

# DEVELOPMENT OF INFORMATION SYSTEMS

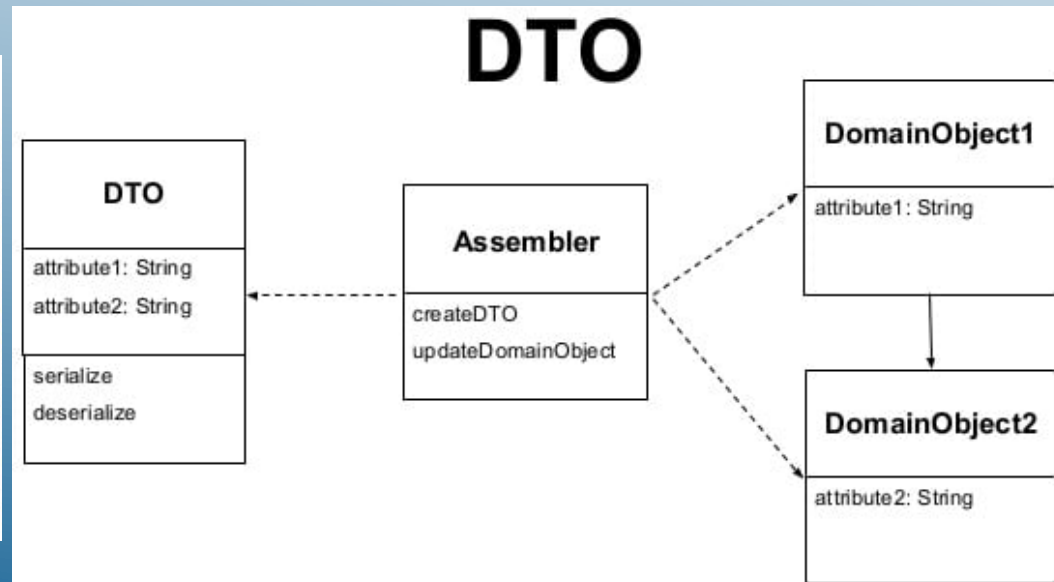
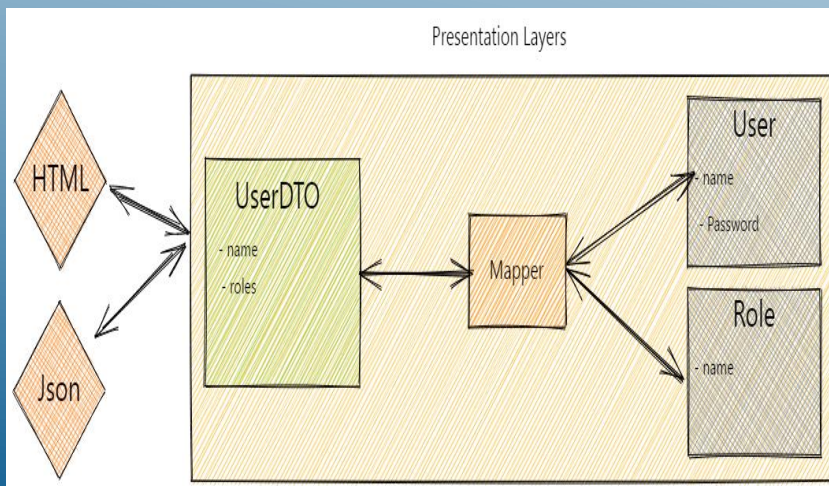
## Lecture 6

# Review of the last lecture

## Data Transfer Objects

An object that carries data between processes in to reduce the number of method calls.


**DTO are flat data structures that contain no business logic**, only storage, accessors, and eventually methods related to serialization or parsing. The data is mapped from the domain models to the DTOs, normally through a mapper component in the presentation or facade layer



# **Review of the last lecture**

## **Object-Relational Behavioral Patterns**


These patterns are designed to support and reduce problems like:

- managing of transactions
  - managing of in-memory objects
  - preserve data integrity
- 
- A decorative graphic consisting of several parallel white lines of varying lengths, slanted diagonally upwards from left to right, located in the bottom right corner of the slide.

# Review of the last lecture

## Object-Relational Behavioral Patterns


Unit of Work  
Identity Map  
Lazy Load

A decorative graphic consisting of several parallel white lines of varying lengths, slanted upwards from left to right, located in the bottom right corner of the slide.

# Review of the last lecture

## Unit of Work

Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

Several thin, white, parallel diagonal lines are located in the bottom right corner of the slide, extending from the right edge towards the bottom.

