Object Relational Mapping
Java Persistence Layer

Ákos Horváth
István Ráth
Dániel Varró
Model Driven Software Development
Lecture 4
Introduction: Obj2Rel mapping

- **Goal:**
  - Persisted objects over RDBMS
  - Transparent handling of RDBMS from an OO programming language

- **Input:**
  - Class diagram

- **Output:**
  - Database schema
  - Query and manipulation operations are embedded into class methods

- Automated SQL code generation
Object Relational Mapping
Performance Optimization Tools

- **Object caching**
  - Decrease the number of direct RDBMS calls

- **Connection pooling**
  - Manage RDBMS connections for later usage

- **Transaction handling**
  - Definition of business level transactions
  - Hiding RDBMS level transaction (from programmers 😊)
Metamodel

Person
- name: String
- passwd: String

Player
- class: String

Organizer
- phone: String
- address: String

Championship
- id: String
- name: String
- minParticipant: int
- maxParticipant: int

participants
- *

organizer
- 1

champs
- *

organized
- *
Mapping Classes

- **General guidelines**
  - class ➔ table (relation)
  - attribute ➔ column (attribute)
  - (unique identifier) ➔ primary key

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>minP</th>
<th>maxP</th>
</tr>
</thead>
<tbody>
<tr>
<td>hu1</td>
<td>NB1</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>de1</td>
<td>BL</td>
<td>10</td>
<td>22</td>
</tr>
</tbody>
</table>
Attributes of generalization

- Completeness
  - Is there a person who is not a player or an organizer?
  - Partial vs. complete coverage

- Disjunction
  - Can a person be a player and an organizer at the same time? (multiple inheritance)
  - disjoint vs. overlapping classes

- Multiple mappings
Generalization I.

- Vertical mapping
  + No restrictions
- Steps of the Mapping
  - 1 class → 1 table
  - New column: supertype ID, which is a foreign key from the Supertype’s ID
• Add/remove
  – Foreign key constraints
• Query
  – JOIN
Generalization II.

- **Horizontal mapping**
  - Only for disjoint subclasses
  - Only for complete coverage

- **Steps of the Mapping**
  - 1 subclass $\Rightarrow$ 1 table
  - All attributes from the superclass and the subclass within the table
Generalization II. (cont.)

Person
- name:String
- passwd:String

Player
- class:String

Organizer
- id
- name
- passwd
- phone
- addr

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>passwd</th>
<th>phone</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>02</td>
<td>Nagy</td>
<td>edcba</td>
<td>1223</td>
<td>Ó u. 22.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>passwd</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>Szabó</td>
<td>abdce</td>
<td>M</td>
</tr>
</tbody>
</table>

- Simple add/remove operation
- Simple Querying using a Select
Filtered Mapping
- Only for disjoint subclasses
- Suboptimal storage usage, in case of large number of attributes

Steps of the Mapping
- Common table: 1-1 column for the attributes of the super- and the subclasses
- One additional column for the type information
Generalization III/a. (cont.)

- Simple add/remove operation
- Simple Querying using a Select with type based filtering
Generalization III/b.

- **Filtered Mapping**
  - For overlapping classes
  - Suboptimal storage usage, in case of large number of attributes

- **Steps of the Mapping**
  - Common table: 1-1 column for the attributes of the super- and the subclasses
  - Boolean type columns for indicating instance of relation
Generalization III/b. (cont.)

- Simple add/remove operation
- Simple Querying using a Select with type based filtering
Association 1..n (1..1)

Organizer

phone: String
address: String

1 organizer

Championship

id: String
name: String
minParticipant: int
maxParticipant: int

per_id | phone | address
--- | --- | ---
02 | 1223 | Ó u. 22.
04 | 3549 | Új u. 3.

org_id | id | name | minP | maxP
--- | --- | --- | --- | ---
04 | hu1 | NB1 | 6 | 18
02 | de1 | BL | 10 | 22

