O/R mapping with Hibernate

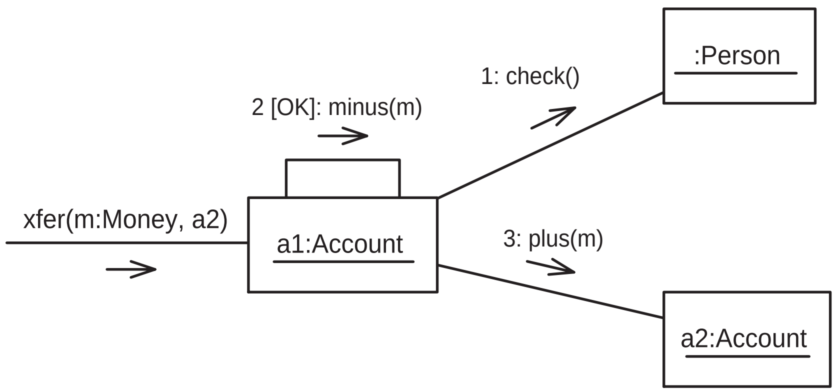
Geert Vandevenne - Abis Training & Consulting

Geert Vandevenne - Abis Training & Consulting

Classes and objects

OO-applications are composed of objects which

- consist of **data** and **behaviour**
- are connected to each other
- send messages to each other



Classes and objects

- objects are described in classes
- classes are instantiated at runtime and populated with data
- these data must be preserved i.e. persisted

Features of persistence mechanisms

- **persist** the information (data) in the object model, i.e.
  - the data in the objects described as attributes in the class model
  - links between objects described as relationships in the classes model
- **synchronization** between application and data
  - data in memory must be synchronized with data in data store
  - data in data store must be synchronized with data in memory
  - important if different applications access the same data
- **transactions**
  - set of actions that move data from one consistent state to another
  - key features: Atomicity, Consistency, Isolation, Durability
- **concurrency control**
  - different users/applications must reach the same data at the same time...
  - ...while keeping the data in a consistent state

Features of persistence mechanisms (cont.)

- query mechanism
  - need for some mechanism to retrieve data selective from the data store
- identity support
  - avoid multiple copies of the same data
- security
  - unauthorized people must not see sensitive data

Standard mechanisms in java to persist these objects:

- serialization
- JDBC

Serialization

Standard component in each JVM

- lightweight persistence mechanism
- classes implement interface Serializable or Externalizable
- persist with writeObject(Object) of ObjectOutputStream

Features

- uses the class-model as data model
- serialization of complete object graph to e.g. file
- use of keyword transient

Serialization

Drawbacks

- no transaction management or concurrency control
- no queries possible against data
- granularity is entire object graph
- no identity support or coordinated management of instances in storage

  -> multiple copies of same instances can exist and manipulated
- no automatic synchronization between application and data
- not scalable

Conclusion:

-> not suitable to store large quantities of data

Java DataBase Connection (JDBC)

Standardized framework to interact with Relational Database Management Systems (RDBMS)

- classes and interfaces in packages **java.sql** and **javax.sql**
- implementations provided by RDBMS vendors

Features

- uses data model of relational system:
  - data reside in set of tables
- query database with Structured Query Language (SQL)
- makes use of features of the RDBMS regarding:
  - transaction management and concurrency control
  - identity support through primary keys (PK)
  - security

Java DataBase Connection (JDBC)

Drawbacks

- Existence of many SQL dialects
- developers must understand very well the relational model
- procedural approach of database
- No support for object model!!!

You have to make a choice in your application:

- consider entities as rows in database

  -> loose of object capabilities of java
- consider entities as objects

  -> must be mapped to relational structure

Java DataBase Connection (JDBC)

Conclusion:

- supports a lot of interesting features
- widely supported
- mismatch between object model and relational model

  -> need for mapping (called O/R mapping)

Object - relational mapping

Data mapping involves a lot of problems:

- mapping of inheritance trees
  - inheritance is not supported in RDBMSs
- difference in identification of entities
  - in OO: not directly supported
  - in RDBMS: through primary keys (PK)
- difference in relationship management between entities:
  - in OO: links between objects
  - in RDBMS: PK - FK relations

Object - relational mapping

Miscellaneous concerns:

- transaction management

  -> in application (application environment) or by datastore?

