1

Introduction to Mapping Objects to Relational Databases

In the computer industry, we commonly have discussions, disagreements, and even shouting matches over which of the latest languages are the best and where they should be used. Discussions turn to the best platform currently available and who's suing whom. However, each of us is also thinking about the latest project we've been given and its strict deadline. Overall, the project isn't complex, but we have to support numerous database backend systems. This means we need to incorporate a persistence layer to handle the database differences. That part isn't too difficult, but what about the data itself? Do we just store it in some proprietary database and deal with the details later? No, we need to have a strategy that works with our application and the language the application is written in.

Today's programming languages take advantage of the latest in object-oriented techniques. As you know, an *object* is a construct to enforce encapsulation. The problem is how to store an object for use by the same application at a later time or by another application. Some of the common solutions to this problem are:

- Serialization
- XML
- Object-oriented database systems mapping

Let's consider these possible solutions and determine their advantages and disadvantage before looking at the solution around which this book is written.

Serialization

You can save data stored in an object by creating a flat-file representation of the object. In the Java language, this process is called *serialization*. An object that is serialized has all its attributes and their class types saved into a file or a buffer. The resulting information string can be saved to a file, placed in a column in a traditional relational database, or sent to another machine. An example is the CD class:

```
public class CD implements Serializable {
String title;
String artist;
public CD(String title, String artist) {
this.title = title;
this.artist = artist;
    }
}
```

The CD class has two attributes that need to be saved when the object is serialized. In Java, all primitive types as well as many of the foundational classes are defined such that they implement the Serializable interface. The system automatically recurses into each of the attributes as needed.

The serialization of the CD class results in a binary representation of the data currently contained in the represented object. The binary representation could be placed in a BLOB column type of a relational database; this process would allow other applications to access the data. However, if a legacy application has been tweaked to access the column where the object is held, it won't be able to make sense of the data unless the legacy application can deserialize Java objects. Even worse, the serialization process doesn't migrate well from Java application to Java application.

Further, the serialization process isn't fast, and a large object can take a considerable amount of time to be put into its binary representation. Thus, serialization as a practice has a specific place in the development process; but as a mechanism for persisting data, it should be avoided.

XML

In the past few years, XML has been one of the hottest technologies. With this popularity comes the issue of using XML in both objects and mapping to a database. First, consider an XML document like the following:

```
<cd>
    <title>
        Grace Under Pressure
    </title>

    <artist>
        Rush
    </artist>
</cd>
```

A database to handle the XML document can be created with the following schema:

```
create table cd (
  title varchar,
  artist varchar
);
```

From the XML, we can easily build a class like this:

```
public class cd {
  String title;
 String artist;
}
```

Having the XML representation creates an additional level of complexity and processing required to go from an object to the database. This processing can be extensive, given the complexity of the objects in an application.

Object-Oriented Database Systems

When object-oriented programming first began to be used, the issue of storing objects for later use was an important topic. The most widely used component for storage is a database, so several companies started down the path of developing a new database technology used specifically to store objects. The object-oriented database system handles the storing and loading of objects in a transparent manner. The complexity in the system comes from querying the database. For example, we might have a query like this:

```
select x from user x where x.name = \"John Doe\"");
```

The database will access all the user objects in the database and return those whose name attribute is equal to "John Doe". The database needs to be designed in a manner that automatically allows the query to access the attributes of the stored objects.

Although these new database systems can transparently store and load objects to Java or other object-oriented languages, there is typically an issue when a legacy system or a quick RAD application needs to access the same information. In addition, the OO databases haven't made a large impact in the database market; they're still coming of age when compared to those offered by Oracle, Microsoft, and others.

Mapping

The three solutions we've just covered can work, but they present issue when put into the mix of legacy applications and traditional relational database systems. If you're working with an application that uses a database, you'll most likely need to use databases having names like Oracle, MySQL, Sybase, Microsoft SQL Server, and others. These databases are based on the traditional relational model; somehow we need to use them along with our Java objects.

An object can be placed in a relational database through the process of *mapping*. Mapping is a technique that places an object's attributes in one or more fields of a database table. For example, the earlier CD

class has two attributes that would need to be mapped to a relational database table for permanent storage. The title and artist fields can be mapped to a schema like the following:

```
create table CD (
ID int not null primary key auto_increment,
title varchar(256),
artist varchar(256)
);
```

The ID field is used to create unique rows in the database. Each title and artist field holds the appropriate value from the CD object. This is an example of a *one-to-one mapping* between the class and the database. Figure 1.1 shows the appropriate UML diagram to accompany the class and the resulting database table.

