SOEN 387: Web-Based Enterprise Application Design

Chapter 12. Object-Relational Structural Patterns

Identity Field

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

Relational databases tell one row from another by using key—in particular, the primary key.

However, in-memory objects don't need such a key, as the object system ensures the correct identity under the covers .

Reading data from a database is all very well, but in order to write data back you need to tie the database to the in-memory object system.

Choosing Your Key The first issue is what kind of key to choose in your database.

The first concern is whether to use meaningful or meaningless keys.

A **meaningful key** is something like the. Social Insurance Number for identifying a person.

A **meaningless key** is essentially a random number that is not intended for human use.

The danger with a meaningful key is that, while in theory they make good keys, in practice they do not. To work at all, keys need to be unique; to work well, they need to be immutable. While assigned numbers are supposed to be unique and immutable, human error often makes them neither.

The database should detect the uniqueness problem, but it can only do that after my record goes into the system, and of course that might not happen until after the mistake. As a result, meaningful keys should be distrusted.

A **simple key** uses only one database field; a **compound key** uses more than one.

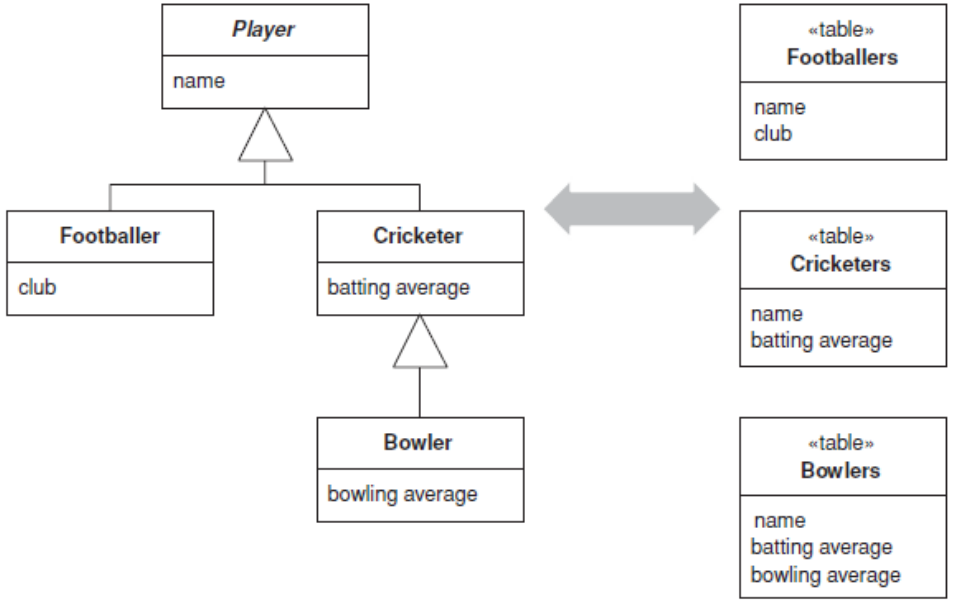
You can have keys that are unique to the table or unique database-wide.

A **table-unique key** is unique across the table, which is what you need for a key in any case.

A **database-unique key** is unique across every row in every table in the database.

A table-unique key is usually fine, but a database-unique key is often easier to do and allows you to use a single *Identity Map*

Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.



Concrete Table Inheritance, where there is a table for each concrete class in the inheritance hierarchy.

Getting a New Key To create an object, you'll need a key. This sounds like a simple matter, but it can often be quite a problem. You have three basic choices:
get the database to **auto-generate**, use a **GUID**, or generate **your own**.

The **auto-generate** route should be the easiest. Each time you insert data for the database, the database generates a unique primary key without you having to do anything. But not all databases do this the same way. Many that do, handle it in such a way that causes problems for object-relational mapping.

The most common auto-generation method is declaring one **auto-generated field**, which, whenever you insert a row, is incremented to a new value.

The problem with this scheme is that you can't easily determine what value got generated as the key. If you want to insert an order and several line items, you need the key of the new order so you can put the value in the line item's foreign key.

