Chapter XXV
Hibernate:
A Full Object Relational Mapping Service

Allan M. Hart
Minnesota State University, Mankato, USA

ABSTRACT

This chapter presents a brief overview of the object/relational mapping service known as Hibernate. Based on work provided in the book Java Persistence with Hibernate, it is argued that the paradigm mismatch problem consists of five problems: the problem of granularity, the problem of subtypes, the problem of identity, the problem of associations, and the problem of data navigation. It is argued that Hibernate, if it is to be considered a successful object/relational mapping service, must solve the paradigm mismatch problem and, hence, each of the five problems noted above. A simplified version of an order entry system is presented together with the mapping files required to store persistent objects to a database. Examples are given for one-to-one, one-to-many, and many-to-many mappings. The distinction between value and entity types is explained and the mapping technique required for value types is introduced into the order entry system application. The n+1 selects problem is explained and a strategy for solving that problem using Hibernate's support for lazy, batch, and eager fetching strategies is discussed.

INTRODUCTION

The purpose of this chapter is to provide the reader with an introduction to Hibernate. Hibernate, as described at its Web site, is "a powerful, high performance object/relational persistencee and query service" (Hibernate, 2008).

Hibernate is one among a number of so-called persistence frameworks. Other notables include TopLink, iBATIS, and Java data objects (JDO) (Oracle TopLink, 2008; IBATis, 2008; JDO, 2008). The basic responsibility of any persistence framework is to manage persistent data, that is, data that needs to be saved to persistent storage (usually a relational database) from one invocation of the application to the next. Not all objects created by a given application constitute persistent data but many of them do. For example, in an ecommerce application, information regarding the customer's name, address, and credit card information as well as the particular products the customer has ordered (and the quantity of each), constitutes data that needs to be saved to persistent storage.

Persistence frameworks can generally be divided into those that are based on an approach known as object/relational mapping (ORM) and those that are not. Both Hibernate and TopLink, for example, are correctly classified as ORM services. iBATIS, on the other hand, though often listed as an ORM service, is, strictly speaking, not an ORM service at all, but rather a data mapping service. With an ORM service like Hibernate, what are mapped, *very* roughly speaking, are classes to tables. Instances of classes become rows in a database table and associations between classes become foreign key constraints between database tables. What are mapped with iBATIS's data mapping services, on the other hand, are not tables to classes, but rather the parameters and results of SQL statements to classes.[1]

The need for ORM services grew out of the realization over the last several decades that a paradigm mismatch problem exists between the world of object-oriented programming languages and relational databases. In this chapter we will explore some of the details of this problem as well as the ways in which an ORM service like Hibernate attempts to solve the problem.

BACKGROUND

The evolution of programming languages over the last 50 years has seen a large growth not only in the *number* of different programming languages but also in the number of different *types* of programming languages. During the 60s procedural languages such as FORTRAN, BASIC, and Pascal were the rage. While object-oriented programming (OOP) was in its infancy in the 60s, its first implementation language, Smalltalk, appeared. During the 70s and 80s SmallTalk continued to evolve and newer OOP languages such as Object Pascal and C++ appeared. The appearance of Java during the mid 90s solidified OOP's position as the predominant programming language paradigm. During this same period, changes in the database world were also afoot. Edgar F. Codd's seminal paper "A Relational Model of Data for Large Shared Data Banks" was published in 1970. This paper triggered much work in the 70s and, during this time, implementations of the relational model began to appear. Soon, thereafter, relational databases had largely replaced their hierarchical and network predecessors.

As the development of OOP languages and relational database management systems proceeded, it soon became clear that there was a paradigm mismatch problem.[2] The details of the problem will be presented in the next section. For now, suffice it to say that there were primarily two responses to this problem. Some have argued that the relational model should be abandoned and that object-oriented databases should be embraced. Others have argued that relational databases should instead be expanded to include at least some of the features found in object-oriented programming languages, for example, user-defined types and inheritance. This bifurcation in the historical road down which database development has gone, constitutes, one might say, a fork in that road. During the 90s, one saw much development being done in the area of pure object-oriented databases. However, while this work deserves much praise, the effort did not bear much fruit. While pure object-oriented database management systems were developed, none of them caught on in the marketplace and much of that effort has now been abandoned. During roughly the same time period, much effort was also expended toward the other fork. The notion of a user defined type (UDT) was brought into the database world with implementations being provided for both Oracle and SQL server. While this has served to alleviate the paradigm mismatch problem to some degree (at least for those platforms), it has not successfully overcome the problem in all of its detail. Moreover, the solutions provided are not portable among those platforms.

PARADIGM MISMATCH PROBLEM

In order to gain an understanding of the way in which an ORM service like Hibernate solves the paradigm mismatch problem, we need first to gain a better understanding of just what the problem is. Christian Bauer and Gavin King (2007, pp. 12-18) break down the paradigm mismatch problem into five essential parts. The first part is the problem of granularity. Here, we reuse their example of an address to illustrate this part of the problem. Suppose, for example, that in our domain model we have Customer and Orders classes as outlined in Listing 1.

Listing 1 Customer and Orders

```
public class Customer {
 private String customerName;
 private String address;
 private Set orders;

 //other stuff
 …
}

public class Orders {
 private int orderNum;
 private Date orderDate
 private Customer customer;

 //other stuff
 …
}
```

The database tables that we might design to store persistent instances of the above classes can be defined as in Listing 2.

Listing 2 CUSTOMER and ORDERS tables

```
create table CUSTOMER (
CUSTOMERNAME VARCHAR(20) PRIMARY KEY,
ADDRESS VARCHAR(110)
);

create table ORDERS (
ORDERNUM INT PRIMARY KEY,
ORDERDATE DATE NOT NULL,
CUSTOMER VARCHAR(20) FOREIGN KEY REFER-
ENCES CUSTOMER
)
```

While this simple design seems quite benign, there is a problem with the representation of the address field. Realistically, this should be broken down into street, city, state, country, and ZIP code fields. Those fields *could* simply be added to the Customer class as a replacement for the address field but, because it is likely that *other* classes (e.g., Employee, Shipper, Vendor, etc.) will require an address field as well, it makes more sense, from an object-oriented (OO) point of view, to simply create another class named address and to include those fields in that class. The address field in the customer class then becomes a reference to an instance of the Address class. The Address and (modified) Customer class appear in Listing 3.

Listing 3 Address Class

```
public class Address {
 String street;
 String city;
 String state;
 String country;
 String zip;

 //maybe other stuff
 …
}

public class Customer {
 private String customerName;
 private Address; // this might be a set
of Addresses
 private Set orders;

 //other stuff
 …
}
```

What changes need to be made to the relational schemata to accommodate the above change to the object domain model? Should we create a separate ADDRESS table with a foreign key reference from the CUSTOMER table? Doing so might *seem* appropriate but it is not the norm in the *relational* model. Rather, the norm is to modify the CUSTOMER table as in Listing 4.

Listing 4 CUSTOMER Table

```
create table CUSTOMER(
CUSTOMERNAME VARCHAR(20) PRIMARY KEY,
STREET VARCHAR(50),
CITY VARCHAR(20),
STATE CHAR(2),
ZIP VARCHAR(10),
COUNTRY VARCHAR(20)
)
```

By not creating a separate ADDRESS table we avoid the cost involved in joining the CUSTOMER table to the ADDRESS table every time we want to look up a user's address.

As Bauer and King note (2007, p. 12), some database management systems include a facility for creating user defined types (UDTs). For such systems, it might be possible to define an ADDRESS UDT and to include it as a single column in the USER table. Doing so would restore, to some extent, the symmetry between our object domain model and our relational model (the Address class would map to a *single* column in the CUSTOMER table). However, UDT support in the database industry is neither uniform nor portable.

The above example illustrates the problem of granularity. Granularity concerns the relative size of our objects (Bauer & King, 2007, p. 12). On the object domain side, our classes come in different levels of granularity. On the one hand, we have a coarse grained class named Customer and, on the other hand, we have a finer grained class named Address. In a more realistic object domain model we might have many classes each exhibiting different levels of granularity. By contrast, on the relational side, we have a very limited amount of granularity. This contrast is the inevitable result of two very different type systems. Java's type system is quite flexible and rich; every new class represents a new data type. The type system found in RDBMSs is, on the other hand, much less flexible. Since it is obviously impossible to impose the more flexible system on the less flexible, too often the reverse is true, that is, the less flexible is imposed on the more flexible. The net result is that the object domain model becomes bound by the relational model. In other words, the object domain model becomes a mere reflection of the relational model. Inevitably, the application developer loses much of the advantage of working in an OOP language environment.

The second part of the paradigm mismatch problem noted by Bauer and King (2007, pp. 13-14) is the problem of subtypes. As is well known, one of the biggest advantages of OOP languages over procedural languages is inheritance. Given a preexisting class, one can define a class (called a subclass) that inherits the properties and methods of the preexisting class along with new properties and methods unique to the subclass. Doing so goes a long way toward preventing us from "reinventing the wheel." Instead of starting from scratch for each new application, we can reuse classes designed originally for other applications. These new classes are not "second rate" classes somehow detached from the language. They are first rate classes that are effectively part of the language itself. To illustrate the idea, suppose that we want to countenance in our object domain model two different kinds of orders: corporate customer orders and ordinary customer orders. Corporate orders come from corporations, may be eligible for special discounts (perhaps because corporate orders are usually large volume orders), can be paid by a purchase order, require information regarding a particular corporation office, and so forth. Ordinary customer orders are not eligible for special discounts, can be paid only by cash, check, or credit card, and require no special

information beyond what is already available in the CUSTOMER table. Because of inheritance, the required adjustment to our object domain model is quite simple. Perhaps we make the Orders class an abstract base class and develop concrete Corporate and Ordinary subclasses from it. Listing 5 shows the required changes.