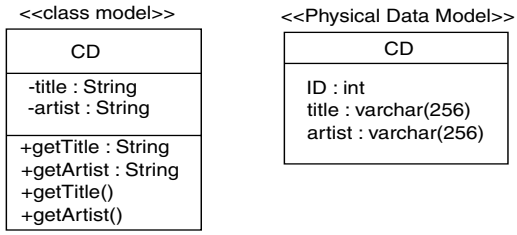

Figure 1.1

From a database standpoint, consider a CD object instantiated using the following constructor:

```
new CD("Grace Under Pressure", "Rush");
```

When the object is mapped to the database table defined earlier, the values in the database might look like this:

```
+----+--------------------------+-------+
| ID | Title                    | Artist|
+----+--------------------------+-------+
| 1  | Grace Under Pressure     | Rush  |
+----+--------------------------+-------+
```

For any additional CD objects that are instantiated, new rows are created in the database; the ID column maintains their uniqueness.

Typically, the classes you're dealing with in a production application will include more than just simple attributes. Consider the following CD class:

```
public class CD implements Serializable {
String title;
String artist;
```

```
ArrayList tracks;

public CD(String title, String artist) {
this.title = title;
this.artist = artist;

tracks = new ArrayListO;
 }

public void addTrack(String track) {
tracks. add(track);
 }
}
```

In this new CD class, we've added an ArrayList to hold the name of each title on the CD. As you might expect, this additional attribute introduces a hidden complexity to the mapping between the objects instantiated from the CD class and the permanent storage: The ArrayList attribute can contain no values or hundreds of values. There might be 8 to 10 tracks on a CD; an MP3 CD could include hundreds of songs. In either case, we need to be able to map the ArrayList attribute into a database structure so it can be permanently recorded when the entire CD is committed.

A common solution is to create a second database table just for the attribute. For example, we might create a table like this:

```
create table CD_tracks (
ID int not null primary key.
track varchar(256)
);
```

Using the Rush CD object created earlier, we can make a couple of calls to the `addTrack()` method and fill the tracks ArrayList:

```
rush.addTrack("Distant Early Warning");
rush.addTrack("Afterimage");
rush.addTrack("Red Sector A");
```

When the CD object is saved to the database, information is placed in two different tables: a CD table

```
+----+--------------------------+-------+
| ID | Title                    | Artist|
+----+--------------------------+-------+
| 1  | Grace Under Pressure     | Rush  |
+----+--------------------------+-------+
```

and a CD_tracks table

```
+----+--------------------------+
| ID | Track                    |
+----+--------------------------+
| 1  | Distant Early Warning    |
+----+--------------------------+
| 2  | Afterimage               |
+----+--------------------------+
| 3  | Red Sector A             |
+----+--------------------------+
```

If we have another CD to store, should another track table be added to the system, or should the tracks be added to the existing CD_tracks table? If we add a track to the CD_tracks table, how do we keep track of the fact that some of the tracks relate to specific CDs in the CD table?

We need to add a *foreign key* to the CD_tracks table; it will relate to the primary key in the CD table. Here's what the new CD_tracks table looks like:

```
create table CD_tracks (
ID int not null primary key auto_increment,
cd_id int,
track varchar(256)
);
```

Using this schema, the Rush CD's tracks appear as follows:

```
+----+------+----------------------------+
| ID | cd_id| Track                      |
+----+------+----------------------------+
| 1  |   1  | Distant Early Warning      |
+----+------+----------------------------+
| 2  |   2  | Afterimage                 |
+----+------+----------------------------+
| 3  |   3  |  Red Sector A              |
+----+------+----------------------------+
```

With the addition of the cd_id column, we can relate the two tables needed to fully map the CD object to permanent storage. The addition of the new CD_tracks table expands our UML diagram, as shown in Figure 1.2.

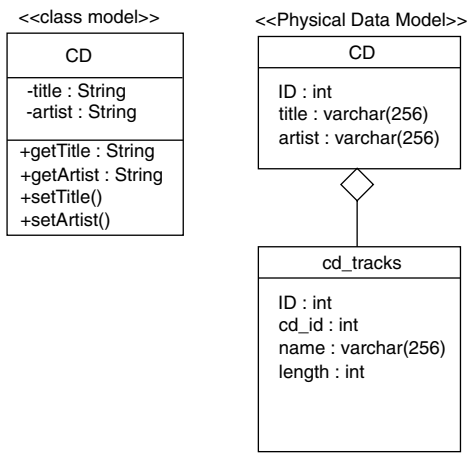

Figure 1.2

Our CD class can be further complicated by adding an attribute based on another class type. For example:

```
public class CD implements Serializable {
String title;
String artist;
ArrayList tracks;

public CD(String title, String artist) {
this.title = title;
this.artist = artist;

tracks = new ArrayList();
}

public void addTrack(Track track) {
tracks.add(track);
 }

private class Track {
String name;
int length;

public track(String name, int length) {
this.name = name;
this.length = length;
        }
 }
}
```

