relational: Data focus, based op mathematical model
  - stored procedures added later
  - 1-1 relations are few
  - SQL is a declarative language
    - Specify WHAT you want
    - DBMS determines steps (plan) to manage data



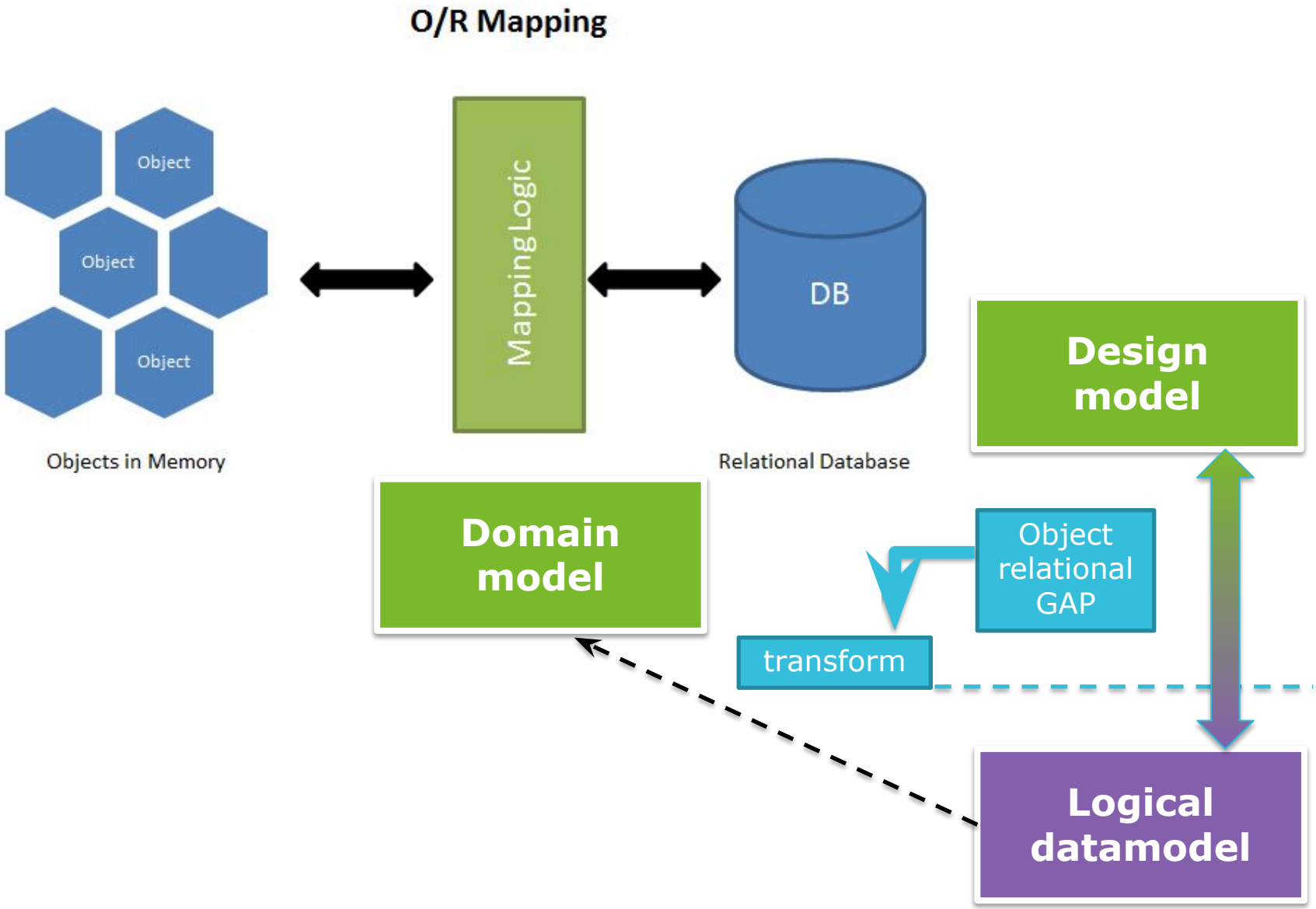
OR Gap: Data ⇔ Behaviour

- Object oriented: combines data and behaviour
  - encapsulation and data hiding
  - An attributes with its own behaviour gets a separate class, referenced in a -> 1 relation
  - methods are imperative
    - Specify HOW your want to do something
    - has steps, loops…
- Extra association type: inheritance
  - polymorphism, abstraction



How to bridge the Object-Relational gap?

