4-28-2007

Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications

Jeffrey M. Barnes
Macalester College, J242110559@aol.com

Object-Relational Mapping as a Persistence Mechanism for Object-Oriented Applications

An honors project presented by

Jeffrey M. Barnes

to

the Department of Mathematics and Computer Science
at Macalester College
in Saint Paul, Minnesota,

in partial fulfillment of the requirements
for the major in computer science

Advisor: Professor Elizabeth Shoop
Second reader: Professor G. Michael Schneider
Third reader: Professor Daniel Kaplan

April 28, 2007

Abstract

Many object-oriented applications created today, especially Web applications, use relational databases for persistence. Often there is a straightforward correspondence between database tables on the one hand and application classes on the other. Application developers usually write a great deal of code to connect to and query the database, code which differs little from class to class and is often tedious to write. Moreover, the parallel class and table structures constitute a duplication of information, which requires duplication of work and increases the likelihood of errors. Ideally, we could automate this duplication, rendering it invisible to developers. This is the idea behind object-relational mapping (ORM), which achieves the mapping between the object-oriented world and the relational world automatically. Many existing ORM tools do not realize the goal of fully transparent persistence, however, and ORM tools have not become pervasive in the software industry. We survey ORM technology, probing issues of ORM system architecture and examining real-world ORM systems. We consider the state of the art in ORM, asking why it is not more popular than it is and anticipating the future course of ORM system development.

Contents

Chapter 1

Introduction

A huge number of applications created today, especially Web applications, rely on two broad technologies: object-oriented programming (OOP) languages and relational database management systems (RDBMSs). Object-oriented technologies allow applications to be built out of logical objects that act on one another. Meanwhile, relational database technologies store data in tabular structures, capture the relationships between these tables, and support querying of data. These two technologies are superficially disparate but are related in two ways. The first way is practical: object-oriented languages and RDBMSs complement one another nicely. RDBMSs provide a framework for object-oriented applications to persist their data, and conversely object-oriented applications provide an interface between the data and the user and also allow for processing and manipulation of data beyond what can be accomplished by an RDBMS. The second way is theoretical. There is an obvious correspondence between database tables on the one hand and OO classes on the other. A typical application might include a `Customer` table in the database with `Name`, `Address`, and `TelephoneNumber` columns and a corresponding `Customer` class in the application code with `Name`, `Address`, and `TelephoneNumber` member data.

Object-relational mapping (ORM) is a technology that seeks to automate the bridge between the object-oriented world and the relational world, eliminating this duplication of data and the maintenance cost and susceptibility to error associated with it. A great variety of ORM tools exists today, and new tools are developed every day. However, many existing ORM tools do not realize the goal of fully transparent data persistence, and ORM tools have not gained overwhelming popularity in the software industry.

This paper reviews the state of the art of ORM and consider issues of deployment and transparency. Its structure is as follows. Chapter 1 introduces the concepts underlying ORM. It begins with an in-depth introduction to object-oriented and relational technologies. (Software developers familiar with these technologies can probably skim or skip to Section 1.4.) Next, it discusses the way in which these two technologies relate to one another and the so-called

impedance mismatch between them. Finally, it introduces and motivates ORM and discusses some foundational, operational issues that all ORM systems must consider.

Chapter 2 discusses components and features of ORM technologies in much greater depth. It presents issues that ORM system developers face and dimensions along which existing ORM technologies vary.

Chapter 3 is an in-depth survey of the most popular and interesting ORM technologies that exist today. ORM systems have appeared not only in classical object-oriented languages like Java and C#, but also in surprising contexts, in languages that can only barely be considered object-oriented at all. Chapter 3 concludes with a head-to-head comparison of the features that the most widespread and most capable ORM systems provide.

Chapter 4 concludes the paper by summarizing the present state of ORM technology and suggesting the course of its future development.

1.1 The object-oriented paradigm

Object-oriented programming (OOP) is today the dominant paradigm in mainstream software development, although it had a tentative beginning. Simula introduced many of the fundaments of OOP in the 1960s, but it is only within the last couple of decades that OOP has become popular in the mainstream. It has indeed become popular, though; it is hard to overstate the prevalence of OOP in industrial, scientific, and didactic applications.

OOP expresses itself by means of analogies with the real world. The idea is to construct programming abstractions that model objects in the real world. As a programming abstraction, an *object* is a discrete entity with state, behavior, and identity. Ideally, it corresponds to a well-defined, meaningful "business object" in the problem domain, such as a particular customer that a company wishes to keep track of. Objects can interact with each other, exchange messages, and process data.

In an object-oriented language, objects are grouped together into *classes*, which encapsulate functionality common to many objects. For example, many customer objects may be members of a single "customer" class, which represents the set of all customers. The customer class defines the structure and behavior common to all customers. For example, it records that all customers have a "name" property, and all customers have the behavior of "buying" an item.

More explicitly, a class defines the state (i.e., the abstract data structure that all class instances share) and behavior. The abstract data members defined by a class are called its *attributes*, *properties*, or simply *data members*. The behaviors it defines are called its *methods*. The following code snippet is a class definition in Simula 67, the first object-oriented language.

```
Class Airplane (WingspanInMeters, LengthInMeters);
    Real WingspanInMeters, LengthInMeters;
Begin
```

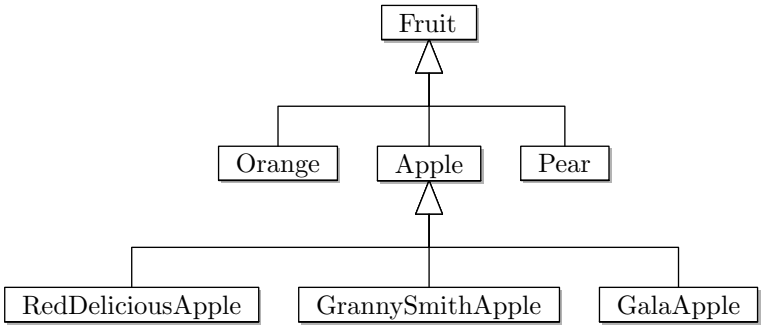

Figure 1.1: A simple inheritance hierarchy

```
    Real WingspanInFeet, LengthInFeet;

    Procedure Convert;
    Begin
        WingspanInFeet := 0.3048 * WingspanInMeters;
        LengthInFeet := 0.3048 * LengthInMeters;
    End of Update;

    Convert;
    OutText("New airplane has wingspan ");
    OutText(WingspanInMeters);
    OutText("meters (");
    OutText(WingspanInFeet);
    OutText(" feet).");
    OutImage
End of Airplane;
```

1.1.1 Inheritance

One of the most important techniques of OOP is *inheritance*, in which the programmer creates new classes by extending existing ones. The derived classes *inherit* the behavior and structure of the base class and add functionality on top of this. A less common name for inheritance is *generalization*, as base classes can be viewed as generalizations of their derived classes. For example, there could be classes `Apple`, `Orange`, and `Pear` that have `Fruit` as a base class. Often, inheritance relations form a tree: `RedDeliciousApple`, `GrannySmithApple`, and `GalaApple` could be derived classes of `Apple`, forming a tree as depicted in Figure 1.1. (Of course, there would need to be a practical reason that `RedDeliciousApple` needs additional functionality beyond that offered by `Apple`; otherwise, there is no purpose in creating a new class for it.) In practice, software architects try to limit the depth of inheritance trees for the benefit of

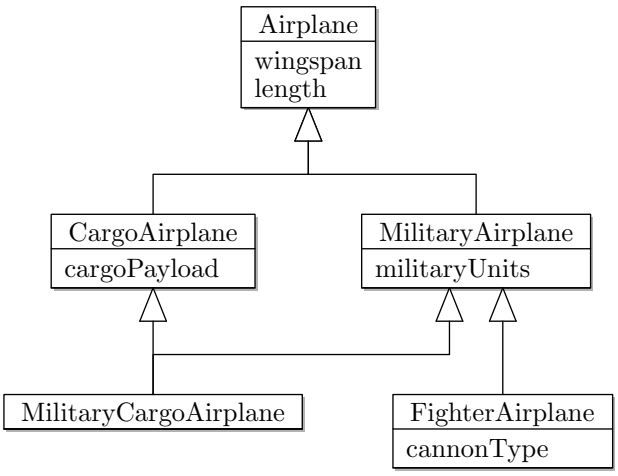

Figure 1.2: A diamond-shaped inheritance graph of the sort that could occur only in a language with multiple inheritance. Multiple inheritance affords additional flexibility; in this example, the inheritance structure allows us to record that some airplanes are both cargo airplanes and military airplanes. Moreover, for such a military cargo airplane, we can record both its cargo airplane information (its cargo payload) and its military airplane information (the military units that use it). Of course, this introduces complexity.