We've added another class called Track, which is added to the track ArrayList instead of a single string. With the Track class added to the class model, we have a new situation that needs to be handled through an appropriate mapping. The most obvious choice is to add another database table for mapping between the class and the permanent storage. For example:

```
create table tracks (
  ID int not null primary key auto_increment,
  name varchar(256),
  length int
);
```

The new database table looks similar to the CD_tracks database created in the previous example but includes a little more information. (We could have used the CD_tracks schema and added the length column.)

After these examples, it should be clear that saving a set of objects to permanent storage isn't a simple task. In order to correctly put an object's attributes in a database, you must have a clear plan in place with the proper databases and mappings. As you can see, though, once a class has been defined, the database tables and mappings become clear. Thus, in most design situations, the database modeling should occur after the classes have been defined.

Primary Keys, Timestamps, and Version Numbers

In the examples presented so far, all the database tables have included a primary key that isn't part of the original object being mapped. The primary key is needed in order for the database server to uniquely distinguish and manage the objects stored in the database. Without the primary key, the database might have duplicate rows in a table because two objects in the system have identical attribute values. The primary key also gives us the ability to determine whether or not an object has actually been added to the database and if the object needs to be updated.

Depending on the system used to handle the mapping from objects to permanent storage, there are different ways to determine whether an object is up to date. One way is to use a timestamp in the database table. When the persistence layer needs to determine whether an object should be persisted to the database, it can check the timestamp in the table row for the object. If the timestamp is less than a timestamp kept in the object itself, the persistence layer knows the object should be persisted. If the timestamps are the same, the object hasn't been changed and doesn't need to be saved.

Another technique involves a version number stored in the database. When the object is pulled from the database, it has an associated version number. If the application changes the object in any way, the persistence layer updates the version number by a single digit. The layer can then use the version number to determine whether the object needs to be persisted.

Handling Inheritance

Obtaining efficiencies in the software development process means using all of a methodology's features. This is especially true when you're developing the classes for an application. During the development of the class structure, a clear hierarchy can sometimes be created through inheritance. For example, for our CD class, we might have another class called SpecialEditionCD that inherits its foundational attributes from the CD class:

```
public class SpecialEditionCD extends CD {
String newFeatures;
int cdCount;

public SpecialEditionCD(
String title,
String artist,
String newFeatures,
int cdCount) {
this.title = title;
this.artist = artist;
this.newFeatures = newFeatures;
this.cdCount = cdCount;
    }
}
```

The SpecialEditionCD class adds two more attributes that need to be persistent to our permanent storage in addition to the attributes from the CD parent class. We can't easily store the SpecialEditionCD information in the CD database table because of these new attributes. How do we perform the mapping? There are several solutions:

- ❑ Create a single table using the attributes from the lowest child.
- ❑ Create a table per class.
- ❑ Create a table per concrete class.

Let's consider each of these inheritance mappings using the CD and SpecialEditionCD classes as examples.

Lowest-Child Inheritance Mapping

If we have a hierarchy of classes, our mapping needs to be able to handle both the CD and SpecialEditionCD classes using a single table. We can accomplish this by taking all the attributes for the *lowest child* in the inheritance chain, mapping them, and then moving up the chain until we've mapped the topmost parent. The result is a table with attributes from all classes. Using our two classes, this process produces a table like the one in Figure 1.3.

```
create table cd (
ID  int not null primary key auto_increment,
type varchar(256),
title varchar(256),
artist varchar(256),
newFeatures varchar(256),
count int
);
```

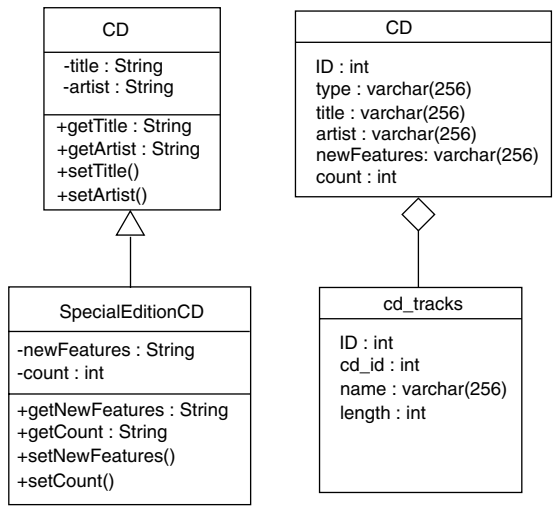

Figure 1.3

If we have an object of type SpecialEditionCD, all the necessary columns are available to persist it to the database. If we have a CD object, all the necessary columns are still available; however, both the

newFeatures and count columns need to be populated with a value such as null. But will this work? Can't we have a SpecialEditionCD object where both the count and newFeatures attributes are null? In that case, how can we tell which object has been stored? We add to the database schema an addition field called type that the mapping software uses to keep track of the class type to which a particular row belongs.

