Using Relational Databases to Provide Object Persistence

Learning Objectives

After studying this chapter, you should be able to:

- Concisely define each of the following terms: persistence, serialization, object-relational mapping (ORM), object-relational impedance mismatch, object identity, accessor method, call-level application programming interface, transparent persistence, separation of concerns, pooling of database connections, entity class, fetching strategy, N+1 selects problem, declarative mapping schema, and value type.
- Understand the fundamental mismatch between the object-oriented paradigm and the relational model and the consequences of this mismatch for the use of relational databases with object-oriented development environments.
- Understand the similarities and differences between the approaches that are used to address the object-relational impedance mismatch.
- Create a mapping between core object-oriented structures and relational structures using Hibernate.
- Identify the primary contexts in which the various approaches to addressing the object-relational impedance mismatch can be most effectively used.
- Understand possible effects of the use of the object-relational mapping approaches on database performance, concurrency control, and security.
- Use HQL to formulate various types of queries.

INTRODUCTION

In Chapter 13, you learned about the object-oriented approach to data modeling, leading to a conceptual domain model represented with the UML class diagram notation. As briefly discussed in Chapter 13, the object-oriented approach is not limited to data modeling but has been applied increasingly often to systems analysis, design, and implementation in recent years. It is useful to understand the broader context of object-oriented thinking before we discuss the core topic of this chapter: the integration between object-oriented application development models and relational databases. Object-orientation first emerged as a programming model in the context of languages such as Simula (as early as in the 1960s) and Smalltalk. It became mainstream and reached a dominant position in the 1990s, particularly through the widespread use of languages such as C++ and Java.
Simultaneously with its strengthening impact in the world of programming, the object-oriented approach started to have an increasingly strong role in analysis and design, to the extent that in early and mid-1990s the proliferation of object-oriented modeling techniques and tools made them very difficult for anybody to manage. This led to efforts to control the propagation of object-oriented modeling approaches through a concentrated, eventually successful attempt to integrate them into one, standard set. As a result, the Unified Modeling Language (UML), discussed in greater detail in Chapter 13, was born in 1997. UML, together with variants of the closely related Unified Process iterative process model, is one of the major reasons object-oriented analysis and design have become very popular. The two most widely used application development frameworks, Java EE and Microsoft .NET, are both based on object-oriented concepts and support object-orientation as the foundation for development.

One of the key characteristics of the object-oriented development approach is that the same core concepts can be applied at all stages and throughout the entire process of development. The same domain model that is identified at the conceptual level during requirements specification and analysis (as you learned in Chapter 13) will be directly transformed into a model of interconnected software objects. Many of the core object-oriented concepts (modeling the world with classes of objects, integrating behavior and data, inheritance, encapsulation, and polymorphism) can be applied seamlessly at different levels of abstraction. The object-oriented principles are applied across a broad spectrum of systems development activities, with one glaring exception: data management. For a long time, it was widely believed that object-oriented database management systems (OODBMSs) would gradually become very popular. These systems were intended to provide direct, transparent persistence for objects in object-oriented applications, and they were expected to become as widely used as object-oriented languages and systems analysis and design methods are.

For a variety of reasons, OODBMSs never took off. One of the reasons is simply organizational inertia: Starting in the 1980s, companies, government entities, and other users of large-scale database management systems (DBMSs) began to invest large amounts of money in relational database management systems. Moving to a new DBMS technology is, in practice, much more difficult than starting to use a new application development environment. Object-oriented databases also did not initially have the same types of powerful query capabilities as relational databases do. Moreover, the theoretical model underlying object-oriented databases is not quite as sophisticated and mathematically precise as it is in the relational world. In practice, potential user organizations clearly didn’t find the OODBMS technologies to be highly beneficial. These technologies were typically created and represented by smaller companies, and thus they seldom had the type of backing that would have made it possible for them to convince user organizations that these products are scalable and reliable for all types of uses.

It is not practical for object-oriented applications to maintain all relevant objects in run-time memory all the time; therefore, despite the failure of the OODBMSs to catch on, it is clear that object-oriented development environments need a mechanism for storing object states between the application execution sessions. Storing the state of an object between application execution sessions is called providing persistence to the object. Object-oriented languages provide a built-in mechanism for storing a persistent state of an object: Serialization refers to the process of writing an object onto a storage medium or a communication channel as a data stream. Serialization is not, however, a realistic mechanism to be used for storing large amounts of structured data, as is needed in most administrative applications: Its performance simply is not sufficiently scalable for purposes that require fast and constant response times, and it does not provide adequate support for the management of shared access to data, as database management systems do.

Thus, the problem caused by the existence of two fundamentally different ways to model the world remains. We have to find ways to address it; otherwise,
the object-oriented approach cannot be a realistic option for the development of large-scale administrative applications. In this chapter, we will first discuss the conceptual differences between the object-oriented and relational approaches (the “object-relational impedance mismatch”) in more detail. We will continue by describing the general characteristics of the different mechanisms that have been developed to close the gap between these two approaches. The chapter continues with a comprehensive example using Hibernate, a widely used object-relational mapping (ORM) technology, and a systematic review of how various object-oriented structures are mapped to the relational world.

**OBJECT-RELATIONAL IMPEDANCE MISMATCH**

The conceptual gap between the object-oriented approach to application design and the relational model for database design and implementation is often labeled as a mismatch. Chris Richardson (2006) and Scott W. Ambler (2006) call this phenomenon the “object relational impedance mismatch,” Christian Bauer and Gavin King (2006) refer to it as the “object/relational paradigm mismatch,” and Ted Neward (2005) simply calls it the “object/relational mismatch.” These authors have identified a large number of dimensions to this problem. They are summarized in Table 14-1.

The rest of this section discusses these dimensions of the object-relational impedance mismatch and illustrates them with examples.

An illustration often used to describe the problem related to the nature and granularity of data types is the way an address is expressed in object-oriented and relational data models. Let’s assume that we have a generic Person class (in the object-oriented world) or entity type (in the relational world) that has typical attributes such as Last Name, First Name, Date of Birth, etc. In addition, let’s assume that a person has two addresses (home and business) consisting of Street, City, Zip, State, and Country. In a typical object-oriented solution, Address would be modeled as a separate class with the attributes listed previously. Each object in the Person class would, in turn, include one or several address objects (see Figure 14-1a).

In a typical relational database solution, however, the components of the addresses would in most cases be represented as individual columns of the PERSON relation (as discussed in Chapter 4 in the context of representing composite attributes and illustrated in Figure 14-1b). Please note that we do not consider address here as being a multivalued attribute but treat the two addresses as two separate attributes. The object-oriented and relational representations of the domain capture the same information about the real-world situation, but their internal structures are quite different.