Additional Column and constraints
Association m..n

Player
- class:String
- *participants
  - champs

Championship
- id:String
- name:String
- minParticipant:int
- maxParticipant:int

Player_champ
- ch_id
- play_id

<table>
<thead>
<tr>
<th>ch_id</th>
<th>play_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>hu1</td>
<td>03</td>
</tr>
<tr>
<td>hu1</td>
<td>04</td>
</tr>
<tr>
<td>de1</td>
<td>04</td>
</tr>
</tbody>
</table>

Player
- per_id
- class

<table>
<thead>
<tr>
<th>per_id</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>03</td>
<td>M</td>
</tr>
<tr>
<td>04</td>
<td>GM</td>
</tr>
</tbody>
</table>

Championship
- id
- name
- minP
- maxP

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>minP</th>
<th>maxP</th>
</tr>
</thead>
<tbody>
<tr>
<td>hu1</td>
<td>NB1</td>
<td>6</td>
<td>18</td>
</tr>
<tr>
<td>de1</td>
<td>BL</td>
<td>10</td>
<td>22</td>
</tr>
</tbody>
</table>

New table and constraints
Java Persistence API
ORM frameworks

- Many players
  - ActiveObjects
    - Inheritance and annotations
  - Torque
    - Codegeneration from XML configurations
  - JPA
    - Annotations and/or XML
Java Persistence API

- Part of EJB 3 specification
- Hides RDBMS specific parts
- Provides a transparent runtime API for managing Objects that are persisted in an RDBMS
JPA providers

- JPA is only an API specification
- Various implementations
  - Hibernate
  - OpenJPA
  - Toplink
  - **EclipseLink** (official specification implementation)
Usage of JPA

- Java classes (POJO) with annotations
  - Alternate: directly from XML
    - Overwrites annotations
    - Only for Experts (do not use)
- Basic building block: Entity = persisted class
- All jar that contains a *persistence.xml* in its META-INF folder is a persisted module
- `javax.persistence` package
Defining an Entity

- Java class with `@Entity` (javax.persistence.Entity) annotated with default constructor
- Usually serializable (implements Serializable)
- Mandatory primary key attribute: `@Id`
  - Different ID generalization strategy can be defined in the strategy parameter
Attributes of an Entity

- The persisted attributes can only be managed using getters/setters (JavaBean convention)
- Non persisted (transient) attributes: @Transient
- Types of attributes
  - Primitive types:
    String, BigInteger, BigDecimal, java.util.Date, java.util.Calendar, java.sql.Date, java.sql.Time, java.sql.Timestamp, byte[], Byte[], char[], Character[]
  - Enum
  - Other entity, collection of other entities
  - Inner class
Parameters of the Mapping

- Default
  - the name of the columns and tables are identical of the name of the attributes’ and classes’ names, respectively.
- `@Table(name=“MyTable”)`
  - `@SecondaryTable(s)`: can be separated into multiple tables
- `@Column(name=“MyColumn”)`
- Other parameters for columns
  - nullable
  - unique
  - length
Generalization

- Supported from EJB3.0
- Supported modes:
  - One table for one class hierarchy → filtered mapping
  - Separate tables for subclasses with references → vertical mapping
  - One table for one concrete entity → horizontal mapping
JPA - Generalization

- **Filtered mapping**
  - Discriminator column defines the type
  - Requires nullable columns for subclass attributes
  - On the top of the hierarchy:
    - @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
    - @DiscriminatorColumn(name=<columnname>)
  - On all other classes:
    - @DiscriminatorValue(<value representing the type>)

- **Vertical mapping**
  - @Inheritance(strategy=InheritanceType.JOINED)

- **Horizontal mapping**
  - Not part of the EJB3.0 specification
Other Generalization Modes

- **Supertype as a non-entity**
  - `@MappedSuperClass`: attributes from the annotated class can be used in the subtypes. Will not have a dedicated table in the RDBMS, however, its attributes will be persisted.
  - Non marked will not be persisted

- **Abstract Entity**
  - Cannot be instantiated, but can be mapped to a table
  - Can be queried
Relations

- Based on multiplicity four different:
  - @OneToOne
  - @OneToMany
  - @ManyToOne
  - @ManyToMany

- Based on direction:
  - unidirectional
  - bidirectional (both entities will have getter/setter methods to manipulate the relation): mappedBy parameter

