5

Using Hibernate for Persistent Objects

Raj: When do you want to meet with Susan for the questions on the screens we are working on currently?

Steve: Let's just call her right now since it impacts our database persistence design. After all, she wants to be an active stakeholder since her department is paying for this application.

(c) Visual Patterns, Inc.

WITHIN THE FIRST FEW YEARS OF my software development career, I came to realize more and more that information (data) is an organization's main asset; however, many developers tend to lose sight of this fact and get caught up in the latest cool tools.

We often use the words "information technology" (or simply, IT) but have you ever stopped to think about these two words? In my opinion, the definition is obvious: technology to manage information. Simply put, data is at the core of what we do in our industry because we are constantly moving data from point A to point B. No matter how many systems the data travels through, it originates on one end (point A; for example, a UI) and is typically viewed on the other end (point B; for example, reports). Furthermore, the data and its structure typically outlive the applications built around it; hence, it should arguably be the most important component of an overall software application's architecture.

Given my emphasis on data and databases, I will cover Hibernate before the other key products discussed later in this book, such as the Spring Framework and the Eclipse SDK.

What's Covered in This Chapter

In this chapter, we develop the classes we need to implement functionality for the first five user stories (page 63) for our sample application, Time Expression. So, we will do the following:

- Understand what object-relational mapping technology is and the benefits it offers.
- Install HSQLDB, a Java-based, lightweight relational database.
- Design our database.
- Write a Data Definition Language (DDL) script to create our database tables using Ant and test out some sample queries.
- Set up Hibernate, understand its basic concepts, and begin working with it.
- Demonstrate a simple and then a slightly more complex example (along with a corresponding unit test suite class) of using Hibernate for Time Expression's Department and Timesheet tables.
- Discuss advanced Hibernate areas for you to explore (should you need them).

Note
The complete code for the examples used in this chapter can be found within this book's code zip file (available on the book's website).

We will cover a lot of material in this chapter, so let's get started.

An Overview of Object-Relational Mapping (ORM)

It is no secret that relational databases are the most common type of databases in a majority of organizations today when compared to other formats (for example, object-oriented, hierarchical, network). Product names such as Oracle, Microsoft SQL Server, MySQL, IBM DB2, and Sybase are common terms used by developers in our line of work.

On the computer languages side of things, object-oriented (OO) programming has become the norm. Languages such as Java, C#, C++, and even OO scripting languages are common discussion topics among developers.

A majority of the software applications that use relational database and OO languages end up writing code to map the relational model to the OO model. This can involve anywhere from cumbersome mapping code (because of the use of embedded SQL or stored procedure calls) to heavy-handed technology, such as EJB's entity beans.

Because most of us seem to like both relational databases and OO, Object-Relational Mapping (ORM) has become a natural choice for working with POJOs (plain old Java objects), especially if you don't need the distributed and secure execution of EJB's entity beans (which also map object attributes to relational database fields).

Although you still need to map the relational model to the OO model, the mapping is typically done outside of the programming language, such as in XML files. Also, once this mapping is done for a given class, you can use instances of this class throughout your applications as POJOs. For example, you can use a `save` method for a given object and the underlying ORM framework will persist the data for you instead of you having to write tedious `INSERT` or `UPDATE` statements using JDBC, for example.

Hibernate is one such ORM framework and given its popularity in the world of Java today, we will use it for Time Expression. A few others, such as JDO, iBATIS, Java, and Apache ObJectRelationalBridge, are listed at the end of this chapter under "Recommended Resources."

Hibernate also supports the EJB 3.0 standard, so should you need to move to EJB 3.0, it'll be an easy transition (in fact, EJB 3.0 is based on many of the concepts and techniques found in Hibernate). EJB 3.0, as you might already know, aims to simplify working with EJB technology prior to this release; for example, EJB 3.0 provides a lighter-weight persistent API similar to the one provided by Hibernate. However, if you do not need the many services provided by EJB technology, you can use the Hibernate core technology by itself (without needing a big EJB container product such as an application server).

Before delving into Hibernate, let's review some basic concepts common across ORM technologies. Later we will look at lots of Hibernate Java code and XML file examples. After you have the hang of coding using an ORM framework, you will almost certainly not turn back to the old ways of working with relational databases.

Relationships and Cardinality

Database relationships are typically defined in terms of direction and cardinality (multiplicity in OO terminology). From an OO perspective, relationships are defined as association, inheritance, or aggregation. Many software development projects use ORM either with existing databases or are required to conform to standards established by a database group within the organization; hence, I will approach our relations discussion from a database perspective.

Note

Relationships can be viewed as unidirectional or bidirectional for objects. On the other hand, relations in a relational database are bidirectional by definition because related tables know of each other. However, if we were designing objects that map to the database, we would factor in both types of relations because object relationships have to be made bidirectional explicitly. So for the sake of our discussion on relationships and cardinality, we will pretend that the database can have both—unidirectional and bidirectional—relations.

Unidirectional is when one table knows of another, but not vice versa. For example, you might have a record that uses a unique primary key; this same primary key can be used as a foreign key by records in a child table, thereby establishing a unidirectional relationship. In a *bidirectional* relationship, records in both tables would know about each other. For example, assume we have two tables named Employee and Project to store information about which employees worked on which project. In the Project record, we might have an EmployeeId foreign key. On the flip side, we might have a ProjectId key in the Employee table.

Cardinality can be defined as *one-to-one*, *one-to-many* (or many-to-one depending on which direction you look at the relationship), and *many-to-many*. We look at each briefly:

- A one-to-one relationship is when a record in table 1 can have exactly one associated record in table 2. For example, a record in a Person table might have exactly one related record in a JobTitle table.
- A one-to-many relationship is typically seen in parent-child relationships where a parent record can have several related records in a child table (for example, related via the parent's primary key).
- A many-to-many relationship is where a record in table 1 can have several related records in table 2 and vice versa. For example, an Employee table might have more than one record in a Project table (because an employee can be involved in multiple projects). On the flip side, a record in the Project table might have several related records in the Employee table because a project can have multiple employees assigned to it. Also, this type of relationship is typically achieved by using an (extra) association table (for example, a ProjectEmployee table that contains foreign keys pointing to the two main tables).

Note

We will be looking at examples of relationships in this chapter from various perspectives, namely diagrams, code, and mappings. For example, Figures 5.1 and 5.2 show examples of one-to-many relationships.

Object Identity

An object identity (or simply, object id) is something that uniquely defines a persisted object (that is, a record in the database). It is commonly mapped to the primary key of a database table.

Cascade

Cascading can be defined as an action on a given entity flowing down to related entities. For example, if we wanted to maintain referential integrity between related parent-child tables in a database, we would delete records from a child table whenever its related parent is deleted, so that no orphan records are left lingering in the database. Similarly, when you read a parent record in, you may also want to read in all its children records. Cascading can be defined for each of the four CRUD operations—that is, create, read, update, and delete. Also, cascading is often handled via the use of database triggers.

Mapping

Before we can begin working with objects that store and retrieve data from a relational database, we must create mappings (usually in an XML file) between the database tables and Java classes. The mapping file typically contains properties, which essentially map an attribute (variable) in a class to a column in database. If you are new to some of these concepts, don't worry; after you see some examples later in this chapter, it'll start to become a bit clearer.

There are various mapping strategies we can employ, such as horizontal mapping, vertical mapping, and union mapping. In *vertical mapping*, each class in a hierarchy (abstract or concrete) is mapped to a different table. For example, if we have concrete classes named Dog and Cat, both inheriting from an abstract class named Animal, we would end up having three tables in the database—one for each class. In *horizontal mapping*, each concrete class is mapped to a table. In *union mapping*, many classes (presumably part of the same hierarchy) map to a single table.

Although vertical mapping is more flexible, it is also more complex because it requires multiple tables to extract all the data. Hence, we will use horizontal mapping because it is a simpler design and can provide faster performance, especially for simple to reasonably complex applications. To be more specific, our approach will involve one table per class mapping strategy.