There are several differences in representing structural relationships between the core modeling elements in the object-oriented and relational worlds. We will discuss briefly two of them: inheritance and associations. As you learned in Chapter 13, inheritance and the related generalization-specialization hierarchy are some of the most important concepts in the world of object-oriented modeling. These same principles are applied

<table>
<thead>
<tr>
<th>TABLE 14-1 Elements of the Object-Relational Impedance Mismatch</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Nature and granularity of data types</td>
</tr>
<tr>
<td>• Structural relationships:</td>
</tr>
<tr>
<td>• Inheritance structures</td>
</tr>
<tr>
<td>• Representation of associations</td>
</tr>
<tr>
<td>• Defining the identity of objects/entity instances</td>
</tr>
<tr>
<td>• Methods of accessing persistent data</td>
</tr>
<tr>
<td>• Focus on data (relational databases) versus integrated data and behavior (the object-oriented approach)</td>
</tr>
<tr>
<td>• Architectural styles</td>
</tr>
<tr>
<td>• Support for managing transactions</td>
</tr>
</tbody>
</table>
Part V • Advanced Database Topics

FIGURE 14-1 Examples of different ways to represent Address

(a) Object-oriented representation

(b) Relational representation

In the relational world, associations between instances of entities are represented with foreign key values, as you learned in Chapter 4. These links are not directional by nature: Using relational joins, it is as easy for us to determine all orders that a customer has as it is to determine the customer for a specific order (see Figure 4-12). In the object-oriented world, associations are, however, always directional in nature. This is easy to see when you think about the way they are implemented in object-oriented programming languages, such as Java or C#. Let's assume that we have two classes, Customer and Order, which are associated with each other as follows:

```java
public class Customer {
    private Set<Order> orders;
    ...
}
public class Order {
    private Customer customer;
    ...
}
```

Even if you are not familiar with the syntax of the Java programming language, this example is probably clear enough to illustrate how customers and orders are associated with each other. Orders are represented as a collection (“Set”) of instances of class Order in Customer; this collection is called orders. A customer is, in turn, represented as an attribute customer of Order that has to be an object in the class Customer. Thus, both sides of this one-to-many relationship have to be represented explicitly. Every mechanism that is created to address the object-relational impedance mismatch issue has to address this issue in one way or another.

The identity of the core elements (objects/entity instances) is specified differently in the object-oriented and relational worlds. As you know based on Chapter 4, every row in a relational table has to have a unique primary key value, which determines the...
Object identity
A property of an object that separates it from other objects based on its existence.

Accessor method
A method that provides other objects with access to the state of an object.

identity of the row. As discussed in Chapter 13, in the object-oriented world, each object has its own identity simply based on its existence (i.e., its location in memory), and this **object identity** is not dependent on the values of any of the attributes of the object. Therefore, any system that provides a capability to map between relational and object-oriented representations has to include a mechanism to convert between these two approaches to separate objects/entity instances from each other.

Object-oriented environments and relational databases have a very different navigational model for accessing persistent data. In the object-oriented world, a typical way to access a data item is to call the **accessor method** associated with a specific attribute. Using the structures represented in Figure 14-2, we would first locate the correct order (anOrder) in the set orders and then call anOrder.getCustomer().getDiscountPercentage() to access the discount percentage for the order’s customer. The system would navigate from the anOrder object to the customer object associated with this order, using the getCustomer() accessor method of the order, and then call its getDiscountPercentage() method. The actual attribute is hidden (encapsulated) within the customer object and accessible only through the public accessor method.

In the case of a relational database, however, the same discount percentage would be accessed by using a single query that connects ORDER and CUSTOMER tables to each other using the CustomerID of ORDER as a foreign key that refers to CustomerID in CUSTOMER. The difference is not necessarily very clear when we evaluate just one order and customer, but the situation becomes much more complex if we need to access discount percentages for a large number of customers. In an object-oriented language, we would use some way to iterate over a collection of order objects and navigate separately to each customer. This would not, however, be a feasible approach in the relational database context: It would be highly inefficient to retrieve each ORDER–CUSTOMER row pair separately. Instead, we would declare the resulting set of values in advance in the SQL query and retrieve them all at the same time.

Relational databases and the processes that are used to design them focus primarily on data (with the exceptions of stored procedures and some object-relational extensions of relational databases), whereas object-oriented environments by definition integrate data and behavior. In Chapter 1, you learned about the benefits of separating data from applications as one of the key characteristics of the database approach. The entire object-oriented philosophy contradicts the separation between data and behavior that is so central for the database approach. This philosophical difference contributes to the gap between the relational and object-oriented approaches.

The relational database approach is based on a different set of architectural assumptions than the object-oriented approach. The relational approach fits very well with the

**FIGURE 14-2** Accessing a customer’s discount percentage with navigation
client/server model, as discussed in Chapter 8: Relational database management systems have been designed to respond to service requests that arrive in the form of an SQL query. The requests might be coming from an application server, from a Web server, from a Web client, or from a human user through a very simple text-based interface, but the idea is the same: A DBMS receives a query asking for either a specific set of data or an operation on data, executes it, and returns a response back to the client. In the object-oriented world, the situation is quite different: Data and the behavior that manipulates the data are intricately linked to each other and designed to be inseparable. Thus, when a relational database is used to store the persistent state of an object, the system as a whole has to take into account the linkage between data and behavior in the object-oriented world.

Finally, as you learned in Chapter 11, all database management systems have to offer a mechanism to manage transactions so that an abnormal interruption of a sequence of actions that belong together does not lead to an inconsistent state of the database. Object-oriented environments typically do not have an inherent, built-in concept of boundaries between transactions.

Let’s summarize the challenge that systems architects and application developers are facing: In application development, the object-oriented approach has gradually reached a dominant position, and a large percentage of software projects that include development of new applications is based on the object-oriented philosophy in some way. The most commonly used application development frameworks, Java EE and Microsoft .NET, are both object-oriented. At the same time, relational databases are almost invariably used as the mechanism to provide long-term persistence for organizational data. This is unlikely to change any time soon. As we demonstrated previously, these two approaches have significant conceptual differences, which require careful attention if we want them to coexist. We have no choice but to provide long-term object persistence for any realistic organizational application: The key reason why we have information systems in organizations is to maintain long-term information about the objects that are important for the business. Object-oriented applications need object persistence, and in the foreseeable future, the only technology that will provide that in a reliable, scalable way in the enterprise context are relational database management systems. Therefore, solutions for closing the gap between these two approaches are an essential component of any modern computing infrastructure.

**PROVIDING PERSISTENCE FOR OBJECTS USING RELATIONAL DATABASES**

Many different approaches have been proposed for addressing the need to provide persistence for objects using relational databases. Most modern relational database management systems offer object-oriented extensions, which are typically used for dealing with nonstandard, complex, and user-defined data types. In this chapter, however, our focus is on mechanisms that provide persistence support to a genuine object-oriented design and implementation model, and we will review the most common of those here.

**Common Approaches**

The most typical mechanisms for providing persistence for objects include call-level application programming interfaces, SQL query mapping frameworks, and object-relational mapping frameworks.

**CALL-LEVEL APPLICATION PROGRAMMING INTERFACES**

Since the early days of Java, Java Database Connectivity (JDBC) has been an industry standard for a call-level application programming interface (API) with which Java programs can access relational databases. If you are developing software using Microsoft’s .NET Framework, ADO.NET provides similar types of capabilities for providing access to relational databases. Open database connectivity (ODBC) is another widely used API for accessing data stored in relational databases from different types of application programs. All of these mechanisms are based on the same idea: An SQL query hand-coded by a developer is passed as a string parameter to the driver, which passes it on to the DBMS, which, in turn, returns the result.
as a set of rows consisting of (untyped) columns. The mechanisms have their differences (e.g., ADO.NET provides an intermediate DataSet construct), but conceptually they are very similar.

**SQL QUERY MAPPING FRAMEWORKS** The next category of tools provides additional support and a higher level of abstraction for using a relational database to provide object persistence by linking classes in an object-oriented solution to parameters and results of SQL queries (instead of database tables). These tools are not full-blown object-relational mapping tools because they do not generate the needed SQL based on a mapping between descriptions of tables and classes. They are, however, an “elegant compromise” (in the words of Tate and Gehtland, 2005) that hide some of the complexity of a pure JDBC or ADO.NET solution but still give the developers full access to SQL. The best-known tools in this category are iBATIS and iBATIS.NET. They consist of two components: iBATIS Data Mapper/SQL Maps, which are structures used to create a bridge between an SQL query and a Java object, and iBATIS Data Access Objects, which form an abstraction layer between the details of your persistence solution and the actual application.