Example: MySQL Auto-Increment

In MySQL , the `AUTO_INCREMENT` attribute can be used to generate a unique identity for new rows:

```
CREATE TABLE animals (
id MEDIUMINT NOT NULL AUTO_INCREMENT, name CHAR(30) NOT NULL, PRIMARY KEY (id)
);

INSERT INTO animals (name) VALUES ('dog'),('cat'),('penguin'), ('lax'),('whale'),('ostrich');

SELECT * FROM animals;
```

id	name
1	dog
2	cat
3	penguin
4	lax
5	whale
6	ostrich

```
INSERT INTO animals (id,name) VALUES(0,'groundhog');

INSERT INTO animals (id,name) VALUES(NULL,'squirrel');

INSERT INTO animals (id,name) VALUES(100,'rabbit'); INSERT

INTO animals (id,name) VALUES(NULL,'mouse');

SELECT * FROM animals;
```

id	name
1	dog
2	cat
3	penguin
4	lax
5	whale
6	ostrich
7	groundhog
8	squirrel
100	rabbit
101	mouse

A **GUID** (Globally Unique IDentifier) is a number generated on one machine that's guaranteed to be unique across all machines in space and time.

Often platforms give you the API to generate a GUID (or UUID) .

All that matters is that the resulting number is completely unique and thus a safe key.

The only disadvantage to a GUID is that the resulting key string is big, and that can be an equally big problem. There are always times when someone needs to type in a key to a window or SQL expression, and long keys are hard both to type and to read.

Example: MySQL UUID

```
SELECT UUID();
```

Output:

```
1f7cf749-5fde-11ed-99a8-ecf4bb26a039
```

Consider the below ‘Persons’ table.

ID	FirstName	MiddleName	LastName	Mobile	Work	HomePhone	City	DOB
1	Rahul	NULL	Khanna	9866552233	NULL	2045887896	Pune	1999-03-16
2	Devika	P	Kulkarni	4577122256	NULL	NULL	Navi Mumbai	1999-09-10
3	Aaliyah	Ishrat	Shaikh	NULL	2445996560	NULL	Chennai	1999-01-05
4	Park	Lee	NULL	NULL	NULL	NULL	Seoul	2000-08-25

ALTER TABLE Persons **ADD** UID text;
UPDATE Persons **SET** UID=**UUID**();
SELECT * **FROM** Persons;

ID	FirstName	MiddleName	LastName	Mobile	Work	HomePhone	City	DOB	UID
1	Rahul	NULL	Khanna	9866552233	NULL	2045887896	Pune	1999-03-16	de879101-a2ac-11eb-93d3-54e1ad0a7440
2	Devika	P	Kulkarni	4577122256	NULL	NULL	Navi Mumbai	1999-09-10	de879391-a2ac-11eb-93d3-54e1ad0a7440
3	Aaliyah	Ishrat	Shaikh	NULL	2445996560	NULL	Chennai	1999-01-05	de87942f-a2ac-11eb-93d3-54e1ad0a7440
4	Park	Lee	NULL	NULL	NULL	NULL	Seoul	2000-08-25	de8794b4-a2ac-11eb-93d3-54e1ad0a7440

UPDATE Persons **SET** UID=**UUID**() **WHERE ID**=2;
SELECT * **FROM** Persons;

ID	FirstName	MiddleName	LastName	Mobile	Work	HomePhone	City	DOB	UID
1	Rahul	NULL	Khanna	9866552233	NULL	2045887896	Pune	1999-03-16	de879101-a2ac-11eb-93d3-54e1ad0a7440
2	Devika	P	Kulkarni	4577122256	NULL	NULL	Navi Mumbai	1999-09-10	84b5bca6-a2af-11eb-93d3-54e1ad0a7440
3	Aaliyah	Ishrat	Shaikh	NULL	2445996560	NULL	Chennai	1999-01-05	de87942f-a2ac-11eb-93d3-54e1ad0a7440
4	Park	Lee	NULL	NULL	NULL	NULL	Seoul	2000-08-25	de8794b4-a2ac-11eb-93d3-54e1ad0a7440

See: https://mysqlcode.com/mysql-uuid/ for more details on MySQL UUID

The last option is generate **your own**. A simple staple for small systems is to use a **table scan** using the SQL max function to find the largest key in the table and then add one to use it.

But this read-locks the entire table while you're doing it, which means that it works fine if inserts are rare, but your performance will be toasted if you have inserts running concurrently with updates on the same table.

You also have to ensure you have complete isolation between transactions; otherwise, you can end up with multiple transactions getting the same ID value.