code comprehensibility. The following Simula 67 code creates a subclass of the `Airplane` class we just created.

```
Airplane Class PassengerAirplane (NumberOfPassengers);
    Integer NumberOfPassengers;
Begin
    comment Attributes and methods specific to
        passenger airplanes belong here;
End of PassengerAirplane;
```

Some languages, such as C++, Python, and Common Lisp, allow *multiple inheritance*, in which classes can inherit features from more than one base class. In such a language, we could realize an inheritance hierarchy such as the one depicted in Figure 1.2 (although we might not want to; diamond-shaped inheritance graphs often elicit confusion). Others, such as Smalltalk and Modula-3, exclude multiple inheritance. Many languages take some sort of intermediate approach. For example, Java, the .NET Framework, and Objective-C have both classes and *interfaces* (called *protocols* in Objective-C). An interface specifies a set of methods that inheritors must implement, but omits their implementation entirely, leaving the specification of the implementation entirely to derived classes. These three environments support multiple inheritance for interfaces but only single inheritance for classes. That is, a class in any of

these environments inherits from exactly one class and implements zero or more interfaces. For example, in the ASP.NET architecture of the .NET Framework there is a `Button` class that models a push button control on a Web page. The `Button` class inherits from the `WebControl` class, which defines methods, properties, and events common to all Web controls (such as a `Font` property and a `RenderControl` method) but also the `IButtonControl` interface and the `IPostBackEventHandler` interface. The `IButtonControl` interface defines properties and events that must be implemented to allow a control to act as a button, such as a `Click` event. The `IPostBackEventHandler` interface defines the `RaisePostBackEvent` method, which server controls must implement to handle post-back events. Mixed approaches to multiple inheritance such as this can alleviate much of the confusion that multiple inheritance can cause.

1.1.2 Encapsulation

Another often-cited property of object-oriented systems is *encapsulation*. This term may have any of a few different meanings. Armstrong [4] mentions three primary concepts of encapsulation that exist in the literature. Here we consider each concept in turn and the degree to which it applies to the object-oriented languages in existence.

- Encapsulation is a process used to package data together with the functions that act on it. In the context of OOP, encapsulation refers to the property that data and methods are packaged together into classes. Thus, all OOP languages support encapsulation, as all OOP languages have classes with both data and methods.

  More strictly, though, we could say that for an object to be called encapsulated, *all* operations on its state must be part of its interface. In this strict sense, few languages enforce or even accommodate encapsulation. In Java, for example, to calculate the sine of a number `x`, one writes `Math.sin(x)`. Strict encapsulation would demand `x.sin()`.

- Encapsulation means hiding the details of objects' implementation so that clients access objects only via their public interfaces. Encapsulation is beneficial because it frees clients from concerning themselves with classes' implementations. If the implementation of a class changes, clients can ignore the change without risk as long as the public interface and logical behavior of the class remain the same.

  Many object-oriented languages accomplish encapsulation in two ways. First, clients have no way of examining the implementation code of classes' methods, only their signatures, so method implementation is hidden from clients. Second, many object-oriented languages have a feature called *access*, which allows class members (including both attributes and methods) to be declared as *private* instead of public, making them accessible only to the class itself, not to clients. Often there is also a `protected` access specifier that makes members accessible only to derived classes, and sometimes

there are other access specifiers such as `friend` access in C++, which grants access only to declared "friends" of the class, and `internal` access in C#, which grants access only to clients in the same assembly. Because of the benefits that encapsulation affords, software architects generally make most data members private and expose only those methods that are necessary for clients to use the class. The following C++ class has a private data member with accessor functions that allow that data member to be accessed and set, as well as other functions for doing calculations.

```
const double PI = 3.14159;

class Circle {
private:
    double radius;
public:
    Circle();
    double getRadius();
    void setRadius(double r);
    double Circumference();
    double Area();
}

Circle::Circle() {
    radius = 0;
}

double Circle::getRadius() {
    return radius;
}

void Circle::setRadius(double r) {
    radius = r;
}

double Circle::Circumference() {
    return 2 * PI * radius;
}

double Circle::Area() {
    return PI * radius * radius;
}
```

If the implementation of the class should ever change, clients need not even know about it, let alone adapt to it. Of course, in such a simple example, this is not a serious concern, but this is important in more substantial cases.

- Encapsulation is a combination of these two ideas. More explicitly, encapsulation entails packaging data with the functions that act on it into a discrete unit whose implementation details are hidden from the outside world.

Henceforth we will refer to these three senses by the names *separation of concerns*, *information hiding*, and *encapsulation*, respectively. Separation of concerns and information hiding both predate OOP, but they are cornerstones of OOP nonetheless.

1.1.3 Polymorphism

Another oft-cited concept in definitions of OOP is *polymorphism*, which allows a single interface to be used with different types of data. There are several types of polymorphism that appear in object-oriented (and other) languages:

Ad hoc polymorphism This is the most limited form of polymorphism. It allows a function (or class, etc.) to be parameterized on a small (explicitly enumerated) set of types. The clearest example is *method overloading*. The `Math` class in the `java.lang` Java package, for example, has several `abs` methods. All the `abs` methods take a single argument, but the parameters have different types. There is one `abs` method that takes an `int` and returns its absolute value, also as an `int`; another takes a `double` and returns its absolute value as a `double`; and so on. When a client calls `Math.abs`, Java calls the appropriate overload based on the type of the parameter. If Java did not support ad hoc polymorphism in the form of overloading, this functionality would require differently named methods, one for each type to be supported: `absInt`, `absDouble`, `absFloat`, and so on.

Overloads can vary not only on the type of parameter, but also on the number of parameters. In the .NET Framework the `Int32` class (for storing 32-bit integers) has a `ToString` method that converts an integer into its string representation. This method has four overloads with different numbers and types of parameters. The most elaborate is

```
public string ToString (string format, IFormatProvider provider)
```

which executes the conversion using the specified numeric format string (for controlling whether the integer is converted into decimal or hexidecimal format, with or without leading zeroes, etc.) and culture information. The simplest is `public string ToString ()`, which executes the conversion using the default formatting and the currently active culture.

Type coercion is sometimes considered another form of ad hoc polymorphism. Most languages support some implicit type conversion, and in some languages, such as C++ and C#, class developers can define new coercions. Then a function that takes an argument of type T can accept arguments of any type that can be implicitly converted to T. This is the weakest sort

of polymorphism, as arguments are actually converted to the target type before the function is executed on them.

Parametric polymorphism This is a much more powerful form of polymorphism in which generic code is written to accommodate a range of types. C++ templates provide a good example. C++ programmers can write code like

```
template <class anyType>
anyType GetMax (anyType a, anyType b) {
   return (a>b ? a : b);
}
```

This function takes two arguments of any type on which the `>` operator is defined—including primitive types and classes that have overloaded `>`—and returns whichever is greater. Generic types, which are similar to C++ templates, were added to Java in 2004 [43, JSR 14] and the .NET Common Type System in 2005 [24].

Parametric polymorphism is in no way specific to the object-oriented paradigm (nor is ad hoc polymorphism). Haskell, Ada, and many other non–object-oriented languages support parametric polymorphism, and in fact ML, a functional language, was the language that introduced the feature.

Subtype polymorphism This sort of polymorphism is specific to languages that support subtypes, most prominently object-oriented languages, which can support subtype polymorphism through inheritance. It is related to type coercion and refers to the fact that in many languages, functions that take a parameter of type T will also accept an instance of any subtype of T. This means that functions that can work with type T are guaranteed to support subtypes of T as well.