Listing 5

```
public abstract class Orders {
 private int orderNum;
 private int Date orderDate;
 private Customer customer;

 //other stuff
 …
}

public class Corporate extends Orders {
 private float discount;
 private String po;
 private String office;

 //other fields and methods unique to a
corporate order
 …
}

public class Ordinary extends Orders {
 //fields and methods unique to ordinary
orders
 …
}
```

Subclassing in this way is quite usual in an object domain model. Our customer class has an association to our (now) abstract orders class. Because our Orders class is subclassed by the Corporate and Ordinary classes, the association is *polymorphic*. During runtime, a given instance of Customer might be associated with instances of either or both of Corporate and Ordinary. So how do we make the adjustments on the relational model side to accommodate our new object domain model? Bauer and King (2007) answer this question as follows:

We can take the short route here and observe that SQL database products do not generally implement type or table inheritance, and if they do implement it, they do not follow a standard syntax and usually expose you to data integrity problems (limited integrity rules for updatable views).
SQL databases also lack an obvious way (or at least a standardized way) to represent a polymorphic association. A foreign key constraint refers to exactly one target table; it isn't straightforward to define a foreign key that refers to multiple tables. We'd have to write a procedural constraint to enforce this kind of integrity rule.
The result of this mismatch of subtypes is that the inheritance structure in your model must be persisted in an SQL database that doesn't offer an inheritance strategy. (p. 14)

The third part of the paradigm mismatch problem noted by Bauer and King (2007, pp. 14-16) is the problem of identity. In an OOP language like Java, there are two different notions of sameness for objects. These are provided by the == operator on the one hand and by the implementation of the equals() method on the other hand. Scores of beginning Java programmers have had their programs slashed to pieces by their instructors because they checked for the equality of two objects using == instead of equals(). The == operator, when used to compare objects, returns true if there is exactly one object and the references (as in A == B where A and B are references) involved refer to one and the same object. The equals() method, on the other hand, returns true if the implementation contract of the equals() method is satisfied. The following code snippet gives the idea.

```
String A = new String("Now is the time");
String B = A;
System.out.println(A==B); // prints true
since these is only one object
```

```
…
String C = new String("Now is the time");
System.out.println(A==C); // prints false
System.out.println(A.equals(C)); // prints
true since the contract of
                    // the equals() method
for String is
                    // satisfied
```

Database identity, in contrast to the above, is rather simple. The primary key value of a row in a database table governs the identity of that row. So the question becomes how does *database* identity relate to Java *object* identity? Neither the == operator nor the equals() method equates well to the notion of a primary key.

The fourth part of the paradigm mismatch problem noted by Bauer and King (2007, pp. 16-17) is a collection of problems relating to associations. There are large differences between the ways in which classes in an object domain are associated with one another and the ways in which tables in a relational database are associated with one another. Relationships between classes in an object model are represented by references and collections of references to objects. Consider the following class snippets:

```
public class A {
 private B referenceToB;
 // other stuff omitted
 …
}


public class B {
 private A referenceToA;
 //other stuff omitted
 …
}
```

The relationship between A and B in this example is bidirectional. Given an A object, we can get to the associated B object (assuming there is one) via the referenceToB variable. Conversely, given a B object, we can get to the associated A object (assuming there is one) via the referenceToA variable. If the referenceToA variable is eliminated from class B, then there is no way to get to an A from a B. On the other hand, in the relational model, navigation between tables is handled by foreign key associations and is inherently *non*directional. Given a foreign key association between table A and table B, one can get to table A from table B and vice versa simply by performing a join between table A and table B. The relationship between class A and class B depicted above represents a one-to-one association between As and Bs; a given instance of A is associated with at most a single instance of B (and vice versa). However, there is nothing to prevent us from depicting a many-to-many association between As and Bs. The following code snippet demonstrates how:

```
public class A {
 private Set referencesToBs;
 //other stuff omitted
 …
}

public class B {
 private Set referencesToAs;
 //other stuff omitted
 …
}
```

As every beginning database student soon learns, while we often have many-to-many associations represented in an entity-relation or UML diagram, the actual *implementation*

of many-to-many associations (using SQL) are handled via link or association tables. Given tables A and B, implementing a many-to-many association between them requires that we introduce a table C having separate many-to-one associations to tables A and B (via separate foreign keys linking to table A and table B).

Given the differences between how associations are represented in the object model and how they are represented in the relational model, the question becomes how do we relate the two models when it comes to associations? Do we create a link *class* even though, from the point of view of the object model, it is unnecessary? If we do so, are we not just letting our relational model dominate our object model? On the other hand, if we do not do so, just what *is* the connection between our object model and our relational model when many-to-many relationships are involved?

The fifth and last part of the paradigm mismatch problem noted by Bauer and King (2007, pp. 18-19) is the problem of data navigation. This problem is probably one of the most serious problems facing any purported persistence solution. In Java, if we want to access the data in an instance of a class B that has an association to a class A where, currently, we have only a handle to A (handleToA), we might call handleToA.getB().getMyData(). Navigation through an object graph is accomplished by simply following associations between the objects in the graph. In SQL, on the other hand, if we are interested in the data in the A table, we issue a query like this:

```
select * from A
where condition _ to _ satisfy
```

If, later, we are interested in the data in table B as well as data in table A that is related to B, we might issue this query:

```
select * from A
left outer join B on A.id = B.id
where condition _ to _ satisfy
```

Let us assume that our application is constructed using an object model and, thus, is written using an OOP language like Java. Let us assume also that we want the appropriate SQL SELECTs to be issued as we traverse the available object graph at some point in the execution of our application. The problem then is this: we may end up issuing a different SQL SELECT for each node in the graph. This "node at a time" approach to accessing the data in the database will clearly result in a performance problem for any application attempting to support concurrency.[3] Indeed, this problem (or a variant of it) has been labeled the *n+1 selects problem* (Bauer & King, 2007, pp. 18-19). On the other hand, using an approach in which the object graph contains every possible persistent object, fetched in a single access to the database, generates a complementary performance problem. One aspect of the problem is simply that, in a given scenario, one might load the *entire* database into memory when only a small portion of it is required. Another aspect of the problem is that, in so doing, it is quite likely that locks will be issued on the tables and, so, concurrency will be negatively affected. So the problem of data navigation summarizes to: How do we find a "middle ground" between acquiring too little of the data in the database (n+1 selects problem) and acquiring too much of the data in the database? A persistence solution that does not find this middle ground is going to perform extremely poorly.

Having obtained an overview of the paradigm mismatch problem, we now look at the ways in which an ORM service, such as Hibernate, overcomes the problem.

HIBERNATE

As early as 1997 Mark Fussel laid the foundations for ORM services. Scott Ambler (2002) in a well known paper provided further developments. Perhaps the best known ORM service in use today is Hibernate (Relational Persistence, 2008). While it is beyond the scope of this chapter to provide a complete explanation of how Hibernate works or even how it is used, we can say that Hibernate attempts to provide a full object to relational mapping service. Bauer and King (2007), following Fussel, define this as follows:

Full object mapping supports sophisticated object modeling: composition, inheritance, polymorphism

and 'persistence by reachability.' The persistence layer implements transparent persistence; persistent classes do *not* inherit any special base class or have to implement a special interface. Efficient fetching strategies (lazy, eager and prefetching) and caching strategies are implemented transparently to the application. (p. 27)

Hibernate provides full object mapping via a number of avenues. In its original form, Hibernate utilized XML metadata as the medium by which persistent classes are stored in a database. For those who prefer not to use XML, the emergence of annotations with Java 5.0 and the completions of the Java Persistence API (JPA) and the EJB 3.0 specifications provide an alternative (Persistence, 2008; EJB 3.0, 2008).

In this section we examine the pieces that comprise Hibernate and develop some Hibernate code along the way. It is far beyond the scope of this chapter to provide a full exposition of all of Hibernate's features. Instead, we initially focus on a simplified model of an order entry system and utilize but one of several ways that Hibernate employs to enforce persistence. Our goal is to evaluate Hibernate. We will hold Hibernate to the standards developed in the previous section. In particular, we will judge Hibernate's ability to function as a "full object mapping" solution to the problem of persistence and as a solution to the paradigm mismatch problem.

FIRST CONTACT

The sample code in Listing 6 shows a Customer class that might be used in an order entry system application for an online store[4]. There is nothing special about this class. Indeed, it is simply a plain old Java object (POJO) (POJO, 2008).

Listing 6 A Customer Class

```
package org.mnsu.edu.oes;
public class Customer {
  private int id;
  private String fname;
  private String lname

  …
   // other fields such as street, city,
state, zip,
   // balance, creditCardNum, etc. left
out for
  // the sake of brevity

  private Customer() {}
   public Customer(String fname, String
lname) {
   this.fname = fname;
    this.lname = lname;
  }
  public int getId() {
   return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getFname() {
    return fname;
  }
  public void setFname(String fname) {
   this.fname = fname;
  }
  public String getLname() {
   return lname;
  }
  public void setLname(String lname) {
   this.lname = lname;
  }
}
```

As it stands, the Customer class is simply a POJO and could easily be instantiated in an application and its fields accessed with calls such as:

```
Customer customer = newCustomer("Gavin",
"King");
Sytem.out.println(customer.getFname() + "
" + customer.getLname());
```

As a POJO, this class has no claims to persistence. However, because this class would be central to an order entry system application, we will definitely require that it *be* a persistent class. One Hibernate mechanism for enforcing object persistence is a metadata mapping file with a ".hbm.xml" suffix.[5] This file maps the instantiated class to a relational database table. Typically, this file's prefix is the same as the class file's name.[6] For our current version of the Customer class, the following Customer.hbm.xml file (Listing 7) will do.

Listing 7 Customer.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//
EN"
   http://hibernate-sourceforge.net/hiber-
nate-mapping-3.0.dtd>
<hibernate-mapping>
 <class
   name="org.mnsu.edu.oes.Customer"
   table="CUSTOMERS">
   <id
     name="id"
     type="int"
     unsaved-value="0"
     <column name="CUST _ ID"
     sql-type="int"
     not-null="true"/>
     <generator class="hilo"/>
   /id>
   <property
    name="fname"
    column="FNAME"/>
   <property
     name="lname"
     column="LNAME"/>
 </class>
</hibernate-mapping>
```

Our mapping file specifies that the Customer class will be persisted to a relational database table named CUSTOMERS. The properties of the Customer class, namely "id," "fname," and "lname" will be mapped to columns in the CUSTOMERS table named "CUST_ID," "FNAME," and "LNAME," respectively.

While the mapping file specifies what *will* be done, it still remains to see how that will be accomplished in the application itself. The following code fragment gives the idea:

```
…
Session session = getSessionFactory().openSes-
sion();
Transaction tx = session.beginTransac-
tion();
Customer customer = new Customer("Gavin",
"King");
session.save(customer);
tx.commit();
session.close();
…
```

This short piece of code fulfills the promise of the mapping file. The result of executing this fragment results in this SQL code being executed against the database:

```
insert into CUSTOMER (CUST _ ID, FNAME,
LNAME) values(1, "Gavin", "King")
```

It is important to note that the value for the id property of the Customer class was not set during the creation of the customer object. So one might wonder why the above SQL command contains a value, namely 1, for the CUST_ID attribute of the CUSTOMERS table. The answer is that the id property of the Customer class is an *identifier property*, that is, it is a specially generated unique value that is assigned to the instance of the Customer class, not when that class's constructor is called, but, rather, when the call to "session.save(customer)" is made[7]. It is important to note

also that, while this call makes the customer object a *persistent object*, it does *not* thereby *persist* the object to the database, that is, the generated SQL insert command is not necessarily executed at this point.[8] In the code fragment above, the execution of the generated SQL is guaranteed only *after* the call to "tx.commit()."[9] The actual point at which the generated SQL is executed can be controlled, in several ways. Each of the following, for example, will each cause immediate execution:

Certain calls to Session.find() or Sesson.iterate()
net.sf.hibernate.Transaction.commit()
net.sf.hibernate.Session.flush()

Not every persistent object is persisted to the database when these calls are made. Hibernate employs a strategy known as *automatic dirty checking* to decide which persistent objects require a database update or insert. In short, Hibernate keeps track of persistent objects that are either updated in the persistence context (session) or are new to the persistent context. Only these require an update or insert.

The delayed execution of the generated SQL and the order in which they are executed can cause problems when triggers are used in the database. By using Session.flush(), the developer can ensure that, in such circumstances, the generated SQL is executed at the proper time and in the proper order.

Note the "unsaved-value="0""" attribute in the mapping file. The inclusion of this attributes is but one of the mechanisms that Hibernate uses to distinguish between a new transient instance of a class and a *detached* instance. For our present purposes, suffice it to say that this attribute enables transitive persistence for our Customer class and, later, for our Orders class. The idea is simply that, given a persistent Customer instance and a set of Orders instances owned by the Customer instance, the Orders instances in the set will become persistent as soon as they are added to the set without a call to session.save() being required.

SECOND CONTACT: ASSOCIATIONS AND COLLECTIONS

Let us enhance our model a bit and add an Order class. The idea, of course, is that a given customer might submit many orders while a given order should be submitted by, at most, one customer. Thus, there should be a one-to-many association between the Customer class and the Order class and a one-to-many relationship between the CUSTOMERS table and the ORDERS table. The Orders class might be defined as in Listing 8.

Listing 8 Order.java

```
package org.mnsu.edu.oes;

import java.util.date;

public class Order {
  private int id;
  private java.util.Date date;
  private Customer customer;

  public Order {}

 public Order(java.util.Date date, Customer
customer) {
    this.date = date;
    this.customer = customer;
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
  }
  public String getDate() {
    return date;
  }
  public void setDate(java.util.Date date)
{
    this.date = date;
  }
```

```
  public String getCustomer() {
     return customer;
  }
   public void setCustomer(Customer cus-
tomer) {
    this.customer = customer;
  }
}
```

The mapping file for this class appears in Listing 9.

Listing 9 Order.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//
EN"
   http://hibernate-sourceforge.net/hiber-
nate-mapping-3.0.dtd>
<hibernate-mapping>
 <class
   name="org.mnsu.edu.oes.Order"
   table="ORDERS">
   <id
     name="id"
     type="int"
     unsaved-value="0">
     column name="ORDER _ ID"
     sql-type="int"
     not-null="true"/>
     <generator class="hilo"/>
   /id>
   <property
     name="date"
     column="ORDER _ DATE"/>
   <many-to-one
     name="customer"
     column="CUST _ ID"
     class="org.mnsu.edu.Customer"
     not-null="true"/>
 </class>
</hibernate-mapping>
```

This is an example of an *unidirectional* many-to-one association. CUST_ID in the ORDERS table is a foreign key referencing the CUST_ID primary key attribute of the CUSTOMERS table. The not-null attribute is set to "true" because you cannot have an order without a customer.

As was noted earlier, associations between classes in an OO model are unidirectional while relationships between tables in the relational model are nondirectional.[10] Since, in an order entry application, we often need to traverse from an order to its associated customer and from a customer to the associated order, we need to modify the Customer class to make the association between the two classes bidirectional. Listing 10 shows the necessary modifications.

Listing 10 Modified Customer Class

```
package org.mnsu.edu.oes;
public class Customer {
  private int id;
  private String fname;
  private String lname;
  private Set orders = new HashSet();

  …
   // other fields such as street, city,
state, zip, balance,
  // creditCardNum, etc. left out for
  // the sake of brevity

  private Customer() {}
   public Customer(String fname, String
lname, Set orders) {
    this.fname = fname;
    this.lname = lname;
    this.orders = orders;
  }
  public int getId() {
    return id;
  }
  public void setId(int id) {
    this.id = id;
```

```
  }
  public String getFname() {
    return fname;
  }
  public void setFname(String fname) {
    this.fname = fname;
  }
  public String getLname() {
    return lname;
  }
  public void setLname(String lname) {
    this.lname = lname;
  }
  public Set getOrders() {
    return orders;
  }
  public void setOrders(Set orders) {
    this.orders = orders;
  }
}
```

Not surprisingly, the mapping file for the Customer class requires modification as well.

Listing 11 Modified Customer.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//
EN"
   http://hibernate-sourceforge.net/hiber-
nate-mapping-3.0.dtd>
<hibernate-mapping>
 <class
   name="org.mnsu.edu.oes.Customer"
   table="CUSTOMERS">
  <id
    name="id"
    type="int"
    unsaved-value="0">
    <column name="CUST _ ID"
     sql-type="int"
     not-null="true"/>
    <generator class="hilo"/>
  /id>
  <property
    name="fname"
    column="FNAME"/>
  <property
    name="lname"
    column="LNAME"/>
  <set name="Orders"
    inverse="true"
    cascade="all-delete-orphan">
    <key column="ORDER _ ID"/>
    <one-to-many class="org.mnsu.edu.oes.
Order"/>
  </set>
 </class>
</hibernate-mapping>
```

There are a couple of things in this modified Customer.hbm.xml file that require our attention.

First, notice the inverse attribute. This tells Hibernate that the one-to-many association between the Customer class and the Order class is just a many-to-one association from the Orders class to the Customer class as seen from the Order side of the association. Without this attribute, Hibernate would see the one-to-many association and the many-to-one association as two *different* associations!

Second, notice the "cascade= …" attribute. This attribute controls the nature of the association between the Customer class and the Order class. There are a number of XML options that can be used with this attribute, and they are:

- None (Default)
- Save-update
- Persist
- Merge
- Delete
- Remove
- Lock
- Replicate
- Evict
- Refresh
- All
- Delete-orphan

Since exploring all of these options would take us too far from our goal of providing an overview, we instead explain a few of the more commonly used options.

The "cacade=save-update" attribute relieves the developer of the need to make a call to save() in the application code every time an object referenced by a separate persistent object is created or updated. In our application, for example, had we used this attribute, as soon as a new Order instance is created, it would be automatically made persistent if it is referenced by a persistent Customer instance. Without the use of this attribute, creating and persisting Customer and Order instances would be accomplished by code such as the following:

```
…

Session session = getSessionFactory().
openSession();
Transaction tx = session.beginTransac-
tion();

Customer customer = new Customer("Gavin",
"King");
Order order = new Order(new java.util.Date,
customer);
Customer.getOrders().add(order);
Session.save(order);
session.save(customer);

tx.commit();
session.close();

…
```

By having this attribute enabled, the code reduces to the following:

```
…

Session session = getSessionFactory().
openSession();
Transaction tx = session.beginTransac-
tion();

Customer customer = new Customer("Gavin",
"King");
Order order = new Order(new java.util.Date,
customer);
Customer.getOrders().add(order);

session.save(customer);

tx.commit();
session.close();

…
```

While it is true that, in this example, the developer is saved from writing only a single line, it is also clear that, in a more substantial application, the savings can be considerable.

Like the "cascade=save-update" attribute, the "cascade=delete" attribute can save the developer a considerable amount of work. Instead of having to manually delete associated objects when a given persistent object is deleted, if "cascade=delete" is enabled, associated objects are automatically deleted. Suppose, for example, that we want to delete a Customer instance and all of the customer's related Order instances. Without "cascade=delete" enabled, our code might appear as follows:

```
Session session = getSessionFactory().
openSession();
Transaction tx = session.beginTransac-
tion();

customer = // load a Customer from the
database;

for (Iterator<Order> iter = customer.
getOrders().iterator(); iter.hasNext();) {
  Order order = iter.next();
  iter.remove();     // remove a single or-
```

```
der from the
// collection of orders
  session.delete(order);    // remove the
order from the database
}
session.delete(customer);   // remove the
customer from the database

tx.commit();
session.close();
```

With "cascade=delete" enabled, our code simplifies to this:

```
Session session = getSessionFactory().
openSession();
Transaction tx = session.beginTransac-
tion();

customer = // load a Customer from the
database;

session.delete(customer);   // remove the
customer from the database

tx.commit();
session.close();
```

The various "cascade=…" attributes can be combined so that, for example, "cascade=save-update, delete" can be used to enable both "cascade=save-update" and "cascade=delete." One can even enable all of them (save the last) by using "cascade=all."

The "cascade=delete-orphan" attribute needs to be considered separately from the other attributes since it is *not* enabled by "cascade=all." To understand its purpose we need to consider first the difference between entity and value types. As noted by Bauer and King (2007),

An object of entity type has its own database identity (primary key value). An object reference to an entity instance is persisted as a reference in the database (a foreign key value). An entity has its own lifecycle; it may exist independently of any other entity. An object of value type has no database identity; it belongs to an entity instance and its persistent state is embedded in the table row of the owning entity. Value types don't have identifiers or identifier properties. The lifespan of a value type instance is bounded by the lifespan of the owning entity instance. A value type doesn't support shared references. (pp. 159-60)

Earlier we noted that creating an Address class separate from a Customer class has its utility from an object-oriented point of view. After all, it is likely that we will require not only customer addresses but also shipper addresses, vendor addresses, supplier addresses, and so forth. Not having a separate Address class can result in massive amounts of code duplication. On the other hand, creating a separate Address *table* in the database is possibly not something that we want to do. For example, it might be the case that most, if not all, of our queries for the tables corresponding to customers, shippers, vendors, and suppliers require the address value(s) in question. Not having a separate Address table increases performance because we are not required to perform any joins to such an Address table. Given this roughly sketched scenario, a Customer class is an entity type and an Address class is a value type.

The reader may have noticed that we left out any address attribute in the Customer class so far developed. This, of course, was deliberate. Suppose now that we want to enhance our Customer class so that it contains a reference to an Address class. We will not here include any code for an Address class. Suffice it to say that such a class would likely contain string properties such as street, city, state, and zip code together with appropriate constructors, getters, and setters. It would *not* contain an id attribute. Note that, because it is a value type, with no persistent identity of its own, there is no Address.hbm.xml file corresponding to it. Instead, Hibernate's mechanism for persisting the data in an Address instance to a Customer table is the notion of a component. By including a reference to

an Address object in our Customer class, providing getters and setters for that reference, adjusting our constructors appropriately ,and including the following code in our Customer.hbm.xml file, we effectively map both the Customer and Address classes to the same database table:

```
<component name="address" class="org.mnsu.
edu.oes.Address">
  <property name="street" type="string"
    column="STREET" not-null="true"/>
  <property name="city" type="string"
    column="CITY" not-null="true"/>
  <property name="state" type="string"
    column="STATE" not-null="true"/>
  <property name="zipcode" type="string"
    column="ZIPCODE" no-null="true"/>
</component>
```

With the distinction between entity and value (component) types in mind, we can return to our discussion of "cascade=delete-orphan." Suppose that instead of mapping our Order class as an entity, we mapped it as a component. Suppose also that we wanted to delete an order for a particular customer. This can be accomplished by:

```
order =  //load a particular order
customer = //load a particular customer
customer.getOrders().remove(order);
```

Because order is a value type, no other object holds a reference to it and it can safely be removed. On the other hand, if, as we have done, the Order class is mapped as an entity type, the code to remove an order becomes slightly more complex as in:

```
order =  //load a particular order
customer = //load a particular customer
customer.getOrders().remove(order);
session.delete(order)
```

Because order is here an entity it can exist independently of the collection from which it is removed. Thus, we must also delete it from the *session* to guarantee that no other references to it are held. By enabling the "cascade=delete-orphan" attribute in our mapping file, we are telling Hibernate that order can be safely removed from the collection in which it resides because order is the only reference to the instance it references. The code we used to remove order when it is mapped as a component will now work when it is mapped as an entity. Failure to enable the "cascade=delete-orphan" will result in a foreign key constraint exception in the following code:

```
order =  //load a particular order
customer = //load a particular customer
customer.getOrders().remove(order);
session.remove(customer);
```

It is common in OO design circles to make a distinction between association, aggregation, and composition. Associations between classes are very loose and simply represent the ability of one class to "talk" to another. This is typically implemented in a given class with a simple reference variable to another class or, perhaps, as a method argument or local variable. Aggregation, on the other hand, while an association, represents a somewhat tighter coupling between the classes involved and typifies a "part-whole" relationship. Composition, like aggregation, is also an association but represents a situation in which the class representing the "whole" is responsible for the creation and destruction of the "part." We can suppose that our business rules dictate that the association between our Customer and Orders classes is a composition. If a given instance of the Customer class is deleted, so also should any instances of the Orders class initiated by the customer in question. By using "cascade=all-delete-orphan" we have not only enabled all of the "cascade=…" attributes including "cascade=save-update" and "cascade=delete" but enforce the semantics of composition even when the associated object is an entity type.

ONE-TO-ONE ASSOCIATIONS

Let us suppose for the moment that our order entry system is for a store that offers its customers large discounts. It is able to do so because it deals in large volumes of the products that it offers for sale. However, each customer is required to fill out a membership application after which, if they qualify (and have paid the appropriate fee), they are issued a membership card. The code for a MemberCard class could appear as in Listing 12.

Listing 12 MemberCard

```
package sales;
import java.util.Date;

public class MemberCard {
  private Long id;
  private int cardNumber;
  private Date start;
  private Date end;
  private Customer customer;

  public MemberCard() {}

  public MemberCard(int cardNumber, Date
start, Date end, Customer
    customer) {
    this.cardNumber = cardNumber;
    this.start = start;
    this.end = end;
    this.customer=customer;
  }

  public Long getId() {
    return id;
  }

  public void setId(long id) {
    this.id = id;
  }

  public int getCardNumber() {
    return cardNumber;
  }

  public void setCardNumber(int cardNum-
ber) {
    this.cardNumber = cardNumber;
  }

  public Date getStart() {
    return start;
  }

  public void setStart(Date start) {
    this.start = start;
  }

  public Date getEnd() {
    return end;
  }

  public void setEnd(Date end) {
    this.end = end;
  }

  public Customer getCustomer() {
    return customer;
  }

  public void setCustomer(Customer cus-
tomer) {
    this.customer = customer;
  }
}
```

A given card is issued to exactly one customer and a given customer is allowed to have exactly one membership card. Thus, the association between out Customer class and our new MemberCard class is a one-to-one. To incorporate the MemberCard into our model, the Customer class needs, of course, to be augmented with an attribute of type MemberCard called memberCard, changes to the constructor for Customer need to be made, and getter and setter methods need to be included. We

do not include here these changes to the Customer class for reasons of obviousness.

Hibernate offers two strategies for mapping a one-to-one association. The first strategy is based on identical primary keys for the CUSTOMERS and MEMBERCARDS tables. The MemberCard.hbm.xml file needs to include the following:

```
<hibernate-mapping package="sales">
      <class   name="MemberCard"
table="MEMBERCARDS">
    <id name="id" column="MEMBERCARD_ID"
type="long">
      <generator class="foreign">
        <param name="property">customer</
param>
       </generator>
    </id>
     <property name="cardNumber"/>
     <property name="start"/>
     <property name="end"/>
      <one-to-one  name="customer"
class="Customer"
       constrained="true"/>
  </class>
</hibernate-mapping>
```

Our Customer.hbm.xml file also needs to include the following:

```
<one-to-one   name="memberCard"
class="MemberCard" cascade="all"/>
```

There are two things to notice in the MemberCard.hbm.xml mapping file. First, the key generation strategy is designated as "foreign." Second, note the use of the one-to-one and constrained="true" mapping elements. The net result of these changes is that the database schema for the MEMBERCARD table contains a primary key (MEMBERCARD_ID) that is also a foreign key referencing the primary key (ID) of the CUSTOMERS table.

The second strategy offered by Hibernate for mapping a one-to-one association is based on a foreign key mapped, in our case, from the CUSTOMERS table to the MEMBERCARDS table. The mapping element in Customer.hbm.xml now becomes:

```
<many-to-one   name="memberCard"
class="MemberCard"  column="MEMBERCARD_
ID" cascade="all" unique="true"/>
```

The reader may wonder why the "many-to-one" element is used since this is supposed to be a one-to-one association. The answer is that we have also set the "unique" element to "true." By doing so we guarantee that the association is actually one-to-one.

Using this strategy, if we want the association between the Customer class and the MemberCard class to be bidirectional, the mapping element in MemberCard.hbm.xml changes to:

```
<one-to-one   name="customer"
class="Customer"   property-
ref="memberCard"/>
```

MANY-TO-MANY ASSOCIATIONS

As noted above, the association between our Customer and Orders classes is a one-to-many association. Since our application is an order entry system, we clearly need some products for our customers to order. For the sake of argument, let us suppose that we are selling woodworking equipment. We have the usual machinery for sale, namely table saws, band saws, jointers, planers, routers, drills, and so forth. A given customer can order a table saw, and a band saw and a jointer all in the same order. Clearly we need a Product class. The question is: What kind of association is there between our Product class and our Order class? Given that a customer can order a table saw, and a band saw, and a jointer all in the same order and that table saws can be present in orders from many customers, the association is clearly a many-to-many. Now the question becomes:

How do we implement such an association? On the relational database side, the answer is clear: since only one-to-one and one-to-many relationships are countenanced in a relational database, we must introduce an intermediary table (ORDERLINE) called a *junction table* that stands in two separate many-to-one relationships to the PRODUCT table and the ORDERS table.[11] The attributes of this ORDERLINE table will include, as a minimum, the primary key attributes of the ORDERS and PRODUCT tables. It is important to note that we have no choice about this. What about the object model side; must we introduce an Orderline class? The short answer is "No." The Hibernate mapping system is more than robust enough to handle many-to-many class associations without the introduction of "junction classes."[12] The long answer is a bit more involved. In the real world, a many-to-many association not only associate entities but usually conveys information about the *association* as well. For example, in the association between the Product class and the Orders class, we would usually want to include information such as the quantity of a given product involved in the order and perhaps the quoted price for that particular product.[13] A little thought quickly reveals that this information does *not* belong in either the Product or Orders classes. As Bauer and King (2007) put it:

Our experience is that there is almost always other information that must be attached to the link between associated instances (such as the date and time when an item was added to a category) and that the best way to represent this information is via an intermediate association class. (pp. 297-298)

Before we implement our Product and Orderline classes and their mapping files, one other issue deserves our attention, namely composite keys. In an introductory database class, students are often told that the ORDERLINE schema should contain the primary key attributes of the PRODUCT and ORDERS tables as foreign keys and that, jointly, they should form a composite primary key for the ORDERLINE table. An alternative, not often taught, is to have a separate noncomposite primary key for the ORDERLINE table. This has certain advantages from Hibernate's point of view; indeed, the Hibernate team discourages the use of both composite keys and natural keys. We will follow their advice and not use a composite key for our implementation of the ORDERLINE table.[14] The following represents an acceptable schema for the ORDERLINE and PRODUCT tables:

ORDERLINE(ORDERL_ID, PROD_ID, ORDER_ID, QUANTITY)

PRODUCT(PROD_ID, DESCRIPTION, UNIT_PRICE, QOH)[15]

Our Product and Orderline classes can now be implemented as in Listings 13 and 14.

Listing 13 The Product Class

```
package org.mnsu.edu.oes;
public class Product {
  private int id;
  private String description;
  private float unitPrice;
  private int qoh;
  private Set orderline = new HashSet();

  public Product {}
  public Product(String description, float
unitPrice, int
     qoh, Set orderline) {
    this.description = description;
    this.unitPrice = unitPrice;
    this.qoh = qoh;
    this.orderline = orderline;
  }
  …

  //getters and setters for description,
etc.

  …
}
```

Listing 14 The Orderline Class

```
package org.mnsu.edu.oes;
public class Orderline {
  private int id;
  private int orderid;
  private int productid;
  private int quantity;

  public Orderline {}
   public Orderline(int orderid, int pro-
ductid, int quantity) {
    this.orderid = orderid;
    this.prodid = prodid;
    this.quantity = quantity;
  }
  …

   // getters and setters for orderid,
etc.