# Review of the last lecture

## Unit of Work - when to use?

### Pros:

- If we can postpone saving objects into the DB
- Objects are changing frequently.

### Cons:

- Critical systems

```
class UnitOfWork...

    public void registerNew(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
        Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
        Assert.isTrue("object not removed", !removedObjects.contains(obj));
        Assert.isTrue("object not already registered new", !newObjects.contains(obj));
        newObjects.add(obj);
    }


    public void registerDirty(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
        Assert.isTrue("object not removed", !removedObjects.contains(obj));
        if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
            dirtyObjects.add(obj);
        }
    }

    public void registerRemoved(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
        if (newObjects.remove(obj)) return;
        dirtyObjects.remove(obj);
        if (!removedObjects.contains(obj)) {
            removedObjects.add(obj);
        }
    }
}
```

# Review of the last lecture

## Identity Map

Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.

Several thin, parallel white lines are drawn diagonally across the bottom right corner of the slide, extending from the right edge towards the bottom left.



# Review of the last lecture

## Identity Map – when to use

### Pros :

- If we don't want two objects to represent the same record in DB.

### Cons:

- When objects do not change their state.


```
private Map people = new HashMap();  
public static void addPerson(Person arg) {  
    soleInstance.people.put(arg.getID(), arg);  
}  
public static Person getPerson(Long key) {  
    return (Person) soleInstance.people.get(key);  
}  
public static Person getPerson(long key) {  
    return getPerson(new Long(key));  
}
```



# Review of the last lecture

## Lazy Load

An object that doesn't contain all of the data you need but knows how to get it.

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the bottom left.

# Review of the last lecture

## Lazy Load – when to use?

### Pros:

- When we need an extra call to get an object from DB.

### Cons:

- When objects are not in complex compositions.

```
class Supplier...
```

```
    public List getProducts() {  
        if (products == null) products = Product.findForSupplier(getID());  
        return products;  
    }
```

# Object-Relational Structural Patterns



# Object-Relational Structural Patterns

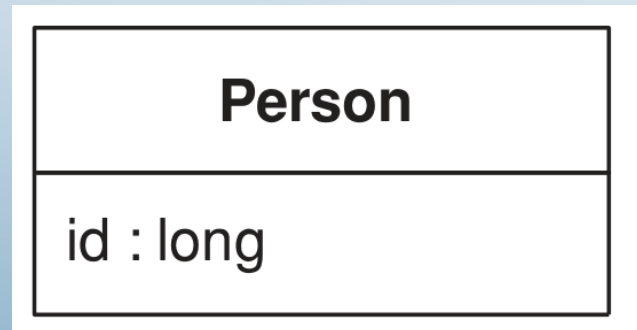
These patterns deal with the structural differences between in memory objects and database tables/rows.

The patterns include:

- Identity Field
  - Foreign Key Mapping
  - Association Table Mapping
  - Dependent Mapping
  - Embedded Value
  - Serialized LOB
- 

# Identity Field

Saves a database ID field in an object to maintain identity between an in-memory object and a database row.



In essence, Identity Field is mind-numbingly simple. All you do is store the primary key of the relational database table in the object's fields.

# Identity Field – how it works?

Although the basic notion of Identity Field is very simple, there are oodles of complicated issues that come up.

**Choosing Key** - often dealing with an existing database that already has its key structures in place.

- **A meaningful key** – ID card number for identifying a person.
- **A meaningless key** – Autoincrement sequential number, GUID

# Identity Field – how it works?

The most common operation you'll do with a key is equality checking, so you want a type with a fast equality operation.

- **A simple key** uses only one database field. It can be used use the same code for all key manipulation.
- **A compound key** uses more than one. Carry a bit of meaning, so be careful about the uniqueness and particularly the immutability rule with them. (A good example is orders and line items, where a good key for the line item is a compound of the order number and a sequence number makes a good key for a line item)




# Identity Field – how it works?

You have to choose the type of the key

- A long integer type is often the best choice.
- Strings can also work, but equality checking may be slower and incrementing strings is a bit more painful.
- Beware about using dates or times in keys. Not only are they meaningful, they also lead to problems with portability and consistency.
- **Table unique key** - is unique across the table
- **Database unique key** - is unique across every row in every table in the database

# Identity Field – how it works?

## Representing the Identity Field in an Object

- The simplest form of Identity Field is a field that matches the type of the key in the database.
  - Compound key – The best bet with them is to make a key class.
- 
- A decorative graphic consisting of several parallel white lines of varying lengths, slanted diagonally upwards from left to right, located in the bottom right corner of the slide.