There is more to subtype polymorphism than this, though. *Method overriding* makes subtype polymorphism a powerful tool. In languages that support method overriding, subtypes can provide their own implementations for interfaces in their base types. In the following C++ example, an abstract base class provides for a function `Area` that is implemented in derived classes.

```
const double PI = 3.14159;

class Shape {
   public:
      virtual double area (void) = 0;
}

class Circle : Shape {
   private:
```

```
        double radius;
    public:
        // Radius accessors go here.
        double area (void) {
            return PI * radius * radius;
        }
}

class Rectangle : Shape {
    private:
        double length, width;
    public:
        // Accessors go here.
        double area (void) {
            return length * width;
        }
}
```

Now we can build a function that accepts a `Shape` parameter. If this function calls the `area` function of the argument, the overrides will be called when an instance of a derived type is passed to the function. The function will thus be able to obtain the correct area regardless of whether the shape happens to be a circle or a rectangle. Of the three types of polymorphism, subtype polymorphism is what is most often meant in the context of OOP, although many object-oriented languages have the other types of polymorphism too.

1.1.4 Competing definitions of OOP

Many competing definitions of *object-oriented* exist. Kristen Nygaard, the co-creator of Simula 67 and hence one of the inventors of OOP, defines OOP as programming in which "a program execution is regarded as a physical model, simulating the behavior of either a real or imaginary part of the world" [21, p. 16]. Alan Kay, the inventor of Smalltalk and coiner of the phrase *object-oriented* defines OOP much more narrowly:

> OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things. It can be done in Smalltalk and in LISP. There are possibly other systems in which this is possible, but I'm not aware of them. [18]

Armstrong [4] surveys the literature and identifies a number of "quarks" that frequently appear in definitions of OOP. The most popular were *inheritance*, *object*, *class*, *encapsulation*, *method*, *message passing*, *polymorphism*, and *abstraction*.

Rees [38] provides an "a la carte menu of features" that appear as components of definitions of OOP, observing that different definitions of OOP appear to choose subsets of the list almost ad hoc. Here I reproduce the list and elaborate

on each item, because some of them lead to further discussion of object-oriented features.

Encapsulation Rees defines encapsulation as "the ability to syntactically hide the implementation of a type." Many languages enforce encapsulation in some way, including C++ and Java, but some, such as Python, do not.

Protection Rees defines protection as "the inability of the client of a type to detect its implementation." This is closely related to what we are referring to with the term *encapsulation*. Again, C++ and its descendants support protection, but some languages, including Simula 67, do not.

Ad hoc polymorphism Most object-oriented languages do support ad hoc polymorphism, but then so do many non–object-oriented languages in, for example, functional and pure imperative paradigms.

Parametric polymorphism The same is true of this.

Everything is an object. Some theorists, possibly including Alan Kay, believe that for a language to be considered object-oriented, *everything* must be an object. This is true of Smalltalk (and arguably C# and Visual Basic .NET). It is not true of C++ and Java, which have primitive data types such as `int` that are not objects.

An even stronger restriction is that all objects fit into a singly rooted inheritance tree. That is, there must be an `Object` superclass from which *all* entities ultimately inherit. Such a superclass exists in Smalltalk and in the .NET Common Type System, but in C++ there is no superclass from which all objects inherit. (In Java there is an `Object` superclass, but as mentioned, primitive data types do not inherit from it.)

All you can do is send a message. There is no direct manipulation of objects, only communication with or invocation of them. Very few languages implement this actor model, although E comes close. Java and C#, for example, fail because they allow for direct manipulation of public fields.

Inheritance with subtype polymorphism Essentially all OOP languages support inheritance with some form of subtype polymorphism, all the way back to Simula 67, so something like this is part of most OOP definitions.

Implementation inheritance This means that subtypes inherit implementation from their base types with no need to copy and paste code. Most modern object-oriented languages have this feature, but some, such as E, do not.

Sum-of-product-of-function pattern Objects are restricted to be effectively functions that take as their first argument the name of the method to invoke from a finite list. For example, consider a Java class `C` with methods `M1`, `M2`, and `M3`. Then we can think of `C` as a function that takes a method name as its first argument. In this conception, we can mandate that

the first argument is in the list {`M1`, `M2`, `M3`}. In some languages, such as JavaScript, clients are allowed to send messages to objects at will and see what happens.

The point in considering all these definitions and elements of definitions is that there is no universally accepted definition of *object-oriented*. We have lists of features, but it seems that for any definition we can come up with, someone will point to a language that ought to be object-oriented but isn't, or isn't but ought to be. It seems that we may be relegated to the same position as Justice Potter Stewart, confronted with the problem of defining obscenity, who famously said, "I shall not today attempt further to define the kinds of material I understand to be embraced within that shorthand description; and perhaps I could never succeed in intelligibly doing so. But I know it when I see it" [41].

Fortunately, a perfect definition of OOP is beyond the scope of this paper. The purpose of this section is to provide readers with an intuition for OOP and an appreciation for the complexities of the issue of object orientation and the diversity of object-oriented languages. In fact, the tools that we will describe can be applied to a wide variety of languages, some of which are certainly object-oriented and some of which many might claim are not. It is not important for us to establish a dichotomy between object-oriented and not. Instead, let us describe the tools that fit naturally under the umbrella of ORM and see how they apply in a variety of contexts.

To conclude, OOP offers a number of benefits. It reduces the need for duplicate code, as functionality common to many business objects can be stored in a single place (the class definition), and indeed functionality common to many classes can be encoded at the level of base classes. This also increases the reusability of code. In addition, well-structured, object-oriented code is relatively readable and understandable, as the abstraction inherent to OOP invites engineers to take a high-level view of the architecture of the software, examining the code at the highest level of abstraction appropriate to the task at hand. Finally, many software architects feel that the object-oriented paradigm has value as an intuitive framework, as objects and classes in an object-oriented system may accord well with real-world objects and categories of objects in the problem domain.

OOP is quite pervasive. Today, the most important object-oriented languages in a classical mold, at least in industry, are languages such as C++, Java, C#, and Visual Basic .NET, but Smalltalk, Python, Eiffel, and innumerable others have their devotees. OOP has been so successful that object-oriented features have been incorporated into a wide range of languages, even including scripting languages such as JavaScript and server-side Web languages such as PHP and ASP (now ASP.NET). Given the ubiquity of OOP, it seems inevitable that this paradigm will be with us for many years, and probably decades, to come.

1.2 Relational databases

Just as OOP is the dominant paradigm in modern application programming, so is the relational model the dominant paradigm for queryable information storage. Like OOP, the relational model has many competitors, but none have deposed it from its throne (though they have clearly and deeply influenced it). This section provides historical and conceptual background on relational database management systems (RDBMSs).

1.2.1 The relational model

The relational model for database management was born in 1969 in the guise of a now-famous paper [11] by the English computer scientist Edgar F. Codd. In the relational model, all data are represented as mathematical relations. In mathematics, a relation is a subset of the Cartesian product of several sets. In other words, given a collection of sets $S_1, S_2, \ldots, S_n$, a relation is a set of n-tuples of the form $\langle s_1, s_2, \ldots, s_n \rangle$, where $s_i \in S_i$ for $i = 1, 2, \ldots, n$.

In Codd's model, a database relation comprises two parts: a heading and a body. The body is a collection of n-tuples (i.e., a mathematical relation). The heading is a set of n attributes describing the sort of data that the relation can accommodate. Specifically, an *attribute* is an *attribute name* together with a *type name*.

This is quite abstract and seems far removed from our casual understanding of a database. An example will clarify the model. Consider a relation of customers who shop at a particular store. We want to record each customer's name, street address, postal code, country, and telephone number. In this case, our attributes might be

	Attribute name	Type name
1.	Name	String
2.	Street address	String
3.	Postal code	String
4.	Country	Country
5.	Telephone number	String

In this example, we declare that customers' names, street addresses, postal codes, and telephone numbers may be arbitrary strings, but customers' countries must belong to the `Country` data domain. (This domain would presumably include elements such as China and India and exclude other entities, such as Karachi, which is not a country.) The body of the relation would then be a set of pentuples such as $\langle$Mary Smith, 123 Main St., 12345, USA, +1 555-555-0123$\rangle$. For convenience, we usually think of relations in a tabular format, as in the following example.