Object – Relational Mapping



Object identification

- OO Objects identified by memory location
- RD Rows identified by key

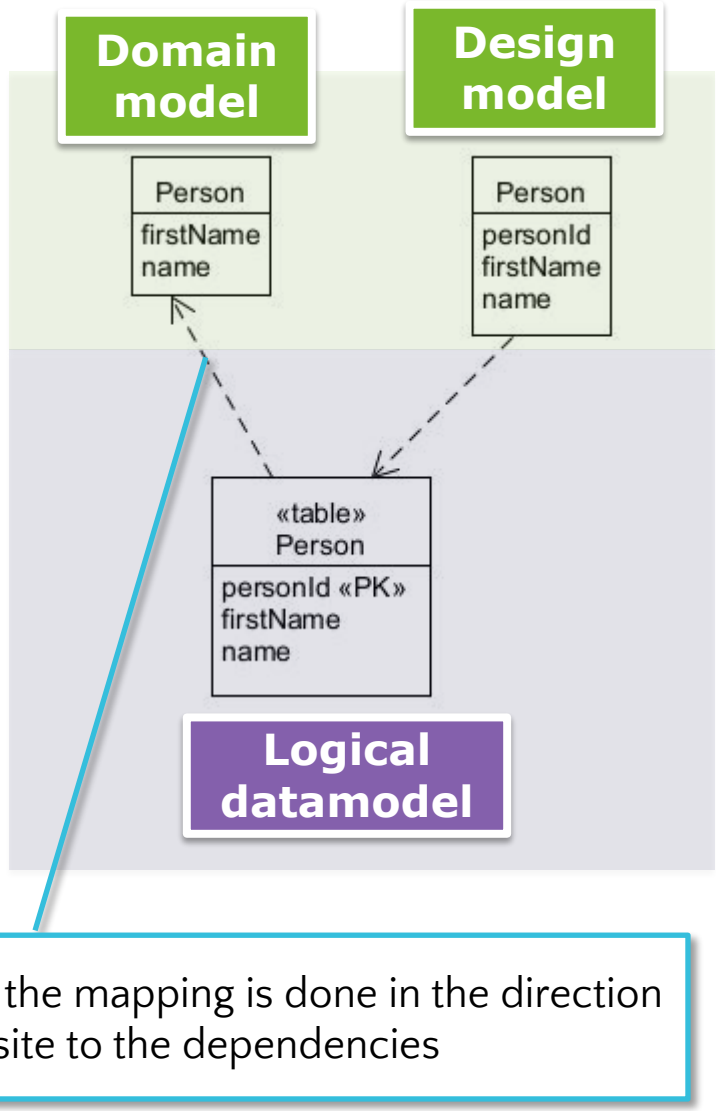
☐ OO -> RD requires adding identity (PK)

- Surrogate key (artificial identity)