In-Memory Versus Persisted Objects

When we are working with ORM technologies, there is a distinction between database objects we have in memory versus persisted ones. If the object does not exist in the database, or its attribute values do match the corresponding column values in the database, it is considered an in-memory object. For example, Hibernate distinguishes object states as persistent, detached, or transient (each is explained later in this chapter).

Another way to look at this distinction is that if we remove an object from memory (for example, by removing it from a Java collection), it does not necessarily mean the record has been physically deleted from the database (unless, of course, we mapped the collection in Hibernate to have automatic cascading during parent deletes).

Design of Our Sample Database

Now that we have covered some OR concepts, it is time to set up our database so that we can move one step closer to building an application's user interface with the help of the Spring Web MVC Framework.

As I mentioned in earlier chapters, the focus of this book is more on development and less on infrastructure. Given Java's vendor product portability (for example, operating system, web/application server, databases), in theory, it should be relatively easy to develop your application using one product but deploy to another application. In light of this, I chose the easiest (and consequently lightest-weight) products to set up. HSQLDB, a relational database, is one such product (discussed later in this chapter), and we will use it for Time Expression.

Denormalization

Before we look at HSQLDB, let's revisit our domain model from Chapter 3, "XP and AMDD-Based Architecture and Design Modeling," shown in Figure 5.1.

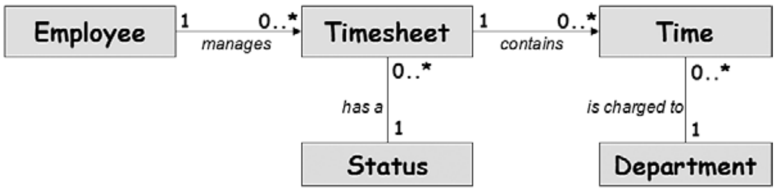


Figure 5.1 Domain model for Time Expression.

Given the simplicity of our sample application, Time Expression, and its domain model, we could create a physical database model (PDM), also known as an Entity-Relationship (ER) diagram, which contains entities identical to ones in our domain model, with the addition of columns and data types and other database constraints. However, let's denormalize it just a bit for performance and ease of development purposes.

Figure 5.2 shows a PDM, denormalized a bit from our Domain Model and with data types (for example, varchar) added to it. The denormalization is related only to the Timesheet and Time tables.

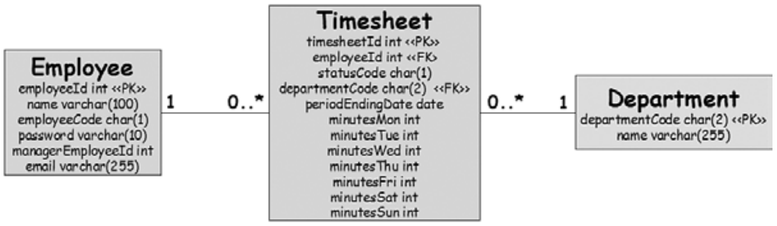


Figure 5.2 Physical database model for Time Expression.

Naming Convention

You will notice we are using Java-like naming conventions for the table and column names. This makes our job easier because we can use the same names across all artifacts related to Time Expression while also gaining consistency across them. In other words, we have matching names from User Story tag/name to controller classes to model (domain) objects to the Hibernate persistent bean Java code and finally, to the database tables and columns (shown in Figure 5.2).

This naming approach makes our job easier in two ways. First, we don't need to think about the naming convention for each layer, and second, it reduces the amount of mapping details we need to specify in our Hibernate class mapping files because we do not have to specify a corresponding column name for each property being mapped (as we will see later in this chapter).

However, in the real world, you might not have control over the database table/column naming because a database group might have their own set of naming standards. In this case, it is easy to use Hibernate's *column* attribute to specify the database column name. I would also encourage following your organization's naming standards for consistency sake.

Note that for database objects (such as tables and sequences), I tend to use names starting with an uppercase letter, whereas column names start with a lowercase letter.

Database Design Disclaimers

The following are some disclaimers and/or explanations for the PDM we looked at in Figure 5.2.

Unused Columns

By combining the Timesheet and Time entities into one physical table, there is the possibility of wasted database space by unused columns. For example, there is a good chance that MinutesSat and MinutesSun will be less frequently used (unless employees in this company work most or all weekends). However, the advantages of the simpler design and performance arguably outweigh the disadvantages of a bit of wasted space.

Int Versus Float

We have used Minutes<Day> columns to store fractional hours worked (for example, 30 minutes or 0.5 hour) in the Timesheet table versus Hours<Day> columns or even float data types. The reason we did this is because I want to demonstrate how we can use the Spring Web MVC framework (in Chapter 7, "The Spring Web MVC Framework") to do automatic data conversions between the UI and the database. Also, an `int` will typically take up less physical storage space than a `float` will (for example, 2 bytes versus 4 bytes).

Password

We have a Password column in the Employee table. Typically, in larger organizations, you might end up using something like a central Lightweight Directory Access Protocol (LDAP) authentication service. However, this works well for our small (and sample) application, Time Expression.

DDL Script

Now that we have a PDM (see Figure 5.2), we can move to the next level down, which is to write a DDL script that can be used to create the actual databse. Our DDL script is embedded inside one of our Ant scripts, named `timexhsqldb.xml`. The table names, column names, and data types in our DDL script closely match the PDM in Figure 5.2, as they should.

Our DDL file primarily contains CREATE TABLE statements. However, I would like to point out a couple of additional notable items.

First, the primary key column of the `Timesheet` table is of data type `identity`, as shown in this code excerpt:

```
CREATE TABLE Timesheet
(
   timesheetId IDENTITY NOT NULL,
```

As you might already know, an `identity` is an auto increment database column (and is directly supported by Hibernate). For databases that do not support identity types, we can use a sequence type instead.

Second, we have seen some test data being inserted; this is for use by our JUnit test cases covered later in this chapter. For the sake of simplicity, I have not created any primary or foreign key constraints, as we typically should in a real-world application. Also, the focus of this chapter is to demonstrate features of Hibernate and not necessarily database design.

Where HSQLDB and Hibernate Fit into Our Architecture

Before we get too far along with HSQLDB and Hibernate, it is a good idea to revisit our architecture diagram that we developed earlier in this book. Figure 5.3 shows the diagram; notice where HSQLDB and Hibernate fit into the big picture (top-right).

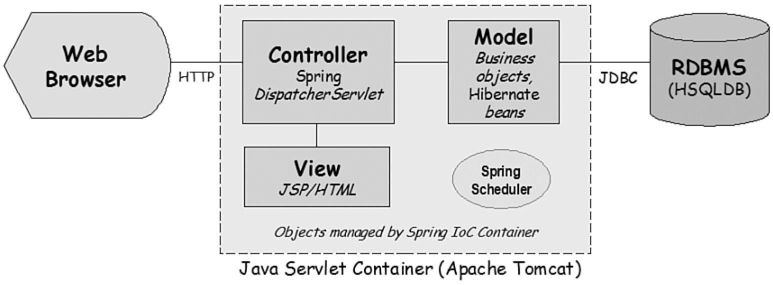

Web Browser
HTTP
Controller
Spring
DispatcherServlet
Model
Business objects, Hibernate beans
JDBC
RDBMS
(HSQLDB)
View
JSP/HTML
Spring Scheduler
Objects managed by Spring IoC Container
Java Servlet Container (Apache Tomcat)

Figure 5.3 High-level architecture for Time Expression.

In later chapters, when we develop our web and schedule job-related code, we will need the classes and database we will create in this chapter.

HSQLDB

HSQLDB is a lightweight Java database engine that has been around since 2001. However, because it is a continuation of Thomas Mueller's closed Hypersonic SQL Project, it has actually been around longer than 2001. In short, the product is fairly mature.

HSQLDB provides a good amount of ANSI-92 SQL-compliant features (and many enhancements from more recent SQL standards)—more than we will need in this book. Furthermore, most of the features defined by JDBC 2, and some from JDBC 3, are also supported. HSQLDB's popularity has grown significantly since its inception a few years ago, and it is commonly found bundled with open source and commercial Java-related products such as JBoss, OpenOffice.org, Atlassian's JIRA, and many more.