Name	Street address	Postal code	Country	Telephone number
Mary Smith	123 Main St.	12345	USA	+1 555-555-0123
Budi Dianputra	45 Pejompongan	67890	Indonesia	+62 0123-456-789
Fernanda Silva	Av. Ibirapuera 1234	02468-000	Brazil	+55 21-0123-4567
Kamal Das	12 Nayapaltan	9630	Bangladesh	+880 2-9876543

Often one attribute of a relation (or a combination of attributes) contains values that uniquely identify each element of the relation. For example, a customer relation might have a `CustomerNumber` attribute such that no two customers share the same customer number. Such an attribute (or collection of attributes) is called a *superkey*. A minimal superkey (i.e., a superkey K such that no proper subset of K is also a superkey) is a *candidate key*. Often, a single candidate key is identified as the *primary key* of a relation.

Often an attribute of one relation cross-references another relation. An attribute (or combination of attributes) A is a *foreign key* of a relation R if A is a primary key of some relation S. (Usually, but not always, $R \neq S$.) Foreign keys allow for relationships between relations. In our example, we could have a relation `Country` as follows:

Country code	Name	Capital city
050	Bangladesh	Dhaka
076	Brazil	Brasília
360	Indonesia	Jakarta
566	Nigeria	Abuja
643	Russia	Moscow
840	USA	Washington
⋮	⋮	⋮

(We mark the primary key by printing it boldface.) Now, our `Customer` relation can have `CountryCode` as a foreign key:

Name	Street address	Postal code	Country code	Telephone number
Mary Smith	123 Main St.	12345	840	+1 555-555-0123
Budi Dianputra	45 Pejompongan	67890	360	+62 0123-456-789
Fernanda Silva	Av. Ibirapuera 1234	02468-000	076	+55 21-0123-4567
Kamal Das	12 Nayapaltan	9630	050	+880 2-9876543

This may seem like a cosmetic change, but in fact it empowers us to ask more sophisticated questions of the database, such as: Which customers live in the country whose capital is Moscow?

The relational model itself is rather rigid, and few so-called RDBMSs follow Codd's relational model exactly. In fact, Codd bemoaned this trend in a 1985 article [10], publishing a list of twelve rules to which a DBMS ought to conform in order to be considered truly relational:

1. All information is represented explicitly and in exactly one way: by values in tables. This includes not only application data, but even the names of the tables and columns themselves.

2. Every datum in a relational database is accessible via a combination of table name, column name, and primary key value.
3. Null values are fully supported for representing missing information and are distinct from empty values such as zero-length strings and the value 0.
4. The database description is logically represented in the same way as ordinary data and can be accessed via the ordinary query language.
5. There is a language that comprehensively supports data definition, view definition, data manipulation, integrity constraints, authorization, and transaction boundaries.
6. All theoretically updatable views are also updatable by the system.
7. Insert, update, and delete operations are supported for any retrievable set rather than just for single rows.
8. The logical behavior of the database is isolated from the underlying physical architecture.
9. Database users are isolated from changes to the logical structure of the database.
10. Integrity constraints can be defined through the query language and are stored in the catalog.
11. Database users are isolated from the introduction of data distribution or the redistribution of data.
12. If there is a way to manipulate data other than via the query language, it cannot be used to bypass the integrity rules and constraints expressed in the query language.

Almost no modern "RDBMSs" conform to all these rules, and certainly no widely used DBMSs conform to them all. Rules 9 through 12 are especially poorly supported. Most DBMSs exhibit strong ties between the user's view of the data and the logical structure of the tables. Many DBMSs do not provide complete support for integrity constraints. The nature of distributed databases makes distribution independence difficult. Finally, many DBMSs allow low-level data manipulation that bypasses integrity rules and constraints.

Besides these twelve rules, there are many other departures that modern DBMSs make from the relational model. For example, the model treats a relation as a *set* of tuples, meaning that the records in a relation have no intrinsic order and, in particular, that it is impossible to represent duplicate records. Most modern DBMSs accommodate duplicate rows in their tables, and many systems order result sets predictably even when not asked to sort the results (e.g., returning table records in the order in which they were inserted).

Some theorists dismiss modern, commercial DBMSs that fail to realize the relational model completely as merely "pseudo-relational" DBMSs. However,

most practitioners use the term *RDBMS* to describe all DBMSs based on the relational model, including systems such as Oracle, Microsoft SQL Server, and MySQL. Henceforth, we follow the latter convention.

1.2.2 The entity-relationship model

Although at a certain level we must conceive of a relational database in terms of its relations and their attributes and keys, database designers often find it convenient to operate at a higher level of abstraction. In the entity-relationship model [9], which was introduced at the 1975 International Conference on Very Large Data Bases, individual tuples in a relation (or records in a table) are viewed as *entities*—"things" that can be distinctly identified (akin to objects in OOP). Entities are grouped into *entity sets* such as `Employee` and `Country`. Entity sets are analogous to relations in the relational model (or classes in the object-oriented paradigm). Entities can be associated with each other via *relationships*, such as a parent-child relationship between two person entities. RDBMS, of course, realize these relationships via foreign keys, but this detail is abstracted away at the entity-relationship level. Formally, a relationship is a tuple of entities $\langle e_1, e_2, \ldots, e_n \rangle$. Relationships are grouped into *relationship sets*, which are mathematical relations among n entities. The *role* of an entity in a relationship is the function it performs, such as *child* in a parent-child relationship.

It is important not to confuse the concepts *relation*, *relationship*, and *relationship set*. A relation is a creature of the relational model, representable as a table and analogous to an entity set (in the entity-relationship model) or a class (in the object-oriented paradigm). A relationship is an association between entities (i.e., an association between relations). A relationship set is a mathematical relation (just like a relation in the relational model) that describes the associations that exist between two entity sets.

In the entity-relationship model, entities have *values* such as "Mary Smith," "Budi Dianputra," and "123 Main St.," which are grouped into *value sets* that we can label "Name," "Street address," and so on. (Value sets in the entity-relationship model are analogous to data domains in the relational model.) Formally, an *attribute* is a function that maps from an entity set or a relationship set into a value set. For example, a `DateOfBirth` attribute may map from a `Person` entity to the `Date` value set.

The entity-relationship model is often discussed in the context of entity-relationship diagrams, which have been part of the entity-relationship model from the beginning [9, §3]. Entity-relationship diagrams allow database designers to appreciate the structure of a database at a glance. Different conventions exist for drawing entity-relationship diagrams. In the classical notation, boxes represent entity sets, diamonds represent relationship sets, and ovals represent attributes. However, we instead use a crow's-feet notation that we believe more clearly reflects the structure of the relational database, similar to the notation described in depth by Carlis and Maguire [7].

1.2.3 SQL

Theoretically, it is possible to devise many languages that could act as query languages for relational databases. In practice, SQL is the only language worth discussing. SQL is used as the query language in practically all RDBMSs today. SQL, then called SEQUEL for "Structured English Query Language," was born in the 1970s to support System R, an RDBMS developed by an IBM research group. After other implementations of SQL appeared, including the Oracle DBMS, SQL was standardized by the American National Standard Institute in 1986 and the International Organization for Standardization in 1987. Revisions appeared in 1989, 1992, 1999, and 2003, and the SQL standards are often called by names such as SQL-86, SQL-89, SQL-92, and so on. Despite standardization, significant differences between implementations of SQL remain. As a consequence, SQL scripts are generally not portable between DBMSs.

SQL (as originally conceived) is a declarative programming language, not an imperative language like C or BASIC. SQL includes commands for data retrieval, data manipulation, and data definition, as well as administrative commands supporting functionalities such as database transactions, permissions, and database triggers. The `SELECT` command is used for data retrieval and has many syntactic variants that accommodate sophisticated queries. For example, the question on page 18, "Which customers live in the country whose capital is Moscow?", can be expressed

```
SELECT Customer.* FROM Customer
JOIN Country ON Customer.CountryCode = Country.CountryCode
WHERE Country.CapitalCity = 'Moscow'
```

The primary data manipulation commands in SQL are `INSERT`, `UPDATE`, and `DELETE`. `INSERT` adds a record to a table, as in the following example:

```
INSERT INTO Customer
(Name, StreetAddress, PostalCode, Country, TelephoneNumber)
VALUES ('Yelena Ivanova', '868 Pochtovaya', '287242',
    643, '+7 495 375-76-55')
```

`UPDATE` changes values that already exist in the database, as in the following command, which might have been executed when Zaire's name changed to the Democratic Republic of the Congo:

```
UPDATE Country SET Name = 'Democratic Republic of the Congo'
WHERE CountryCode = 180
```

`DELETE`, of course, removes existing records from the database.