# Identity Field – how it works?

## Getting a New Key

- **Database auto-generate** - auto-generated field or database counter.
- **GUID** (Globally Unique Identifier) - is a number generated on one machine that's guaranteed to be unique across all machines in space and time. API functions to get it.
- **Generate your own** - A simple staple for small systems is to use a table scan using the SQL max function. (read-locks whole table). Better approach is separate key table.

# Identity field - when to use it

Use Identity Field when there's a mapping between objects in memory and rows in a database. This is usually when you use:

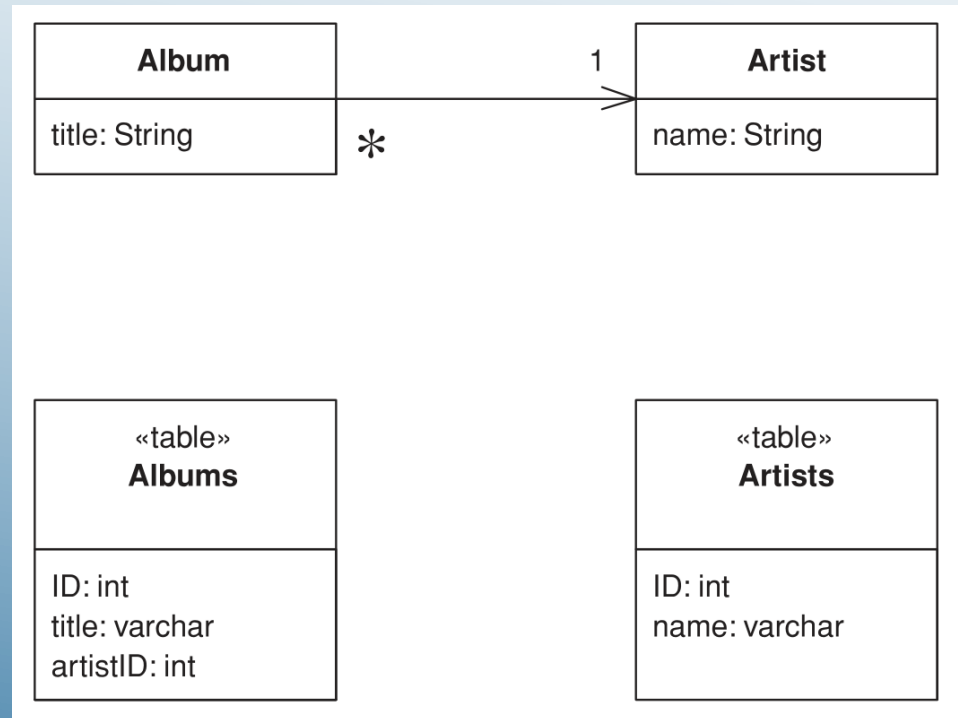
- Domain Model
- Row Data Gateway

You don't need this mapping if you're using Transaction Script, Table Module, or Table Data Gateway.

Several thin, white, parallel diagonal lines are positioned in the bottom right corner of the slide, extending from the right edge towards the bottom.

# Foreign Key Mapping

Maps an association between objects to a foreign key reference between tables.



A Foreign Key Mapping maps an object reference to a foreign key in the database.

# Foreign Key Mapping – how it works?

The obvious key to this problem is Identity Field.

Each object contains the database key from the appropriate database table. If two objects are linked together with an association, this association can be replaced by a foreign key in the database.

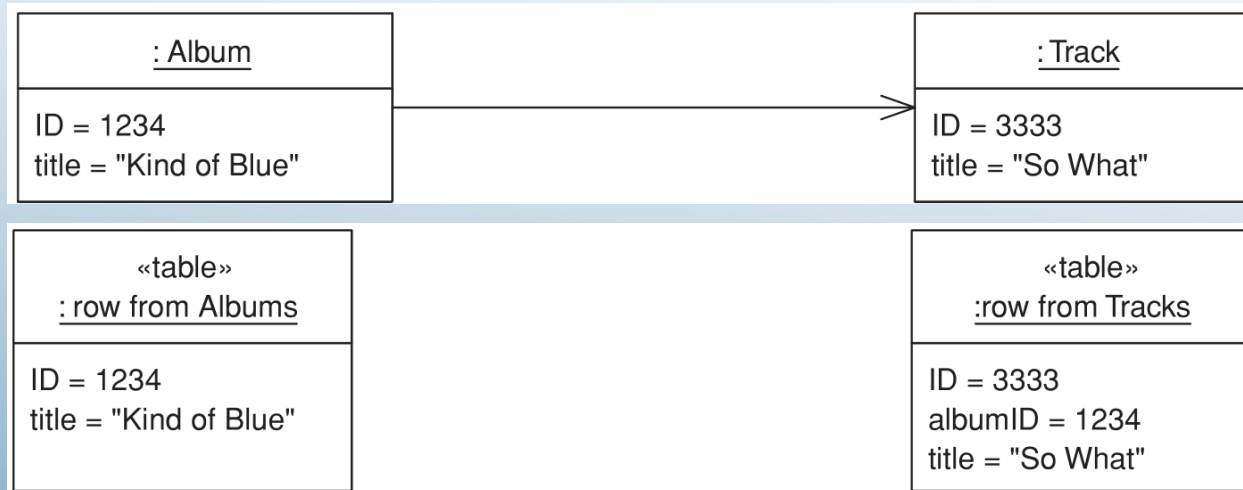
An Example:

Put simply, when you save an album to the database, you save the ID of the artist that the album is linked to in the album record.



# Foreign Key Mapping – how it works?

A more complicated case turns up when you have a collection of objects.



You can't save a collection in the database, so you have to reverse the direction of the reference. Thus, if you have a collection of tracks in the album, you put the foreign key of the album in the track record.


Update is complicated – delete and insert or add and backpoint or diff the collection.



# Foreign Key Mapping – when to use it?

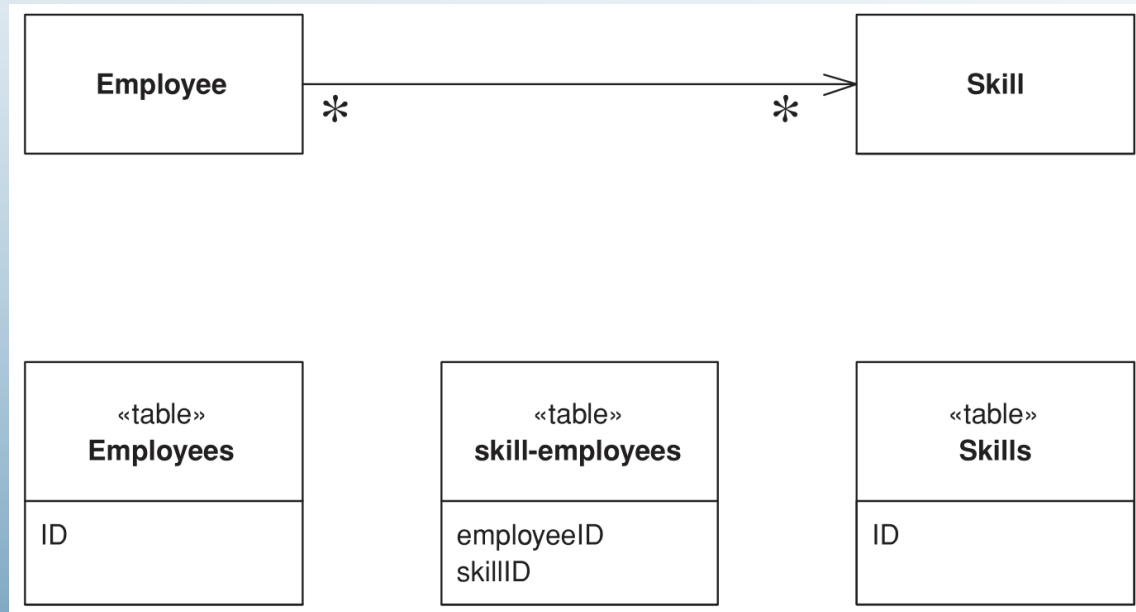
A Foreign Key Mapping can be used for almost all associations between classes.

The most common case where it isn't possible is with many-to-many associations. Foreign keys are single values, and first normal form means that you can't store multiple foreign keys in a single field. Instead you need to use *Association Table Mapping*.

Several white lines of varying lengths and angles are drawn in the bottom right corner of the slide, creating a modern, abstract graphic element.

# Association Table Mapping

Saves an association as a table with foreign keys to the tables that are linked by the association.



To solve a many-to-many association there exists the classic resolution that's been used by relational data people for decades: create an extra table to record the relationship. Then use Association Table Mapping to map the multivalued field to this link table.

# Association Table Mapping – how it works?

The basic idea behind Association Table Mapping is using a link table to store the association.