- synchronization

  -> how to keep the data in the objects in sync with the database an vice versa

O/R mapping solutions:

- do it yourself: extremely difficult and cost intensive
- use existing solutions: Hibernate, JDO, Toplink, EJB,...

Important considerations

Domain logic

- organization determines the way of mapping

Architectural aspects

- prescribe how the domain logic talks to the database

Structural aspects

- describe the actual mapping of an OO model to a relational database

Behavioural aspects

- explain how objects are loaded from and saved into the database

Organization of domain logic

Different ways to organise domain logic (Martin Fowler):

- **Transaction Script**
  - **organisation of domain logic around transactions**
  - **procedural - not object oriented**
  - **suitable for simple CRUD applications**
- **Table Module**
  - **organisation of domain logic around database tables**
  - **organisation of procedures in objects**
  - **suitable for manipulating result sets of data**
- **Domain Model**
  - **organisation of domain logic around business objects**
  - **fully object oriented**
  - **suitable for applications with complex business logic**
  - **Hibernate can be used in case a real Domain Model is used**

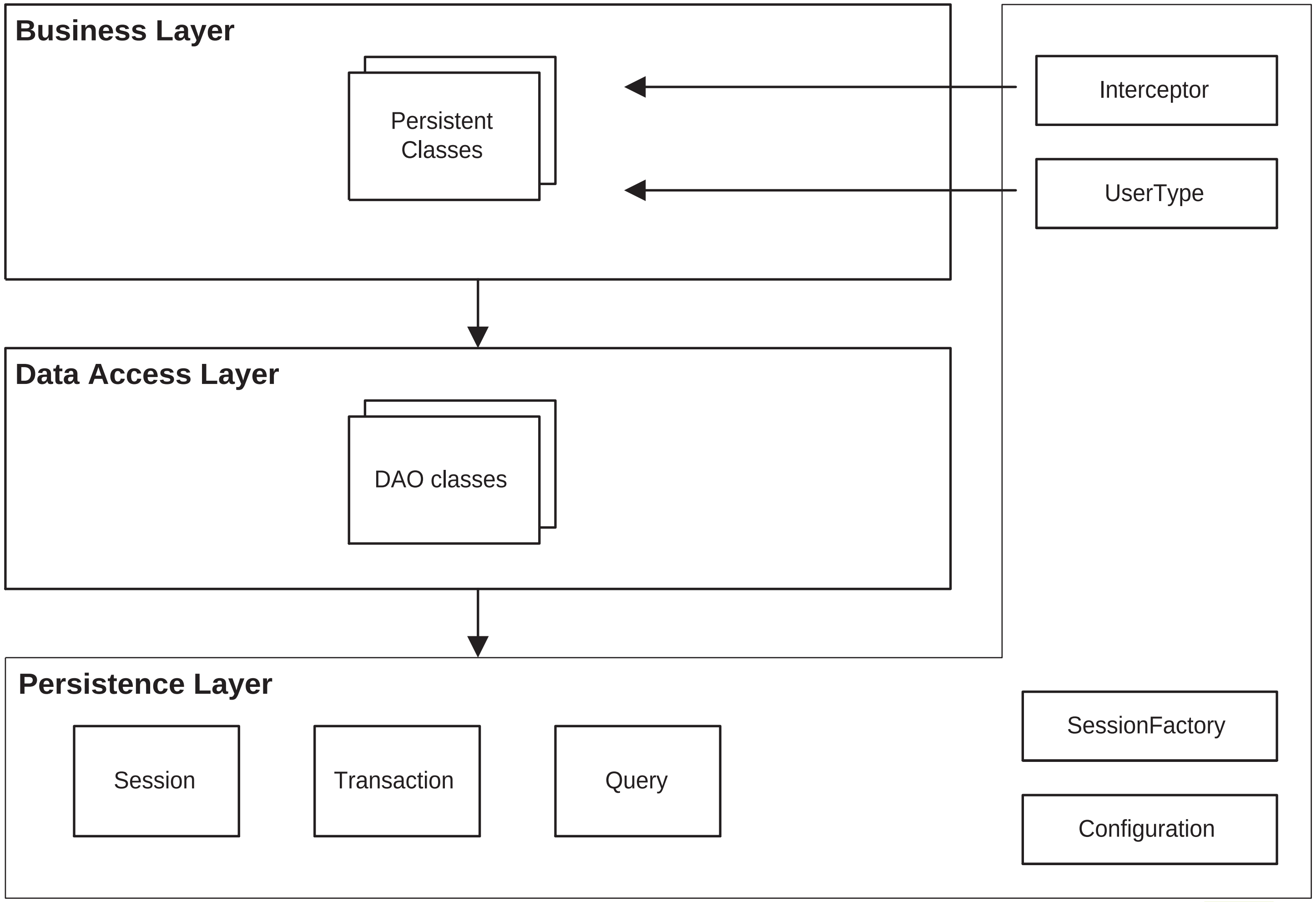
Architectural considerations

Domain logic and Data Access logic should be separated in different layers

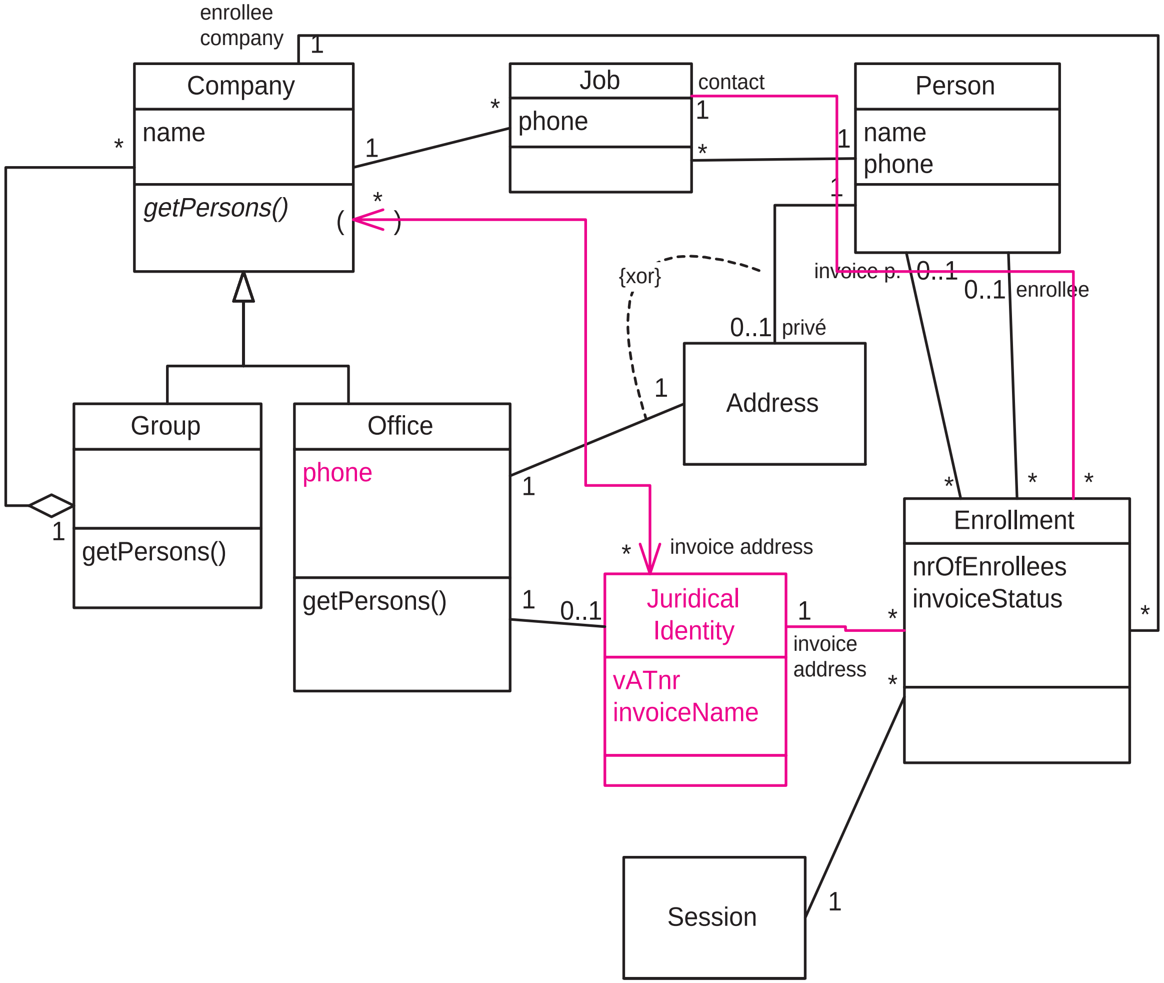
Several possibilities:

- **DataMapper pattern**
- **Data Access Object design pattern**

Hibernate Architecture



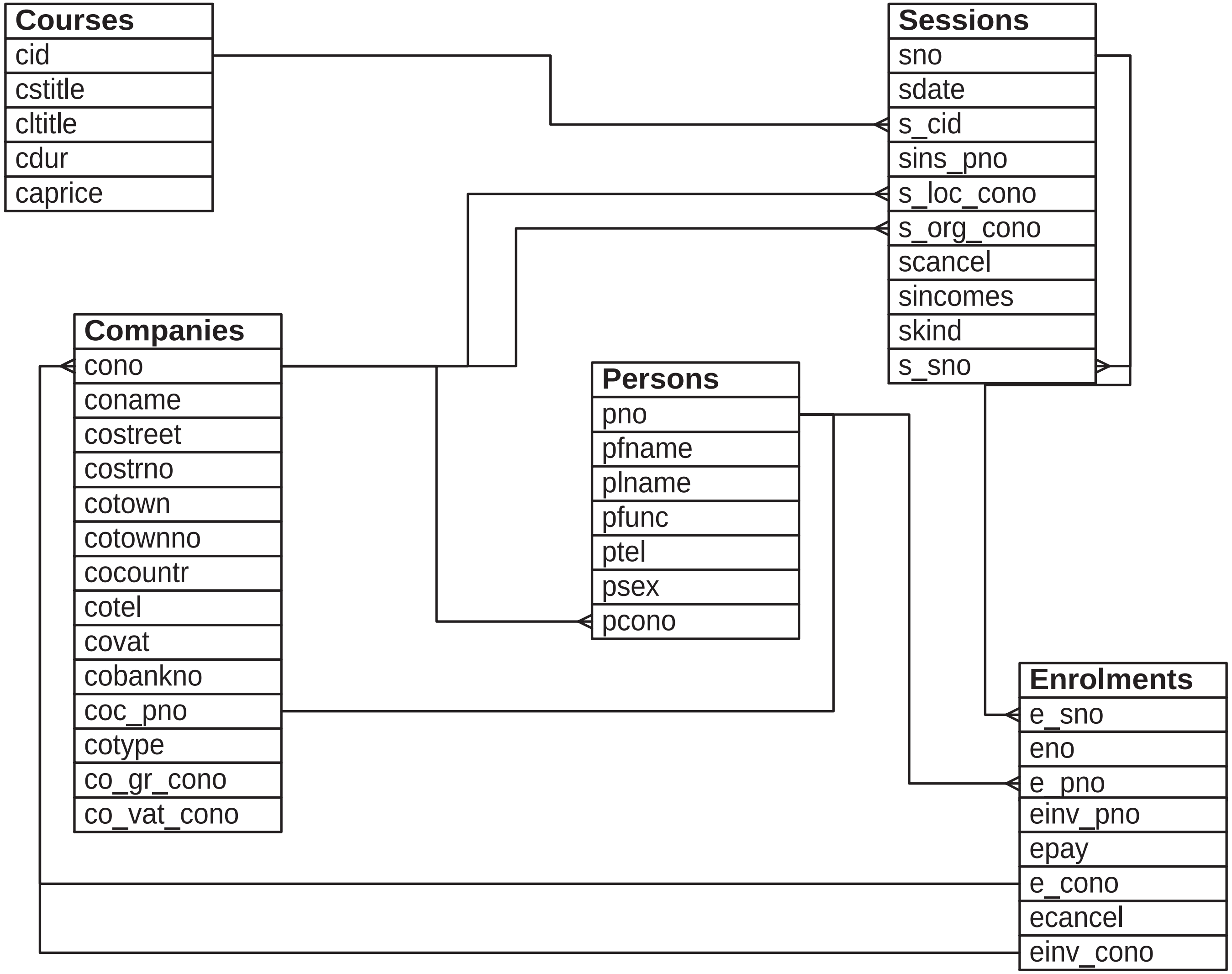
Structural aspects - domain model



O/R mapping with Hibernate

O/R mapping with Hibernate

Structural aspects - database model



Courses
cid
cstitle
cltitle
cdur
caprice

Sessions
sno
sdate
s_cid
sins_pno
s_loc_cono
s_org_cono
scancel
sincomes
skind
s_sno

Companies
cono
coname
costreet
costrno
cotown
cotownno
cocountr
cotel
covat
cobankno
coc_pno
cotype
co_gr_cono
co_vat_cono