Besides these data manipulation commands, SQL also includes data definition commands for establishing the structure of content in the database—creating and revising tables, rows, columns, and indices. Our `Customer` table, for example, would have been created with a command such as

```
CREATE TABLE Customer (
    Name nvarchar(50) NOT NULL,
```

```
    StreetAddress nvarchar(100),
    PostalCode varchar(15),
    Country smallint,
    TelephoneNumber varchar(25)
)
```

Here `nvarchar`, `varchar`, and `smallint` are data types (Unicode string, non-Unicode string, and integer), and `NOT NULL` indicates that no customer record may have a null name. There are also `DROP TABLE` and `ALTER TABLE` commands to delete and modify tables.

A third category of SQL commands is for controlling access to data. These allow database administrators to `GRANT`, `DENY`, `REVOKE`, and `UPDATE` database users' access to specific database objects.

1.2.4 Object-relational and other advanced features of modern DBMSs

Data manipulation, data definition, and data control remain central to SQL, but SQL has come a long way since its 1986 standardization. Today's SQL implementations include features that make it a powerful and versatile tool for a variety of data processing problems. In particular, SQL has gained object-like features that bring it closer to the object-oriented paradigm of programming. These features have brought the major RDBMSs to the verge of being object-relational DBMSs, combining the best features of the relational model and the object-oriented paradigm. SQL:1999, in particular, introduced an impressive array of object-like features [22], some of which are not yet well supported [14, 16]. Specific DBMSs have also introduced many new features to their own implementations ([26], [29, §1.6], [32], [36], [37, App. E]).

One important change has been the addition of traditional programming language features such as variable assignment and flow control structures such as conditionals and loops. Transact-SQL (the proprietary SQL extension used in Microsoft and Sybase products) includes `IF-ELSE` conditionals; `WHILE` loops; and `GOTO`, `BREAK`, and `CONTINUE` statements [25]. PL/SQL (Oracle's SQL extension) similarly includes `IF-THEN-ELSE` conditionals, `CASE` statements, `WHILE` and `FOR` loops, and the `GOTO` statement [32, §4]. These extensions, along with many other implementations, also include support for variable declarations and assignment and other features of programming languages. Such features allow users of these SQL implementations to use the language in an imperative way, as the following Transact-SQL snippet illustrates.

```
/* Check to make sure that there is at least a $20 difference between
   our least and most expensive items. Customers hate it when they
   don't have a large enough range of prices from which to choose. */
DECLARE @minPrice money, @maxPrice money
SELECT @minPrice = MIN(Price) FROM Item
SELECT @maxPrice = MAX(Price) FROM Item
IF @minPrice IS NULL
```

```
BEGIN
   /* If the table is empty, populate it. */
   INSERT INTO Item (Name, Price) VALUES ('A cheap item', 5)
   INSERT INTO Item (Name, Price) VALUES ('A pricier item', 30)
END
ELSE IF @minPrice < @maxPrice − 20
BEGIN
   /* If there is insufficient price variance,
      insert a new item to satisfy the postcondition. */
   INSERT INTO Item (Name, Price)
   VALUES ('Really expensive item', @minPrice + 30)
END
/* Postcondition: The table contains a pair of items
   with a price difference over $20. */
```

Another feature of modern SQL implementations is *cursors*, control structures that support traversal and row-at-a-time (instead of set-based) processing of records. This allows for more sophisticated processing than can be accomplished via the traditional declarative data manipulation commands, empowering database users to apply control structures such as cnoditionals and loops to attain finer control over data processing tasks.

SQL:1999 provides for *stored procedures*, self-contained modules of SQL code that can accept parameters and accomplish data-driven tasks, as well as *user-defined functions*, which accept parameters and return a single value. These features, akin to subroutines or functions in procedural languages, are well supported in modern implementations of SQL and widely used. SQL:1999 also includes *triggers*, stored procedures that fire automatically when a specific data modification occurs. These constructs, like all content in a relational database, can be created, dropped, and modified through SQL itself, with commands such as `CREATE PROCEDURE` and `DROP TRIGGER`.

SQL:1999 also includes *user-defined types*, which at their most sophisticated look like classes in object-oriented languages.

```
CREATE TYPE Address AS (
   StreetAddress nvarchar(100),
   City nvarchar(50),
   State nvarchar(20),
   PostalCode varchar(15),
   Country smallint
) NOT FINAL
```

Attributes of structured types can be accessed easily via a dot notation reminiscent of object-oriented languages such as C++. If we have a table `Customer` with a column `HomeAddress` of type `Address`, we can write a query like

```
SELECT DISTINCT C.HomeAddress.State FROM Customer AS C
```

to get a list of customers' home states.

In fact, SQL:1999 allows database designers to provide not only the structure, but also the behavior of user-defined types, via routines such as methods. Suppose we want a method that returns a formatted address of the form

> 123 Main St.
> Schenectady, NY 12345

Then we could add a method using syntax similar to the following.

```
CREATE INSTANCE METHOD formatted_address ()
    RETURNS nvarchar(200)
    FOR Address
RETURN SELF.StreetAddress + CHR(13) + CHR(10) +
    SELF.City + ', ' + SELF.State + ' ' + SELF.PostalCode
```

Even more impressively, the SQL:1999 type system allows for inheritance. To create a subtype `Customer` of an existing type `Person`, we would use syntax such as

```
CREATE TYPE Customer UNDER Person AS (
  -- Customer-specific attributes go here.
  )
  INSTANTIABLE
  NOT FINAL
```

SQL:1999 also supports polymorphism. Method definitions can overload other method definitions in the same class (with different parameters) and override method definitions in supertypes.

Unfortunately for us, none of these object-oriented features are well supported at present, but they do reflect a shift in the object-oriented direction for SQL. In general, then, while SQL is retaining its roots as a declarative, set-based language, it is being transformed by the addition of imperative and object-oriented capabilities.

1.3 The use of relational databases in object-oriented programs

Not only are object-oriented software and RDBMSs exceptionally common, but they are especially commonly used in conjunction with one another. Many applications need to retain the content of data structures over long periods of time—and, in particular, between program executions. The capacity to do so is called *persistence*. Without persistence, data exists only in memory and is lost when the computer shuts down.

To achieve persistence, an application must store the data to non-volatile storage such as a hard drive. For example, a software application that runs on consumer computers (a game, for example) might persist data to the file system, saving and loading data files to the hard drive in order to maintain state (e.g.,

user settings and saved games) between program executions. Many applications, though, especially applications that run on servers or other powerful machines (e.g., Web applications), use relational databases for persistence of data.

Typically, developers map classes to database tables in a rather straightforward way, creating one table for each class that needs to be persistent. For example, given a `Customer` class such as the following Java example

```
public class Customer {
    private int customerNumber;
    private String name;
    private String streetAddress;
    private String postalCode;
    private short countryCode;
    private String telephoneNumber;

    /* Accessors and other methods go here. These are generally not
       part of the class-to-table mapping process. Remember, the
       database is for storing the data of the class, not its behavior. */
}
```

a developer might create a table in the application database using code similar to the following. (This example is in MySQL.)

```
CREATE TABLE Customer (
    CustomerNumber int NOT NULL PRIMARY KEY,
    Name nvarchar(50) NOT NULL,
    StreetAddress nvarchar(100),
    PostalCode varchar(15),
    Country smallint,
    TelephoneNumber varchar(25)
)
```

Typically, the developer adds `Save` and `Load` methods to the `Customer` class that store and load the data in the database. It is worth spending a few minutes examining how this is accomplished.

1.3.1 Database drivers and APIs

In the simplest terms, modern, object-oriented software applications connect to databases via application programming interfaces (APIs), which provide programmatic access to database drivers, the software components that actually connect to and communicate with the database. In fact, it can be more complicated than this, with several layers of APIs, drivers, and bridges, but from the application developer's point of view, all database access is performed through an API, which is often built into the language or readily available through standard class libraries. (In principle, an application could bypass the API and communicate directly with the database driver, or even connect to the database directly by implementing a communication protocol that the database supports, but in

modern times this is never done, as it would be an unthinkable waste of time to reinvent these facilities.)

