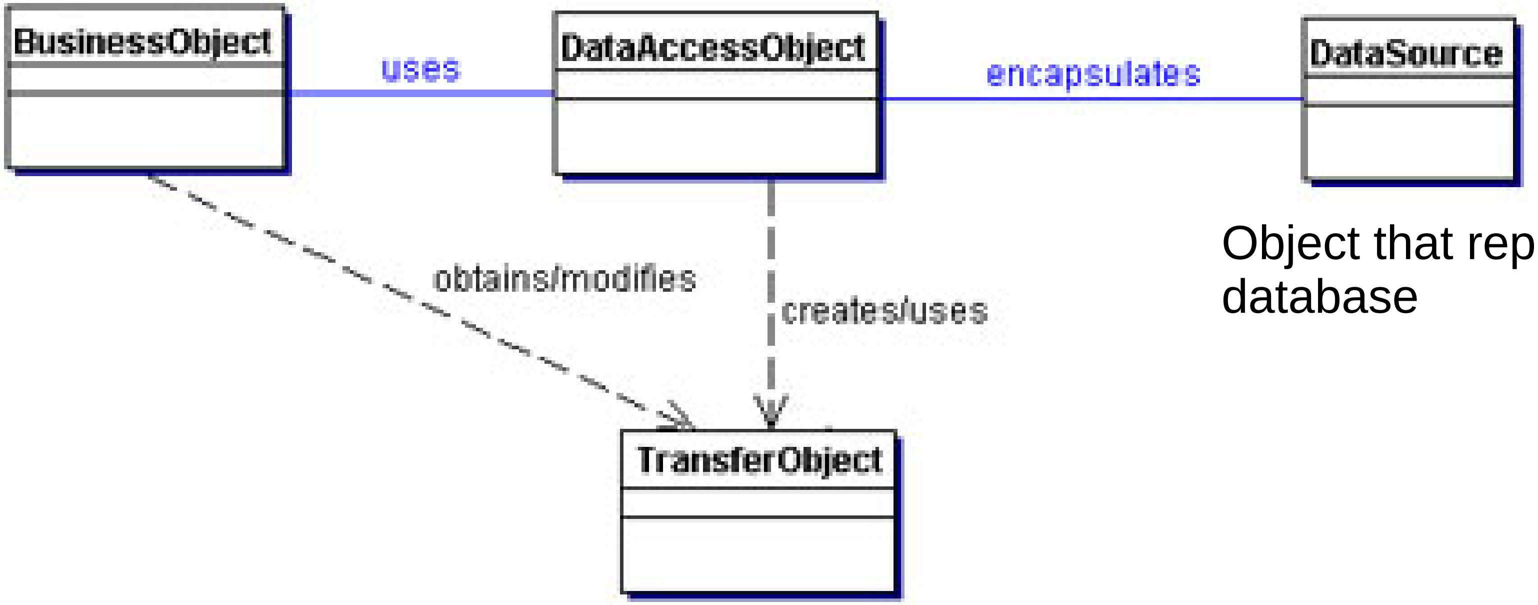
Object Persistence

Jerome David
Université Grenoble Alpes

Master MIASHS
2017-2018

Data Access Object

- This is a design pattern allowing to abstract and to encapsulate persistence mechanisms



Application Object:
servlet, bean for JSF, etc.