**OBJECT-RELATIONAL MAPPING FRAMEWORKS** Comprehensive object-relational mapping (ORM) frameworks, such as the Java Persistence API (JPA) specification and its implementations Hibernate, OpenJPA, and EclipseLink, hide the relational data access methods from the object-oriented applications and provide an entirely transparent persistence layer. These frameworks, when integrated with an object-oriented application, move the management of the concerns related to persistence outside the core structure of the object-oriented applications. They often provide a declarative mapping schema that links domain classes needing persistence to relational tables and mechanisms for managing database transactions, security, and performance in ways that are hidden from the applications; alternatively, they create the mapping based on specific notes added to Java code called annotations. The classes for which an ORM framework provides persistence do not know that they are persistent: Persistent objects in these classes are created, loaded, and deleted by the ORM framework. Many ORM frameworks also include a query language, improve performance by optimizing the time when objects are loaded from the database, use caching to manage performance, and allow applications to detach objects that can be modified and, at a suitable time, made persistent again (Richardson 2006). The number of options in this category is quite large. The most widely used of them is Hibernate (and its .NET counterpart NHibernate), which is one of several implementations of the JPA. In addition to Hibernate, Apache’s OpenJPA and Eclipse Foundation’s EclipseLink (together with Oracle’s older, closely related TopLink) are widely used JPA implementations. The past few years have seen the parallel development of multiple ORM frameworks. At this time, JPA has emerged as the overall framework specification and Hibernate as the most popular implementation. In this chapter, we have chosen to use Hibernate as our vehicle for presenting the examples because of its long-standing status as the most widely used ORM framework and because its XML-based mapping specifications provide us with more visibility to the internal mapping structures.

**PROPRIETARY APPROACHES** Finally, there are many proprietary approaches for integrating data access directly into object-oriented environments and languages, such as Microsoft’s Language Integrated Query (LINQ), which is a component of the .NET Framework. The goal of LINQ is to very closely integrate data access queries into programming languages, not limiting the access to relational databases or XML but offering access any type of data store. The first version of LINQ, titled LINQ to SQL, was released as part of the first version of the .NET Framework 3.5; a more sophisticated but also more complex version of the technology, called LINQ to Entities, was released with .NET 3.5 SP1. LINQ to Entities is significantly closer to offering a full set of comprehensive ORM framework capabilities than LINQ to SQL, and it appears to form the foundation of Microsoft’s future efforts in this area.

---

**Transparent persistence** A persistence solution that hides the underlying storage technology.

**Declarative mapping schema** A structure that defines the relationships between domain classes in the object-oriented model and relations in the relational model.
Selecting the Right Approach

Which one of the four principal approaches to providing persistence for objects using relational databases should be used in a specific project? To help you understand the issues affecting this decision, we will continue by discussing the advantages and disadvantages of the first three approaches. We will not include the proprietary approaches (such as LINQ) in the comparison because none of them has become widely used yet, but we encourage you to follow developments in this area. Tables 14-2, 14-3, and 14-4 preview the advantages and disadvantages of each of these approaches.

CALL-LEVEL APIs JDBC and other call-level APIs with which object-oriented applications can connect directly to relational databases are still widely used, and many developers and organizations continue to utilize them because they allow the most direct access to the capabilities provided by the DBMS through SQL (see Table 14-2). Specifically, they do this

### TABLE 14-2 Advantages and Disadvantages of the Call-Level API Approach

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Low overhead</td>
<td>• Proliferation of code related to database connectivity</td>
</tr>
<tr>
<td>• Highest level of control over the details of the database connection</td>
<td>• Need to write a lot of detailed code</td>
</tr>
<tr>
<td></td>
<td>• Little reuse of code</td>
</tr>
<tr>
<td></td>
<td>• Developers need a detailed understanding of DBMS capabilities and the database schema</td>
</tr>
<tr>
<td></td>
<td>• SQL code not generated automatically</td>
</tr>
<tr>
<td></td>
<td>• The approach does not provide transparent persistence</td>
</tr>
</tbody>
</table>

### TABLE 14-3 Advantages and Disadvantages of the SQL Query Mapping Frameworks

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Direct access to all DBMS capabilities provided through SQL</td>
<td>• More overhead than with call-level APIs</td>
</tr>
<tr>
<td>• Mapping to legacy database schemas easier</td>
<td>• Developers need a detailed understanding of DBMS capabilities and the database schema</td>
</tr>
<tr>
<td>• Amount of code required significantly less than with call-level APIs</td>
<td>• SQL code not generated automatically</td>
</tr>
<tr>
<td>• Database access code easier to manage than with call-level APIs</td>
<td>• The approach does not provide transparent persistence</td>
</tr>
</tbody>
</table>

### TABLE 14-4 Advantages and Disadvantages of the Object-Relational Mapping Frameworks

<table>
<thead>
<tr>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>• They provide the highest level of persistence transparency</td>
<td>• There is more overhead than with call-level APIs and with query mapping frameworks</td>
</tr>
<tr>
<td>• Developers do not need to have a detailed understanding of the DBMS or the database schema</td>
<td>• Complex cases often need detailed attention</td>
</tr>
<tr>
<td>• The implementation of persistence is fully separated from the rest of the code</td>
<td>• Legacy databases lead to difficulties</td>
</tr>
<tr>
<td>• They enable true object-oriented design</td>
<td></td>
</tr>
</tbody>
</table>
without requiring the processing overhead that the other approaches unavoidably have (the others are, after all, built on top of a call-level API). The call-level APIs have, however, significant weaknesses: They expose the database structure to the application developers and require that the developers understand the underlying database. They also require that application developers be able to write database access (SQL) code manually. They distribute data access/persistence code to a potentially very large number of methods making it difficult to maintain. Finally, they violate the idea of separation of concerns (such as persistence) by including the persistence-related code in all objects. Using JDBC, ADO.NET, or ODBC to embed SQL code directly to an object-oriented solution is a very labor-intensive approach that is prone to errors, particularly in large applications. Often, these mechanisms work well when an application is small and in the early stages of its life cycle, but later they lead to bloated and heavily layered code. Thus, call-level APIs can be recommended primarily for small, simple applications that are unlikely to grow.

**SQL QUERY MAPPING FRAMEWORKS**  SQL query mapping frameworks (primarily iBATIS and iBATIS.NET) have several strengths compared to the call-level APIs (see Table 14-3): They significantly reduce the amount of code that is required to manage database connectivity. They allow application developers to operate at a higher level of abstraction and avoid the need to re-create low-level code repeatedly, thus reducing the probability of errors. Compared to the call-level APIs, their primary disadvantage is the extra overhead they unavoidably create, but that overhead appears to be at a reasonable level. A significant strength of these frameworks is that they give the developer full access to all the capabilities that a DBMS offers through SQL. Thus, they work well with legacy database schemas that might not have been designed to provide a good fit with object-relational mapping tools. They also allow developers to write sophisticated SQL queries, access stored procedures, and use nonstandard features of a DBMS. Whether the use of these options is considered good practice will, of course, depend on the user organization, but the SQL query mapping frameworks give developers these options. Particularly in the case of complex joins, SQL query mapping frameworks give more opportunities for fine-tuning the queries (assuming that the developer has a high level of competency in the use of SQL).

One disadvantage of these tools is that they are not genuine object-relational mapping tools because they do not create a conceptual connection between classes and tables. By definition, they require an in-depth understanding of the characteristics of the DBMS and the database schema, and an ability to write SQL queries. Unlike the genuine object-relational mapping frameworks, they don’t generate the SQL code automatically. As a student of data management topics this should, of course, not be a concern for you. SQL query mapping frameworks such as iBATIS appear to be particularly strong in situations where there is a complex, potentially nonstandard existing database structure and the task requires the execution of sophisticated queries resulting in a large number of rows.