  …
}
```

Listing 15 and 16 shows the mapping files for our new classes.

Listing 15 Product.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//
EN"
  "http://hibernate.sourceforge.net/hiber-
nate-mapping-3.0.dtd">

<hibernate-mapping>

  <class  name="org.mnsu.edu.oes.Product"
table="PRODUCT">

   <id name="id" type="int" unsaved-val-
ue="0">
   <column name="PROD _ ID"
sql-type="int"
not-null="true"/>
   <generator class="hilo"/>
  </id>

    <property  name="description"
type="string">
      <column  name="DESCRIPTION"
length="50"/>
  </property>

    <property  name="unitPrice"
type="double">
     <column  name="UNIT _ PRICE"  sql-
type="double"/>
  </property>

  <property name="qoh" type="int">
   <column name="QOH" sql-type="int"/>
  </property>

  <set name="Orderline"
  table="ORDERLINE"
  lazy="true"
  inverse="true"
  cascade="all"
  sort="unsorted">
    <key column="PROD _ ID"/>
    <one-to-many class="org.mnsu.edu.oes.
Orderline"/>
  </set>
 </class>
</hibernate-mapping>
```

Listing 16 Orderline.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//
EN"
"http://hibernate.sourceforge.net/hiber-
nate-mapping-3.0.dtd">

<hibernate-mapping>
```

```
 <class name="org.mnsu.edu.oes.Orderline"
table="ORDERLINE">

   <id name="id" type="int" unsaved-val-
ue="0">
      <column  name="ORDERLINE _ ID"  sql-
type="int" not-null="true"/>
     <generator class="hilo"/>
  </id>

  <property
   name="quantity"
   column="QUANTITY"/>

  <many-to-one
   name="Orders"
   column="ORDER _ ID"
   class="org.mnsu.edu.oes.Orders"
   not-null="true"/>

  <many-to-one
   name="Product"
   column="PROD _ ID"
   class="org.mnsu.edu.oes.Product"
    not-null="true"/>
 </class>
</hibernate-mapping>
```

Since we must connect our Orders class with our Orderline class in a one-to-many association we need to modify the mapping file for the Orders class as in Listing 17.

Listing 17 Orders.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD//
EN"
  "http://hibernate.sourceforge.net/hiber-
nate-mapping-3.0.dtd">

<hibernate-mapping>

  <class  name="org.mnsu.edu.oes.Orders"
table="ORDERS">
    <id  name="id"  type="long"  unsaved-
value="null">
  <column name="ORDER _ ID" sql-type="bigint"
not-null="true"/>
   <generator class="hilo"/>
  </id>

  <property name="date" type="date">
   <column name="ORDER _ DATE" />
  </property>

  <many-to-one
   name="customer"
   column="CUST _ ID"
   class="org.mnsu.edu.oes.Customer"
   not-null="true"/>
  <set name="Orderline" table="ORDERLINE"
lazy="true"
   inverse="true"
   cascade="all" sort="unsorted">
    <key column="ORDER _ ID"/>
    <one-to-many class="org.mnsu.edu.oes.
Orderline"/>
  </set>
 </class>
</hibernate-mapping>
```

With the above class and mapping files in place, the following code (Listing 18) can be used to create Customer, Product, Order, and Orderline instances.

Listing 18 Driver Code

```
…
// configuration code omitted
try {
   session  =  app.sessionFactory.openSes-
sion();
  tx = session.beginTransaction();
  // Create and make persistent a Customer
and a Product
```

```
  // The Product and possibly the Customer
objects
  // would normally be read from the da-
tabase using
  // a HQL query
  Customer cust1 = new Customer("Gavin",
"King");
  session.save(cust1);
 Product prod1 = new Product("Delta DJ20",
1400.00,
  20);
  session.save(prod1);
  prod1.setQoh(prod1.getQoh()-5);
  // Create an Order object
 Orders order1 = new Orders(new java.util.
Date(),
  cust1);
  // It is initially a transient object
  cust1.getOrders().add(order1);
  // Now it is a persistent object be-
cause
  // it is referenced by a persistent Cus-
tomer object
  // Create an Orderline object
 Orderline ordline1 = new Orderline(order1,
prod1, 5);
  // It is initially a transient object
  order1.getOrderline().add(ordline1);
  prod1.getOrderline().add(ordline1);
  // Now it is a persistent object be-
cause
  // it is referenced by persistent Order
and Product
  // objects
  tx.commit(); // cust2, prod1, order1,and
ordline1
  // are now garanteed to be in the da-
tabase
  } catch (HibernateException e) {
   tx.rollback();
   e.printStackTrace();
  } finally {
   if (session != null)
     session.close();
  }
…
  // other code
```

The above code creates Customer and Product instances, makes them persistent via the calls to session.save(), creates Orders and Orderline instances, adds them to the set collections in Customer and Product, and then commits all of these instances to the database. Note that there was no need to call session.save() on the Orders and Orderline instances. This is the result of the fact that we enabled transitive persistence for those instances.

INHERITANCE STRUCTURES

It is no secret, of course, that one of the hallmarks of any OOP language worthy of the name is inheritance. A given class can be a subclass of one or more superclasses. The subclass inherits the fields and methods of the superclass(es) and can add its own new fields and methods. In this way, the superclass can be used over and over again and the OO developer is relieved of the need to start from scratch and reinvent the wheel. If a class inherits from more than one superclass, we have *multiple* inheritance. If only one superclass is involved, we have *single* inheritance. In practice, multiple inheritance has led to problems and, as a result, the folks who developed Java opted for a single inheritance model but, in deference to the usefulness of multiple inheritance, included the notion of an *interface*. An interface is rather similar to an abstract class but allows no implementation whatsoever. While a Java class can extend at most one other Java class, it can implement as many Java interfaces as needed (Arnold, Gosling, & Holmes, 2006, Chapter 4).

Hibernate's strategy for inheritance support comes in four forms, namely table per concrete class with implicit polymorphism, table per concrete class, table per class hierarchy, and table per subclass. The names describe the strategies. The Hibernate team prefers the table per class hierarchy

strategy for reasons of performance and simplicity (Bauer & King, 2007, p. 199). Using this strategy, the table created for a particular class hierarchy contains columns for all of the properties of all classes in the hierarchy. Additionally, the table contains a discriminator column. This column is responsible for identifying which rows in the table correspond to instances of which class in the hierarchy. The downside of this strategy is that the table columns mapped by the properties of the subclasses involved must be nullable.

We illustrate the table per class hierarchy strategy by subclassing the Customer class with CorpCustomer (corporate customer) and PrivateCustomer. Though a corporate customer is a customer, it needs to be distinguished from a private customer by virtue of the fact that corporate customers have tax-id numbers while private customers have social security numbers. There would be other differences, of course, but this will suffice for our purposes. The code for CorpCustomer and Private Customer appears as in Listing 19 and 20.

Listing 19 CorpCustomer

```
package org.mnsu.edu.oes;
public class CorpCustomer extends Customer {
  private String taxId;
  public CorpCustomer() {}
 public CorpCustomer(String fname, String lname, String taxId) {
   super(fname,lname);
   this.taxId = taxId;
  }
  public void setTaxId(String taxId) {
   this.taxId = taxId;
  }
  public String getTaxId() {
   return taxId;
  }
}
```

Listing 20 PrivateCustomer

```
package org.mnsu.edu.oes;

public class PrivateCustomer extends Customer {
  private String ssnum;

  public PrivateCustomer() {}
   public PrivateCustomer(String fname, String lname, String ssnum) {
   super(fname, lname);
   this.ssnum = ssnum;
  }

  public void setSsnum(String ssnum) {
   this.ssnum = ssnum;
  }
  public String getSsnum() {
   return ssnum;
  }
}
```

The necessary modifications for the Customer.hbm.xml file are included in Listing 21.

Listing 21 Customer.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
   "-//Hibernate/Hibernate Mapping DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
  <class
    name="org.mnsu.edu.oes.Customer"
        table="CUSTOMERS" discriminator-value="CT">

   <id
    name="id"
     type="long"
```

```
    unsaved-value="null">
    <column name="CUST _ ID"
      sql-type="bigint"
       not-null="true"/>
      <generator class="hilo"/>
  </id>

  <discriminator
   column="CUSTOMER _ TYPE"
   type="string"/>

  <property
     name="fname"
     column="FNAME"/>
  <property
     name="lname"
     column="LNAME"/>
  <set name="Orders"
   table="ORDERS"
   lazy="true"
   inverse="true"
   cascade="all"
   sort="unsorted">
   <key column="ORDER _ ID"/>
       <one-to-many class="org.mnsu.edu.
oes.Orders"/>
  </set>
  <subclass
   name="org.mnsu.edu.oes.CorpCustomer"
   discriminator-value="TXID">
   <property
     name="taxId"
     column="TAX _ ID"/>
  </subclass>
   <subclass
      name="org.mnsu.edu.oes.PrivateCus-
tomer"
     discriminator-value="SS">
  <property
   name="ssnum"
   column="SS _ NUM"/>
  </subclass>
 </class>
</hibernate-mapping>
```

The following code snippet instantiates and makes persistent CorpCustomer and PrivateCustomer objects.

```
PrivateCustomer cust1 = new
PrivateCustomer("Gavin", "King", "555-55-
5555");
CorpCustomer cust2 = new CorpCustomer("Steve",
"Ebersole", "41-1234567");
session.save(cust1);
session.save(cust2);
```

When the transaction commits, we have the following in the database (Box 1).

mysql> select * from customers;

HIBERNATE QUERIES

Though not specifically mentioned in our earlier discussion of full object mapping, most ORM services, purporting to implement full object mapping, include an object-oriented query language. Note that this is in contrast to a *data* mapping service like iBATIS. The query language for iBATIS is SQL itself. Hibernate supports three different query "languages": hibernate query language (HQL), the criteria API, and native SQL. HQL is a full object-oriented query language that is deliberately designed to resemble SQL. Indeed, new Hibernate users are often confused by this and wonder just what the difference is between HQL and SQL. The criteria API allows one to express query by criteria (QBC) and query by example (QBE) queries (Bauer & King, 2007, p. 615). Native (direct) SQL queries are also supported (Bauer & King, 2007, p. 615). In this chapter, we will focus on HQL.

A HQL query is created by instantiating the query interface. This can be done as simply as follows:

```
Query query = session.createQuery("from
Customer");
List<Customer> queryResult = query.
list();
```

Or, more simply, using method chaining:

```
List<Customer> queryResult = session.
createQuery("from Customer").list();
```

In this code, queryResult is a list of available Customer instances and we can, for example, iterate through queryResult and print out customer names as in this code:

```
for (Customer customer: queryResult) {
  System.out.println(customer.getFname() +
" " + customer.getLname();
}
```

Hibernate also supports parameter binding in the form of positional parameters and named parameters. The following is an example using positional parameters:

```
String qString = "from Customer c where
c.fName like ?";
Query query = session.createQuery(qString).
setString(0, "Gavin King");
```

The named parameter(s) equivalent of this is:

```
String qString = "from Customer c where
c.fName like :fName";
Query query = session.createQuery(qString).
setString("fName", "Gavin King");
```

Note that in the above queries, we are using "c" as an alias for Customer. A HQL alias is a shorthand name and functions in HQL much like its SQL counterpart.

Hibernate supports a number of different join types, including all of the following:

- Inner join
- Left (outer join)
- Right (outer join)
- Full join
- Join fetch

Each of these, save the last, should be familiar to readers already familiar with SQL. We will examine join fetches in a later section.

As an example of a left outer join in HQL, consider the following code snippet:

```
Query query = session.createQuery("from
Customer c left join c.orders o");
```

This query returns a list of customers together with their associated orders. The list contains customers who have orders together with those who do not. The list actually consists of a collection of Object[] arrays with customer values at index 0 and order values at index 1.

To obtain a list of only those customers who have placed orders, use this query:

Box 1.