As you can see from this example, adding new classes to the class model as well as the database is as simple as adding columns to the single hierarchy table. There is complete support for polymorphism because of the type column. Changing the type value from one class to another changes the object class type. We don't need to keep track of additional tables since all the information for the class hierarchy can be found in the single table.

If you have a large hierarchy, the single-table option isn't ideal because of the potential for wasted space from row to row. If an object is higher in the hierarchy, there will be quite a few nulled column values. Further, the database may potentially have a large number of columns. This isn't usually a problem for database-management software, but the database can be difficult to read and maintain.

Table-Per-Class Inheritance Mapping

If you want to eliminate some of the issues with the single-table approach to mapping an inheritance hierarchy, consider creating a single table for each of the classes. Figure 1.4 shows an example of the object-to-table mapping.

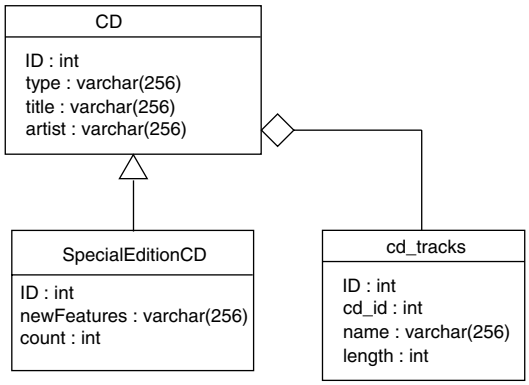

Figure 1.4

In Figure 1.4, we've created three tables to handle the CD inheritance hierarchy. The CD and `cd_tracks` tables monitor the attribute needed for a `CD` object. For the SpecialEditionCD class, we create another table to hold the information new to this class. If the object we're mapping to the database is a CD class, then we just need to access the two base tables. However, if we have a SpecialEditionCD, we need to access three tables to pull out all the information needed to save or load the object from the database. If the persistence layer is intelligent, we won't necessarily hit all three tables for a SpecialEditionCD—only the tables needed at a particular moment.

If we add more classes to the hierarchy, we need to build the appropriate database table to fulfill the necessary mappings. Various relationship must be maintained between the rows for each subchild object

created. Maintaining these relationships can be complex, depending on the hierarchy being mapped. We aren't wasting database space if we have a CD object instead of a SpecialEditionCD since we won't need a row for the child object.

Table-Per-Concrete-Class Inheritance Mapping

In our example CD class hierarchy, it's clear that we'll be creating objects of type CD and SpecialEditionCD. Thus both of these classes are concrete. However, you might have a hierarchy in which one or more classes aren't concrete. In this type of situation, mapping all the classes to database tables isn't necessary; instead, you can make the concrete classes only. This technique will save you time and database space. The real advantage comes in the time needed to access the database for attributes from the nonconcrete classes when an object needs to be saved or loaded.

Working With Relationships

In almost all applications, numerous objects relate to one another in some fashion. For example, we might have an account object that holds information about a specific account in an accounting application. Each account might be associated with a single owner object; this association is a *one-to-one* relationship. Each account might have numerous addresses where account information could be sent; this association is a *one-to-many* relationship. Finally, each account can be assigned to numerous securities, and securities can be assigned to numerous accounts; this association creates a *many-to-many* relationship.

These three different relationships are called *multiplicity*; however, there is another relationship type called *directionality*. To illustrate, consider the account/securities many-to-many relationship just discussed. If you have an account, you can determine which securities it's associated with; and if you have a security, you can determine which accounts it's associated with. This is an example of a *bidirectional* relationship. For the account-to-address association, an account knows about the addresses it needs, but the addresses don't know which account they're bound to; this is an example of a *unidirectional* relationship.

Mapping a One-to-One Relationship

In our one-to-one relationship example, we'll consider a situation where we have a single account and a single owner. Figure 1.5 shows an example of the classes we'll map.

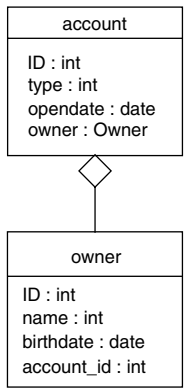

Figure 1.5.