**ORM FRAMEWORKS**  The genuine object-relational mapping frameworks have clear advantages from the perspective of the object-oriented approach (see Table 14-4). Specifically, the conceptual mapping between the classes and associations in the object-oriented world and the tables and relationships in the relational world has to be specified only once. After the mapping has been completed, a developer operating in the object-oriented world is not, at least in principle, required to write SQL queries or have an in-depth understanding of the underlying database structure. In addition, the frameworks have been designed to take care of caching and query optimization so that, in most cases, performance is at least at the same level as any homegrown solution based on direct use of call-level APIs. The ORM frameworks also have the advantage of allowing true object-oriented application design.

The ORM frameworks are not, however, without disadvantages: It is often difficult to map legacy database schemas to genuinely object-oriented application designs. With more complex designs, even clean-slate development is not as straightforward and automatic as simple examples might suggest; finding an optimal mapping solution requires detailed work. ORM frameworks create more overhead than the other two...
approaches because of the higher level of abstraction they create and, therefore, they potentially impose a performance penalty in certain situations. Reliable empirical data regarding ORM framework performance is, however, difficult to obtain. The ORM frameworks are particularly strong in situations where you have an opportunity to create a new database schema to provide persistence to your objects and the required database operations are not hugely complex.

It is also important to point out that particularly in the ORM framework category, you have multiple options from which to choose. Currently, Hibernate is the de facto industry standard and the most important implementation of the Java community standard JPA (which, in turn, is part of the Enterprise JavaBeans [EJB] 3.0 standard). Apache OpenJPA is the latest EJB 3.0/JPA implementation, and it also appears to be gaining users. In addition, there are probably dozens of other, non-JPA ORM frameworks; for example, Cayenne, TJDO, Prewayler, Speedo, and XORM are just a few of the open source frameworks. All of these implementations have strengths and weaknesses, which are likely to change over time. A detailed comparison of the products is beyond the scope of this text. It is, however, important that you know what the most important implementation options are and continuously evaluate their fit with your own development environment and projects.

OBJECT-RELATIONAL MAPPING EXAMPLE USING HIBERNATE

This section gives a practical example of object-relational mapping using Hibernate. In this example, we will present an object-oriented domain model for a simple university domain, a relational database representation of the same data, and the configuration files that are needed to implement the mapping. We will not describe here all the configuration details that are needed to create a functional solution because our focus is on helping you understand the conceptual ideas underlying the mapping.

Foundation

Figure 14-3 shows a UML class diagram that represents an object-oriented conceptual domain model of our area of interest, and Figure 14-4 shows a UML class diagram that represents a design model for the same set of classes. Despite the differences in notation, Figure 14-3 is relatively close to an enhanced ER model: The inheritance structure between Person and Student/Faculty can be expressed with the generalization/specialization structure in an EER model, and the association class Registration is very similar to an associative entity. This domain model does not yet include any behaviors. The similarity is not surprising because an EER model and an object-oriented domain model serve the same purpose: Each provides a conceptual representation of a real-world domain. Figure 14-4 is a design model that, instead of describing a real-world domain, uses the same diagram language to represent the design of an object-oriented software solution. It is important to note the difference: In Figure 14-3 the rectangle labeled Faculty refers to the real-world concept of a faculty member, whereas in Figure 14-4 the rectangle labeled Faculty (which looks exactly the same) refers to a software class called Faculty.

The inclusion of a particular software class in the software solution is, of course, directly derived from the domain model, and the objects in this software class directly correspond to the real-world faculty members. Still, the domain model and the design model describe different spaces. The objects that are instances of the design model classes are the ones for which our solution needs to provide persistence. (We have little control over the persistence of the real-world objects in the domain model.) Note that while some of the classes have identifying attributes, such as sectionNbr, there are no primary keys. Also, several of the associations between the classes indicate directional navigation. This means that to access objects in the Registration class, for example, the application has to navigate to them through the appropriate Section. With this design solution, there is no way to access a specific section through a specific registration.

Figure 14-5 presents a Java representation of the design model included in Figure 14-4. Note that each of the classes would also need a constructor without parameters
(so called no-arg constructor) and getter and setter methods; Hibernate requires these to operate correctly. Figure 14-6 (page 14-13), in turn, includes a possible relational model for a database serving this application. With both the object solution and the relational solution defined, we can now analyze the characteristics of the solution that links the two, using Hibernate as the object-relational mapping tool.

Mapping Files
The core element of Hibernate that defines the relationship between the object-oriented classes and relational tables is XML mapping files, which are typically named \(<Class name>.hbm.xml\). The following example appears to be relatively simple, but it reveals interesting mapping challenges.

In some cases, mapping files are very straightforward, as in the case of Course:

```xml
<class name = "registrations.Course" table = "Course_T">
  <id column = "CourseID">
    <generator class="native"/>
  </id>

  <property name = "courseNbr" column = "CourseNbr"/>
  <property name = "courseTitle" column = "CourseTitle"/>
  <set name = "sections" inverse = "true" table = "Section_T">
    <key column = "CourseID"/>
    <one-to-many class="registrations.section"/>
  </set>
</class>
```
Note that the mapping is based on the classes in the programming language (in this case, Java), not on the database structure. Therefore, the fundamental element is the class, followed by its attributes name and table, specifying the name of the programming language class (Course) and the corresponding table (Course_T). The <id> element specifies the primary key of the database table, which in this case is a nonintelligent key, CourseID. The <generator> element gives the DBMS instructions regarding how to create the primary key values. The <property> tags specify a mapping between an attribute of the programming language class and the name of the database column. Finally, we need to specify that a course has multiple sections (maintained in the Java attribute sections) and that those sections are persistently stored in table Section_T.

In the same way, we will specify the mapping for the class Section:

```xml
<class name = "registrations.Section">
  <id name = "id" column = "SectionID">
    <generator class = "native"/>
  </id>
  <property name = "sectionRegNbr" column = "SectionRegNbr"/>
  <property name = "sectionNbr" column = "SectionNbr"/>
  <property name = "semester" column = "Semester"/>
  <many-to-one name = "course">
    class = "registrations.Course"
    column = "CourseID"/
  </many-to-one>
</class>
```
FIGURE 14-5 Java implementation of the design model

```java
public abstract class Person {
    private Long id;
    private String lastName;
    private String firstName;
}

public class Student extends Person {
    private int yearMatriculated;
    private String studentID;
}

public class Faculty extends Person {
    private String office;
    private String facultyID;
}

public class Course {
    private Long id;
    private String courseNbr;
    private String courseTitle;
    private Set<Section> sections;
}

public class Section {
    private Long id;
    private String sectionRegNbr;
    private String sectionNbr;
    private String semester;
    private Faculty facultyMember;
    private Set<Registration> enrolledStudents;
    public double getAvgGrade() {
        // the body of the method is intentionally missing
    }
}

public class Registration {
    private Long id;
    private Student student;
    private String status;
    private String grade;
    private float numGrade;
}
```

FIGURE 14-6 Relational representation of the design model

```
<many-to-one name="faculty"
    class="registrations.Faculty"
    column="FacultyID"
    not-null="true"/>

<set name="enrolledStudents" table="Registration_T">
    <key column="SectionID"/>
    <composite-element class="registrations.Registration">
        <parent name="Section"/>
        <many-to-one name="student" column="StudentPersonID"
            class="registrations.Student" not-null="true"/>
        <property name="status" column="Status"/>
        <property name="grade" column="Grade"/>
        <property name="numGrade" column="NumGrade"/>
    </composite-element>
</set>
</class>
```
In this mapping, we are using the `<many-to-one>` tags to tell Hibernate that there is one course and there is one faculty member per course but that a course can have multiple sections, and a faculty member can be responsible for multiple sections. In addition, we are mapping the table `Registration_T` to the class `Registration`. They both represent the many-to-many relationship between `Student` and `Section`. In the Hibernate configuration file, this structure is called `composite-element`.