At the time of this writing, the HSQLDB project was one of the top 50 ranking in more than 100,000 SourceForge.net projects.

HSQLDB can be found at http://hsqldb.org. There are ample setup instructions on this site to download, install, and configure it. I'm using version 1.8.x in this book.

HSQLDB Server and Convenient Ant Tasks

Now we need to start the HSQLDB server and create the database using our DDL file. However, first, let's copy the `hsqldb.jar` file from the HSQLDB install directory to our `lib/` directory; for example, on my Microsoft Windows XP-based system, I typed the following:

```
copy \hsqldb\lib\hsqldb.jar \anil\rapidjava\timex\lib\
```

We will use our Ant script, `timexhsqldb.xml`, to start the server and also to create the database. This file is placed in the top-level directory of our sample application (in my case, this is `C:\anil\rapidjava\timex`).

Assuming our HSQLDB configuration is set up correctly, we can now type the **`ant -f timexhsqldb.xml starthsql`** command to start the HSQLDB server, as demonstrated here:

```
C:\anil\rapidjava\timex>ant -f timexhsqldb.xml starthsql
```

From another command window, we can type the **`ant -f timexhsqldb.xml execddl`** command to execute our DDL script for creating our database within HSQLDB, as demonstrated here:

```
C:\anil\rapidjava\timex>ant -f timexhsqldb.xml execddl
```

Before we move on, let's review parts of `timexhsqldb.xml`.

The following properties are related to HSQLDB; that is, the `hfile` property points to a local set of files under the `timex/data/` directory, the `halias` is the alias we will use in our client applications to connect to the HSQLDB server, and the `hport` is the port number the HSQLDB server will listen to:

```
<property name="hfile" value="-database.0 data/timexdb"/>
<property name="halias" value="timex"/>
<property name="hport" value="9005"/>
```

Next, the `starthsql` Ant target starts the HSQLDB server using the built-in Ant `java` task, as shown here:

```
<java fork="true"
      classname="${hclass}" classpath="${hjar}"
      args="${hfile} -dbname.0 ${halias} -port ${hport}"/>
```

The `execddl` Ant target uses the built-in `sql` task to execute our SQL DDL script, as shown here:

```
<sql classpath="${hjar}"
     driver="org.hsqldb.jdbcDriver"
     url="jdbc:hsqldb:hsql://localhost:${hport}/${halias}"
     userid="sa" password=""
     print="yes">
```

HSQLDB Database Manager and SqlTool

HSQLDB is bundled with two tools you should read about in the HSQLDB documentation: HSQL Database Manager (GUI) and SqlTool (command-line based). These are nice tools for working with our database. Meanwhile, you will find two convenient ant tasks in our `timexhsqldb.xml` file, `hsqldm` and `sqltool`, which can be used to start these two tools. For example, to start HSQL Database Manager, type the following on the command line:

```
ant -f timexhsqldb.xml hsqldm
```

After the Database Manager comes up, we can change the following parameters on the screen and work with our database in a GUI fashion, instantly (assuming the HSQLDB server is running in another window):

Type: HSQL Database Engine Server

URL: jdbc:hsqldb:hsql://localhost:9005/timex

HSQLDB Persistent and In-Memory Modes

Be sure to read about the various modes HSQLDB can run in (such as local versus server and in-memory versus persistent); we will use the server and persistent mode. For example, we could also use the very same HSQLDB database files (found under our `timex/data/` directory) as follows:

```
jdbc:hsqldb:file:${catalina.base}/webapps/timex/WEB-INF/data/timexdb
```

Incidentally, this is a feature that ties in nicely with the next section on bundling HSQLDB in an archive file.

Bundling HSQLDB in a Deployable Archive File

As an added benefit, HSQLDB has a small enough footprint to run entirely in memory. For example, we could deploy our sample application with HSQLDB, bundled in the same web archive (WAR) file, essentially making the WAR file a fully self-contained system with no need for an external database!

Personal Opinion: Data Is the Customer's Most Valuable Asset!

After all my years developing software, I still find that some people miss the whole point of Information Technology (IT). In my words, IT means technology for managing information. Information. As in data (databases).

Data is the customer's asset and hence the single most important component of a system. The data outlives most programs written to use it. This is precisely why the domain model, physical data model, and database refactoring are more important aspects of software development than, for example, *cool tools* or adding layers of unnecessary abstractions in your code.

When you're designing the database, one important thing to keep in mind is that the database can be used by multiple applications, not just a single, well-designed, object-oriented, n-tier application. For example, querying and reporting tools could also access the database for customer reports. So, as much as possible, the structure of the database should be somewhat independent of a single application. Furthermore, even the original application designed for the database can be retired after a few years, but the database will likely live on for a long time to come on.

For further reading on this matter, visit the agiledata.org website to learn more about database refactoring techniques. You may also want to visit the domaindrivendesign.org website, which is complementary to the AM website. For example, I found this line from an article by Eric Evans on this website, "the complexity that we should be tackling is the complexity of the domain itself—not the technical architecture, not the user interface, not even specific features."

To summarize, the data is the customer's asset, so focus on getting the domain model and database structure right using a combination of some upfront design and database refactoring as necessary.

Working with Hibernate

Hibernate has recently gained a lot of momentum in the world of Java database application development. Although products such as Toplink and others have been around for many years, Hibernate is open source (hence, free), stable, mature, well documented, and relatively easy to learn; these are probably just a few reasons why it is as popular as it is. Hibernate has been around for several years but was recently acquired by the JBoss group. (However, it continues to operate autonomously as an open source project.)

The Hibernate persistence framework can make working with relational databases using Java a pleasant experience. This is especially true if you have been developing using JDBC or using heavy-handed type entity beans. Defining the mappings can seem like a slight pain initially, but as you will see in later in this book, there are tools to generate these mapping files.

No Need for DAOs or DTOs

The extra work of defining mappings is well worth it because our persistence code will be cleaner and we will have automatically eliminated the need for Data Access Objects (DAOs), which typically are objects that know how to persist themselves. We also won't need Data Transfer Objects (DTOs), which are objects used to encapsulate business data and get transferred between layers of an application.

Supported Databases

As of the writing of this book, Hibernate supported the following databases (other databases are supported via community efforts):

- DB2
- HSQLDB
- Microsoft SQL Server
- MySQL

- Oracle
- PostgreSQL
- SAP DB
- Sybase
- TimesTen

Note

The databases are supported via Hibernate's *SQL Dialect* classes such as org.hibernate.dialect.HSQLDialect, org.hibernate.dialect.OracleDialect, org.hibernate.dialect.MySQLDialect, and so on.

Hibernate and EJB 3.x

One thing worth mentioning here is that members of the Hibernate/JBoss team are part of the EJB 3.0 expert group, a group that helped simplify the EJB specifications. It should come as no surprise, then, that the latest version of Hibernate supports the EJB 3.0 specification. However, we will not cover the EJB 3.0 here because it is outside the scope of this book. The focus of this book is on lighter-weight (and open source) frameworks, not heavy-handed specifications that require commercial application servers to use these features.

Simple Test for Hibernate Setup

Before diving into Hibernate concepts and terminology, let's look at a simple hibernate program and the setup involved. The following sections outline the steps required to get our first test program, SimpleTest, working. But first, let's take another look at the development directory structure we established in Chapter 3.

Figure 5.4 shows the development directory structure for Time Expression. It is important to review this again because we will create several files in this chapter and refer to them using their relative path names—for example, `model/Department.java` means file `Department.java` in the `timex/src/java/com/visualpatterns/timex/model/` directory.

Hibernate XML Files and Related Java Files

We will place the three types of Hibernate files (discussed next), a Hibernate configuration file, related Java classes, and table mapping files, in the same directory. This is the practice recommended in Hibernate documentation and examples.

The naming convention for the Hibernate mapping files is typically the name of the Java class name with a suffix of `.hbm.xml`—for example, `Timesheet.hbm.xml`.