- Table has only the foreign key IDs for the two tables that are linked together
- It has one row for each pair of associated objects.
- The link table has no corresponding in-memory object. As a result it has no ID.
- Its primary key is the compound of the two primary keys of the tables that are associated.

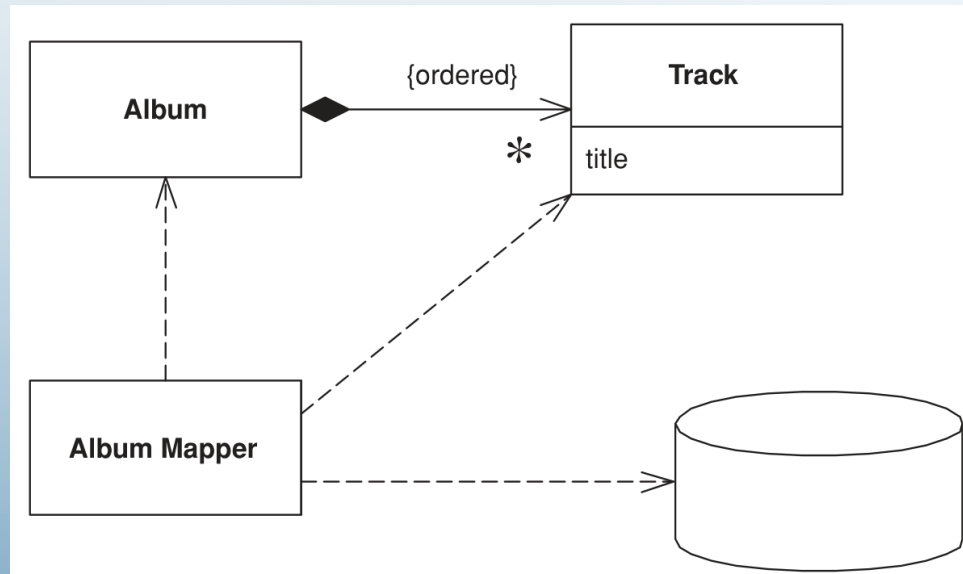
# Association Table Mapping – when to use it?

The canonical case for Association Table Mapping is a many-to-many association, since there are really no any alternatives for that situation.

- In case of need to link two existing tables, but you aren't able to add columns to those tables. In this case you can make a new table and use Association Table Mapping.
- Sometimes an existing DB schema uses an associative table, even when it isn't really necessary. In this case it's often easier to use Association Table Mapping than to simplify the database schema.

# Dependent Mapping

Has one class perform the database mapping for a child class.




Tracks on an album may be loaded or saved whenever the underlying album is loaded or saved. If they aren't referenced to by any other table in the database, you can simplify the mapping procedure by having the album mapper perform the mapping for the tracks as well-treating this mapping as a dependent mapping.

# Dependent Mapping – how it works?

The basic idea behind Dependent Mapping is that one class (the **dependent**) relies upon some other class (the **owner**) for its database persistence.

- Each dependent can have only one owner and must have one owner. This manifests itself in terms of the classes that do the mapping.
- For Active Record and Row Data Gateway, the dependent class won't contain any database mapping code; its mapping code sits in the owner.
- With Data Mapper there's no mapper for the dependent, the mapping code sits in the mapper for the owner.
- In a Table Data Gateway there will typically be no dependent class at all, all the handling of the dependent is done in the owner.

# Dependent Mapping – how it works?

- A dependent may itself be the owner of another dependent.
  - In this case the owner of the first dependent is also responsible for the persistence of the second dependent.
  - You can have a whole hierarchy of dependents controlled by a single primary owner.
  - Using Dependent Mapping complicates tracking whether the owner has changed.
- 



# Dependent Mapping – when to use it?

- Use it when you have an object that's only referred to by one other object, which usually occurs when one object has a collection of dependents.
- Situation where the owner has a collection of references to its dependents but there's no back pointer