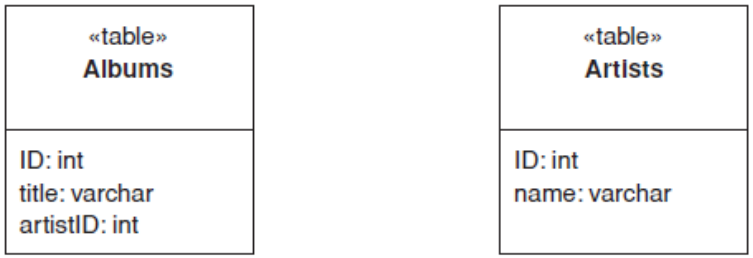
A better approach is to use a separate **key table**. This table is typically one with two columns: name and next available value.

If you use database-unique keys, you'll have just one row in this table. If you use table-unique keys, you'll have one row for each table in the database.

To use the key table, all you need to do is read that one row and note the number, increment the number and write it back to the row. This cuts down on expensive database calls and reduces contention on the key table.

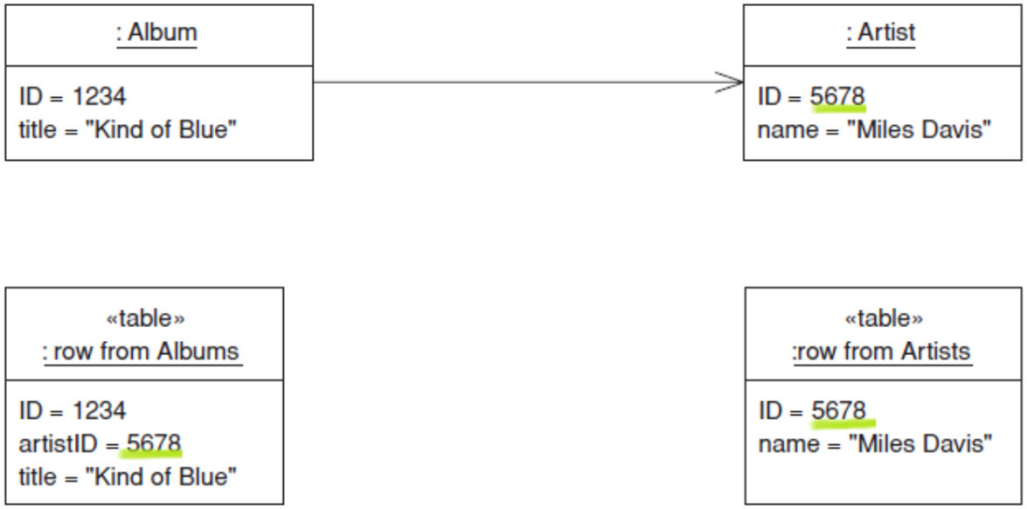
Foreign Key Mapping

Maps an association between objects to a [foreign key](#) reference between tables.



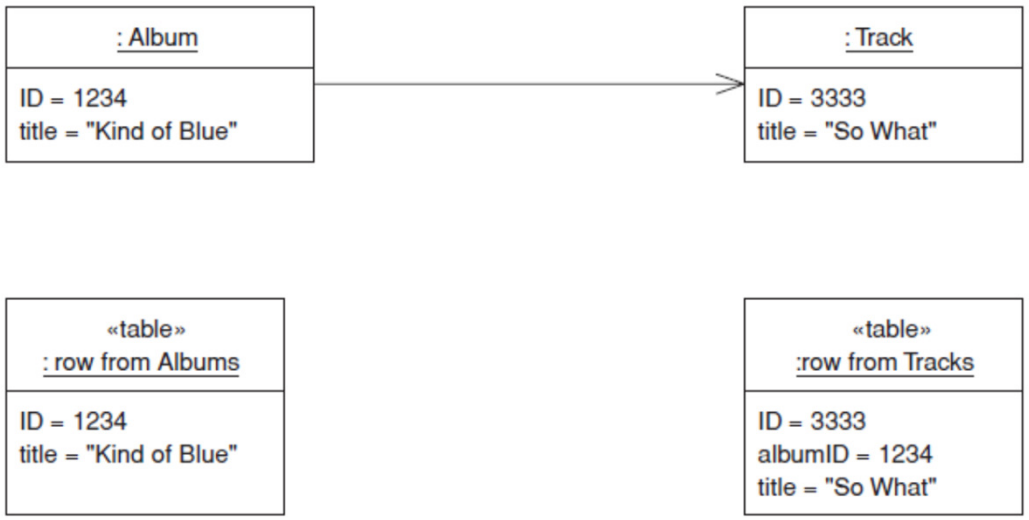
Each object contains the database key from the appropriate database table.

If two objects are linked together with an association, this association can be replaced by a foreign key in the database.
Example, 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, as in Figure below.