Modern object-oriented languages' standard libraries, such as the Java Platform, Standard Edition (Java SE), and the .NET Base Class Library, provide ready support for a variety of database platforms. Since these libraries provide excellent examples of database APIs, we will be using them exclusively for the rest of this section, but many other languages also provide facilities for database access. In Java, applications normally access databases via the Java Database Connectivity (JDBC) API. .NET applications use ADO.NET. A discussion of each of these technologies will illuminate the way in which object-oriented software connects to databases and reveal differences that exist between different database APIs.

JDBC has been part of Java SE since version 1.1 of the Java Development Kit was released in 1997 [42]. In the basic JDBC architecture, a Java application sends instructions to the JDBC API, which communicates with the database server via a DBMS-specific, proprietary protocol using a JDBC driver. In other situations, the communication may go through additional levels, such as through vendor-specific middleware. In this case, the JDBC driver translates JDBC calls into the protocol of the middleware, which in turn uses a proprietary protocol to communicate directly with the DBMS. Another typical architecture involves a JDBC-ODBC bridge, which converts JDBC calls into ODBC calls. Open Database Connectivity (ODBC) is a language-independent API that can connect to many DBMSs (and even other data sources such as XML files and spreadsheets). Much of this complexity is hidden from the application, which only needs to call the JDBC API.

So how does this look from the application's perspective? In a typical JDBC interaction with a database, the application loads the correct database driver, connects to the database, creates a `Statement` object, executes a query that returns a result set, processes the result set, and cleans up. The following Java snippet provides a `Load` method for the `Customer` class definition given above.

```
// This function loads a customer. It takes the unique customer number
// as input and returns a Customer object as output.
public static Customer Load(int customerNumber) {
    // Load the JDBC database driver for MySQL.
    Class.forName("com.mysql.jdbc.Driver");

    // Connect to a remote MySQL database.
    Connection conn = DriverManager.getConnection(
        "jdbc:mysql://example.com/customerdb",
        "databaseUserName", "databasePassword");

    // Create a Statement object to execute SQL code.
    Statement stmt = conn.createStatement();

    // Execute a query and get the results.
```

```
        ResultSet rs = stmt.executeQuery(
            "SELECT * FROM Customer WHERE CustomerNumber = ?");

        Customer c; // This will be the return value.
        if (rs.next()) { // If there is a result,
            // populate a new Customer object with the data from the database.
            c = new Customer(customerNumber);
            c.name = rs.getString(1);
            c.streetAddress = rs.getString(2);
            c.postalCode = rs.getString(3);
            c.countryCode = rs.getShort(4);
            c.telephoneNumber = rs.getString(5);
        }
        else // If no customer with specified identifier exists,
            c = null; // this function will return null.

        rs.close(); stmt.close(); conn.close(); // Clean up.
        return c;
    }
```

(This snippet is simplified. A live example would need to handle connection errors and SQL exceptions as well as null values in the database.) This code is rather dull to write, and even more so once exception handling and null value handling are added. The structure of code like this varies little from class to class and even from application to application. As you can imagine, in a class with many attributes, it can become tedious to write. As we will see, this is one of the problems that object/relational mapping addresses.

Roughly speaking, the .NET equivalent to JDBC is ADO.NET, which is a reinvention of an earlier product called ActiveX Data Objects (ADO), a set of Component Object Model objects that could be used for accessing (via OLE DB, Object Linking and Embedding Database) data stores including relational databases from languages such as VBScript and Visual Basic, as well as (less often) Delphi, C++, and other languages. ADO.NET differs substantially from its predecessor. Perhaps most conspicuously, rather than being a standalone component, it is integrated into the Base Class Library of the .NET Framework, meaning that it is immediately accessible from any of the .NET languages, such as C# and Visual Basic .NET. At present, ADO.NET includes four data providers: a proprietary-protocol data provider for Microsoft SQL Server, an Oracle data provider, an ODBC data provider, and an OLE DB data provider. (The ODBC and OLE DB data providers allow .NET to use a variety of data sources including many relational databases.) All four data providers have `Connection`, `Command`, `Parameter`, `DataReader`, and `DataAdapter` classes to support database operations.

Despite some architectural differences, from a developer's point of view ADO.NET bears many similarities to its Java counterpart, JDBC. The following snippet is the C# equivalent of the Java code above.

```
public static Customer Load(int customerNumber) {
    // Connect to a Microsoft SQL Server database.
    SqlConnection conn = new SqlConnection(
        "server=localhost;uid=sa;pwd=;database=example");
    using (conn) {
        conn.Open();

        // Create a SqlCommand object to execute SQL code.
        SqlCommand cmd = new SqlCommand(
            "SELECT * FROM Customer WHERE CustomerNumber = ?",
            conn);

        // Execute the query and get the results.
        SqlDataReader rdr = cmd.ExecuteReader();

        if (rdr.Read()) { // If there is a result,
            // return a new Customer object with the data from the database.
            Customer c = new Customer(customerNumber);
            c.name = rdr.getString(1);
            c.streetAddress = rdr.getString(2);
            c.postalCode = rdr.getString(3);
            c.countryCode = rdr.getInt16(4);
            c.telephoneNumber = rdr.getString(5);
            return c;
        }
        else // No customer with specified identifier exists.
            return null;
    }
}
```

As you can see, some of the names and protocols are different, but the idea is basically the same.

1.4 The impedance mismatch

The so-called *object-relational impedance mismatch* is simultaneously a motivation and an obstacle for object-relational mapping. The term *impedance mismatch* is borrowed from electrical engineering, where impedance measures opposition to the flow of an alternating current in a circuit. The most efficient exchanges between electrical systems happen when their impedances are closely matched. Impedance mismatch is a problem that occurs when two circuits with different impedances are connected, which can ultimately result in attenuation and noise. The problem can be correct through impedance mismatching.

The analogy is that object-oriented systems often seem to be mismatched to relational database systems. To clarify the metaphor, impedance mismatch in

both electrical engineering and software engineering occurs when one system is unable to interact with another system efficiently.

Object-relational impedance mismatch occurs because the object-oriented and relational database paradigms have different conceptions of data. The object-oriented paradigm views data primarily in the context of actions performed on it. That is, objects are important not just because of the data they contain, but because of their ability to perform tasks on the data and exchange information with other objects. The relational paradigm is data-focused. It places importance on the data itself and its structural (not behavioral) relationship with other data.

Less abstractly, the impedance mismatch manifest itself in several specific ways:

1.4.1 Inheritance and polymorphism

The notion of inheritance or generalization is central to OOP. Indeed, in many of the most clearly object-oriented languages and type systems, such as Smalltalk, Java, and the .NET Common Type System, all objects ultimately derive from an `Object` superclass, meaning that all objects in the entire system exist together in an enormous, intricate inheritance hierarchy. In practice, things are not quite so bad as this might sound, as many business objects inherit directly from `Object`, and many inheritance hierarchies of business objects are quite shallow, but inheritance is nonetheless an integral concept in an object-oriented system. By contrast, inheritance is not intrinsic to the classical relational model, which instead concerns itself with structural relationships between data.

How would we map an inheritance hierachy such as the one in Figure 1.1 to a database? Our default inclination to map each class to its own table, resulting in a `Fruit` table, an `Apple` table, an `Orange` table, a `RedDeliciousApple` table, and so on. But are there better ways of mapping such an inheritance hierarchy? As it turns out, there are several options, which we will discuss in depth in Section 2.3.

Even if we achieve an inheritance mapping, complications remain. The inheritance mechanisms in object-oriented languages often provide important features, the most substantial of which is subtype polymorphism. To flesh out our fruit example, suppose the software we are building is an application for simulating the dieting patterns of lemurs, so we have a `Lemur` class with data members for modeling its nutrition (caloric intake, dietary protein consumption, vitamin deficiencies, and so on). It has an `Eat` method which takes a `Fruit` argument. Subtype polymorphism allows it to update its nutritional information in a manner appropriate to the specific type of fruit it is eating—namely by using members of the `Fruit` class that its dervied types override.

Relational databases have no straightforward way to represent this polymorphism. This can be problematic. Suppose, for example, that for each lemur we want to persist the fruits it has eaten. If we were only dealing with a `Lemur` table and a `Fruit` table, this would mean that there should be a foreign key in the `Fruit` table that refers to the primary key of the `Lemur` table, creating

a one-to-many relationship. But how does this work if there are several tables that we are thinking of as somehow subtypes of `Fruit`? The answer depends on what sort of inheritance mapping we use, but it is a complex issue in need of resolution.