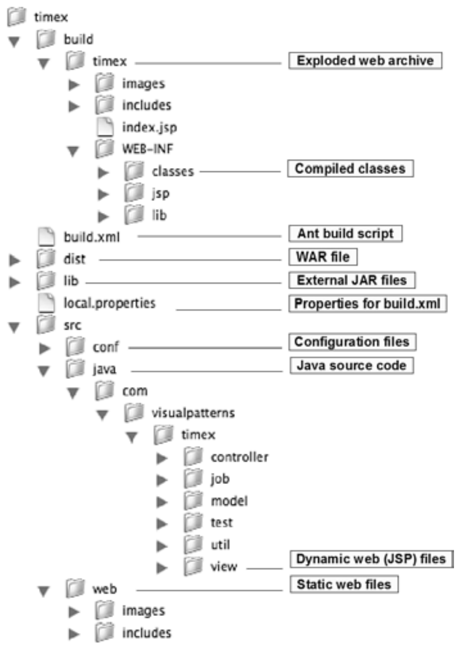


Figure 5.4 The Time Expression development directory structure.

Hibernate Configuration File (`hibernate.cfg.xml`)

First we will create a file named `hibernate.cfg.xml` in the `timex/src/java/com/visualpatterns/timex/model/` directory. This file will contain a SessionFactory definition (discussed later in this chapter) and reference to our first mapping file, `Department.hbm.xml`. Let's review some of the interesting lines from this file:

The following lines show the HSQLDB-related configuration (as we saw in `timexhsqldb.xml`, previously):

```
<property name="connection.driver_class">
    org.hsqldb.jdbcDriver
</property>
<property name="connection.url">
    jdbc:hsqldb:hsql://localhost:9005/timex
</property>
<property name="connection.username">sa</property>
```

The following lines from our `hibernate.cfg.xml` show the reference to the mapping files we will create in this chapter:

```
<mapping resource="Department.hbm.xml"/>
<mapping resource="Employee.hbm.xml"/>
<mapping resource="Timesheet.hbm.xml"/>
```

Using the complete `hibernate.cfg.xml` file, we will be able to create a Hibernate SessionFactory (discussed later in this chapter).

Mapping File (`Department.hbm.xml`)

We will create our first mapping file, `Department.hbm.xml`, in the `timex/src/java/com/visualpatterns/timex/model/` directory.

To keep things simple, I chose to start with the Department table because it is one of the simpler tables, and we will also use it in our slightly more complex example later in this chapter. Let's review the `Department.hbm.xml` file a bit closer.

The following line maps our Java class to the database table:

```
<class name="com.visualpatterns.timex.model.Department" table="Department">
```

The following line establishes `departmentCode` as the object id (as we discussed earlier) and the database primary key, and also maps the two:

```
<id name="departmentCode" column="departmentCode">
```

The `generator class="assigned"` value shown next tells Hibernate that we will be responsible for setting the value of this object id in our Java class, and Hibernate does not need to do anything special, such as get the next sequence from an auto-increment type column (for example, HSQLDB's identity data type):

```
<generator class="assigned"/>
```

This line maps the remainder of the Department table—that is, the `name` column to a name property in the `Department.java` class file (discussed next):

```
<property name="name" column="name"/>
```

Java Code

We will write two Java classes, one called `com.visualpatterns.timex.model.Department` and another called `com.visualpatterns.timex.test.HibernateTest`.

Department.java

The `Department.java` (under `src/java/com/visualpatterns/timex/model`) contains a simple JavaBean class, which provides *accessors* (*get methods* or *getters*) and *mutators* (*set methods* or *setters*) for these two variables:

```
String departmentCode;
String name;
```

HibernateTest.java

Now we will write some simple code to accomplish two things: test the Hibernate setup and also look at a basic example of how to use Hibernate. Let's review our `HibernateTest.java` file (under `src/java/com/visualpatterns/timex/test`) step-by-step.

The first few lines show how we obtain a Hibernate SessionFactory class and get a single Department record back for the departmentCode "IT":

```
SessionFactory sessionFactory = new Configuration().configure()
        .buildSessionFactory();
Session session = sessionFactory.getCurrentSession();
Transaction tx = session.beginTransaction();
Department department;
department = (Department) session.get(Department.class, "IT");
System.out.println("Name for IT = " + department.getName());
```

The following lines shows how to get and process a `java.util.List` of Department objects.

```
List departmentList = session.createQuery("from Department").list();
for (int i = 0; i < departmentList.size(); i++)
{
    department = (Department) departmentList.get(i);
    System.out.println("Row " + (i + 1) + "> " + department.getName()
            + " (" + department.getDepartmentCode() + ")");
}
```

The remaining notable line closes out theSessionFactory.

```
sessionFactory.close();
```

Note the `HibernateTest.java` file provides a simple example of using Hibernate by bunching all the code in a single (main) method. Later in this chapter, we will look at a better way of building a SessionFactory and subsequently obtaining Session objects from it.

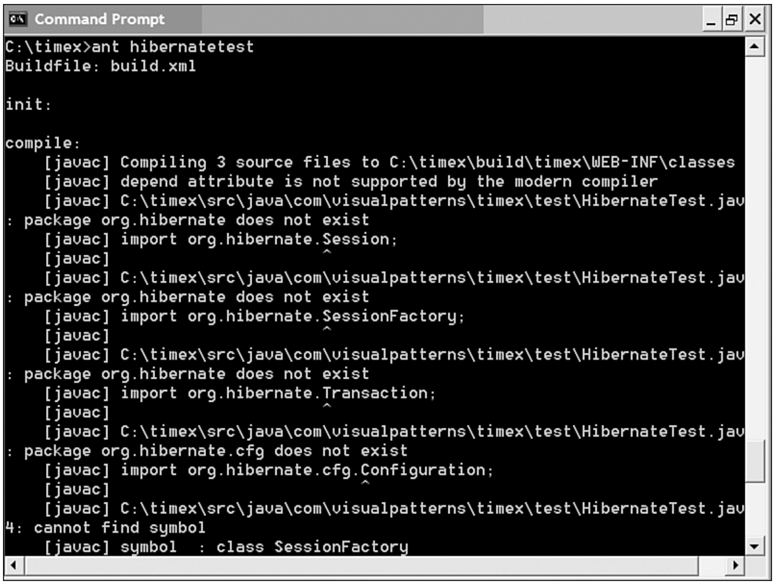
Now we are going to try running our test using our Ant `build.xml` file, introduced in Chapter 4, "Environment Setup: JDK, Ant, and JUnit." Our Ant target, `hibernatetest`, is as follows:

```
<target name="hibernatetest" depends="build">
  <java fork="true" classpathref="master-classpath"
        classname="com.visualpatterns.timex.test.HibernateTest"/>
</target>
```

To run our test, we need to:

- Change (`cd`) to the `timex/` (top-level) directory.
- Type the **`ant hibernatetest`** command, as shown in Figure 5.5.

Notice that there are errors on the screen, such as `package org.hibernate does not exist`. This means it is time to download and set up Hibernate in our environment!



```
C:\timex>ant hibernatetest
Buildfile: build.xml

init:

compile:
    [javac] Compiling 3 source files to C:\timex\build\timex\WEB-INF\classes
    [javac] depend attribute is not supported by the modern compiler
    [javac] C:\timex\src\java\com\visualpatterns\timex\test\HibernateTest.jav
: package org.hibernate does not exist
    [javac] import org.hibernate.Session;
    [javac]                      ^
    [javac] C:\timex\src\java\com\visualpatterns\timex\test\HibernateTest.jav
: package org.hibernate does not exist
    [javac] import org.hibernate.SessionFactory;
    [javac]                      ^
    [javac] C:\timex\src\java\com\visualpatterns\timex\test\HibernateTest.jav
: package org.hibernate does not exist
    [javac] import org.hibernate.Transaction;
    [javac]                      ^
    [javac] C:\timex\src\java\com\visualpatterns\timex\test\HibernateTest.jav
: package org.hibernate.cfg does not exist
    [javac] import org.hibernate.cfg.Configuration;
    [javac]                          ^
    [javac] C:\timex\src\java\com\visualpatterns\timex\test\HibernateTest.jav
4: cannot find symbol
    [javac] symbol  : class SessionFactory
```

Figure 5.5 Ant errors due to missing `lib/hibernate3.jar` file in classpath.