Let’s take a closer look at this mapping. The Java class `Section` includes as one of its attributes a set of objects that belong to the class `Registration`, called `enrolledStudents`. As the name suggests, this set includes one `Registration` object per student who is enrolled in a particular section. The relational representation of this set is defined by the configuration file segment that is inside the `<set>` </set> tags. The specification for the set includes its name (`enrolledStudents`), the table to which it is mapped (`Registration`), and the attributes that will be mapped to the columns of the table. `Status`, `Grade`, and `NumGrade` are attributes with a value, but `student` refers to an object in the class `Student`. This association is implemented as a foreign key relationship in the database. Hibernate also understands that the `Registration_T` table needs a composite primary key consisting of `SectionID` and `StudentPersonID`, which are both nonintelligent, system-generated primary key columns of the `Section_T` and `Student_T` tables, respectively.

The final configuration file that is needed for mapping the original Java representation to relational tables describes the mapping for the abstract superclass `Person` and its two subclasses, `Student` and `Faculty`. It is as follows:

```xml
<class name = "registrations.Person" table = "Person_T">  
  <id name = "id" column = "PersonID">  
    <generator class = "native"/>  
  </id>  
  <property name = "firstName" column = "FirstName"/>  
  <property name = "lastName" column = "LastName"/>  
  <joined-subclass name = "registrations.Student" table = "Student_T">  
    <key column = "StudentPersonID"/>  
    <property name = "studentID" column = "StudentID"/>  
    <property name = "yearMatriculated" column = "YearMatriculated"/>  
  </joined-subclass>  
  <joined-subclass name = "registrations.Faculty" table = "Faculty_T">  
    <key column = "FacultyPersonID"/>  
    <property name = "facultyID" column = "FacultyID"/>  
    <property name = "office" column = "Office"/>  
  </joined-subclass>  
</class>
```

Hibernate offers multiple ways to take care of the mapping of an inheritance hierarchy. In this case, we have chosen to use an approach often called “table per subclass.” This name is somewhat misleading because the approach requires a table for each class and subclass that requires persistence. The configuration file first specifies the way the superclass is mapped and then uses the `<joined-subclass>` tag to map the subclasses. Note that you do not need a separate configuration file for the `Student` or `Faculty` subclasses; this is all that is needed to map them.

Hibernate includes a tool (SchemaExport) that can be used to create SQL data definition language (DDL) scripts for creating a relational database schema described in a specific set of mapping files. The specific nature of the generated SQL will depend on the DBMS in use. For our example, using MySQL, a popular open-source DBMS, Hibernate generated the SQL DDL included in Figure 14-7.

Note how the tables `Student_T`, `Faculty_T`, and `Registration_T` do not have auto-generated primary keys because `Student_T` and `Faculty_T` get their primary keys from `Person_T`, and the primary key of `Registration_T` is a composite of the primary keys of `Section_T` and `Student_T`. Also, it is interesting to see how Hibernate names the
constraints so that they can be referenced later if the schema is updated. Finally, you should pay attention to the way Hibernate generates the foreign key constraints separately in a specific order. (Can you tell how this order is determined?)

**Hibernate Configuration**

You might have wondered how Hibernate knows to which DBMS and database it should connect and what the specific characteristics of the connection are. These characteristics are specified in a configuration file called hibernate.cfg.xml. This XML text file has multiple sections, focusing on different aspects of the connection. We will review the most important of them here. The first segment of the file specifies the characteristics of the database connection:

```xml
<!— Database connection settings —>
<property name="connection.driver_class">com.mysql.jdbc.Driver</property>
<property name="connection.url">jdbc:mysql://localhost/universityTest</property>
<property name="connection.username">username</property>
<property name="connection.password">password</property>
```

These settings include the driver to be used (in this case, the JDBC driver for MySQL), the URL for the database connection string (in this case, MySQL running on localhost), and the username and password to connect to the DBMS.
Another important segment specifies the <Class name>.hbm.xml files that Hibernate can use as its resources. In this example, we have three of them:

```xml
<!— list of the mapping configuration files —>
<mapping resource="registrations/Course.hbm.xml"/>
<mapping resource="registrations/Person.hbm.xml"/>
<mapping resource="registrations/Section.hbm.xml"/>
```

The most complex set of parameters is associated with the process of pooling of database connections. The designers of Hibernate recognize that it is not feasible to open a new connection to the DBMS every time an application wants to interact with the database. Every new active connection has a cost associated with it and, thus, it makes sense to use a pool of connections that are kept open and allocated to different users and users, depending on the need at a specific time. In this case, we chose to allocate four parameters, following Bauer and King (2006):

```xml
<!— Using the C3P0 connection pool —>
<property name = "hibernate.c3p0.min_size">10</property>
<property name = "hibernate.c3p0.max_size">30</property>
<property name = "hibernate.c3p0.timeout">250</property>
<property name = "hibernate.c3p0.max_statements">100</property>
```

The min_size parameter specifies the number of connections that are always open. The max_size parameter provides the upper limit for connections; more than max_size connections will lead to a runtime exception. Timeout specifies (in seconds) the amount of time a connection can be idle before it is removed from the pool. max_statements configures the maximum number of prepared statements that can be cached to improve Hibernate’s performance.

### MAPPING OBJECT-ORIENTED STRUCTURES TO A RELATIONAL DATABASE

In this section, we will evaluate the mapping between a core set of object-oriented structures and relational database designs. This collection is intended to be illustrative and not exhaustive. The discussion will not cover the details of the Hibernate configuration files or the Java implementation of the structures; the intent is to review the structures conceptually.

#### Class

In most cases, the relationship between a class in the object-oriented world and a relational table is one-to-one: There is one table for storing objects in each class. Specifically, this is true for *entity classes* or classes that represent real-world entities, such as Course, Section, or Person in our previous example. There are, however, situations in which a single table provides persistence to two or more classes. This is the case when a Java class is used to specify a *value type* instead of representing an instance of a real-world entity. Typical examples of value types could be, for example, PersonName, Address, or Grade. In each of these cases, the class has been created to specify a detailed representation of a value associated with entity instances. For example, the class PersonName exists so that all names of people in this system are represented in the same way (e.g., with the elements prefix, firstName, middleInitial, lastName, and suffix and with the method printFullName(), which specifies how a person’s full name should be shown). Objects in the value type classes are typically included in the same table with the object that “owns” them. Thus, if a Person class has among its attributes objects in PersonName and Address classes, the attribute values of these objects will be included in the same table with the attribute values that belong to a specific person (see Figure 14-8).
Inheritance: Superclass–Subclass

There are at least four ways in which an inheritance structure can be represented in a database (see Figure 14-9). They are all based on the same object-oriented representation (see Figure 14-9a). We discussed one of them, called table per subclass, in our example earlier in this chapter (see Figure 14-9b). In it, the abstract superclass (Person) and the concrete subclasses (Faculty and Student) were represented in separate tables. Bauer and King (2006) specify three other approaches: table per concrete class with implicit polymorphism (see Figure 14-9c), table per concrete class with unions (see Figure 14-9d), and table per class hierarchy (see Figure 14-9e). In both table per concrete class approaches, the attributes from the superclass are included in all the tables representing the subclasses and thus, there is no table representing the abstract superclass. In the table per class hierarchy approach, the model is reversed and the attributes from the subclasses are included in one table, the instances of which represent objects in all subclasses. Obviously, this table has to include attributes for both subclasses. Each of these approaches has its advantages and disadvantages; a detailed discussion of these is beyond the scope of this text.