- Bidirectional OneToMany = Bidirectional ManyToMany

- A relation always has only one container entity
Example relation

- **Employee:**
  ```java
  @ManyToOne
  @JoinColumn(name="company_id")
  private Company company;
  ```

- **Company:**
  ```java
  @OneToMany
  mappedBy="company_id"
  private Collection<Employee> employees;
  ```

  + getters, setters

- Instead of the `@JoinColumn` the `@JoinTable` is used when a separate table is responsible for the relation (e.g., `ManyToMany`)

- The `@ManyToOne` relation is required to be defined on the container side! (does not have a `mappedBy` parameter)
Cascade type of Relations

- What to do with related entities?
  If you insert, update or delete an object, related objects are inserted?, updated? or deleted?

- Can be defined for any relations
  ```java
  @OneToMany(cascade=
  CascadeType.PERSIST, CascadeType.MERGE)
  ```

- Possible values:
  - PERSIST
  - MERGE
  - REMOVE
  - REFRESH
  - ALL

- Default: no cascade, everything have to be persisted by hand
What to do with relating entities when we load an entity? Load all entities on its relations?

Can be defined for all four relations e.g., `@OneToMany(fetch=FetchType.LAZY)`

**LAZY** : will not be loaded only if they are explicitly referred
- Does not consume memory but requires +1 select

**EAGER** (default): load all entities on its relations
- Faster but requires more memory

Fine tuning options:
- Set LAZY in general and only use EAGER when we know that we will use the entities from that particular relation.
  Use fetch join in the EJB-QL query, e.g.,

```
SELECT c from Customer c LEFT JOIN FETCH c.orders
```
Problems with Lazy fetch

- In case of detached state only those objects will be present that were used before.
- If we merge an entity back after a detached state then all relations (their target objects) that were not fetched will be deleted from the RDBMS.
- The Lazy is just an advice. The persistence provider may switch to Eager.
Persistence context

- The set of entities handled by the persistence provider
- Identification with the name of the persistence unit
- Getting the Entity manager e.g.: 

```java
EntityManagerFactory factory = Persistence.createEntityManagerFactory(
    PERSISTENCE_UNIT_NAME); //parameter in the persistence.xml

EntityManager entityManager = factory.createEntityManager();
```
Entity Manager

- Responsible for handling the entities

  Responsible:
  - Life-cycle of the entities
  - Synchronization with the RDBMS
  - Querying the entities
Transaction handling

- **Properties:**
  - Atomic
  - Consistent
  - Isolated
  - Durable

- **API call:**
  - `entityManager.getTransaction().begin()`
  - `entityManager.getTransaction().commit()`
  - `entityManager.getTransaction().rollback()`
Entity Life-cycle

- **new**: will be in this state when created using the `new` command, exists only in the memory. Will not be synchronized to the RDBMS.

- **managed**: the entity is present in the database and is part of a persistence context. Manipulations will be executed on the database side either at the end of the transaction or at an explicit flush() call.

- **detached**: the entity is present in the database but is **NOT** part of a persistence context. Similar like a DTO (Data Transfer Object)

- **removed**: part of the persistence context, however it is marked for deletion from the database
Entity Life-cycle

- new
  - persist()
  - refresh()
  - managed
    - remove()
      - persist()
      - merge()
      - Persistence context
        - ends
      - detached
  - removed
Entity life-cycle callbacks

- Annotations for callback methods
  - @PrePersist
  - @PostPersist
  - @PreRemove
  - @PostRemove
  - @PreUpdate
  - @PostUpdate
  - @PostLoad

- Persistence provider will execute the callbacks

- Can be defined in separate class
  - Binding using the @EntityListener
  - Its methods receive the entity as their input parameter
Database synchronization

- In general executed in all commit calls
- Can be explicitly executed using the Entity Manager:
  - `flush(entity)`: writes the manipulations to the RDBMS
  - `refresh(entity)`: Reads the changes from the RDBMS
Queries

- Simple query based on the primary key:
  \[
  \langle T \rangle \ T \ \text{find}(\text{Class}\langle T \rangle \ \text{entityClass}, \ \text{Object} \ \text{primaryKey})
  \]