OR

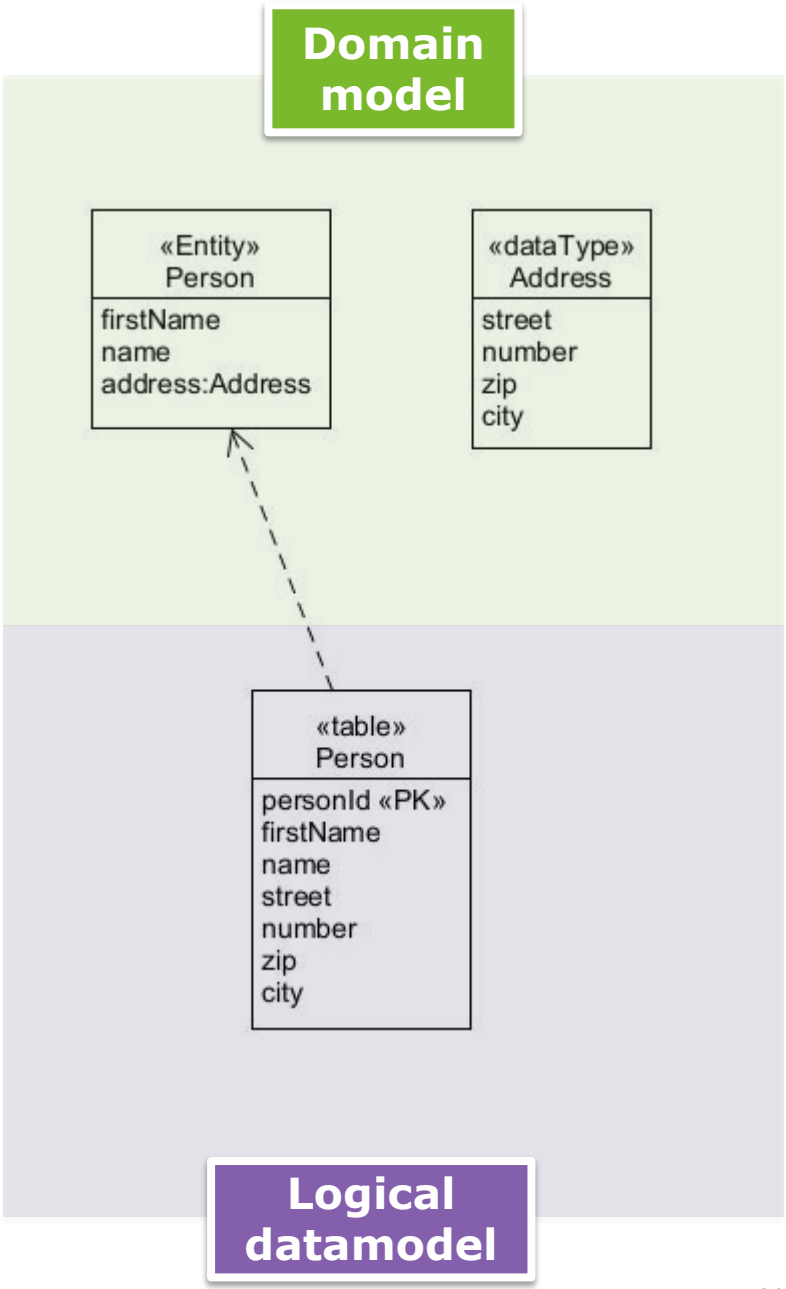
- One or more existing attributes

 If a surrogate key is added in the RD, it needs to be added to the OO design model as well