1.4.2 Associations

Aside from subtyping, classes in object-oriented systems relate to each other in two important ways. First, they can pass messages to each other. Most simply, this can mean that they call each others' methods, but it can also mean that they catch each others' events. This sort of relationship is behavioral and hence normally immaterial to object-relational mapping. The other important way classes relate to each other is via membership associations.

Consider a software application for modeling employees and their professional certifications. Suppose that each employee can have a certification, but multiple employees have the same certification. There are several ways we could model this relationship in an object-oriented language. First, we could place a reference to the `Certification` class in the `Employee` class, as follows:

```
public class Employee {
    private string name;
    private string payRate;
    private Certification certification;
}
```

Alternatively, the `Certification` class could reference a collection of `Employee` objects, recording the employees who have each certification.

```
public class Certification {
    private string name;
    private string grantingAuthority;
    IList<Employee> employees;
}
```

(These examples are in C#. As a matter of syntax, it is worth noting that these references are indeed *references* to objects in memory, not actual copies of those objects. In C++, these examples would be written using pointers—`Certification *certification` and so on.)

If the association is to be traversable in both directions (i.e., if the application needs to be able to determine both the certification that an employee has and the employees who have a certification), then both of these associations must be implemented simultaneously (and application logic must ensure that the two directions preserve the same data). The correct choice in this situation—a `Certification` reference in `Employee`, a list of `Employee` references in `Certification`, or both—is a question of software architecture. The important point to note is that all three of these cases ought to be mapped to the same database structure, namely an `Employee` table and a `Certification` table with a

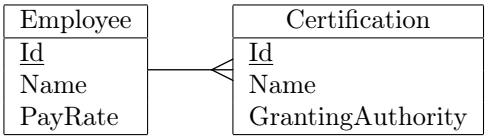

Figure 1.3: A one-to-many association in a relational database

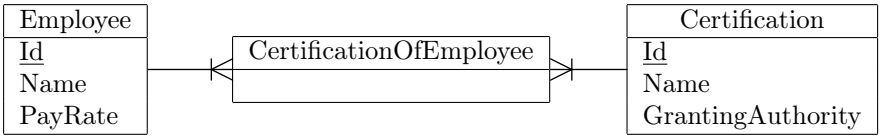

Figure 1.4: Representing a many-to-many relationship in a relational database requires a utility table.

many-to-one relationship between them (i.e., a foreign key in the `Employee` table associated with the primary key of the `Certification` table), as in Figure 1.3.

What if an employee can have multiple certifications, while at the same time multiple employees can have the same certification (which seems to be the most likely relationship between the two entities)? The object-oriented code for this case would result in an `Employee` class with references to multiple Certifications, a `Certification` class with references to multiple employees, or both. However, the relational model does not naturally accommodate many-to-many relationships (although they can be drawn in an entity-relationship diagram). Establishing a many-to-many relationship in a relational database requires the creation of a utility table that represents the relationship itself. In this case, the utility table would be called something like `CertificationOfEmployee` and would be related to `Employee` and `Certification` in the way depicted in Figure 1.4.

1.4.3 Data types

A more mundane sort of object-relational impedance mismatch is caused by data types. Different RDBMSs and different object-oriented languages all have their own peculiar data types, and it is not always completely clear how to map between them. Certainly, a 32-bit integer type (`int` in Java and C#) should map to a 32-bit integer type (`int` in most SQL implementations). But what about less commonplace types? Many languages and RDBMSs have date/time types and currency types, but they vary in range and precision. .NET has a `DateTime` type that represents times from 1 C.E. through 9999 C.E. with accuracy to a tenth of a microsecond. Few DBMSs can match this precision with their date types. The `datetime` type in Transact-SQL, the most precise date/time type available in that language and the one with the greatest range, supports dates from 1753 through 9999 with a precision of 1/300 of a second. This is ample for

storing purchase records or appointment times, but for applications that rely on the greater range and precision of the `DateTime` type in the .NET Base Class Library (historical applications that use the type to record events in the Middle Ages, or physiological event simulators that require sub-millisecond precision) mapping the .NET `DateTime` type to the Transact-SQL `datetime` type would be insufficient.

Other languages have primitive types that do not map directly to standard SQL types. In Python, complex numbers are primitives. In Lisp, linked lists are primitives. And how are arrays in Java or C++ to be mapped? With a little thought, these situations can be resolved on a case-by-case basis, but is there a general solution that we can apply systematically to such instances?

An even more mundane issue in this category is character encoding. If the application uses a Windows-1251 encoding for storing Cyrillic characters and the database assumes an ISO 8859-1 encoding, there will be problems. If the application uses UTF-32 and the database supports only ASCII characters, there will be more severe problems. Nationalization issues can crop up with other data types as well (year-month-day, day-month-year, and month-day-year date formats, for example). Issues such as these are not too hard to deal with (as long as both the object-oriented language and the DBMS are modern, both should support Unicode character encodings and culture-invariant international formatting standards anyway), but they remind us that there are not only architectural differences, but also a plethora of banal issues that need to be resolved for an object-oriented application to communicate with a database.

1.4.4 Granularity

We have been implying so far that every class will map to its own database table. In fact, this is not always practical. For example, suppose we have a pair of classes like this:

```
public class Customer {
    private int customerNumber;
    private String name;
    private Address address;
    private String telephoneNumber;

    /* Accessors and other methods would be here. */
}

public class Address {
    private String streetAddress;
    private String city;
    private String state;
    private String postalCode;
    private short countryCode;
```

```
    /* Accessors and other methods would be here. */
}
```

In this example, instead of including the address fields in the `Customer` class itself, we encapsulate them in a new `Address` class, which may be used many times throughout the application. If we were to map each class to a table, we would get a database like that depicted in Figure 1.5. This is workable, but unless individual addresses are likely to be reused many times in many places in the application, storing the addresses in their own, separate table just introduces overhead. (There is a performance penalty for every query in which a join between two tables must be carried out, such as any data selection statement that makes use of a foreign key.) Database designers more typically include all this information in a single table (Fig. 1.6), even if this does not mirror the structure of the classes in the object-oriented language, not only for the performance benefit, but also because it makes better intuitive sense for all a customer's address fields to be stored in one place. (If our DBMS supports user-defined types, we might also considering making `Address` a user-defined type in the database, so that the `Customer` table can have an `Address` column of type `Address` to store address values conveniently, compactly, and efficiently.) So this is one case in which creating a table per class might be a mistake (though not a fatal one).

This is an example of a granularity mismatch. Classes in an object-oriented system may exist at many levels of granularity, from coarse-grained entity classes such as `Customer`, which model important business objects, to finer-grained classes like `Address`. In many languages, basic data types like `DateTime` and `String` and even `Integer` have their own, even more fine-grained classes, but surely these should not be mapped to tables. In order to carry out an object-relational mapping, we must decide which classes deserve tables of their own and which are too fine-grained to merit this treatment. Fine-grained classes should instead be mapped to user-defined types or to collections of columns (or, in the case of sufficiently primitive types like `DateTime` and `String`, to single columns).

1.4.5 Identity

Identity is very improtant to the relational model. Indeed, many relational theorists believe that no table should be without a primary key. Primary keys ensure that all records in the database are unique, even if they are otherwise identical in content to other records. In other words, it gives them a sense of identity.

Objects in object-oriented systems have two instrinsic types of identity, two ways by which objects can be determined to be equal or unequal. These ways are reference comparison and value comparison. Reference comparison checks whether two object references refer to the same object. In languages with pointers, such as C++, this is obvious. To compare whether two pointers `*a` and `*b` are equal in C++, one simply evaluates `a == b` instead of comparing the

Customer			
CustomerNumber	Name	Address	TelephoneNumber
1	Mary Smith	32	+1 555-555-0123
2	Budi Dianputra	33	+62 0123-456-789
3	Fernanda Silva	30	+55 21-0123-4567
4	Kamal Das	38	+880 2-9876543

Address					
Id	StreetAddress	City	State	PostalCode	CountryCode
30	Av. Ibirapuera 1234	Palmas	Tocantins	02468-000	076
32	123 Main St.	Applefield	New York	12345	840
33	45 Pejompongan	Slawi	Central Java	67890	360
38	12 Nayapaltan	Barisāl	Barisāl	9630	050

Figure 1.5: A database designed with too fine a granularity. Figure 1.6 illustrates a more natural design.