Installing Hibernate

Hibernate can be found at http://hibernate.org. At this point, we will follow the setup instructions provided on this site to download and install it to the recommended (or default) directory.

After we have the Hibernate installed, we will copy all the recommended libraries (for example, `hibernate3.jar` and `antlr.jar`) in the Hibernate documentation to the `rapidjava/lib` directory.

Note that I also needed to copy `ehcache-1.1.jar` and `antlr-2.7.6rc1.jar` (which was not mentioned in the Hibernate reference documentation at the time of this writing). Here is what I ended up with, in my `timex/lib/` directory:

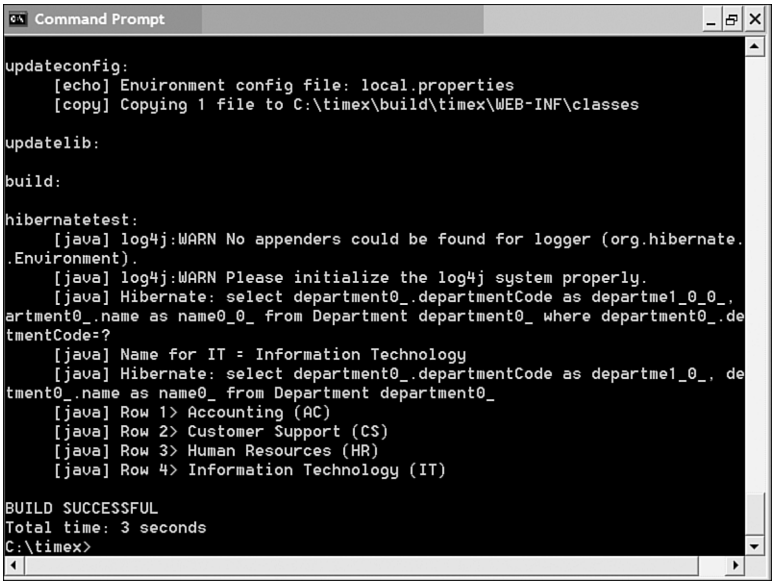
- `antlr-2.7.6rc1.jar`
- `asm-attrs.jar`
- `asm.jar`
- `cglib-2.1.3.jar`
- `commons-collections-2.1.1.jar`
- `commons-logging-1.0.4.jar`
- `dom4j-1.6.1.jar`

- `ehcache-1.1.jar`
- `hibernate3.jar`
- `jta.jar`
- `log4j-1.2.11.jar`

Before rerunning our test, we need to temporarily alter the `hibernate.cfg.xml` file. Because we have only `Department.hbm.xml` implemented, we need to temporarily remove the following lines (to conduct this test) from our `hibernate.cfg.xml` file:

```
<mapping resource="Timesheet.hbm.xml"/>
<mapping resource="Employee.hbm.xml"/>
```

Finally, we can rerun the **`ant hibernatetest`** command. If we run the ant as shown earlier in Figure 5.5, this time our command is successful, as shown in Figure 5.6!



```
updateconfig:
     [echo] Environment config file: local.properties
     [copy] Copying 1 file to C:\timex\build\timex\WEB-INF\classes

updatelib:

build:

hibernatetest:
     [java] log4j:WARN No appenders could be found for logger (org.hibernate.
.Environment).
     [java] log4j:WARN Please initialize the log4j system properly.
     [java] Hibernate: select department0_.departmentCode as departme1_0_0_,
artment0_.name as name0_0_ from Department department0_ where department0_.de
tmentCode=?
     [java] Name for IT = Information Technology
     [java] Hibernate: select department0_.departmentCode as departme1_0_, de
tment0_.name as name0_ from Department department0_
     [java] Row 1> Accounting (AC)
     [java] Row 2> Customer Support (CS)
     [java] Row 3> Human Resources (HR)
     [java] Row 4> Information Technology (IT)

BUILD SUCCESSFUL
Total time: 3 seconds
C:\timex>
```

Figure 5.6 Output of the HibernateTest class.

At this point, we can reinsert the following two lines into the `hibernate.cfg.xml` file:

```
<mapping resource="Timesheet.hbm.xml"/>
<mapping resource="Employee.hbm.xml"/>
```

Notice the log4j warning messages in Figure 5.6. We could ignore these because they are harmless. However, we'll go ahead and create a minimal `log4j.properties` file (available in this book's code zip file) in our `timex/src/conf` directory. Logging will be discussed in more detail in Chapter 9, "Logging, Debugging, Monitoring, and Profiling."

Hibernate Basics

Now that we have looked at a small preview of Hibernate-related Java code and XML files, let's get a high-level understanding of some basic Hibernate concepts before we look at slightly more complex Hibernate code for the Time Expression application.

Dialect

Hibernate provides dialect classes for the various supported databases mentioned earlier. This is essentially to ensure that the correct and most optimized SQL is used for the database product being used. For example, we are using the `org.hibernate.dialect.HSQLDialect` class for HSQLDB.

SessionFactory, Session, and Transaction

SessionFactory, as you might guess, manages a collection of Session objects. Each SessionFactory is mapped to a single database. The Session object essentially is a wrapper for a JDBC connection and is also a factory for Transaction objects. A Transaction is a wrapper for the underlying transaction, typically a JDBC transaction.

Built-In Connection Pooling

A side but important benefit of using Hibernate is that it provides built-in database connection pooling—hence, one less thing for us to worry about. Connection pooling, as you might be aware, is used to create a specified pool of open database connections (see `connection.pool_size` property in our `hibernate.cfg.xml`). By using a pool of connections, we can achieve more efficiency in our use of the database because existing open connections are reused. Furthermore, we get performance gains because we reuse open connections, thereby avoiding any delays in opening and closing database connections.

Working with Database Records (as Java Objects)

Several methods available in Hibernate's org.hibernate.Session interface enable us to work with database records as objects. The most notable methods are *save, load, get, update, merge, saveOrUpdate, delete,* and *createQuery* (several of these are demonstrated later in this chapter).

Another noteworthy interface to mention is org.hibernate.Query, which is returned by calling the Session.createQuery Hibernate method in our `HibernateTest.java` file. The Query class can be used to obtain a group of records in the form of a java.util. Collection object (for example, Hibernate provides mapping elements such as an array, set, bag, and others).

One last interface worth mentioning here is org.hibernate.Criteria, which can be used for database queries in an OO fashion, as an alternative to the Query class (which is HQL based).

We will look at examples of most of these interfaces and methods in this chapter.

Object States

Hibernate defines three states for object instances: *persistent*, *detached*, and *transient*. *Persistent* objects are ones that are currently associated with a Hibernate session; as soon as the session is closed (or the object is *evicted*), the objects become *detached*. Hibernate ensures that Java objects in a persistent state for an active session match the corresponding record(s) in the database. *Transient* objects are ones that are not (and most likely, never were) associated with Hibernate session and also do not have an object identity.

Data Types

Hibernate supports a large number of Java, SQL, and Hibernate types—more than you will probably need for a typical application. Also, you can have Hibernate automatically convert from one type to another by using a different type for a given property in a entity/class mapping file.

The following is a partial list of types supported: integer, long, short, float, double, character, byte, boolean, yes_no, true_false, string, date, time, timestamp, calendar, calendar_date, big_decimal, big_integer, locale, timezone, currency, class, binary, text, serializable, clob, and blob.

Hibernate Query Language (HQL)

HQL is Hibernate's robust SQL-like query language, which is not case sensitive. HQL has many of the features defined in ANSI SQL and beyond, because it is fully object-oriented and supports OO concepts such as inheritance, polymorphism, and more. The following are some basic clauses and features supported in HQL. You will see some examples of these later in the chapter:

- SELECT, UPDATE, DELETE, INSERT, FROM, WHERE, GROUP BY, ORDER BY
- Joins (inner, outer)
- Subqueries
- Aggregate functions (for example, sum and count)
- Expressions and functions (mathematical, string, date, internal functions, and more)

Furthermore, Hibernate provides methods that enable you to use native SQL (discussed in Chapter 10, "Beyond the Basics") for the somewhat rare occasions when HQL is insufficient.