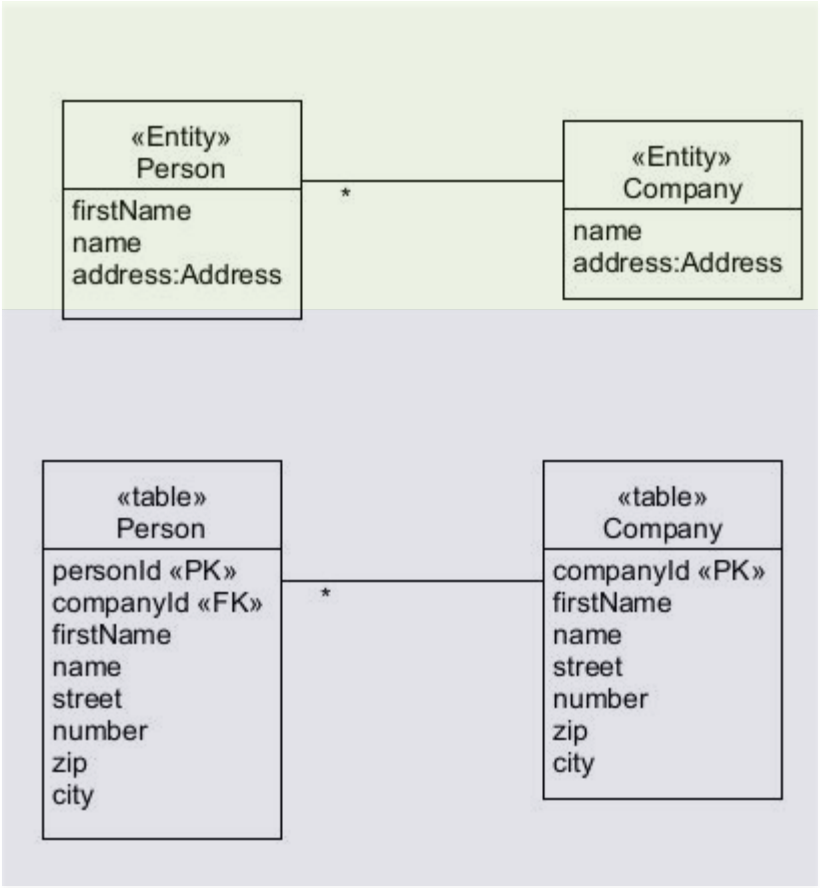
Mapping relations: 1- 1

- Add the attributes of the Value Object to the Entity table
- 1-1 DDD aggregates can also be mapped like this