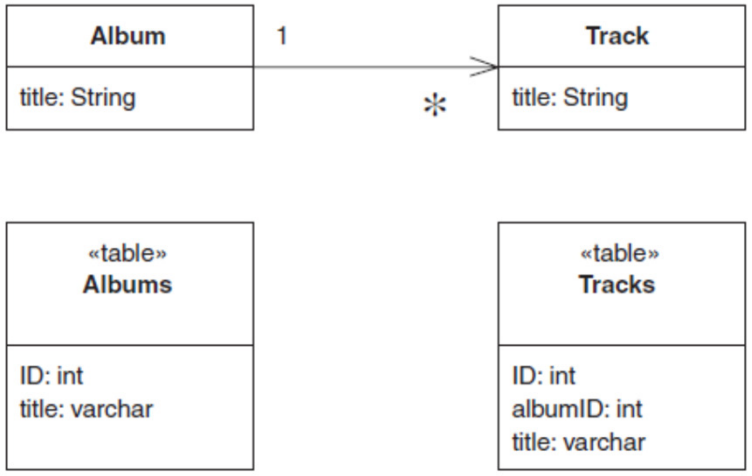


Mapping a collection to a foreign key.

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, as in Figures below.



Mapping a collection to a foreign key.



Classes and tables for a multivalued reference.

The complication occurs when you have an update. Updating implies that tracks can be added to and removed from the collection within an album. How can you tell what alterations to put in the database? Essentially you have three options: (1) **delete and insert**, (2) add a **back pointer**, and (3) **diff** the collection.

With **delete and insert** you delete all the tracks in the database that link to the album, and then insert all the ones currently on the album. At first glance this sounds pretty appalling, especially if you haven't changed any tracks. But the logic is easy to implement and as such it works pretty well compared to the alternatives. The drawback is that you can only do this if tracks are Dependent Mappings , which means they must be owned by the album and can't be referred to outside it.

Adding a **back pointer** puts a link from the track back to the album, effectively making the association bidirectional. This changes the object model, but now you can handle the update using the simple technique for single-valued fields on the other side.

You can do a **diff** with the current state of the database. Diffing with the database involves rereading the collection back from the database and then comparing the collection you read with the collection in the album. Anything in the database that isn't in the album was clearly removed; anything in the album that isn't on the disk is clearly a new item to be added.

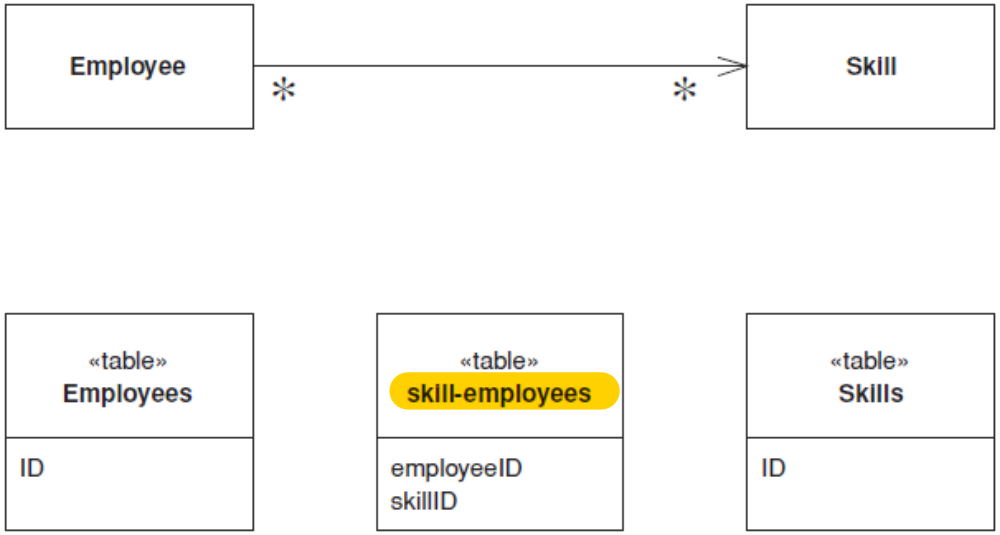
Association Table Mapping

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

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

This 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.

Sometimes you may 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*.