One-to-One Association

A good example of a binary one-to-one association is an association between Employee and Position (Ambler, 2006) (assuming that we are not maintaining position history). In most object-oriented solutions, this would be modeled so that each object in the Employee class contains an object in the Position class, making navigation from an employee object to a position object possible. This association would in most cases be modeled with a foreign key reference from a row in the EMPLOYEE table to a row in the POSITION table (see Figure 14-10).

Many-to-One and One-to-Many Associations

Binary one-to-many types of relationships are by far the most common in both object-oriented application design and relational database design. As you already know, relational design breaks more complicated structures (such as a many-to-many relationship in conceptual data modeling) into multiple binary associations. A typical object-relational mapping context differentiates between many-to-one associations and one-to-many associations, depending on the side from which one is observing the association. Mapping can be done separately for each direction of the association. Regardless of the direction,
the mapping is typically done with two tables, one of which (the many side) includes a foreign key reference to the one side. If, for example, there is a one-to-many association between classes Product and ProductLine, the objects would be stored in tables PRODUCT and PRODUCTLINE and the association would be represented using a foreign key column ProductLineID as part of each row of PRODUCT (see Figure 14-11).
Directionality in Java does not change the relational mapping; unidirectional and bidirectional associations are stored with the same database structures.

**Aggregation and Composition**

Aggregation and composition are structures in object-oriented modeling that capture “has-a” types of associations between objects. Composition is stronger than aggregation: If an association is specified as a composition, it means that one side of the association has been specified as the whole, which manages the life cycle of the parts to the extent that the parts cannot exist without the whole. For example, the objects in Invoice and LineItem classes are associated together with composition: Objects in LineItem should not be able to exist without a corresponding object in Invoice. In practice, this means that the foreign key attribute in table LINEITEM that refers to the row in INVOICE has to be defined as NOT NULL (See Figure 14-12). The database representation of standard aggregation does not have any special features compared to a regular association.

**Many-to-Many Associations**

There are two main types of many-to-many associations in the mainstream object-oriented modeling approaches: a simple association with multiplicities greater than one at both ends and an association class with additional attributes. Still, the database representation is essentially the same in both cases: A new table is created to enable the capture of the many-to-many association, exactly the same way a many-to-many relationship in a conceptual data model is represented with a separate table in the relational representation (see Figures 4-13 and 4-15).
RESPONSIBILITIES OF OBJECT-RELATIONAL MAPPING FRAMEWORKS

Now that we have seen an example of object-relational mapping and reviewed how the core object-oriented structures are mapped to their relational counterparts, it is time to summarize the services object-relational mapping frameworks offer to applications (see also Table 14-5).

First, an ORM framework provides a layer of abstraction that separates object-oriented applications from the details of a specific database implementation. The manipulation of the persistence status of objects takes place using statements of the programming language, not with a separate database language.

Second, although one should not use the ORM frameworks without understanding the characteristics of the underlying databases and DBMSs, the frameworks have the responsibility for generating the SQL code for database access, which means application developers do not have to worry about that. An added benefit is that the code for database access does not have to be written for each of the classes separately, but the relationships between the class structures and the database schema are systematically and centrally defined.

Third, the ORM frameworks include tools for managing database performance in the context of object-oriented applications. As shown earlier in this chapter, a typical ORM framework is capable of using the services of a connection pool (such as C3P0) for the efficient management of expensive database connections. Another performance-related issue that is central in the use of ORM frameworks is the specification of fetching strategies, which define when and how the framework retrieves persistent objects to the run-time memory during a navigation process. A specific issue that has to be addressed is the \textit{N+1 selects problem}, which refers to a situation in which a poorly defined fetching strategy might lead to a separate SELECT statement for each associated object in a one-to-many relationship. For example, Hibernate uses, by default, so-called lazy loading, in which objects are retrieved from a database only when they are needed. The alternative is eager loading, in which all associated objects are always retrieved together with the object to which they are linked. Careful design of fetching strategies is very important from the perspective of achieving a high level of performance in applications based on an ORM framework.

Fourth, the ORM frameworks provide support for transactions and transaction integrity. This topic was covered in Chapter 11, so we will not discuss it again here in detail. The transaction support mechanisms in the ORM frameworks work together with standard transaction management tools, such as Java Transaction API (JTA), that are provided by many application servers (e.g., JBoss and WebSphere). The development of enterprise-level applications would not, in general, be possible without transaction support. It is particularly important in the ORM world because, in many cases, a change in a persistent object leads to cascading changes in the database, which all have to be either accepted or rejected together.

The ORM frameworks provide services for concurrency control, which was also covered in Chapter 11. Hibernate uses, by default, optimistic concurrency control, but its behavior can be modified when more stringent isolation guarantees are needed. The highest level of isolation in Hibernate is fully serializable isolation, which ensures—with a performance penalty—that transactions are executed one after another.

<table>
<thead>
<tr>
<th>TABLE 14-5 Typical Responsibilities of the Object-Relational Mapping Frameworks</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Provide a layer of abstraction between object-oriented applications and a database schema implemented with a DBMS leading to transparent persistence</td>
</tr>
<tr>
<td>• Generate SQL code for database access</td>
</tr>
<tr>
<td>• Centralize code related to database access</td>
</tr>
<tr>
<td>• Provide support for transaction integrity and management</td>
</tr>
<tr>
<td>• Provide services for concurrency control</td>
</tr>
<tr>
<td>• Provide a query language</td>
</tr>
</tbody>
</table>
Finally, the ORM frameworks often include a custom query language, such as HQL in Hibernate, and other mechanisms to run queries, such as direct SQL and the Criteria application programming interface (API) in Hibernate. We will provide here a few examples of ORM queries using HQL.

**HQL**

HQL, the query language in Hibernate, resembles SQL in many ways. Based on what you have learned about SQL in Chapters 6 and 7, you will be able to learn HQL easily. Using our university example presented previously, let’s see how data can be retrieved using HQL.

We will start with a simple example that is limited to one class and database table: We will be listing all available information for all people whose last name is Rosen. The simple HQL query to accomplish this is as follows:

```
from Person
where lastName = 'Rosen'
```

It is noteworthy that the result of this query will return relevant objects from both sub-classes (Faculty and Students) and, because the result is a list of objects, it is possible to determine whether a specific object is a student or a faculty member. Also, note that the select keyword is not mandatory in HQL, as it is in SQL.

If we want the result list to consist of a collection of attributes instead of whole objects in the source class, we can specify the attributes in the query as follows:

```
select firstName
from Person
where lastName = 'Rosen'
```

This is obviously very familiar to you: The statement is essentially the same as it would be as an SQL statement. Hibernate allows you to express a variety of constraints in the where clause, including those that utilize various built-in functions well-known to you from the SQL discussion. Within a single class, aggregate functions (sum, average, count, etc.) work essentially the same way as in single-table SQL queries. We will, however, see interesting differences in how joins are expressed in the HQL environment. According to Bauer and King (2006), HQL provides four different ways of expressing joins:

- An implicit association join
- An ordinary join in the FROM clause
- A fetch join in the FROM clause
- A theta-style join in the WHERE clause

In this chapter, we are primarily interested in the first two.