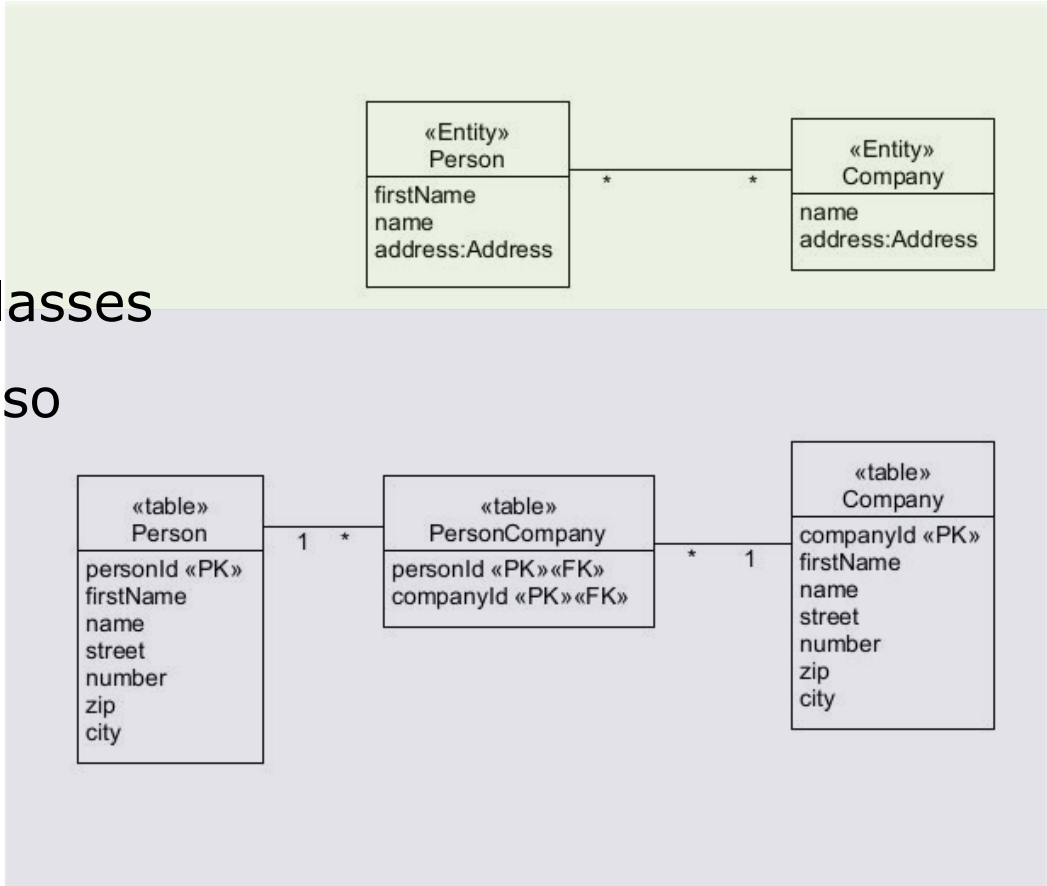
Mapping relations: 1- *

- Each class maps to a table
- FK goes to the *-side



Mapping relations: *-*

- In relational model transform *-* to 2 1-* relations
- Extra table (PersonCompany) has FK for both domain classes
- Association classes are also mapped in this way



Inheritance mapping strategies

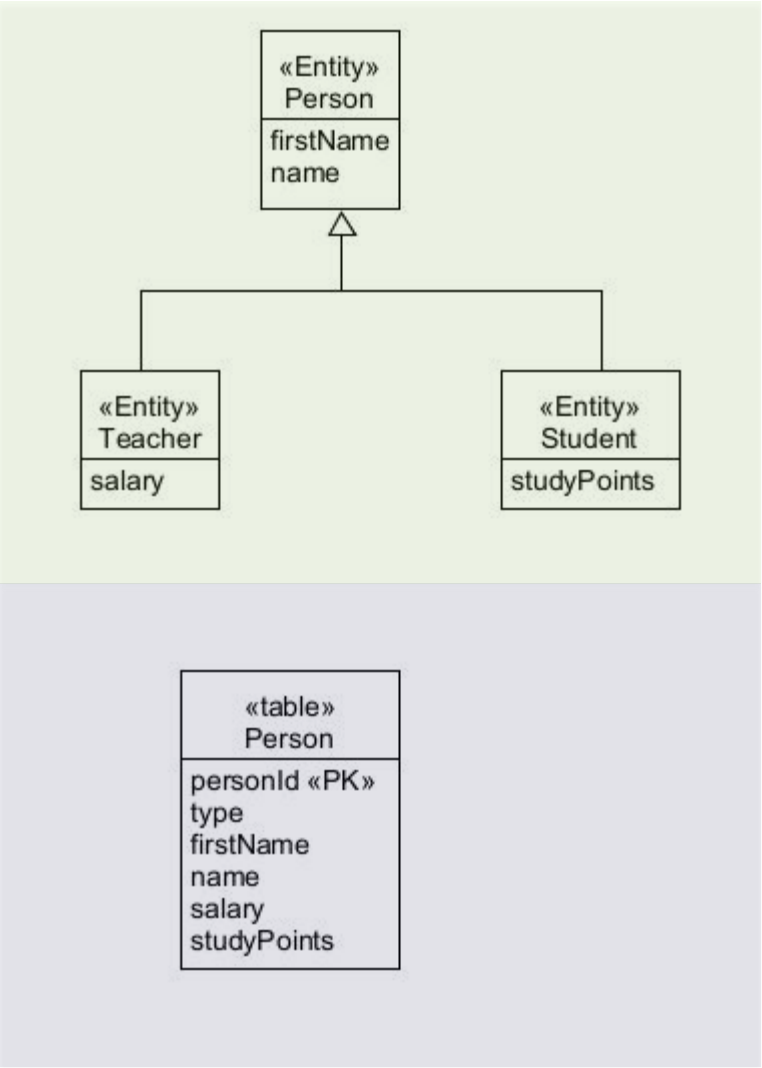
1. Table / hierarchy (1 table)

- Add type attribute
- All subclass attributes are optional
  - No NOT NULL constraints for subclass attributtes
  - Empty fields
  - Large, complex table