Persons
pno
pfname
plname
pfunc
ptel
psex
pcono

Enrolments
e_sno
eno
e_pno
einv_pno
epay
e_cono
ecancel
einv_cono

Persistent classes

Some classes of the Domain Model are persistent

Hibernate is a TRANSPARENT persistency framework

- take care of bi-directional associations yourself

Hibernate works with POJOs (Plain Old Java Objects)

- Serializable interface not needed
- no-argument constructor obligatory (package friendly or higher visibility)
- accessor methods (can be private)

Persistent classes

Hibernate maps domain model and database schema with XML mapping files

What must be mapped:

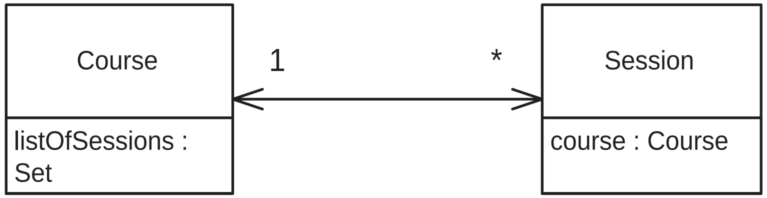
- **properties**
- **associations**
- **hierarchies**

Persistent classes - property mapping

- conversion of java types to sql types
  - rich set of Hibernate built-in types
  - e.g. string, integer, double, date, time, clob,...
- a domain model contains often value types
  - result of fine-grained object model
  - value types do not have a (database) identity
  - e.g. VATnumber, PhoneNumber, Euro,...

  -> Possibility to create your own user types

Association mapping



Course

listOfSessions : Set

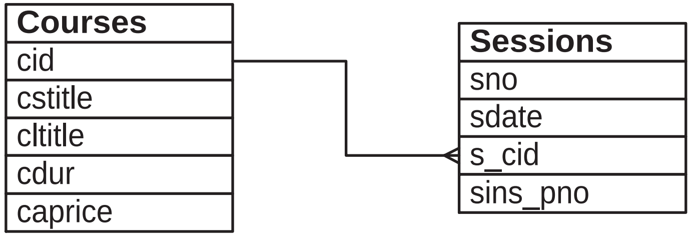
1

*

Session

course : Course

- multiplicity of association
- directionality of association
  - uni- or bidirectional
  - must be implemented in java-code
  - no managed associations in Hibernate
  - relations in RDBMS always bidirectional
- pk - fk relationship in RDBMS



Courses

cid

cstitle

cltitle

cdur

caprice

Sessions

sno

sdate

s_cid

sins_pno

Association Mapping

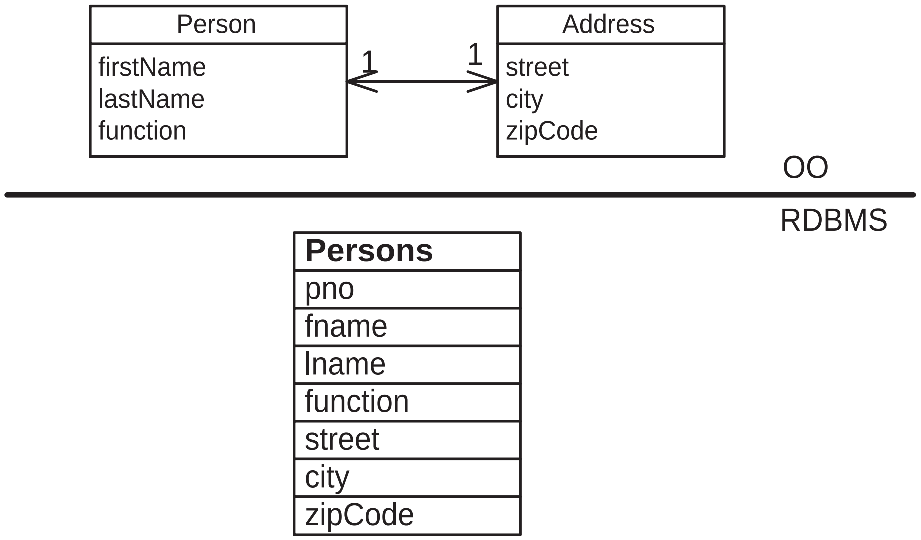
Hibernate supports:

- one-to-one, many-to-one and many-to-many associations
- from a java-perspective it supports mapping of sets, bags, lists and maps
- polymorphic associations
- (polymorphic queries)