```
+---------+--------------+-------+----------+------------+-------------+--------+----------------+
| CUST_ID | CUSTOMER_TYPE | FNAME | LNAME  | TAX_ID   | SS_NUM   |
+---------+--------------+-------+----------+------------+-------------+-------+----------------+
|   1    | SS            | Gavin  | King     | NULL     | 555-55-5555 |
|   2    | TXID          | Steve  | Ebersole  | 41-1234567 | NULL     |
+---------+--------------+-------+----------+------------+-------------+-------+----------------+

 2 rows in set (0.00 sec)
```

```
Query query = session.createQuery("from
Customer c join c.orders o");
```

Just as in SQL, one can use a select clause in HQL to restrict the values returned by a query. In the last two queries we did not use a select clause and, as a result, we would incur the additional overhead of manipulating Object[] arrays if we only wanted, for example, to print out the customer names of customers who have placed orders. We can avoid this overhead with a select clause as in this code:

```
Query query = session.createQuery("select
c from Customer c join c.orders o");
```

Earlier we noted that HQL is an *object-oriented* query language. As such, we expect that polymorphic queries should be possible. HQL does not disappoint us in this regard. Assuming that we have subclassed both CorpCustomer and PrivateCustomer from Customer (as in the section on inheritance strategies), the following query will return instances of all three classes:

```
Query query = session.createQuery("from
Customer c");
```

To underscore the fact that HQL is truly object-oriented and truly supports polymorphic queries, note that all of the following are valid, if somewhat pointless, HQL queries:

```
Query query = session.createQuery("from
java.lang.Object");
Query query = session.createQuery("from
java.util.Set");
Query query = session.createQuery("from
java.util.List");
```

As might be expected, in addition to supporting logical operators such as `and`, `any`, `between`, `exists`, `in`, `like`, `not`, `or`, `some` and comparison operators such as `=`, `>`, `<`, `>=`, `<=`, and `<>`, HQL also supports the use of aggregate functions. In particular, it supports the following five:

- count()
- min()
- max()
- sum()
- avg()

The following is thus a valid HQL:

```
Query query = session.createQuery("select
count(serialNumber) from
Product");
```

SQL students soon learn that the use of an aggregate function needs to be qualified with a "group by" expression if the SELECT clause contains a normal column reference in addition to the aggregate function. The same is true in HQL. For example, suppose our Product class contained a productDescription property of type String. If we wanted to retrieve productDescription(s) as well as a count of the serialNumber(s), we would need to issue the following query:

```
Query query = session.createQuery("select
productDescription, count(serialNumber)
from Product p group by
p.productDescription");
```

As might be expected, Hibernate also supports the use of a HAVING clause to qualify the results obtained from the use of a GROUP BY clause:

```
Query query = session.createQuery("select
productDescription, count(serialNumber)
from Product p group by p.productDescription
having p.description like 'Bandsaw%');
```

Correlated and uncorrelated subqueries are often used in SQL to achieve an economy of expression and power not easily obtained without them. Not surprisingly, HQL supports them as well. HQL supports subqueries in the WHERE clause (Bauer & King, 2007, p. 659). The following are each valid HQL:

```
//correlated – gets customers who have
placed more than 5 orders
from Customer c where 5 < (select count(o)
from c.orders o)
//uncorrelated – gets customers who have
placed an order
from Customer c where c.id in (select
o.customer from Order o)
```

FETCHING STRATEGIES

The astute reader will have noticed that a new attribute appeared in Listings 15, 17, and 21 for the Products.hbm.xml, Orders.hbm.xml, and Customer.hbm.xml files, namely lazy="true." There is in fact no real need to include this in our mapping files since the default for this attribute is "true" as of version 3.x of Hibernate. However, there *is* a definite need to understand what this attribute is and why we want "true" to be the default in most cases. To help in our understanding of this, we need to consider a problem that plagues *any* ORM implementation, namely, the *n+1 selects problem* mentioned in the Paradigm Mismatch Problem section.

Suppose we want a list of Order objects using data stored in the ORDERS table in the database. Each Customer object is associated with a set of Order objects. Does this mean that when we instantiate an Order object, we must also instantiate any associated Customers? If so, then we are likely to generate an object graph that includes much data from the Customer table when all we wanted was a list of Orders objects. By having the lazy attribute set to "true," we are enabling lazy fetching, that is, we are telling Hibernate that, for example, when we obtain an Orders object from the database, we do *not* want the customer attribute of that class to be loaded until we really need it. Instead, a proxy object is created as a stand in for the customer attribute and we thereby avoid the problem of loading customer data into the object graph when we only wanted data from the ORDERS table (see work by Gamma, Helm, Johnson, and Vlissides [1995, pp. 207-217] for a discussion of the proxy design pattern). If later, we decide to "examine" the customer attribute, then a new database select will be issued and the proxy will be replaced by the "real thing." Conversely, if we obtain a Customer object from the database, we may not necessarily want to obtain the list of Orders objects associated with that customer. Instead, with lazy fetching enabled, an uninitialized collection wrapper will be obtained that does not generate a database hit[16].

While having lazy fetching enabled solves one aspect of the n+1 selects problem, it can generate another problem if we are not careful. Suppose we want to obtain some information about a particular customer in our database. Suppose also that the desired information is held in the customer table. We can accomplish this in any number of ways and the particular method chosen is not important so long as it results in a single select query being issued against the customer table in the database.

If now we wish to iterate through our graph of customer objects and examine their orders, the formerly noninitialized collection wrapper will get replaced by the "real thing."

However, when we start to "walk the object graph" and examine a customer's orders, several more selects will be issued, one for each of the customer's orders. Our problem now is not one of having obtained an object graph much larger than needed. The problem is that our application is issuing too many selects against the database. In a highly concurrent application, performance will suffer greatly as a result.

Hibernate's solution to this dilemma comes in the form of programmer controlled fetching strategies. We have already seen part of the solution in the use of the lazy fetching attribute at the level of the mapping files. Hibernate also allows two other modes of fetch control. The first of these is called *batch fetching*. Along with having lazy set to "true," a developer can enable batch fetching in the mapping file. That is, we can modify our Customer.hbm.xml file thusly:

```
<set name="Orders" table="ORDERS"
lazy="true" inverse="true" batch-size="10"
cascade="all"
 sort="unsorted">
 <key column="ORDER_ID"/>
  <one-to-many class="org.mnsu.edu.oes.Or-
ders"/>
</set>
```

Note the inclusion of batch-size="10" in the above. This tells Hibernate to fetch 10 sets when the first set is accessed. While this reduces the size of our n+1 selects problem somewhat, it clearly is not a solution to the problem in general. It is probably impossible for most applications to "know" at the level of the mapping file what the "correct" choice is for the batch size. For some transactions carried out by the application, a particular batch size might be just fine, while for others it might be too large or too small. The third mode of fetch control supported by Hibernate is *eager fetching*. Eager fetching is a bit like batch fetching with a batch size of "all." Using eager fetching at the mapping file level is likely to produce the problem, noted above, of loading the entire database into the object graph when only a small portion is required. Instead, Hibernate recommends that eager fetching be used when needed not in the mapping files, but rather in the application code itself. Since Hibernate supports runtime declarations of association fetching strategies, this turns out to be quite easy to do. Consider an object domain somewhat simpler than our current one. In this scenario, we have associations between courses (in a school) and exams. Each course can have one or more exams. Figure 1 depicts the situation.

Suppose that the course data consists of four courses with the following names:

- CS1
- CS2
- Data Structures
- Design Patterns

Suppose further that each course has associated with it two exams with the following descriptions:

- Midterm CS1
- Final CS1
- Midterm CS2
- Final CS2
- Midterm Data Structures
- Final Data Structures
- Midterm Design Patterns
- Final Design Patterns

The java code for creating these classes, their associated mapping files, and the scripts necessary to create the database tables is quite easy to construct and is not included here for the sake of brevity. What *is* included here is some driver code (and output from it) that shows what happens when we utilize only Hibernate's default lazy loading fetch plan. Suppose, for example, that we want to retrieve the course names from the database and also the exam descriptions associated with each course. The code in Listing 22 shows one way to do this.

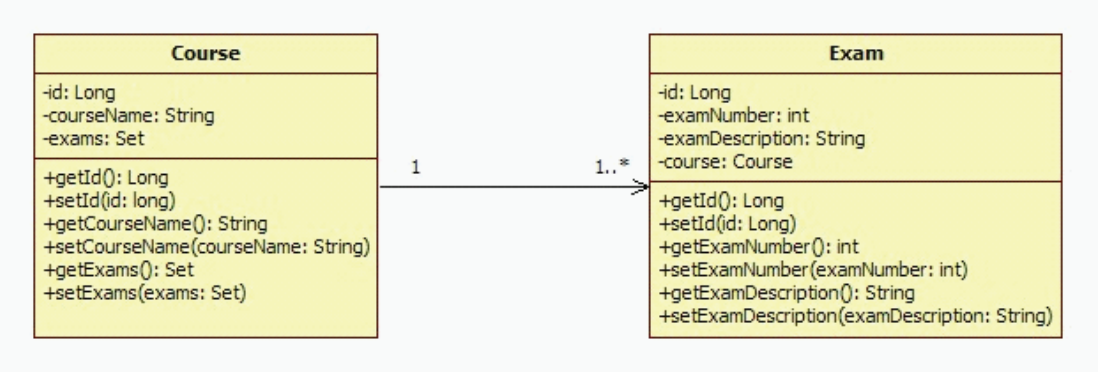
Listing 22 (Lazy Loading Only and a Join)

```
//other stuff omitted
List<Course> courseList = session.
createQuery("select c from Course c left
join c.exams").list();
System.out.println();
for (Course courses : courseList) {
 System.out.println(course.getCourseName()
+ " exams: ");
   for (Exam examination : (Set<Exam>)
courses.getExams()) {
    System.out.println("\t" + examination.
getExamDescription());
  }
System.out.println();
}
```

This code first retrieves a list of courses from the database using lazy fetching. Thus, there are no exams retrieved thus far. It then iterates through the list and prints out the course names. Next, in the inner for-loop, the set of exams associated with the course in question is retrieved via the call to "courses.getExams()" and the exam description for each exam is printed out. The following output shows the SQL code generated by Hibernate during this process:

```
 [java] Hibernate: select distinct course0 _ .COURSE _ ID as COURSE1 _ 0 _ , course0 _ .COURSENAME as COURSENAME0 _ from COURSE course0 _ left outer join EXAM exams1 _ on course0 _ .COURSE _ ID=exams1 _ .COURSE _ ID
  [java]
  [java] CS1 exams:
 [java] Hibernate: select exams0 _ .COURSE _ ID as COURSE4 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ 0 _ , exams0 _ .EXAMDESCRIPTION as EXAMDESC2 _ 1 _ 0 _ , exams0 _ .EXAMNUMBER as EXAMNUMBER1 _ 0 _ , exams0 _ .COURSE _ ID as COURSE4 _ 1 _ 0 _ from EXAM exams0 _ where exams0 _ .COURSE _ ID=?
  [java]  Final CS1
  [java]  Midterm CS1
  [java]
  [java] CS2 exams:
 [java] Hibernate: select exams0 _ .COURSE _ ID as COURSE4 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ 0 _ , exams0 _ .EXAMDESCRIPTION as EXAMDESC2 _ 1 _ 0 _ , exams0 _ .EXAMNUMBER as EXAMNUMBER1 _ 0 _ , exams0 _ .COURSE _ ID as COURSE4 _ 1 _ 0 _ from EXAM exams0 _ where exams0 _ .COURSE _ ID=?
  [java]  Midterm CS2
  [java]  Final CS2
  [java]
  [java] Data Structures exams:
 [java] Hibernate: select exams0 _ .COURSE _ ID as COURSE4 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ 0 _ , exams0 _ .EXAMDESCRIPTION as EXAMDESC2 _ 1 _ 0 _ , exams0 _ .EXAMNUMBER as EXAMNUMBER1 _ 0 _ , exams0 _ .COURSE _ ID as COURSE4 _ 1 _ 0 _ from EXAM exams0 _ where exams0 _ .COURSE _ ID=?
  [java]  Final Data Structures
  [java]  Midterm Data Structures
  [java]
  [java] Design Patterns exams:
 [java] Hibernate: select exams0 _ .COURSE _ ID as COURSE4 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ , exams0 _ .EXAM _ ID as EXAM1 _ 1 _ 0 _ , exams0 _ .EXAMDESCRIPTION as EXAMDESC2 _ 1 _ 0 _ , exams0 _ .EXAMNUMBER as EXAMNUMBER1 _ 0 _ , exams0 _ .COURSE _ ID as COURSE4 _ 1 _ 0 _ from EXAM exams0 _ where exams0 _ .COURSE _ ID=?
```

Figure 1. UML diagram for Course and Exam



```
  [java]  Final Design Patterns
  [java]  Midterm Design Patterns
  [java]
```

Note that there are five SQL SELECTS generated. What we have here is the n+1 selects problem. The "1" in this case refers to the first SQL SELECT where the list of courses is retrieved. The "n" in this case refers to the next four SQL SELECTs required to retrieve the exams (two per course) associated with each of the four courses. Because lazy fetching was utilized to retrieve the list of courses, the set of exams associated with each is not retrieved in the first SQL SELECT but, as noted earlier, an uninitialized collection wrapper (in effect a collection proxy) is retrieved instead. When we iterate through the course list and print the list of exam descriptions, the collection is initialized via additional SQL SELECTs.

There are a number of ways to avoid the n+1 selects in the scenario outline above. One of the simplest ways is to employ a dynamic fetching strategy as in Listing 23.

Listing 23 (Using Join Fetch)[17]

```
List<Course> courseList = session.
createQuery("select distinct c from Course
c left join fetch c.exams").list();
System.out.println();
for (Course courses : courseList) {
 System.out.println(courses.getCourseName()
+ " exams: ");
   for (Exam examination : (Set<Exam>)
courses.getExams()) {
    System.out.println("\t" + examination.
getExamDescription());
  }
  System.out.println();
}

Output:

 [java] Hibernate: select distinct course0 _
.COURSE _ ID as COURSE1 _ 0 _ 0 _ , exams1 _
.EXAM _ ID as EXAM1 _ 1 _ 1 _ , course0 _
.COURSENAME as COURSENAME0 _ 0 _ , exams1 _
.EXAMDESCRIPTION as EXAMDESC2 _ 1 _ 1 _ ,
exams1 _ .EXAMNUMBER as EXAMNUMBER1 _ 1 _ ,
exams1 _ .COURSE _ ID as COURSE4 _ 1 _ 1 _ ,
exams1 _ .COURSE _ ID as COURSE4 _ 0 _ _ , ex-
ams1 _ .EXAM _ ID as EXAM1 _ 0 _ _ from COURSE
course0 _ left outer join EXAM exams1 _ on
course0 _ .COURSE _ ID=exams1 _ .COURSE _ ID
  [java]
  [java] CS1 exams:
  [java]  Final CS1
  [java]  Midterm CS1
  [java]
  [java] CS2 exams:
  [java]  Final CS2
  [java]  Midterm CS2
  [java]
  [java] Data Structures exams:
  [java]  Midterm Data Structures
  [java]  Final Data Structures
  [java]
  [java] Design Patterns exams:
  [java]  Midterm Design Patterns
  [java]  Final Design Patterns
  [java]
```

Note that in this case there is only one SQL SELECT generated.

In the example above, we provided an example of the n+1 selects problem and one way of overcoming that problem (using join fetch). The example was from the perspective of the one side of a one-to-many association. We now demonstrate the problem again but from the perspective of the many side.

Suppose that we want to retrieve and print the exam descriptions data from the database and, for each such description, the course name associated with that description. The code in Listing 24 shows one way to do this.

Listing 24 (Lazy Loading Only and a Join)

```
List<Exam> examsList = session.
createQuery("select e from Exam e left
join e.course").list();
System.out.println();

for (Exam examinations : examsList) {
 System.out.println(examinations.getExam-
Description() + " is a ");
  System.out.println("\t" + examinations.
getCourse().getCourseName()
  + " exam: ");
}

Output:

 [java] Hibernate: select exam0 _ .EXAM _ ID
as  EXAM1 _ 1 _ ,  exam0 _ .EXAMDESCRIPTION
as  EXAMDESC2 _ 1 _ ,  exam0 _ .EXAMNUMBER
as EXAMNUMBER1 _ , exam0 _ .COURSE _ ID as
COURSE4 _ 1 _ from EXAM exam0 _ left outer
join COURSE course1 _ on exam0 _ .COURSE _
ID=course1 _ .COURSE _ ID
  [java]
  [java] Final CS1 is a
 [java] Hibernate: select course0 _ .COURSE _
ID as COURSE1 _ 0 _ 0 _ , course0 _ .COURSENAME
as COURSENAME0 _ 0 _ from COURSE course0 _
where course0 _ .COURSE _ ID=?
  [java]  CS1 exam
  [java] Midterm CS1 is a
  [java]  CS1 exam
  [java] Final CS2 is a
 [java] Hibernate: select course0 _ .COURSE _
ID as COURSE1 _ 0 _ 0 _ , course0 _ .COURSENAME
as COURSENAME0 _ 0 _ from COURSE course0 _
where course0 _ .COURSE _ ID=?
  [java]  CS2 exam
  [java] Midterm CS2 is a
  [java]  CS2 exam
  [java] Final Data Structures is a
 [java] Hibernate: select course0 _ .COURSE _
ID as COURSE1 _ 0 _ 0 _ , course0 _ .COURSENAME
as COURSENAME0 _ 0 _ from COURSE course0 _
where course0 _ .COURSE _ ID=?
  [java]  Data Structures exam
  [java] Midterm Data Structures is a
  [java]  Data Structures exam
  [java] Final Design Patterns is a
 [java] Hibernate: select course0 _ .COURSE _
ID as COURSE1 _ 0 _ 0 _ , course0 _ .COURSENAME
as COURSENAME0 _ 0 _ from COURSE course0 _
where course0 _ .COURSE _ ID=?
  [java]  Design Patterns exam
  [java] Midterm Design Patterns is a
  [java]  Design Patterns exam
```

Again, notice that there are five SQL SELECTS generated. In this case, instead of providing an uninitialized collection wrapper for a set of exams, a proxy for the course associated with a given set of exams is provided. The reader may wonder why there are only four SQL SELECTS for the course names. After all there are eight exams involved and the call to

```
"System.out.println("\t" + examinations.
getCourse().getCourseName()"
```

is, therefore, executed eight times. The answer is that, for a given set of exam descriptions, the *first* call does generate a SQL SELECT but, since the *second* call "retrieves" the very same Course instance as the first call, no additional SQL SELECT is required because the appropriate Course instance is already available. In any case, the cure for this version of the n+1 selects problem is similar to that in the first example. Again, we use dynamic fetching as in Listing 25.

Listing 25 (Using Join Fetch)

```
List<Exam> examsList = session.
createQuery("select e from Exam e left
join fetch e.course").list();
System.out.println();
```

```
for (Exam examinations : examsList) {
 System.out.println(examinations.getExam-
Description() + " is a ");
  System.out.println("\t" + examinations.
getCourse().getCourseName()
  + " exam ");
}

Output:

 [java] Hibernate: select exam0 _ .EXAM _ ID
as EXAM1 _ 1 _ 0 _ , course1 _ .COURSE _ ID as
COURSE1 _ 0 _ 1 _ ,  exam0 _ .EXAMDESCRIPTION
as EXAMDESC2 _ 1 _ 0 _ , exam0 _ .EXAMNUMBER
as  EXAMNUMBER1 _ 0 _ ,  exam0 _ .COURSE _ ID
as  COURSE4 _ 1 _ 0 _ ,  course1 _ .COURSENAME
as COURSENAME0 _ 1 _ from EXAM exam0 _ left
outer  join  COURSE  course1 _  on  exam0 _
.COURSE _ ID=course1 _ .COURSE _ ID
  [java]
  [java] Final CS1 is a
  [java]  CS1 exam
  [java] Midterm CS1 is a
  [java]  CS1 exam
  [java] Final CS2 is a
  [java]  CS2 exam
  [java] Midterm CS2 is a
  [java]  CS2 exam
  [java] Final Data Structures is a
  [java]  Data Structures exam
  [java] Midterm Data Structures is a
  [java]  Data Structures exam
  [java] Final Design Patterns is a
  [java]  Design Patterns exam
  [java] Midterm Design Patterns is a
  [java]  Design Patterns exam
```

Notice, as in the first example, that a single SQL SELECT is generated.

There are many more options that can be used in setting up a fetching plan appropriate for a given application than can be show here. The reader is urged to turn to Bauer and King (2007), Chapters 13-15.

PERSISTENT OBJECTS

In Hibernate, an object instantiated from a class (e.g., using the new operator) may be in one of three states, namely transient, persistent, or detached. The state of an object is *transient* if it is not yet tied to a database table row. As soon as a transient object is dereferenced in its application, it becomes available for garbage collection just as any ordinary Java object would. A transient object has no database identity. The state of an object is *persistent* if it has been saved to the session using either save() or saveOrUpdate(). A persistent object is an object with a database identity and has a primary key value as its database identifier. The state of a persistent object is guaranteed to be synchronized with the database. A persistent object can become a transient object by using delete(). The state of an object is *detached* if it was once a persistent object but the session with which it was previously associated is now closed. The state of a detached object is *not* guaranteed to be synchronized with the database. Additionally, a persistent object can become detached via a call to evict(), close(), or clear() and a detached object can become a persistent object via a call to update(), saveOrUpdate(), or lock().[18] While this discussion suggests a seemingly simple API for manipulating objects between transient, persistent, and detached states, it is, in fact, too simple. Consider the following code (Listing 26) which *appears* to load a Customer object into a session, close the session (thereby transitioning the Customer object from persistent to detached), change the first name of the Customer object, create a new session, and transition the Customer object from detached to persistent via a call to update().

Listing 26