To map the relationship shown in Figure 1.5 to a relational database, we need to use a foreign key that associates the owner with the account. Since there will be a single owner for each account, we know there will be a single row in the database table that holds the owner information. The following table schemas can be used to map the relationship in Figure 1.5:

```
create table account (
  ID int not null primary key auto_increment,
  type int,
  opendate
);

create table owner (
  ID int not null primary key auto_increment,
  account_id int,
  name varchar(256),
  birthdate date
);
```

The one-to-one mapping is created between the account and owner tables through the account_id field in the owner table. When a new row is added to the account table, a new ID is created. This ID uniquely identifies the account to the accounting system. No matter how many accounts are added to the system, no other one will have the same ID value. Therefore, this value can be used to associate the account and its database row with any other necessary data. Some of that data is found in the owner table. Just after the account is added to the database, the owner row is inserted. Part of the owner information to be stored is the ID of the account. The ID is stored in the account_id field, which is a foreign key to the primary key (ID) of the account table.

When the application needs to pull the account, it accesses the account through the primary key or, potentially, through a query that pulls the ID as well. The application can then pull the owner row using the ID value.

Mapping a One-to-Many Relationship

In the previous example, we also have a one-to-many relationship when the account object relates to one or more addresses. Figure 1.6 shows an example of the objects we need to map.

A single account in our application can have one or more addresses. Thus, each account needs an array of address objects that hold the full address where information can be sent and associated with the account. The following schemas show how the information is kept in the database:

```
create table account (
  ID int not null primary key auto_increment,
  type int,
  opendate date
);

create table address (
  ID int not null primary key auto_increment,
  account_id int,
  address varchar(256),
  city varchar(256),
  state varchar(256),
  zip varchar(256)
);
```

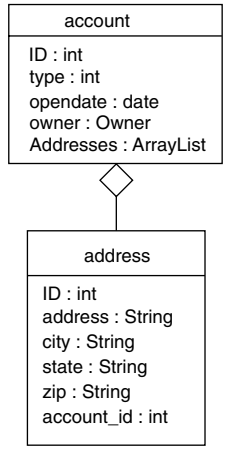

Figure 1.6

As you can see, the foreign key, `account_id`, is used in the mapping just as it is in the one-to-one relationship map. When an account is added to the database, its ID is obtained and used to store one or more addresses in the address table. The ID is stored in the `account_id` foreign key column. The application can perform a query against the address table using the ID as a search value.

Mapping a Many-to-Many Relationship

In the final type of relationship, an account relates to securities and at the same time securities relate to accounts. Figure 1.7 shows the classes in the relationship.

As you might expect, doing the mapping for a many-to-many relationship isn't as easy as it is for the one-to-one and one-to-many cases. For every account and security in the system, we need to be able to relate one or more of the other classes. For example, if we have an account with an ID of 3425, it might be associated with the securities 4355, 3245, 3950, and 3954. How can we associate these securities with the single account? We could have four columns in the account table called sec1, sec2, sec3, and sec4. But what if we added two more securities to the account? We wouldn't have enough columns in the account table.

The solution is to use an association table between the account table and the security table. Here are the table schemas:

```
create table account (
  ID int not null primary key auto_increment,
  type int,
  opendate date,
  securities int
);

create table security (
  ID int not null primary key auto_increment,
  name varchar(256),
```

```
  cost int,
  accounts int
);

create table account_security (
  ID int not null primary key auto_increment,
  account_id int,
  security_id int,
);
```

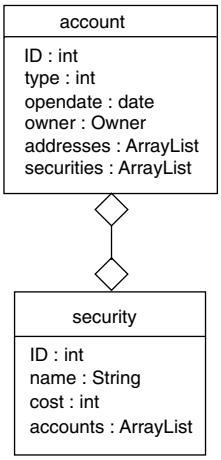

Figure 1.7

For every relationship between an account and a security, there will be a unique row in the `account_security` association table. If we have an account ID, we can query the `account_security` table to find all the securities associated with the account. We can also go the other direction and use a security ID to query for all the accounts using that security.

Summary

When you're writing applications using the object-oriented methodology, real-world information is stored in classes where the information can be encapsulated and controlled. At some point during the execution of the application, the information in each of the objects will need to be persisted. The goal of this chapter has been to provide you with an overview of the issues and techniques involved in object mapping from the application to permanent storage. By far the most popular technique is mapping a class to one or more database tables. We've covered many of the issues involved in object/relational mapping; now we'll push forward and discuss Hibernate as a tool for providing the necessary mapping.