You will see basic examples of HQL throughout this chapter.

Unique Object Identifier (`<id>`)

Hibernate requires mapped classes to identify a table's primary key via the `<id>` element. For example, the following code excerpt from our `Department.hbm.xml` file shows departmentCode defined as the primary key (for the Department table mapping):

```
<id name="departmentCode" column="departmentCode">
    <generator class="assigned"/>
</id>
```

Notice the generator class of type "assigned" in this code excerpt; this means the application will provide a value for this id property prior to any database operations on this object.

Hibernate provides several ways to generate unique ids for inserted records, including increment, identity, sequence, hilo, seqhilo, uuid, guid, native, assigned, select, and foreign. The hibernate reference documentation provides ample explanation of each of these. We will use the assigned and identity generators for our examples.

Mandatory Hibernate Transactions

According to the Hibernate documentation related to working with this API, "transactions are never optional, all communication with a database has to occur inside a transaction, no matter if you read or write data."

Hence, you will find the following types of calls in all our Hibernate-related code:

- `session.beginTransaction()`
- `session.getTransaction().commit()`
- `session.getTransaction().rollback()`

HibernateUtil.java

The Hibernate reference documentation recommends the use of a helper class (named `HibernateUtil`, for example) for setting up a SessionFactory and providing access to it (via a getter method).

A sample `HibernateUtil.java` class file can be found under the `timex\src\java\com\visualpatterns\timex\test` directory. This helper class contains only a few lines of actual code. The first notable lines are the following, which build a SessionFactory object:

```
sessionFactory = new Configuration().configure()
                                    .buildSessionFactory();
```

The only other interesting code in this class is a convenient getter method to return the SessionFactory, as shown here:

```
public static SessionFactory getSessionFactory()
{
   return sessionFactory;
}
```

Then we can obtain a Session object, as demonstrated in this code snippet, which fetches a list of Department database objects:

```
List departmentList=null;
Session session = HibernateUtil.getSessionFactory().getCurrentSession();
session.beginTransaction();
departmentList = session.createQuery("from Department ORDER BY name").list();
session.getTransaction().commit();
```

Further Reading

Again, we have looked only at high-level explanations of the various Hibernate concepts. I will refer you to the Hibernate documentation (found on their website) for detailed information on the concepts discussed here; however, we will be using many of these concepts in our examples, so I will provide additional explanations and code examples along the way.

Now that we have the basics covered, let's begin doing some real work by implementing a full-fledged example to exercise some of Hibernate's many features.

Developing TimesheetManager.java Using Hibernate

If you think about the business requirements of Time Expression, combined with the prototyped screens we looked at in Chapter 2, "The Sample Application: An Online Timesheet System," and the database model shown in Figure 5.2, you can easily see that the Timesheet table is at the heart of our sample application (Time Expression). Hence, I've chosen to use this table as an example in this chapter because our requirements will exercise both read and write type database operations on this table. For example, most screens in Time Expression will either modify data in this table or fetch data from it.

Based on the class/package design we defined in Chapter 3 and the Hibernate configuration files needed, here are the files we will end up with for the example to follow:

- `test/TimesheetManagerTest.java`
- `model/TimesheetManager.java`
- `model/Timesheet.java`
- `model/Timesheet.hbm.xml`

The following sections review each of these files in more detail.

`TimesheetManagerTest.java`

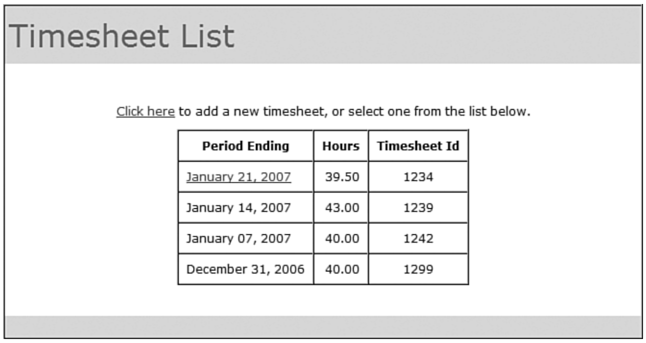
Let's begin by writing a little JUnit test class. Remember, write tests first whenever possible. We already looked at the reasons and benefits of writing tests *first* in Chapter 4, so I won't repeat the same information here.

The only class we need to write a unit test for is TimesheetManager because `Timesheet.java` is a JavaBean and hence has no *real* logic in its methods (just setters and getters).

If we analyze the following two screens from Chapter 2, we can come up with the type of functionality we need our TimesheetManager class to provide:

- Timesheet List—A list of Timesheet records for a given employeeId.
- Enter Hours—The capability to insert and update a single Timesheet record.

Let's look at an example of the functionality we need and how we might implement it. We know we need a list of Timesheet records for the Timesheet List screen (shown in Figure 5.7); this list will be fetched from the database using an employeeId (that is, the employee who is logged in) and also for the current pay period. Hence, I can already picture a method in the TimesheetManager class with a signature that looks something like this: `getTimesheet(int employeeId, Date periodEndingDate)`. So, we can easily write a test case (method) such as `testGetByEmployeeIdAndPeriod()`.



Timesheet List

Click here to add a new timesheet, or select one from the list below.

Period Ending	Hours	Timesheet Id
January 21, 2007	39.50	1234
January 14, 2007	43.00	1239
January 07, 2007	40.00	1242
December 31, 2006	40.00	1299

Figure 5.7 Timesheet list prototype screen (from Chapter 2).

This book's code zip file contains `TimesheetManagerTest.java`, a complete JUnit test suite class to exercise all the methods in our `TimesheetManager` class.

Note

Note that I didn't write the entire test class in one shot. As I mentioned earlier, unit testing and coding happen in the same sitting. So you would write a little test code (perhaps a few lines in a single method), write a little implementation code, compile, try, and repeat the steps until the method has been fully implemented. The idea is to write small methods, which can be tested relatively easily. This technique also enables us to write the minimal code required to satisfy our user requirements (nothing more, nothing less).

Let's review some of the test code behind this class next; we won't walk through the entire file because we only require fetching (get) of Timesheet objects and saving of individual ones, so let's review methods related to these operations next.

`testGetByEmployeeId()`

Let's start with the `testGetByEmployeeId()` method. The first few lines of this code ensure that we get a java.util.List of Timesheet objects back before proceeding:

```
List timesheetList = timesheetManager.getTimesheets();
assertNotNull(timesheetList);
assertTrue(timesheetList.size() > 0);
```

After we know we have at least one Timesheet object, we can fetch Timesheet records using the employeeId found in the first Timesheet object, as shown here:

```
int employeeId=((Timesheet)timesheetList.get(0)).getEmployeeId();
timesheetList = timesheetManager.getTimesheets(employeeId);
assertNotNull(timesheetList);
```

Now we can simply test each Timesheet object in the list to ensure that these records belong to the employeeId we requested, as demonstrated next:

```
Timesheet timesheet;
for (int i=0; i < timesheetList.size(); i++)
{
    timesheet = (Timesheet)timesheetList.get(i);
    assertEquals(employeeId, timesheet.getEmployeeId());
    System.out.println(">>>> Department name = "
                      + timesheet.getDepartment().getName());
}
```

`testSaveSingle()`

Let's review one more test method from our `TimesheetManagerTest.java` file, `testSaveSingle`. The first half of this method sets up a Timesheet object to save; however, the following lines are worth exploring:

```
timesheetManager.saveTimesheet(timesheet);
Timesheet timesheet2 = timesheetManager.getTimesheet(EMPLOYEE_ID,
        periodEndingDate);
assertEquals(timesheet2.getEmployeeId(), timesheet.getEmployeeId());
assertEquals(timesheet2.getStatusCode(), "P");
```

We essentially save a Timesheet object, and then fetch it back from the database and compare the two objects' attributes using the `assertEquals` method.