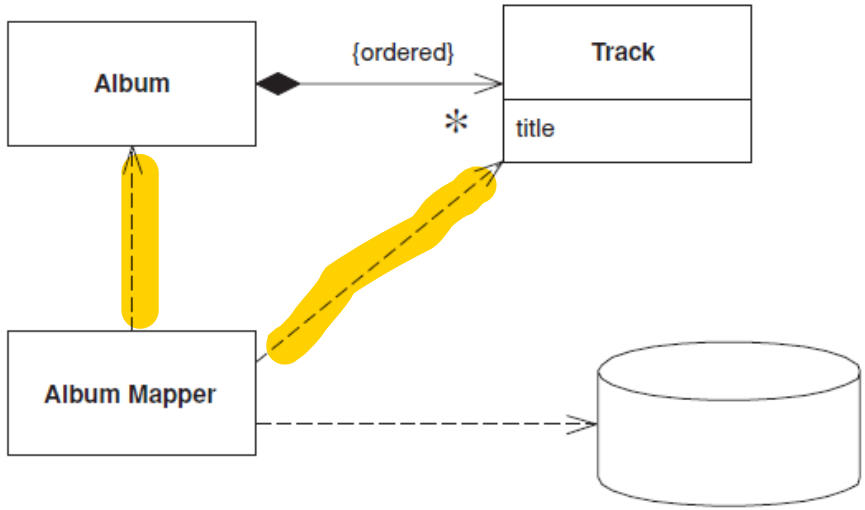
Dependent Mapping

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

Some objects naturally appear in the context of other objects.

Tracks on an album may be loaded or saved whenever the underlying album is loaded or saved.

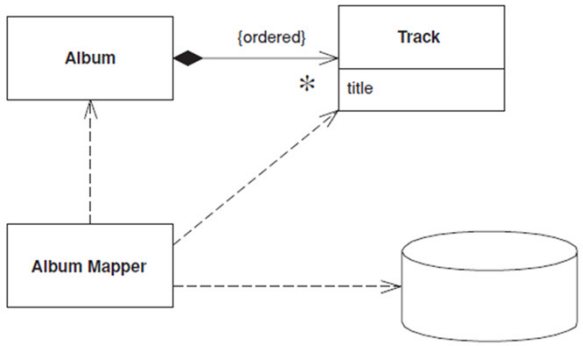
If they are not referenced to by any other table in the database, you can simplify the mapping procedure by having the album mapper perform the mapping for the tracks as well—treating this mapping as a ***dependent mapping***.



In most cases every time you load an owner, you load the dependents too. If the dependents are expensive to load and infrequently used, you can use a *Lazy Load* to avoid loading the dependents until you need them.

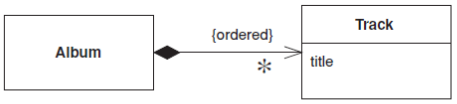
For *Dependent Mapping* to work there are a number of preconditions.

- A dependent must have exactly one owner.
- There must be no references from any object other than the owner to the dependent.



Example: Albums and Tracks (Java)

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*.



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

```
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()]);
}
```

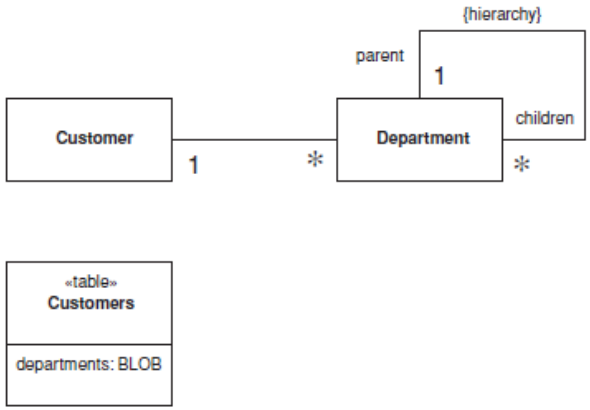
```
class AlbumMapper...
protected String findStatement() {
return
"SELECT ID, a.title, t.title as trackTitle" +
" FROM albums a, tracks t" +
" WHERE a.ID = ? AND t.albumID = a.ID" +
" ORDER BY t.seq";
}

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;
}
```

Serialized LOB

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



Object models often contain complicated graphs of small objects. Much of the information in these structures isn't in the objects but in the links between them.

Consider storing the organization hierarchy for all your customers. An object model quite naturally shows the composition pattern to represent organizational hierarchies, and you can easily add methods that allow you to get ancestors, siblings, descendents, and other common relationships.

There are two ways you can do the serialization: as a binary (**BLOB**) or 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.

The advantages of the BLOB are that it's simple to program and that it uses the minimum of space. The disadvantages are that your database must support a binary data type

The alternative is a **CLOB**. In this case you serialize the department graph 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 works poorly when you have objects outside the LOB reference objects buried in it.