An implicit association join allows us to use object-oriented navigation in the queries. For example, if we wanted to list all sections with course numbers and titles and the names of the faculty members who teach them, we can simply express the query as follows:

```
select s.course.courseNbr, s.course.courseTitle, s.faculty.lastName, s.faculty.firstName
from Section s
```

The beauty of this query is that you don’t have to specify anywhere in the query how section and course or section and faculty are linked together. This information is
available in the Hibernate configuration files and will be used automatically without an explicit join (thus the name). The SQL query that is generated based on this HQL query is as follows:

```sql
select
    course1_.CourseNbr as col_0_0_,
    course1_.CourseTitle as col_1_0_,
    faculty3_1_.LastName as col_2_0_,
    faculty3_1_.FirstName as col_3_0_
from
    Section_T section0_,
    Course_T course1_,
    Faculty_T faculty3_
inner join
    Person_T faculty3_1_
on
    faculty3_.FacultyPersonID=faculty3_1_.PersonID
where
    section0_.FacultyID=faculty3_.FacultyPersonID
    and section0_.CourseID=course1_.CourseID
```

These types of joins work well in relatively simple and straightforward many-to-one cases. In this situation, each section is associated with one course and one faculty member; therefore, navigation to them is easy. We cannot, however, use this notation and navigate from a course object to its section objects, because the association between course and sections is one-to-many when looking at it from the perspective of the course. In these cases, and in many others that require more complex structures, we need to use explicitly expressed joins. Let's look at an example: If we want to list the registration numbers for all sections of a course labeled IS 360 in Spring 2010, we can use the following HQL code:

```sql
select s.sectionRegNbr
from Section as s
    join s.course c
where c.courseNbr='IS 360' and s.semester='Spring 2010'
```

Note that we do not specify here how section and course are linked; the join simply states that the basis for the join is the course attribute of class Section.

HQL converts this query to the following SQL code, which is very similar, except that it explicitly expresses the foreign key (CourseID in Section_T)–primary key (CourseID in Course_T) link between Section_T and Course_T:

```sql
select section0_.SectionRegNbr as col_0_0_
from
    Section_T section0_,
inner join
    Course_T course1_
on section0_.CourseID=course1_.CourseID
where
    course1_.CourseNbr='IS 360'
    and section0_.Semester='Spring 2010'
```
If we want to add the name of the faculty member in the answer set, we need to add another join statement, as follows:

```sql
select s.sectionRegNbr, f.lastName, f.firstName
from Section as s
join s.course c
join s.faculty f
where c.courseNbr='IS 360' and s.semester='Spring 2010'
```

The SQL representation is as follows:

```sql
select
section0_.SectionRegNbr as col_0_0_,
faculty2_1_.LastName as col_1_0_,
faculty2_1_.FirstName as col_2_0_
from
Section_T section0_
inner join
Course_T course1_
on section0_.CourseID=course1_.CourseID
inner join
Faculty_T faculty2_
on section0_.FacultyID=faculty2_.FacultyPersonID
inner join
Person_T faculty2_1_
on faculty2_.FacultyPersonID=faculty2_1_.PersonID
where
course1_.CourseNbr='IS 360'
and section0_.Semester='Spring 2010'
```

Note that there is an additional join that is implicit in the query: Faculty inherits its name attributes from Person, which is expressed in the last join of this query.

Let’s move to a more complicated case: With the following HQL query, we can retrieve the detailed information for all students who took IS 360 in Spring 2010:

```hql
select c.courseNbr, s.sectionNbr, s.sectionRegNbr,
f.lastName, f.firstName, st.lastName, st.firstName,
e.grade
from Section as s
join s.course c
join s.faculty f
join s.enrolledStudents e
join e.student st
where c.courseNbr='IS 360' and s.semester='Spring 2010'
```

As you can see, adding more joins is very straightforward. Course, faculty, and enrolledStudents are all attributes of Section, and the class Student is associated with Section through the enrolledStudents collection. It is not difficult to imagine that the list
of joins is equally long and somewhat more verbose in SQL (because SQL has to represent the foreign key–primary key pairs explicitly):

```
select
course1_.CourseNbr as col_0_0_,
section0_.SectionNbr as col_1_0_,
section0_.SectionRegNbr as col_2_0_,
faculty2_1_.LastName as col_3_0_,
faculty2_1_.FirstName as col_4_0_,
student4_1_.LastName as col_5_0_,
student4_1_.FirstName as col_6_0_,
enrolledst3_.Grade as col_7_0_
from
Section_T section0_
inner join
Course_T course1_
on section0_.CourseID=course1_.CourseID
inner join
Faculty_T faculty2_
on section0_.FacultyID=faculty2_.FacultyPersonID
inner join
Person_T faculty2_1_
on faculty2_.FacultyPersonID=faculty2_1_.PersonID
inner join
Registration_T enrolledst3_
on section0_.SectionID=enrolledst3_.SectionID
inner join
Student_T student4_
on enrolledst3_.StudentPersonID=student4_.StudentPersonID
inner join
Person_T student4_1_
on student4_.StudentPersonID=student4_1_.PersonID
where
course1_.CourseNbr='IS 360'
and section0_.Semester='Spring 2010'
```

We will complete the HQL example with an aggregate query that requires several joins. This query lists the average of the numeric values of the grades given by each instructor in Spring 2010:

```
select f.lastName, f.firstName, avg(e.numGrade)
from Section s
join s.course c
join s.faculty f
join s.enrolledStudents e
join e.student st
where s.semester='Spring 2010'
group by f.facultyID, f.lastName, f.firstName
```

As you see, the aggregation mechanism is very similar to that of SQL. FacultyID is added to the group by clause to make sure that possible name duplicates are identified.

HQL provides a much broader set of capabilities, but most of them should be quite familiar to you, based on your SQL expertise.
Summary

The object-oriented approach has become very popular in application development and systems analysis and design, but object-oriented database management systems never gained widespread acceptance. Instead, relational database management systems continue to maintain their dominant role as the primary data management technology. Therefore, it is essential that relational databases be used effectively with object-oriented application development approaches.

In this chapter, we first reviewed the reasons underlying the object-relational impedance mismatch—that is, the conceptual conflict between the object-oriented and relational models. These reasons include differences in the representation of complex data types and structural relationships (including inheritance and associations), representation of object/entity instance identity, importance and implementation of the transaction concept, and methods of accessing persistent data. In addition, the approaches have a different core focus because the relational model focuses entirely on data whereas the object-oriented approach, by definition, integrates data and behavior. Also, the predominant architectural styles within each approach are different. It is essential that the gap between the two approaches be closed because, in practice, both will be used widely in the foreseeable future.

There are four basic categories of mechanisms through which relational databases can be used to provide persistence to objects. Call-level application programming interfaces (APIs), such as Java Database Connectivity (JDBC), require that application developers embed SQL statements in the program code through a low-level interface. SQL query mapping frameworks, such as iBATiS, raise the level of abstraction by providing a mechanism for declaring links between class specifications and SQL queries and by hiding the details of the call-level APIs, such as Java Persistence API and its implementations Hibernate, EclipseLink, and OpenJPA, provide a transparent persistence solution by creating declarative mapping between classes and database tables. They hide the database structure and the relational query language from developers. Finally, there are many proprietary persistence solutions that intend to integrate data access directly into object-oriented environments and languages, such as Microsoft’s LINQ. Each of the approaches has strengths and weaknesses, and it is essential that you carefully evaluate the specific needs of your project before selecting a tool for linking relational databases to an object-oriented development environment.

An example using Hibernate, a leading ORM framework, demonstrated the use of XML mapping files to declare the mapping between the object-oriented and relational concepts, including both classes and various structural associations between them on the object-oriented side and tables and their relationships on the relational side. Mapping individual, unconnected tables is very straightforward, but the solutions become immediately more complex when various types of associations and other structural relationships are mapped. The mapping process requires an in-depth understanding of both object-oriented application development and relational databases. In addition to the declarative mapping of the concepts, ORM frameworks must be configured to work with a specific database management system.