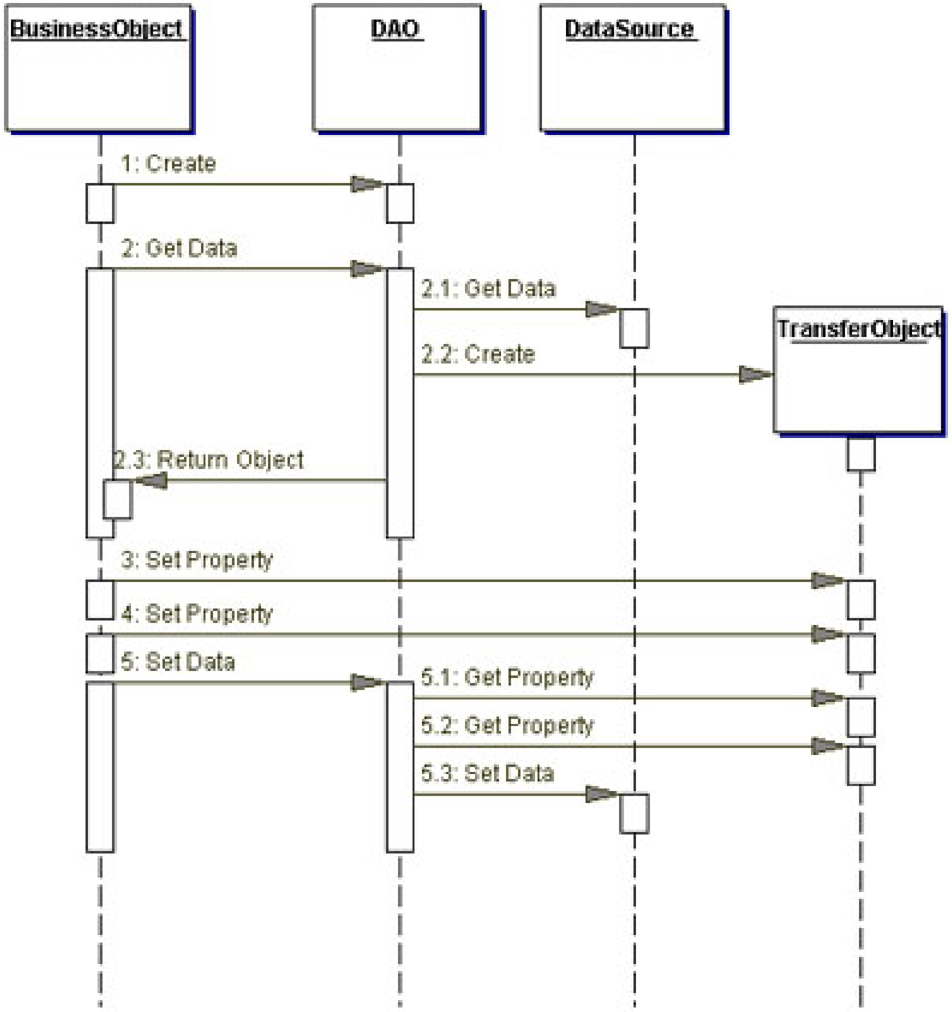
Object that encapsulates
databases queries

Object that represent the
database

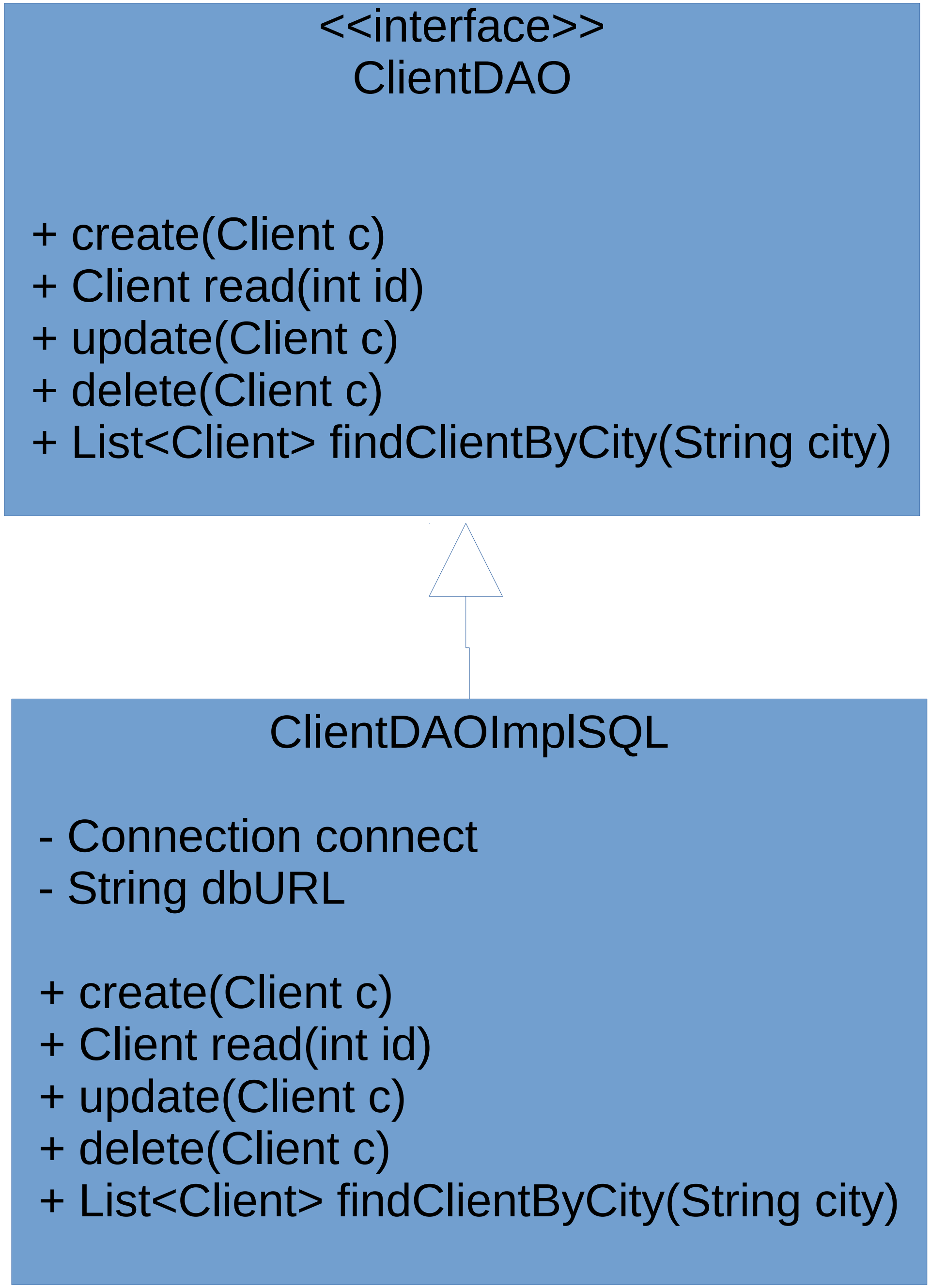
Object from the data model:
User, Client, Bill, etc.