Value types

Difference between

- entity type: has its own database identity (see further)
- value type: depends on database identity of entity type



Person

firstName
lastName
function

1 1

Address

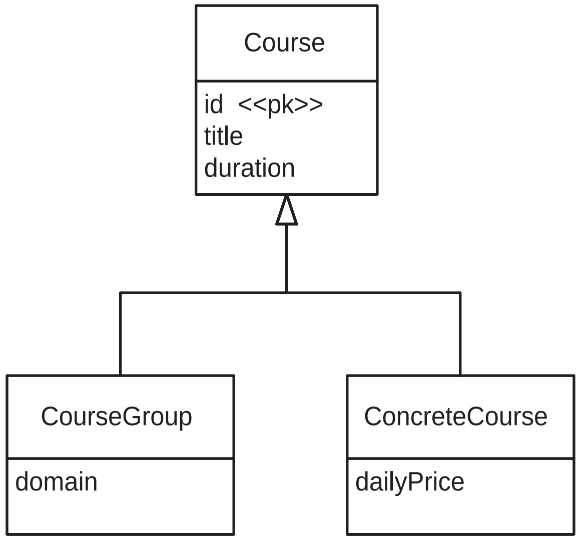
street
city
zipCode

OO

RDBMS

Persons
pno
fname
lname
function
street
city
zipCode

Hierarchy mapping



Hierarchical relations between entities not supported in database

Three alternatives:

- table per concrete class (concrete table inheritance)
- table per class hierarchy (single table inheritance)
- table per subclass (class table inheritance)

Table per concrete class

coursegroups
id <<pk>>
title
duration
domain

concretecourses
id <<pk>>
title
duration
dprice

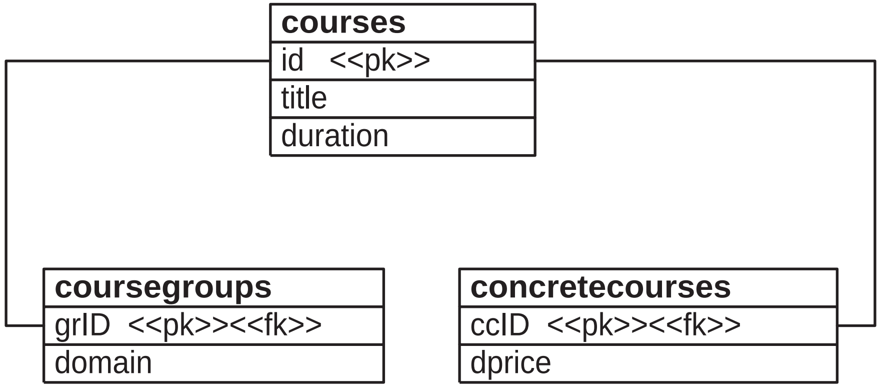
- No special mapping needed
- Create one mapping per class
- used when super class is abstract
- entity integrity can not be enforced by the database
- each change to super class -> change of all subclass tables

Table per class hierarchy

courses
id <<pk>>
title
duration
domain
dprice
ctype <<discriminator>>

- **used with few subclasses with few attributes**
- **gives a lot of null values in table**
- **violates normalisation rules**
- **easy refactoring**
- **discriminator**

Table per subclass



- create pk-fk relationships in database
- lots of joins to compose object
- SQL can not enforce consistency of model

Hierarchy mapping - general remarks

You can not mix strategies within one hierarchy

You can mix strategies in your application

Choose a hierarchy mapping strategy

- No polymorphic queries or associations needed
  -> table-per-class strategy
- Polymorphic queries or associations needed
  - not to many subclasses and not to many attributes in subclasses
    -> table-per-class-hierarchy
  - many subclasses or many attributes in subclasses
    -> table-per-subclass

Persistent classes - example

```
<hibernate-mapping package="be.abis.model" schema="vj">
    <class name="Course" table="COURSES" >
        <id name="course_id" column="cid" type="long">
            <generator class="identity" />
        </id>

        <property name="title" column="stitle" type="string" not-null="true" />
        <property name="price" column="price" type="be.abis.model.Euro"/>

        <discriminator column="cotype" type="string" />

        <set name="abisSessions" inverse="true"  >
            <key column="s_cid" />
            <one-to-many class="AbisSession" />
        </set>

        <subclass name="CourseGroup" discriminator-value="of" lazy="true">
            <property name="domain" column="codom" />
        </subclass>
    </class>
</hibernate-mapping>
```