Object-relational mapping frameworks have multiple responsibilities: They provide a layer of abstraction between object-oriented applications and a database schema implemented with a DBMS to provide transparent persistence for the applications. They generate the SQL code that is needed for database access, and they centralize this code so that it does not proliferate throughout the application. These frameworks provide support for concurrency control and transaction integrity and management. They also typically include a query language (such as HQL in Hibernate) that provides capabilities similar to those of SQL.

Understanding the mechanisms used for linking object-oriented applications and relational databases is very important for both those whose specialty is data management and those who focus on application development. For data management specialists, an increasing number of the applications that they support are developed using the object-oriented approach. To provide high-quality service to these applications (and their developers), it is essential that data specialists understand how these applications connect to relational databases. Application developers, on the other hand, benefit greatly from understanding at least the principles of the mechanisms that provide persistence for the objects in their solutions. It is particularly important that both sides be able to communicate effectively with each other. The quality of the object-relational connection solution directly affects application performance, reliability, and security.

Chapter Review

Key Terms

<table>
<thead>
<tr>
<th>Accessor method 14-5</th>
<th>Fetching strategy 14-20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call-level application programming interface 14-6</td>
<td>N+1 selects problem 14-20</td>
</tr>
<tr>
<td>Declarative mapping schema 14-7</td>
<td>Object identity 14-5</td>
</tr>
<tr>
<td>Entity class 14-16</td>
<td>Object-relational impedance mismatch 14-3</td>
</tr>
<tr>
<td></td>
<td>Object-relational mapping 14-3</td>
</tr>
<tr>
<td></td>
<td>Persistence 14-2</td>
</tr>
<tr>
<td></td>
<td>Pooling of database connections 14-16</td>
</tr>
<tr>
<td></td>
<td>Separation of concerns 14-9</td>
</tr>
<tr>
<td></td>
<td>Serialization 14-2</td>
</tr>
<tr>
<td></td>
<td>Transparent persistence 14-7</td>
</tr>
<tr>
<td></td>
<td>Value type 14-16</td>
</tr>
</tbody>
</table>
Review Questions

1. Define each of the following terms:
   a. object-relational impedance mismatch
   b. object-relational mapping
   c. persistence
   d. call-level application programming interface
   e. transparent persistence
   f. JDBC
   g. iBATIS
   h. Hibernate
   i. N+1 selects problem
2. Compare and contrast the following terms:
   a. object identity; primary key value of a row in a database table
   b. entity class; value type
   c. OODBMS; RDBMS
   d. many-to-one association; one-to-many association
   e. lazy loading; eager loading
   f. JPA; Hibernate
3. Explain the reasons object-oriented database management systems never became very popular.
4. Briefly describe the factors that contribute to the object-relational impedance mismatch.
5. Explain how the object-oriented and relational approaches to accessing data differ from each other.
6. What is the key difference in how entity instance identities are defined in the object-oriented and relational worlds?
7. Why is it essential that relational databases be effectively used to provide persistence for objects in applications developed using the object-oriented paradigm?
8. Why can the link between object-oriented applications and relational databases not be built simply by using JDBC or some other call-level application programming interface?
9. Explain the main conceptual difference between iBATIS and Hibernate.
10. Explain the criteria that you might use to select between iBATIS and Hibernate.
11. Why is transparent persistence so important from the perspective of application developers?
12. Some developers are concerned about the overhead that SQL query mapping frameworks and ORM frameworks add to call-level APIs. Why?
13. What is the relationship between Hibernate and JPA?
14. What is the purpose of the <Class name>.hbm.xml files in Hibernate?
15. How is Hibernate configured for a specific database management system environment?
16. How are attributes specified in the Hibernate configuration files?
17. What is the purpose of the <set> tag in the Hibernate configuration files?
18. Explain how primary keys of the database tables are specified within the Hibernate environment.
19. What is the purpose of the SchemaExport tool in Hibernate?
20. Referring to the SQL code in Figure 14-7, explain why Student_T, Faculty_T, and Registration_T do not have auto-generated primary keys.
21. Explain the importance of pooling database connections.
22. Briefly describe the four different ways in which an inheritance structure can be mapped to a relational schema.
23. Explain why it makes sense to differentiate between many-to-one and one-to-many associations in the object-oriented world.
24. What is the practical impact of specifying an association as composition from the perspective of object-relational mapping?
25. Explain the importance of well-designed fetching strategies.
26. When is the select keyword necessary in HQL?
27. What is an implicit association join?
28. Analyze the queries that include explicit joins in the HQL queries and their SQL counterparts. What is the main difference between these two query types?

Problems and Exercises

1. Create a set of <Class name>.hbm.xml files that would generate a database schema presented in Figure 4-5.
2. Create a domain model (expressed in the UML class diagram notation) corresponding to the <Class name>.hbm.xml files you created in Problem and Exercise 1.
3. Create a domain model (using the UML class diagram notation) corresponding to the EER model in Figure 4-33.
4. Create a relational schema that is compatible with the domain model specified in Problem and Exercise 3 and a set of <Class name>.hbm.xml files that map between the domain model and the relational schema.
5. Create a domain model (using the UML class diagram notation) corresponding to the EER model in Figure 4-38.
6. Create a relational schema that is compatible with the domain model specified in Problem and Exercise 5 and a set of <Class name>.hbm.xml files that map between the domain model and the relational schema.
7. Create a domain model (using the UML class diagram notation) corresponding to the EER model in Figure 4-36.
8. Create a relational schema that is compatible with the domain model specified in Problem and Exercise 7 and a set of <Class name>.hbm.xml files that map between the domain model and the relational schema.

Problems and Exercises 9 through 17 all pertain to the Java implementation specified in Figure 14-5. Write HQL queries for these exercises. If you need additional details regarding HQL, see www.hibernate.org/hib_docs/v3/reference/en/html/queryhql.html.

9. Assume that a course number consists of a two-letter designator (such as IS) and a three-digit number. Find the titles and numbers of all 300-level IS courses.
10. Find the titles and numbers of all 300-level IS courses that are offered in Spring 2010.
11. Find the information that is needed in order to give each faculty member a list of students who take at least one of their courses during the current semester.
12. Find the names and matriculation years for those students who are enrolled in IS 440 in Spring 2010.
13. Find the average grade earned in IS 460 by students who matriculated in 2008, regardless of when they took the course.
14. Find the total number of credit hours (together with basic student information) for each of those students who first matriculated in 2008.
15. Find the names and office locations of those faculty members who taught IS 350 in Spring 2010.

16. Find the names and office locations of those faculty members who taught more than one course in Spring 2010.
17. Find the names and e-mail addresses of the students who took at least one IS course in 2009–2010 or 2010–2011, including in the answer information regarding the number of times the students took a specific course.

Field Exercises

1. Interview database administrators in a few organizations and discuss the mechanisms they have chosen to use for linking object-oriented applications to relational databases. Document the current status and future plans of the organizations.
2. In this chapter, we discussed three different implementations of the Java Persistence API (JPA) standard. Using resources on the Web, evaluate the current competitive situation between these three. Are there any others that are emerging as potential additional competitors?
3. Review Microsoft’s plans for the future of LINQ. Compare LINQ with JPA and its implementations. What are the factors a decision maker should take into account when choosing between the two?
4. Interview database administrators in a few organizations and discuss why the use or don’t use object-oriented database management systems. Document the current status and future plans of the organizations.

References


Richardson, C. 2006. POJOs in Action. Greenwich, CT: Manning.


Further Reading


Web Resources

www.java-source.net/open-source/persistence A collection of links to various open source persistence frameworks.

www.hibernate.org The Hibernate Web site.

http://java.sun.com/javaee/overview/faq/persistence.jsp An official Sun site that provides a description of the persistence standard in Java EE.