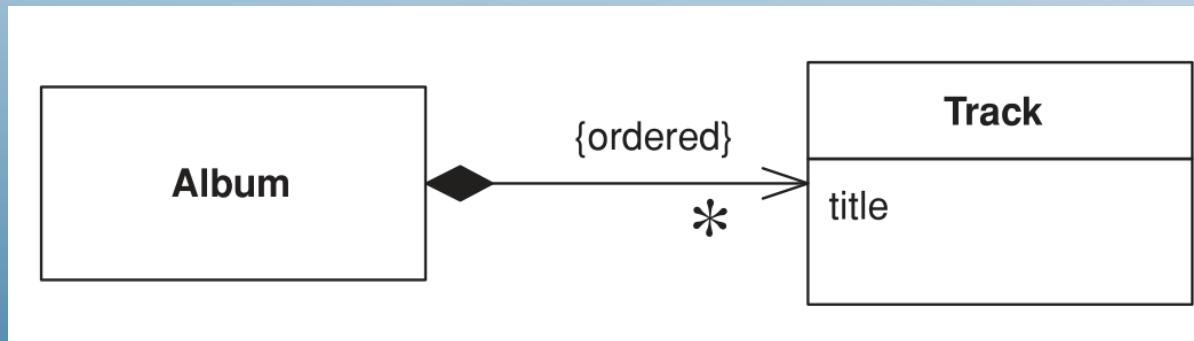
## Preconditions:

- A dependent must have exactly one owner.
- There must be no references from any object other than the owner to the dependent.
- Supposed to use it with Domain Model but avoid large graph of dependents.
- Don't recommend if you're using Unit of Work.

# Dependent Mapping – implementation

An album holds a collection of tracks. This uselessly simple application doesn't need anything else to refer to a track, so it's an obvious candidate for Dependent Mapping. (Just for example purpose)

An album with tracks that can be handled using Dependent Mapping.



This track just has a title. We can define it as an immutable class

# Dependent Mapping – implementation

```
class Track...
```

```
    private final String title;
    public Track(String title) {
        this.title = title;
    }
    public String getTitle() {
        return title;
    }
}
```

```
class Album...
```

```
    private List tracks = new ArrayList();
    public void addTrack(Track arg) {
        tracks.add(arg);
    }
    public void removeTrack(Track arg) {
        tracks.remove(arg);
    };
    public void removeTrack(int i) {
        tracks.remove(i);
    }
    public Track[] getTracks() {
        return (Track[]) tracks.toArray(new Track[tracks.size()]);
    }
}
```

# Dependent Mapping – implementation

```
class AlbumMapper...

    protected String findStatement() {
        return
            "SELECT ID, a.title, t.title as trackTitle" +
            "  FROM albums a, tracks t" +
            " WHERE a.ID = ? AND t.albumID = a.ID" +
            " ORDER BY t.seq";
    }

    protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
        String title = rs.getString(2);
        Album result = new Album(id, title);
        loadTracks(result, rs);
        return result;
    }

    public void loadTracks(Album arg, ResultSet rs) throws SQLException {
        arg.addTrack(newTrack(rs));
        while (rs.next()) {
            arg.addTrack(newTrack(rs));
        }
    }

    private Track newTrack(ResultSet rs) throws SQLException {
        String title = rs.getString(3);
        Track newTrack = new Track (title);
        return newTrack;
    }
}
```