TimesheetManager.java

Next we will look at our bread-and-butter class (so to speak). We will use this class extensively in Chapter 7 when we implement our user interfaces (for example, Timesheet List and Enter Hours).

The key methods we will review here are `getTimesheets`, `getTimesheet`, and `saveTimesheet`.

Let's start with the `TimesheetManager. getTimesheets(int employeeId)` method. The key lines code essentially get a java.util.List of Timesheet objects from the database using the Hibernate `Session.createQuery` method, as shown here:

```
timesheetList = session.createQuery(
        "from Timesheet" + " where employeeId = ?").setInteger(0,
        employeeId).list();
```

The next method, `TimesheetManager.getTimesheet(int employeeId, Date periodEndingDate)`, is slightly different from the `getTimesheets(int employeeId)` method we just looked at; the key difference is the use of Hibernate's `uniqueResult` method, which is a convenient method to get only one object back from a query. The following code shows the notable lines from our *getTimesheet* method:

```
timesheet = (Timesheet) session.createQuery(
        "from Timesheet" + " where employeeId = ?"
                + " and periodEndingDate = ?").setInteger(0,
        employeeId).setDate(1, periodEndingDate).uniqueResult();
```

The last method in TimesheetManager that we will review here is `saveTimesheet(Timesheet timesheet)`. This is a very straightforward method, and the only code worth showing here is the Hibernate's `session.saveOrUpdate` method, which either does an INSERT or UPDATE underneath the covers, depending on whether the record exists in the database:

```
session.saveOrUpdate(timesheet)
```

`Timesheet.java` (and `Timesheet.hbm.xml`)

Before we can successfully compile and use `TimesheetManager.java`, we need to quickly write files it relies on, namely `Timesheet.java` and its mapping file, `Timesheet.hbm.xml` (both available in this book's code file). There is not much to these files; the Java code is a simple JavaBean and the XML file simply maps the bean's properties to the appropriate database columns.

Employee.* and DepartmentManager.java

The other files provided in this book's code zip file but not explicitly discussed here include the following as we will need these to implement our first five user stories (page 36).

At this point, we will create these files in our `src/java/com/visualpatterns/timex/model` directory:

- `DepartmentManager.java`
- `Employee.hbm.xml`
- `Employee.java`
- `EmployeeManager.java`

Files Required in Classpath

The various Hibernate files, such as the `hibernate.cfg.xml` and mapping files (for example, `Department.hbm.xml`) need to be in the CLASSPATH; accordingly our Ant script, `build.xml`, automatically copies these files to the `timex/build/timex/WEB-INF/classes` directory during a build process.

Running the Test Suite Using Ant

Now we can run our test suite (TimesheetManagerTest) discussed previously. However, before we can run the test suite, we need to run HSQLDB in server mode. We can either do it manually as shown here:

```
java -cp /hsqldb/lib/hsqldb.jar org.hsqldb.Server
➥  -database.0 data\time xdb -dbname.0 timex -port 9005
```

Note

I've assumed the HSQLDB directory is installed under the root directory (that is, `/hsqldb`); alter the `java` command shown here according to your environment.

Or we can run it using our handy Ant script, as follows:

```
ant -f timexhsqldb.xml starthsql
```

After we start up the HSQLDB server successfully and have all our files created in the correct directories, we can test our new classes by typing **`ant rebuild test`** on the command line (from the timex/ top-level directory). The output of this command is shown Figure 5.8.

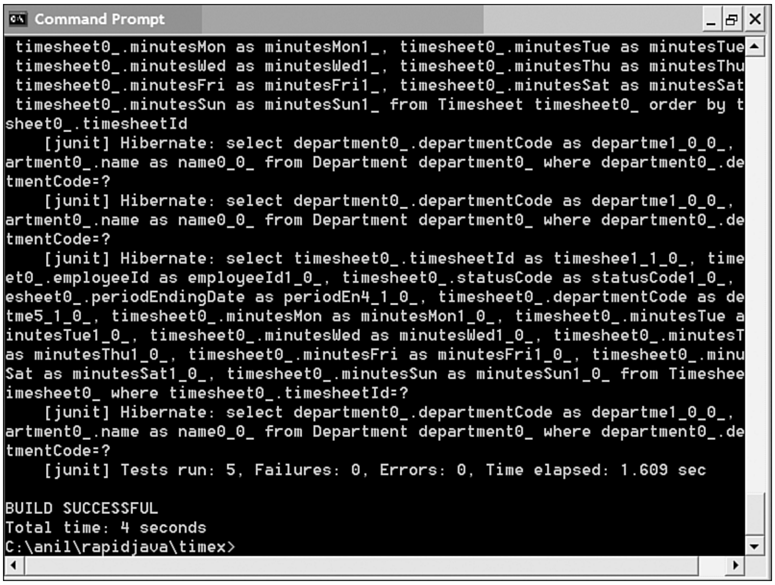
Deleting Records

We covered database reads and writes. However, we have not covered deleting records, a basic need in any CRUD application. Deleting records is not part of our Time Expression application's requirement. Nevertheless, I've added one for demonstration purposes. The only thing different from what we have already seen is the Session.delete, which deletes a database record (object) and Session.load, which fetches a record from the database, as demonstrated here:

```
session.delete(session.load(Timesheet.class, new Integer(timesheetId)));
```

Alternatively, the delete code can be written using the `Query.executeUpdate()` method, useful for bulk processing, as shown here:

```
int updated = session.createQuery("DELETE from Timesheet"
                              + " where timesheetId = ?")
                     .setInteger(0, timesheetId)
                     .executeUpdate();
```



```
 timesheet0_.minutesMon as minutesMon1_, timesheet0_.minutesTue as minutesTue
 timesheet0_.minutesWed as minutesWed1_, timesheet0_.minutesThu as minutesThu
 timesheet0_.minutesFri as minutesFri1_, timesheet0_.minutesSat as minutesSat
 timesheet0_.minutesSun as minutesSun1_ from Timesheet timesheet0_ order by t
sheet0_.timesheetId
    [junit] Hibernate: select department0_.departmentCode as departme1_0_0_,
artment0_.name as name0_0_ from Department department0_ where department0_.de
tmentCode=?
    [junit] Hibernate: select department0_.departmentCode as departme1_0_0_,
artment0_.name as name0_0_ from Department department0_ where department0_.de
tmentCode=?
    [junit] Hibernate: select timesheet0_.timesheetId as timeshee1_1_0_, time
et0_.employeeId as employeeId1_0_, timesheet0_.statusCode as statusCode1_0_,
esheet0_.periodEndingDate as periodEn4_1_0_, timesheet0_.departmentCode as de
tme5_1_0_, timesheet0_.minutesMon as minutesMon1_0_, timesheet0_.minutesTue a
inutesTue1_0_, timesheet0_.minutesWed as minutesWed1_0_, timesheet0_.minutesT
as minutesThu1_0_, timesheet0_.minutesFri as minutesFri1_0_, timesheet0_.minu
Sat as minutesSat1_0_, timesheet0_.minutesSun as minutesSun1_0_ from Timeshee
imesheet0_ where timesheet0_.timesheetId=?
    [junit] Hibernate: select department0_.departmentCode as departme1_0_0_,
artment0_.name as name0_0_ from Department department0_ where department0_.de
tmentCode=?
    [junit] Tests run: 5, Failures: 0, Errors: 0, Time elapsed: 1.609 sec

BUILD SUCCESSFUL
Total time: 4 seconds
C:\anil\rapidjava\timex>
```

Figure 5.8 Running JUnit test suites via Ant.

Criteria Queries

Until now, we have utilized Hibernate's Query interface (via the `Session.createQuery` method) to fetch records from the database. However, there is a slightly more dynamic, and arguably cleaner, way of fetching records using Hibernate's Criteria interface. This provides a more object-oriented approach, which can result in fewer bugs because it can be type checked and can avoid potential HQL-related syntax errors/exceptions. This method is cleaner because developer does use a more object-oriented approach—more objects rather than simple text queries, hence, more type checking, hence fewer bugs, especially, QueryExceptions. It could be a problem though if query syntax is too complex.