  Views can mitigate this

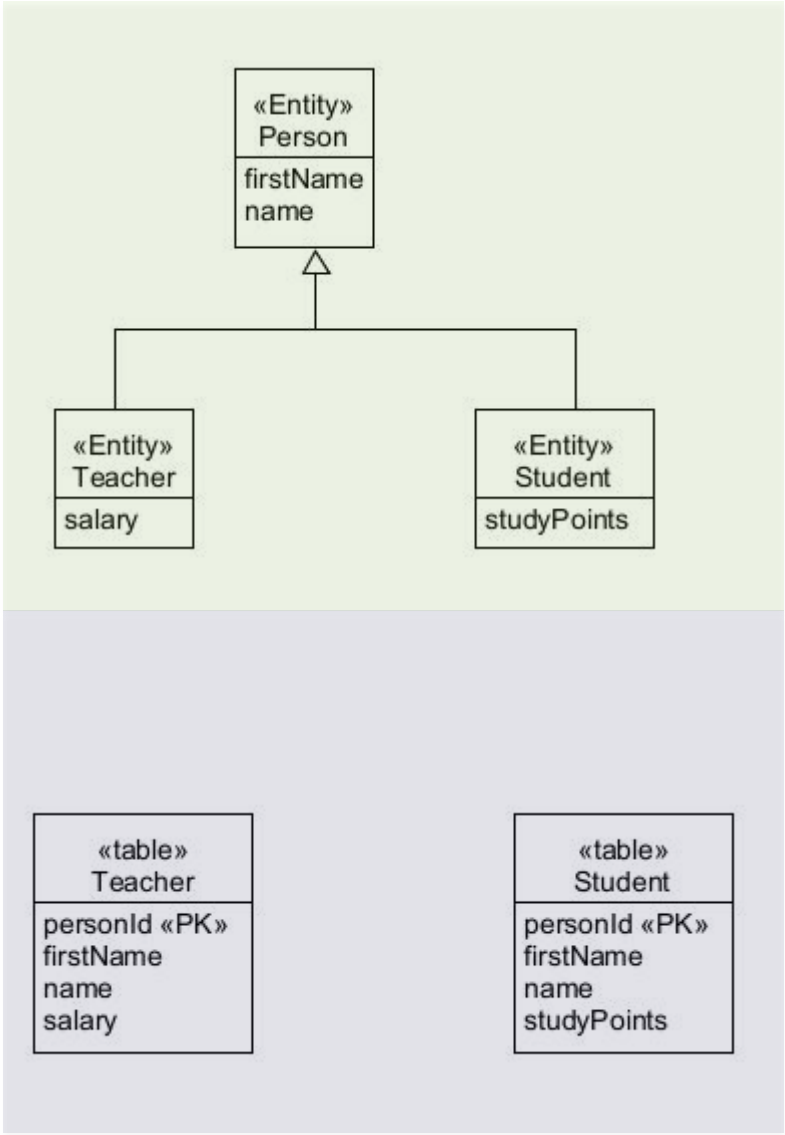
2. Table / subclass (2 tables)