# Dependent Mapping – implementation

```
class AlbumMapper...
```

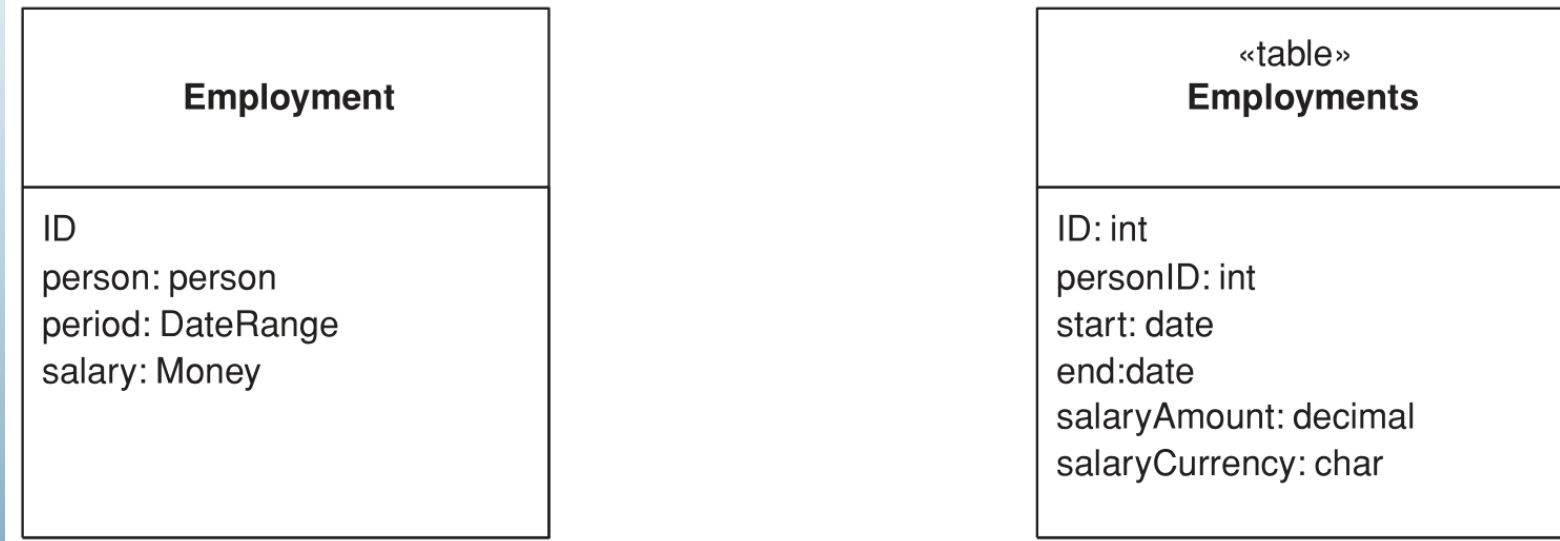
```
public void update(DomainObject arg) {
    PreparedStatement updateStatement = null;
    try {
        updateStatement = DB.prepare("UPDATE albums SET title = ? WHERE id = ?");
        updateStatement.setLong(2, arg.getID().longValue());
        Album album = (Album) arg;
        updateStatement.setString(1, album.getTitle());
        updateStatement.execute();
        updateTracks(album);
    } catch (SQLException e) {
        throw new ApplicationException(e);
    } finally {DB.cleanup(updateStatement);
    }
}

public void updateTracks(Album arg) throws SQLException {
    PreparedStatement deleteTracksStatement = null;
    try {
        deleteTracksStatement = DB.prepare("DELETE from tracks WHERE albumID = ?");
        deleteTracksStatement.setLong(1, arg.getID().longValue());
        deleteTracksStatement.execute();
        for (int i = 0; i < arg.getTracks().length; i++) {
            Track track = arg.getTracks()[i];
            insertTrack(track, i + 1, arg);
        }
    } finally {DB.cleanup(deleteTracksStatement);
    }
}

public void insertTrack(Track track, int seq, Album album) throws SQLException {
    PreparedStatement insertTracksStatement = null;
```

# Embedded Value


Maps an object into several fields of another object's table.



An Embedded Value maps the values of an object to fields in the record of the object's owner.

In this example Field period of the Employment class is mapped into database table fields start and end. Field salary into fields salaryAmount and salaryCurrency.

# Embedded Value – how it works?

- When the owning object (employment) is loaded or saved, the dependent objects (date range and money) are loaded and saved at the same time.
  - The dependent classes won't have their own persistence methods since all persistence is done by the owner.
  - You can think of Embedded Value as a special case of Dependent Mapping, where the value is a single dependent object.
- 
- Three parallel white diagonal lines are located in the bottom right corner of the slide, extending from the middle of the right edge towards the bottom left.