Principles

- All accesses (i.e. queries) are encapsulated into the DAO
- Calls to DAO methods are made only from business objects (not from data objects (TransferObjects)
- Basic DAO operations are
  - Create, Read, Update, Delete



Example



<<interface>>
ClientDAO

+ create(Client c)
+ Client read(int id)
+ update(Client c)
+ delete(Client c)
+ List<Client> findClientByCity(String city)

ClientDAOImplSQL

- Connection connect
- String dbURL

+ create(Client c)
+ Client read(int id)
+ update(Client c)
+ delete(Client c)
+ List<Client> findClientByCity(String city)

Client

+ int id
+ String name
+ String address
+ String city
+ List<Order> orders

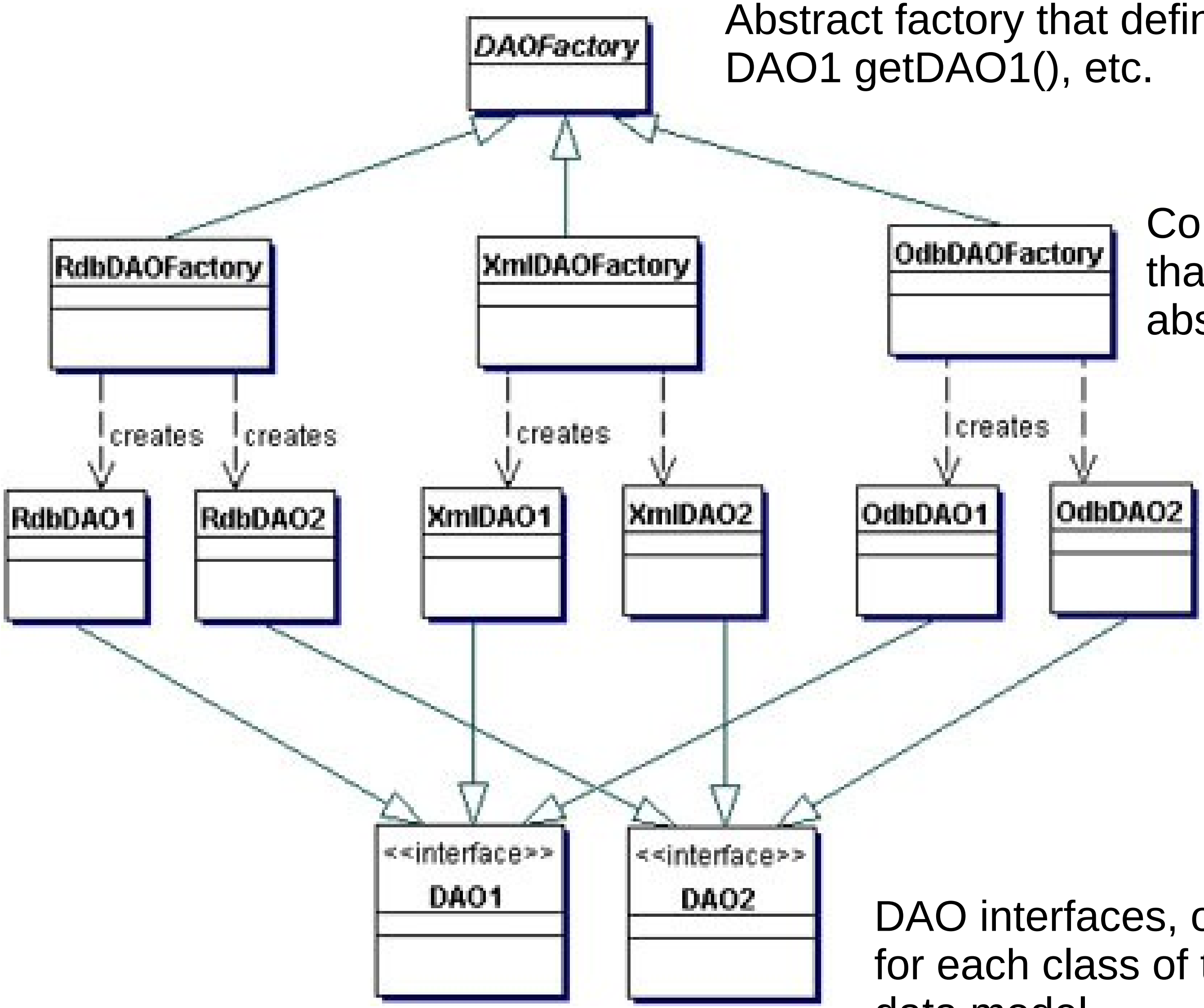
+ int getAmountOrders()
+ hasBuy(Item p)