The Criteria interface can be obtained using the `Session.createCriteria` method as shown in the following code excerpt:

```
timesheetList = session.createCriteria(Timesheet.class)
                      .add(Restrictions.eq("employeeId", employeeId))
                      .list();
```

In addition, Hibernate provides several classes in the org.hibernate.criterion package, which work with the Criteria interface to provide robust querying functionality using objects. Some examples of these classes are Restrictions, Order, Junction, Distinct, and several others.

Exception Handling

Most of the database-related exceptions thrown while using the Hibernate API are wrapped inside `org.hibernate.HibernateException`; more details on this can be found in Hibernate's reference manual. Meanwhile, the following strategy is recommended in Hibernate's reference manual for handling database exceptions:

"If the Session throws an exception (including any SQLException), you should immediately rollback the database transaction, call Session.close() and discard the Session instance. Certain methods of Session will not leave the session in a consistent state. No exception thrown by Hibernate can be treated as recoverable. Ensure that the Session will be closed by calling close() in a finally block."

We are following these guidelines, of course. However, you might also have noticed in our model code that we rethrow any caught exceptions. It is generally a good idea to pass exceptions up the call stack, so the top-level methods can determine how to process the exception. We will discuss exception handling in more detail in Chapter 10.

I think the problem with this piece of code may be that the developer will never know what exception has actually occurred, because all we do in the catch block is roll back the transaction. Some kind of logging mechanism or exception propagation mechanism to outer callers is necessary to make sure the exception is noticed and handled properly (otherwise, we'll never know about the failure details, except by knowing that timesheet did not get saved).

Other Hibernate Features

Up to now in this chapter, we have looked at some basic Hibernate features. Next, let's review some additional, slightly more advanced, Hibernate concepts.

Associations

The physical database design, the mapping, and the Java classes for Time Expression are all fairly straightforward. We have essentially used a one-class-per-table mapping strategy to keep the design simple and fast. However, we have utilized a many-to-one association to fetch the corresponding Department record, which we can use to obtain the name of the department. We will need this functionality on various Time Expression screens that display the full Department.name (versus just the Department.departmentCode).

Let's dissect the Java code and XML mapping related to this association.

First, the persistent attribute Java bean code can be found in `Department.java` file; an excerpt of it, is shown here:

```
private Department department;

public Department getDepartment()
{
     return department;
}
```

```
public void setDepartment(Department department)
{
     this.department = department;
}
```

Secondly, the many-to-one mapping can be found in our `Timesheet.hbm.xml` file:

```
<many-to-one name="department" column="departmentCode"
             class="com.visualpatterns.timex.model.Department"
             lazy="false" not-found="ignore" cascade="none"
             insert="false" update="false"/>
```

Finally, the code on how we obtain the department name can be found in our `TimesheetManagerTest.java` file:

```
System.out.println(">>>> Department name = " +
timesheet.getDepartment().getName());
```

Locking Objects (Concurrency Control)

Database locking can apply to any database applications, not just ones based on ORM technologies. There are two common strategies when dealing with updates to database records, *pessimistic locking* and *optimistic locking*.

Optimistic locking is more scalable than pessimistic locking when dealing with a highly concurrent environment. However pessimistic locking is a better solution for situations where the possibility of simultaneous updates to the same data by multiple sources (for example, users) is common, hence making the possibility of "data clobbering," a likely scenario. Let's look at a brief explanation of each of these two locking strategies.

Pessimistic locking is when you want to reserve a record for exclusive update by locking the database record (or entire table). Hibernate supports pessimistic locking (using the underlying database, not in-memory) via one of the following methods:

- `Session.get`
- `Session.load`
- `Session.lock`
- `Session.refresh`
- `Query.setLockMode`

Although each of the methods accepts different parameters, the one common parameter across all is the `LockMode` class, which provides various locking modes such as NONE, READ, UPGRADE, UPGRADE_NOWAIT, and WRITE. For example, to obtain a *Timesheet* record for updating, we could use the following code (assuming the underlying database supports locking):

```
public Timesheet getTimesheetWithLock(int timesheetId)
{
    Session session =
        HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();
    Timesheet timesheet = (Timesheet)session.get(Timesheet.class,
              new Integer(timesheetId), LockMode.UPGRADE);
    session.getTransaction().commit();
    session.close();0

   return timesheet;
}
```

Optimistic locking means that you will not lock a given database record or table and instead check a column/property of some sort (for example, a timestamp column) to ensure the data has not changed since you read it. Hibernate supports this using a `version` property, which can either be checked manually by the application or automatically by Hibernate for a given session. For example, the following code excerpt is taken verbatim out of the Hibernate reference documentation and shows how an application can manually compare the `oldVersion` with the current version using a getter method (for example, `getVersion`):

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

`StaleObjectStateException`, shown in the previous example, is an exception in the org.hibernate package.

Lots More Hibernate

Although we covered a lot of material in this chapter, there is much more to Hibernate. However, as I mentioned earlier, entire books exist on Hibernate, and we cannot cover everything about this technology in one chapter. Nevertheless, I have given you enough here to build some reasonably complex applications using Java, Hibernate, and relational databases.

Other Hibernate advanced topics not covered here, but ones you might want to explore, include

- Advanced mappings (for example, bidirectional associations, ternary associations, sorted collections, component mapping, inheritance mapping, and more)
- Advanced HQL
- Annotations (and XDoclet)
- Filters

- Hibernate SchemaExport utility
- Inheritance mapping
- Interceptors
- Locking objects
- Performance improvement strategies (for example, fetching strategies, second-level cache)
- Scrollable iteration and pagination
- Transaction management (advanced topics)
- Other areas such as using stored procedures, native SQL, and more

Summary

In this chapter, we developed the classes we need to implement functionality for the first five user stories (page 36). Furthermore, these classes are directly relevant to the code we will develop in Chapter 7 and the scheduled jobs in Chapter 10.

In this chapter, we accomplished the following:

- Learned what object-relational mapping technology is and the benefits it offers
- Installed HSQLDB
- Designed our database
- Used DDL script to create our database tables and some test data
- Setup Hibernate, covered its basic concepts, and began working with it
- Developed a simple and then a slightly more complex example (along with a corresponding unit test suite class) of using Hibernate for Time Expression's Department and Timesheet tables
- Discussed advanced Hibernate topics and other features for you to explore (should you need them)

However, we are not done with Hibernate just yet! For example, we will use some of the classes coded in this chapter in our web application in the next chapter. In addition, I will demonstrate how we can use an Eclipse plug-in to generate the Hibernate mapping files (in Chapter 8, "The Eclipse Phenomenon!").

For now, we are ready to dive into the next two chapters where we enter the world of user interfaces by using the Spring MVC web framework to develop our web UI.

We will also begin working with the Eclipse SDK in Chapter 8 and see how much time IDEs can save. Till now, I have intentionally used the command line because I truly believe learning the fundamentals first by using the manual way will help you better understand how things work behind the scenes. It can also help you drop back to the command line in case the IDE does not provide a certain functionality or it has a known bug.

Recommended Resources

The following websites are relevant to and provide additional information on the topics discussed in this chapter:

- Agile Data http://www.agiledata.org
- Agile Modeling http://www.agilemodeling.com
- Apache ObJectRelationalBridge (OJB) http://db.apache.org/ojb/
- Introduction to Concurrency Control http://www.agiledata.org/essays/concurrencyControl.html
- Cocobase http://www.thoughtinc.com/
- Database refactoring http://www.agiledata.org/essays/databaseRefactoring.html
- Domain-Driven Design http://domaindrivendesign.org/
- Hibernate forums http://forum.hibernate.org/
- Hibernate http://hibernate.org/
- HSQLDB http://hsqldb.org/
- iBATIS Java http://ibatis.apache.org
- JDO and EJB 3.0 http://java.sun.com
- JORM http://jorm.objectweb.org/
- Object Data Management Group http://www.odmg.org/
- SimpleORM http://www.simpleorm.org/
- The Castor Project http://www.castor.org/
- Cocobase http://www.thoughtinc.com/