Object identity

Distinction between

- object identity: a == b
- object equality: a.equals(b)
- database equality: same primary key(pk) in database a.getId().equals(b.getId())

Distinction between

- natural keys
- synthetic keys

Criteria to choose a primary key

- not null
- unique
- value never changes

Object identity - mapping

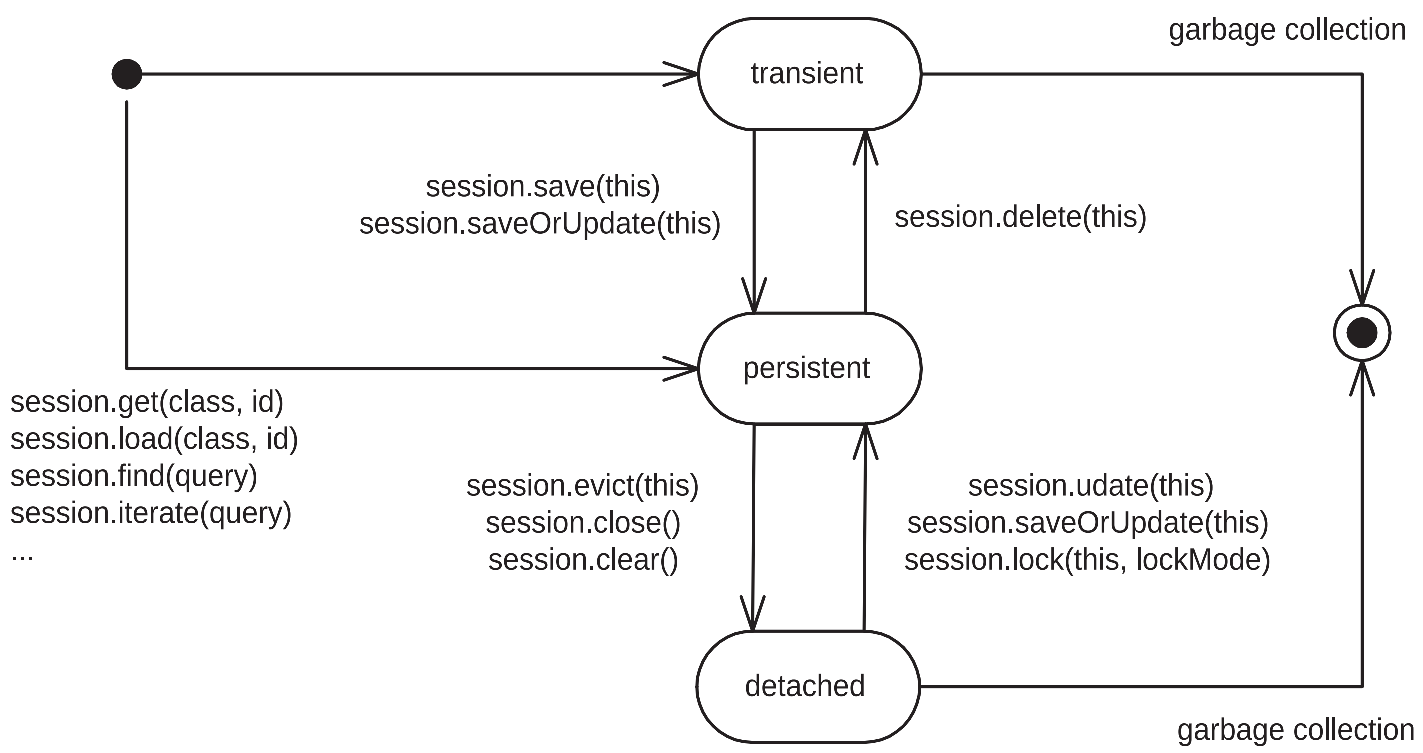
Several possibilities:

- Let database manage identity
- Let Hibernate manage identity
  - difficult if more applications run on same database
- Manage the identity in application
  - difficult if more applications run on same database

Behavioural aspects

- what is the lifecycle of persistent objects
- who is responsible for retrieving and storing objects
- how are transactions defined
- what about caching of objects
- what about lazy loading of objects
- how can we get information out of the database

Persistence lifecycle



Lifecycle states

Transient

- object created with new keyword
- not associated with database

Persistent

- associated with database (persistence manager - Session)
- has database identity (pk)
- transactional - synchronized with db at end of transaction
- Hibernate performs dirty checking