3. Table / class (3 tables)



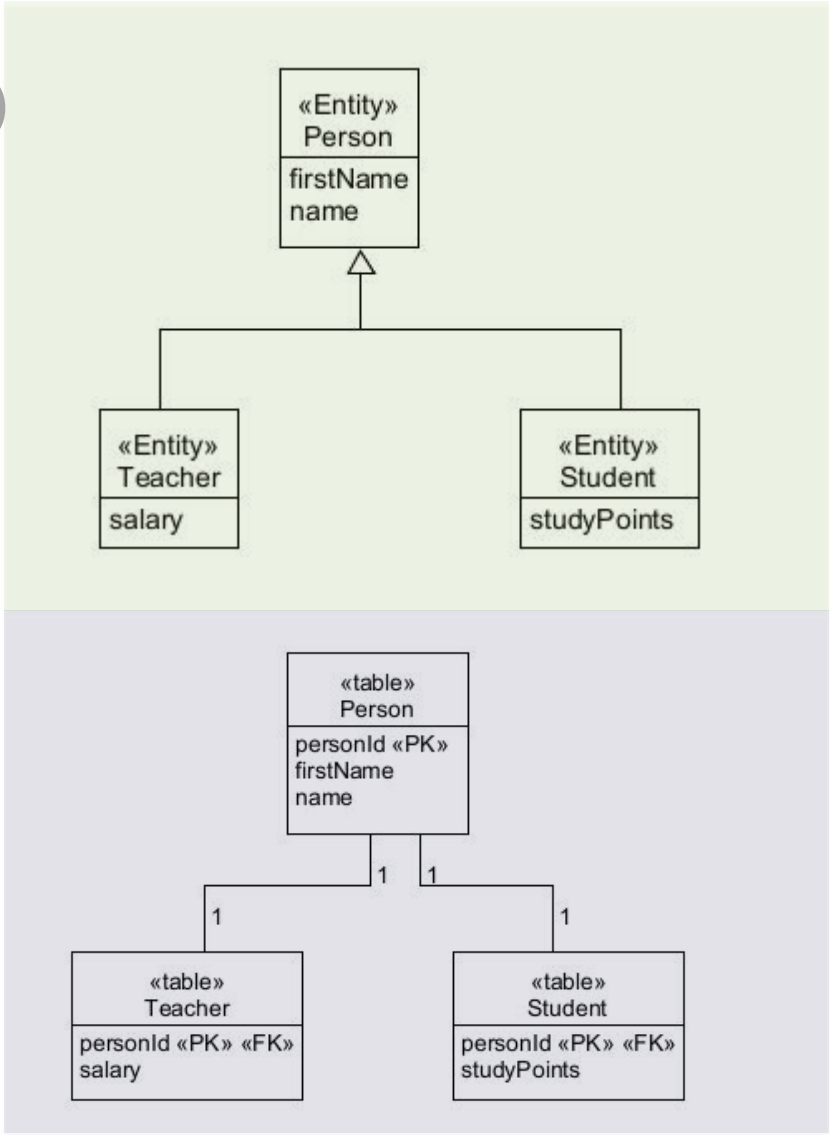
Inheritance mapping strategies

1. Table / hierarchy (1 table)
2. **Table/subclass (2 tables)**
   - superclass query requires UNION
   - Superclass relations need to be implemented for each subclass
3. Table / class (3 tables)



Inheritance mapping strategies

1. Table / hierarchy (1 table)
2. Table/subclass (2 tables)
3. **Table / class (3 tables)**
   - ✔ extra subclasses can be added without modifying existing tables
   - ✔ "multiple inheritance": a teacher can also be a student
   - ✘ JOIN needed



Inheritance mapping strategies

1. **Table / hierarchy (1 table)**
   - Fastest for reading **(most popular)**
2. **Table / subclass (2 tables)**
   - Useful if operations across subclasses are uncommon (example infrequent polymorphic queries)
3. **Table / class (3 tables)**
   - Useful if database structure changes often **(least popular)**

Object

- Transient
- Class
  - Attribute
    - Inheritance
- Object
  - instance attirbute
  - implicit ID: reference
- Relation
  - reference
    - Unidirectional
    - Ordered (List, array…)

Relational

- Persistent
- Table
  - Column
- Row
  - field
  - explicit ID: primary key
- Relation
  - foreign key
    - bidirectional

Object

- Behaviour focus
  - imperative (how)
  - Works with objects
- Methods
  - interfaces
  - encapsulation
  - Data hiding

Relational

- Data focus
  - Declarative (what)
  - works with sets
- Stored procedures*
  - Triggers, constraints
  - Permissions

Some similarity

* Later addition

Summary

1. Persistence stores
2. Repositories
3. Datamodels
4. Transactions
5. Object-Relational Mapping