- Complex queries:
  
  - **Java Persistence Query Language (JPQL, a.k.a. EJB-QL):**
    
    ```java
    public Query createQuery(String ejbqlString)
    
    • Example query:
      
      SELECT DISTINCT OBJECT(p) FROM Player p WHERE p.position = ?1 AND p.name = ?2
    ```

  - **SQL:**
    ```java
    public Query createNativeQuery(String sqlString)
    ```
Queries

- **Safe parameter handling:**
  - Based on name or index
    - `setParameter(String, Object)`
    - `setParameter(int, Object)`

- **Getting the result:**
  - `getSingleResult()`
  - `getResultList()`

- **Manipulation**
  - `executeUpdate()`
  - Can be executed in batch mode
Concurrency

- Two opportunities
  - **Optimistic**
    - Annotate an `int` or `TimeStamp` attribute with the `@Version` tag
    - Persistence provider increments this value at all commits on the entity
    - Throws `OptimisticLockException` if the value is higher in the RDBMS than the one in the memory.
  - **Explicit locks**
    - `entityManager.lock(Object entity, LockMode)`
    - `LockMode`: READ or WRITE
    - Can only be called within a transaction!
JPA 2.0
JPA 2.0 Features

- Richer mappings
- Richer JPQL
- Pessimistic Locking
- Criteria API
- Cache API
- Many more
JPA 2.0: Richer Mapping

• Supports collection of basic types and embeddables
  > In JPA 1.0, only collections of entities were supported
• Supports multiple levels of embeddables
• Embeddables containing collection of embeddables and basic types
• PrimaryKey can be derived entities
• More support for Maps...
@Entity
Public class Item {

    @ElementCollection
    private Set<String> tags;
}

@Entity
Public class Item {

    @ElementCollection
    @CollectionTable(name="TAGS")
    private Set<String> tags;
}
JPA 2.0: Richer JPQL

• Added entity type to support non-polymorphic queries
• Allow joins in subquery FROM clause
• Added new operators
  > INDEX (for ordered lists)
  > CASE (for case expressions)
  > more
• Added new reserved words
  > ABS, BOTH, CONCAT, ELSE, END, ESCAPE, LEADING, LENGTH, LOCATE, SET, SIZE, SQRT, SUBSTRING, TRAILING
Example: JPQL CASE Expression

```java
@Entity public class Employee {
    @Id Integer empId;
    String name;
    Float salary;
    Integer rating;
    // ...
}

UPDATE Employee e
SET e.salary =
    CASE WHEN e.rating = 1 THEN e.salary * 1.05
         WHEN e.rating = 2 THEN e.salary * 1.02
         ELSE e.salary * 0.95
    END
```
JPA 2.0: Locking Enhancements

- JPA 1.0 supports only optimist locking
- JPA 2.0 adds pessimistic locking
- Multiple places to specify lock
  - read and lock
  - read then lock
  - read then lock and refresh

```java
public enum LockModeType {
    OPTIMISTIC,
    OPTIMISTIC_FORCE_INCREMENT,
    PESSIMISTIC,
    PESSIMISTIC_FORCE_INCREMENT,
    NONE
}
```
JPA 2.0: Criteria API

- Strongly typed criteria API
- Object-based query definition objects
  > rather than string-based
- Like JPQL
- Uses a metamodel – Compile time type checking using Generics
  > Each entity X has a metamodel class X_
  > Criteria API operates on the metamodel
JPA 2.0: Caching

• Supports the use of a second-level cache

• Cache API
  > contain(Class, PK)
  > evict(Class, PK), evict(Class)
  > evictAll()

• @Cacheable annotation on entities
References

- Mike Calvo: JPA and Hibernate
  - [http://www.slideshare.net/adorepump/jpa-and-hibernate-presentation](http://www.slideshare.net/adorepump/jpa-and-hibernate-presentation)

- Gordon Yorke: EclipseLink JPA
  - [http://www.slideshare.net/pelegri/eclipselink-jpa-presentation](http://www.slideshare.net/pelegri/eclipselink-jpa-presentation)

- Markus Eisele: New features of JSR-317