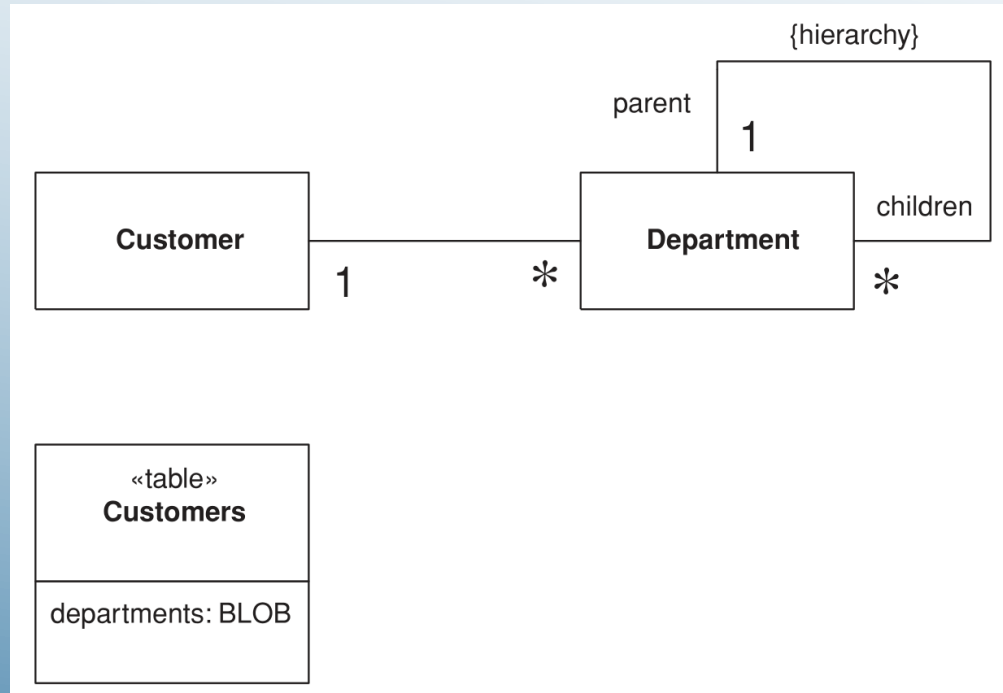


# Embedded Value – when to use it?

- The simplest cases for Embedded Value are the clear, simple Value Objects like money and date range.
- Since Value Objects don't have identity, you can create and destroy them easily without worrying about such things as Identity Maps to keep them all in sync.
- All Value Objects should be persisted as Embedded Value, since you would never want a table for them there.
- Mapping to an existing DB schema, you can use Embedded Value when a table contains data that you split into more than one object in memory.

# Serialized LOB

Saves a graph of objects by serializing them into a single large object (LOB), which it stores in a database field.



Objects don't have to be persisted as table rows related to each other. Another form of persistence is serialization, where a whole graph of objects is written out as a single large object (LOB).

# Serialized LOB – how it works?

There are two ways you can do the serialization:

- as a binary (BLOB)
- as textual characters (CLOB).

The BLOB is often the simplest to create since many platforms include the ability to automatically serialize an object graph.

Saving the graph is a simple matter of applying the serialization in a buffer and saving that buffer in the relevant field.

# Serialized LOB – how it works?

- **BLOB** - it's simple to program (if your platform supports it) and that it uses the minimum of space. The disadvantages are that your database must support a binary data type for it and that you can't reconstruct the graph without the object, so the field is utterly impenetrable to casual viewing. The most serious problem, however, is versioning.
- **CLOB** - you serialize the graph of objects into a text string that carries all the information you need. The text string can be read easily by a human viewing the row, which helps in casual browsing of the database. However the text approach will usually need more space, and you may need to create your own parser for the textual format you use. It's also likely to be slower than a binary serialization.

# Serialized LOB – how it works?

- When you use Serialized LOB beware of identity problems. Example: you want to use Serialized LOB for the customer details on an order. For this don't put the customer LOB in the order table; otherwise, the customer data will be copied on every order, which makes updates a problem.
- Be careful of duplicating data when using this pattern. Mostly the some part of data that overlaps with another one. So pay careful attention to the data that's stored in the Serialized LOB and be sure that it can't be reached from anywhere but a single object that acts as the owner of the Serialized LOB.

# Serialized LOB – when to use it?

- This pattern works best when you can chop out a piece of the object model and use it to represent the LOB. Think of a LOB as a way to take a bunch of objects that aren't likely to be queried from any SQL route outside the application. This graph can then be hooked into the SQL schema.
- Serialized LOB works poorly when you have objects outside the LOB reference objects buried in it. To handle this you have to come up with some form of referencing scheme that can support references to objects inside a LOB - it's by no means impossible, but it's awkward, awkward enough usually not to be worth doing

# Semestral project -Artifact 5

## Domain model design

- Static class diagram (data + methods), association types, inheritance.
- Patterns used in domain model.
- It is possible to separate in diagrams the pure domain design from the design with patterns. Separate layers PL, BL, DAL for better readability.
- Sequence diagrams for key operations (especially object cooperation within patterns).



# Exercise tasks

Continuing of the implementation task from the previous exercise using the patterns from the lecture

- Presentation of prepared user interface sketches.
- Discussion of the domain model with a focus on the patterns used and interaction of objects within the patterns.
- Implementation of object-relational mapping patterns for semester assignment.



# Lecture checking questions

- In what situations would you use the Identity field pattern and why? And when not? Write a fragment of code that shows why you used this pattern.
- What is the difference between the Foreign key mapping and Associate table mapping patterns? Write two fragments of code that show that you used these patterns.
- How does the Dependent mapping pattern differ from the Data mapper pattern? When is it appropriate to use it?
- Write a code snippet that shows that you have used the Dependent mapping pattern. Come up with at least two examples suitable for using the Embedded value pattern. What is the essence of this pattern? Why and when is it appropriate to use it?
- Come up with at least two examples suitable for using the Serialized LOB pattern. What is the essence of this pattern? Why and when is it appropriate to use it?