Why we define a DAO interface and not only the implementation?

Manage several DAO implementations

- Abstracting DAO allows to manage several kinds of persistence frameworks
- We have a generic interface for DAO and one implementation for each persistence technology used (RDB, XML, RDF triplestore, KeyValue Store)
- If we change the persistence technology we only have to redefine a new implementation of the DAO and the business classes does not need to be changed.

DAO Factories



Abstract factory that defines abstract methods DAO1 getDAO1(), etc.

Concrete factories that implements the abstracts methods

DAO implementation, one for each class of the data model and for each kind of persistence

DAO interfaces, one for each class of the data model

DAO: Pros and Cons

+ It allows independence between application logic and the persistence framework to be used

  - It is easier to change the persistence technology
  - The code is more maintainable

- It adds a bit of overhead when we code an application

  - But there are tools that allows to automate their creation

Java Persistence API

- JPA provides an object/relational mapping facility.
  - It consists in automatically maps objects to database records.
- It consists of
  - The API itself
  - A query language: JPQL
  - The Java Persistence Criteria API
  - Object/relational mapping metadata

Entities

- Entities are the classes of the data model
  - An entity class is usually a table in the database and each instance of an entity class is a row
- To make a class an Entity class, we have to
  - Annotate the class with `javax.persistence.Entity`
  - Declare a primary key, i.e. an instance attribute with the annotation `javax.persistence.Id`
  - To not make the class final
    - it will be automatically extended by JPA
  - The class has to be a bean
    - At least a public or protected no-argument constructor
    - Instance attributes must be private
    - And accessible with getters and/or setters

Persistent attributes

- To be stored the type of an instance attribute has to be:
  - A primitive type
  - A String
  - Serializable type
  - Enumeration
  - An entity type (or collection of entity type)
  - An embeddable class

Attribute constraints

- We can add constraints on attributes using annotations
  - Constraints are in the package `javax.validation.constraints`
- For instance:
  - `@NotNull`

  -
    ```
    @Pattern(regexp = "[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\\."
                 + "[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*@"
                 + "(?:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\\.)+
    [a-z0-9]"
                 + "(?:[a-z0-9-]*[a-z0-9])?",
                 message = "{invalid.email}")
    ```
  - `@Column(unique=true)`

Entity relationships

- Entities classes can be in relation.
- There are several multiplicities
  - `javax.persistence.OneToOne`
    -
  - `javax.persistence.OneToMany`
    - Example: a Person can have several Phone
  - `javax.persistence.ManyToOne`
    - Example: a Phone belongs to only one Person
  - `javax.persistence.ManyToMany`
    - Example: A Student follows several Course and a Course is followed by several Student

Direction of relationships

- Relations between two classes can be:
  - **Unidirectional**: only one of the two classes in relation have a reference to the other
  - **Bidirectional**: both the two classes in relation have a reference to the other.
- The directions define how we can navigate between entities using their relationships
- When relations are bidirectional, the inverse side of the relation has to refer to the owning side using `mappedBy` parameter
  - For OneToOne and ManyToMany relation, you are free to choose the owning side
  - For ManyToOne and OneToMany the owning side is Many

Examples

```
@Entity
public class Person implements Serializable {
    @Id
    private long id;

    @NotNull
    private String lastName;

    @NotNull
    private String firstName;

    @OneToMany(mappedBy="owner")
    Collection<Dog> dogs;
}
```

```
@Entity
public class Flea {
    @Id
    private long id;

    @ManyToMany(mappedBy="friends")
    private Collection<Dog> houses;
}
```

```
@Entity
class Dog implements Serializable {
    @Id
    private long id;

    private String name;

    @ManyToOne
    private Person owner;

    @OneToOne
    private Collar collar;

    @ManyToMany
    private Collection<Flea> friends;
}
```

```
@Entity
public class Collar implements Serializable {

    @Id
    private long id;

    private String phoneNumber;

    @OneToOne(mappedBy="collar")
    private Dog dog;
}
```