```
Session session =???//get a session – can
be done in lots of ways
Transaction  tx  =  session.beginTransac-
tion();
```

```
Customer customer = (Customer) session.
load(Customer.class,customerId);

tx.commit();
session.close();//customer is now detached

customer.setName("Jill");
```

However, this code will usually generate an error when run. The specific error reads as follows:

"could not initialize proxy – the owning Session was closed"

This error is actually quite instructive and illustrates one of the mechanisms that Hibernate uses to avoid unnecessary hits to the database and thereby increases performance. Because lazy fetching is enabled for the Customer class, the call to session.load() in the above listing, contrary to appearances, does not generate a database hit that loads the customer whose id is customerId. Instead, a proxy is generated that serves as a stand-in for the "real" customer object. Ordinarily, the "real" customer object would be loaded (and a database hit thereby generated) by the call to customer.setName(). However, that call is not made until *after* the session is closed. Since the session is closed, the proxied customer object is detached and cannot be initialized. Since it cannot be initialized, the call to customer.setName() generates the above error.

There are a variety of ways in which the above problem can be fixed. One *could* enable eager fetching for the Customer class. While this would solve the problem, it would reintroduce the problem of loading too many Customer objects for queries involving customers. What we want, of course, is some way of initializing a proxy *without* having to enable eager fetching. Luckily, Hibernate provides the static initialize() method from the Hibernate API. The solution now becomes very simple. Just insert the following line immediately after the call to session.load():

```
Hibernate.initialize(customer);
```

This line will cause a database hit and the proxied Customer object, with an id of customerId, will be initialized with the proper values. The subsequent call to customer.setName() on the detached customer object can now succeed and will no longer generate the error noted above.

CONCLUSION

We began our overview of Hibernate with the goal of evaluating its success in overcoming the impedance mismatch problem. Specifically, we noted that there are five aspects to this problem. The first aspect—the problem of granularity—is handled by Hibernate's support for fine-grained domain models and its ability to naturally distinguish between entity and value types (Bauer & King, 2007, Sections 4.1 and 4.2). The second aspect—the problem of subtypes—is handled by Hibernate's support for at least four different class mapping strategies and by its support for custom types (Bauer & King, 2007, Chapter 5). The third aspect—the problem of identity—is handled by Hibernate's distinction between entity and value types, its use of identifiers, and its support for transactions and caching (Bauer & King, 2007, Chapters 10 and13). The fourth aspect—problems relating to associations—is handled by Hibernates ability to handle sets, bags, lists, and maps quite transparently (Bauer & King, 2007, Chapter 6). The fifth aspect—the problem of object network navigation and the associated n+1 selects problem—is handled by Hibernate's ability to utilize a variety of different fetching strategies (including lazy loading, batch fetching, and eager fetching), each of which can be utilized at the metadata mapping level or at the application code level (Bauer & King, 2007, Chapter 13).

In the last several years Hibernate has enjoyed a considerable amount of success and growth. Evidence for this can be seen on a number of fronts. Consider, for example, these facts: the revised EJB

specification provides a Java Persistence API that is largely based on Hibernate (EJB 3.0, 2008); highly regarded Java/EE application frameworks such as Spring include facilities for integrating with Hibernate (Spring Framework, 2008); while Hibernate is aimed toward the Java developer, a .net port of Hibernate (NHibernate) is also available (NHibernate, 2008); Hibernate can be used with virtually any database for which a decent jdbc driver is available; and Hibernate is quite flexible and allows a developer to utilize either XML metadata or annotations to fulfill the persistence requirements of their applications.

The above is not to suggest that Hibernate is a "silver bullet." Indeed, for some applications Hibernate is definitely overkill and a simpler data mapping solution such as iBATIS might well be more appropriate. Hibernate is a complex ORM service with a steep learning curve that is not for those looking for a quick and dirty solution to the paradigm mismatch problem. However, for those who are looking for a full ORM service that solves virtually all aspects of the paradigm mismatch problem and who are willing to invest the time required to master it, Hibernate provides rewards in the form of increased productivity, increased maintainability, increased performance, decreased time to deployment, and vendor independence.

REFERENCES

Ambler, S. (2002). Mapping objects to relational databases (white paper).

AmbySoft Inc. Retrieved April 15, 2008, from www.ambysoft.com/mappingObjects.html

Arnold, K., Gosling, J., & Holmes, D. (2006). *The java programming language* (4th ed.). Addison Wesley.

Bauer, C., & King, G. (2007). *Java persistence with hibernate.* Manning

Begin, C., Goodin, B., & Meadors, L. (2007). *iBATIS in action*. Manning.

Codd, E. F. (1970, June). A relational model of data for large shared data banks. *Communications of the ACM, 13*(6), 377-387.

EJB 3.0. Retrieved April 15, 2008, from http://java.sun.com/products/ejb

Fussel, M. L. (1997). Foundations of object relational mapping. *Chimu Corporation.* Retrieved April 15, 2008, from www.chimu.com/publications/objectRelational/

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns elements of reusable object-oriented software.* Addison Wesley.

Hibernate. Retrieved April 15, 2008, from http://www.hibernate.org

iBATIS. Retrieved April 15, 2008, from http://ibatis.apache.org/

Java data objects (JDO). Retrieved April 15, 2008, from http://java.sun.com/javaee/technologies/jdo/index.jsp

Oracle TopLink. Retrieved April 15, 2008, from http://www.oracle.com/technology/products/ias/toplink/index.html

NHibernate for .net. Retrieved April 15, 2008, from www.nhibernate.org

Persistence. Retrieved April 15, 2008, from http://java.sun.com/technologies/persistence.jsp

POJO. Retrieved April 15, 2008, from http://www.martinfowler.com/bliki/POJO.html

Relational persistence for Java and .Net. Retrieved April 15, 2008, from http://www.hibernate.org

Spring framework. Retrieved April 15, 2008, from http://www.springframework.org/

KEY TERMS

Automatic Dirty Checking: A strategy for detecting which persistent objects have been modified by an application.

Entity Type: A class whose instances have their own persistent identity.

Many-to-Many Association: An association between class in which instances of the first class are associated with possibly many instances of the second class. Instances of the second class are associated with possibly many instances of the first class.

N+1 Selects Problem: The problem that occurs when an application retrieves an object from a database and then "visits" an object or collection of objects to which the original object has an association. A single SQL SELECT is issued for retrieving the original object but additional SQL SELECTs are required for the visitations.

Object/Relational Mapping Service: A service for mapping instances of classes in a domain model to tables and foreign key constraints in a relational database.

One-to-Many Association: An association between classes in which instances of the first class are associated with possibly many instances of the second class. Instances of the second class are associated with at most one instance of the second class.

One-to-One Association: An association between classes in which instances of the first class are associated with exactly one instance of the second class and vice versa. The association may be between a single object and itself.

Paradigm Mismatch Problem: A set of problems encountered by applications written in an object-oriented programming language when the application needs to store (retrieve) objects in (from) a relational database.

POJO: Plain old Java object.

Transitive Persistence: A technique that allows for the automatic propagation of persistence to objects associated with a given persistent object.

Value Type: A class whose instances do not have their own persistent identity.

ENDNOTES

1 (Begin, Goodin, & Meadors, 2007 p. 34)

2 Also known as the *object/relational impedance mismatch* problem.

3 Minimizing the number of required calls to the database is clearly a performance goal for an application supporting concurrency.

4 We chose an order entry system example because it is one that many readers will be familiar with. Our particular example is overly simplified of course and differs in a number of respects from the example given on p. 213 of the Hibernate reference manual.

5 As noted earlier, for those who eschew XML, another mechanism that can be used with Hibernate to enforce object persistence is the use of annotations. For reasons of space, we will not explore that approach in this chapter.

6 While it is possible to include the mapping information for several classes in a single .hbm.xml file, best practices dictates that each persistent class have its own .hbm.xml file.

7 This is not to say that the id property of customer has no value immediately after its creation. It certainly does. Since ints in Java default to 0, the value of id is 0. However, this value will more than likely change after customer becomes a *persistent* object.

8 To those whose linguistic sensibilities rebel against the idea of using "persists" in this way, that is, as a transitive verb, I apologize. However, as is very often the case in computer circles, jargon is pervasive in ORM circles and

in Hibernate's circle in particular. So I will persist in using "persists" in this way. ;-)

[9] This delayed execution of the generated SQL has its merits in terms of performance.

[10] or automatically bidirectional if you prefer

[11] These are sometimes also called *link tables* or *association tables*.

[12] Actually, the idea here is quite simple. If we have a many-to-many association between classes A and B, make a set of Bs an instance variable of A and a set of As an instance variable of B. The mapping documents (A.hbm.xml and B.hbm.xml) for A and B still need to ensure that the associations between A and B are bidirectional and that the cascade attributes are properly set. Bauer and King (2007 pp. 297-303) include a discussion of how to implement the mapping classes for a many-to-many association

[13] The quoted price may well be different from the unit price. The unit price property belongs in the Product class. The quoted price might be the result of a special discount (perhaps a special "deal" between the employee who took the order and the customer who submitted the order), or the result of any number of other things. Ultimately, the quoted price is the price that will affect the total cost of the order.

[14] Admittedly we are operating under the luxury of implementing our database from scratch. Were we dealing with a legacy database containing junction tables with composite primary keys and tables with natural keys, our approach would have to change. Be that as it may, Hibernate *does* contain support for both composite and natural primary keys. See Section 8.1 by Bauer and King (2007) for details.

[15] QOH represents quantity on hand. We have left out the schemata for the CUSTOMER and ORDERS tables for reasons of obviousness.

[16] Hibernate contains a set of its own collection wrappers for most of the usual Java collection classes like Set, List and Map. Notably, it also contains a collection wrapper for Bag.

[17] The use of the "distinct" keyword in this query reflects the fact that eager-fetching a collection may return duplicates. Though the keyword is maintained in the generated SQL, it has no effect at that level for this example. See the discussion by Bauer and King (2007, p. 651) for details.

[18] close() and clear() transition all persistent objects in the session to detached. evict() only affects the instance on which it is called.