Detached

- when a persistent object is not associated with a session (close)
- can become “persistent” again

Object changes state through interaction with a Session-object

Persisting and retrieving objects

Three possibilities to select objects:

- Hibernate Query Language (HQL)
- Query By Criteria (QBC)
- Query By Example (QBE)

Other possibilities:

- report queries
  - relational in nature
  - used to export capabilities of RDBMS
- native sql
  - to optimize sql for a specific RDBMS system

Persisting and retrieving objects

Retrieve objects from the database with HQL

- Hibernate Query Language
- resembles Structured Query Language
- no ddl or dml

```
Query query = (Course) session.createQuery(“from Course c where c.title = :title” );
query.setString(“title”, “Hibernate”);
List result = query.list();
```

Persisting and retrieving objects

Retrieve objects from the database with QBC

- Query By Criteria
- more object-like
- no ddl or dml

```
Criteria criteria = session.createCriteria(Course.class);
criteria.add ( Expressions.like(“title”, “Hibernate”) );
List result = criteria.list();
```

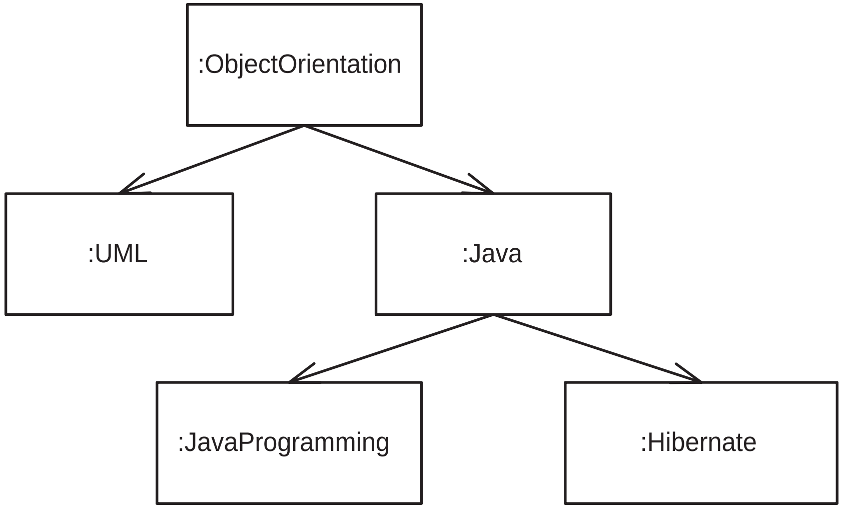
Persisting and retrieving objects

Retrieve objects from the database with QBE

- Query By Example
- not very powerful
- retrieves objects with matching properties
- no ddl or dml

```
Course course = new Course();
course.setTitle(“Hibernate”);
Criteria criteria = session.createCriteria(Course.class);
criteria.add ( Example.create(course));
List result = criteria.list();
```

Transitive persistence



Persistence by reachability

- direction of association is important
- by default Hibernate does navigate associations
- for each association, a cascade style can be specified

Fetching

Different styles of fetching:

- immediate fetching
  - linked objects fetched immediate together with parent
- lazy fetching
  - linked object fetched when link is navigated
- eager (outer join) fetching
  - linked objects fetched immediate together with parent
  - select-clause contains outer join-clause
- batch fetching
  - not strictly a fetching strategy
  - used to improve performance of lazy fetching

Transactions

Unit-Of-Work (unit-of-recovery)

- related activities
  - all successful executed
  - all failed
- ACID

Hibernate has its own transaction API

Hibernate uses underlying transaction mechanism:

- Java DataBase Connectivity (JDBC) in non-managed environment
- Java Transaction API (JTA) in managed environment

Transaction isolation 1.1

Isolation levels can be specified for transactions (cfr. JDBC):

- read uncommitted
- read committed
- repeatable read
- serializable

HQL even understands SELECT... FOR UPDATE

Caching 2

Idea: keep objects (data) close to application

Hibernate has two caching levels

First-level cache

- always available
- accessed through Session object
- objects synchronized with database on *flush()* or *commit()*

Second level cache

- process or cluster scope
- different caching strategies:
  - specifies isolation of objects in the second level cache
  - specifies synchronisation with database

Thank you

abis

TRAINING & CONSULTING

Geert Vandevenne
ABIS Training & Consulting
gvandevenne@abis.be