Customer							
Customer-Number	Name	StreetAddress	City	State	Postal-Code	Country-Code	TelephoneNumber
1	Mary Smith	123 Main St.	Applefield	New York	12345	840	+1 555-555-0123
2	Budi Dianputra	45 Pejompongan	Slawi	Central Java	67890	360	+62 0123-456-789
3	Fernanda Silva	Av. Ibirapuera 1234	Palmas	Tocantins	02468-000	076	+55 21-0123-4567
4	Kamal Das	12 Nayapaltan	Barisāl	Barisāl	9630	050	+880 2-9876543

Figure 1.6: A database designed with a coarser, probably more natural level of granularity than that in Figure 1.5.

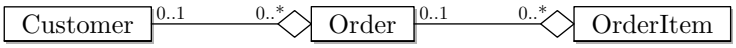

Figure 1.7: An aggregation chain, which could cause a database query explosion

pointer referents `*a` and `*b`. Modern languages such as Java, C#, and Visual Basic .NET have reference-type semantics that obscure the distinction between objects and references to objects. In Java, for example, two objects `a` and `b` are compared using reference equality with the syntax `a == b`. This sort of comparison amounts to comparing whether two objects occupy the same memory location.

The other type of object comparison is value comparison. Value comparison compares objects attribute-by-attribute. In Java, C#, and Visual Basic .NET, two objects `a` and `b` may, under the right circumstances, be compared for value using the syntax `a.Equals(b)` (C# and Visual Basic .NET) or `a.equals(b)` (Java). This sort of comparison indicates whether two objects store the same information.

These three types of identity are all different. Reference comparison differs from database identity comparison in that two objects need not share the same location in memory in order to represent the same database entity. In other words, it is possible to have two copies of the same entity in memory. Value comparison differs from database identity comparison because the point of an identity is that two database entities can be different even if they carry the same content (aside from their identity).

There are a few ways of resolving this mismatch, but usually it is the object-oriented code that is adapted to be in agreement with the relational database. Often, the primary key of a table is made an attribute of the object-oriented class, even if the primary key is a surrogate.

1.4.6 The database query explosion

A final problem with object-relational mapping is that if it is implemented naïvely, navigation of object graphs can result in an exponential explosion in application queries to the database. Suppose that we have an application in which each customer can have multiple orders, and each order can have multiple order items, as depicted in Figure 1.7. Now suppose that in our object-oriented application, we want to list (or otherwise process) all the order items that a certain customer has ever purchased. To accomplish this, we might write a function such as the following:

```
function ProcessOrderItemsOfCustomer(int customerId) {
    Customer c = Customer.Load(customerId);
    foreach (int orderId in c.OrderIds) {
        Order o = Order.Load(orderId);
        foreach (int orderItemId in o.OrderItemIds) {
            OrderItem i = orderItem.Load(orderItemId);
```

```
            Console.WriteLine(i.ToString());
            // Here we could do whatever processing on i that we wished.
        }
    }
}
```

In this example, we are making a number of structural assumptions about the mapping. We assume that the `Customer`, `Order`, and `OrderItem` classes have static `Load` methods that take primary keys as arguments and return class instances. We are assuming that the `Customer` class has an `OrderIds` property that is a collection of the primary keys of the orders that a customer has made. (That is, the `Customer` class only stores the primary keys of the orders associated with, not the entire content of each order.) The `Order` class has an analogous `OrderItemIds` property. These assumptions are reasonable, although the mapper could have made different choices.

The problem is clear. If the customer has made m orders, and each order has n items, then the function will call `Order.Load` m times and `OrderItem.Load` mn times. If each `Load` function establishes its own connection to the database and makes its own `SELECT` query, this function could be an incredible performance sink. Of course, things needn't be so bleak; in fact, we could retrieve the information we want with a single query:

```
SELECT OrderItem.* FROM Customer
    INNER JOIN [Order]
        ON Customer.CustomerId = [Order].CustomerId
    INNER JOIN OrderItem ON [Order].OrderId = OrderItem.OrderId
```

However, architecting a mapping to allow this single-query access to the desired data is not a trivial problem.

In conclusion, changes to SQL and to DBMS features over the years have alleviated the object-relational mismatch to some degree, and increasing support for the object-oriented facilities in recent SQL standards will continue this trend. At the same time, programming conventions have made bridging the gap between the two paradigms easier. For example, objects are given identity values with special behavior that reflect the importance of primary keys in database systems. Nonetheless, the object-relational mismatch continues to be a problem for software architects.

1.5 The need for ORM

In section 1.3.1, we saw how persistence code is written. Typically, an object-oriented application uses a API that in turn calls a driver to communicate commands to the database. This code differs little from class to class and is largely the same even from application to application and is often tedious to write.

Part of the reason this process is tiresome is that it is generally very straightforward. Designing a database to support persistence for a collection of object-

oriented classes, creating the database, writing code to interface with the database (or, more immediately, with the database connection API)—all of these tasks tend to be mechanical. They do not tend to invite creativity. Fortunately, software developers have a strategy for dealing with boring, mechanical, uncreative tasks: engineering software to undertake them. This is one motivation for object-relational mapping.

Saving time is reason enough to wonder about the practicality of object-relational mapping, but it is not the only reason. We have seen that many persistent software applications have parallel class and table structures. If the application has a `Customer` class with `Name` and `Address` member data, the database likely has a `Customer` table with `Name` and `Address` columns. This could reasonably be viewed as duplication of information, which is justifiably viewed by software architects as a bad thing. Perhaps the most obvious problem is that duplication of structure means duplication of work; developers have to create the classes in the application code and then turn around and reproduce the same structures in the database. Slightly more subtly, this duplication of structure increases the likelihood of errors. If a software developer adds a `Title` field to the `Customer` table and forgets to tell the database administrator to change the database, the application will break. Indeed, even if the software developer wants to extend the permissible length of the `Name` attribute from 50 characters to 100 characters, a change in the database must mirror the change that the application developer makes to the validation code in the user interface layer. If not, input over 50 characters will be truncated or cause an exception. (A problem like this may not be caught in testing, unless a quality assurance analyst thinks to test exceptionally long names. Quite likely, it will be an end user who first encounters the problem.) Object-relational mapping offers the prospect of eliminating this duplication—or, more precisely, automating it and hiding it from developers.

There is no universal consensus for the meaning of the term *object-relational mapping*. In the broadest sense, it can refer to any of the tasks mentioned in the previous paragraph—designing object-relational mappings, building databases, generating API code—or any combination of them (often all three). It can even refer to tasks in the reverse direction: taking a database and transforming it into a set of classes. Software already exists and has been deployed to accomplish all of these tasks.

Chapter 2

Issues in ORM system architecture

Thus far we have portrayed the class of ORM tools as a relatively homogenous group of technologies for confronting a fixed set of problems. It should come as no surprise to learn that this is a simplification. In fact, object-relational mappers are remarkably diverse in their range of approaches to the ORM problem and in the features they provide. In this chapter, we consider, one by one, many of the most important points on which ORM systems differ—or, equivalently, the most important questions that ORM system designers face.

2.1 Mapping paradigms

There are a number of perspectives from which one can approach ORM. Choice of paradigm is perhaps the most fundamental decision involved in the development or selection of an ORM system. In this section, we describe and compare three major perspectives that an ORM system can assume.

2.1.1 Metadata-oriented

In the first type of ORM system, the only input of the application developer into the ORM process is metadata. The ORM system itself then generates the code. (This paradigm is often called *code generation*, but we use the term *metadata-oriented* to distinguish it from database-oriented ORM systems.)

The database can either preexist to be described by the metadata, or the mapper can instead generate the database as it does the code. This is likely to vary on an application-by-application basis. In systems that are constructed from the ground up, it is convenient if the mapper can build the database. If the code is to use a legacy database, the mapper need not generate the database, but instead must write the code to be compatible with the existing schema.

The advantage of this method is simplicity for the developer. Not only does it free the developer from worrying about the persistence mechanism, but also from worrying about the internal workings of the mapped object or of the database itself (except to the extent that the developer needs to specify some of these details in the metadata). The disadvantage is a loss of flexibility. Since the code is generated, the developer cannot ordinarily modify it and is limited to the functionality that the mapper provides. Likewise, since the database is either generated or preexisting, the developer cannot update it directly either.

2.1.2 Application-oriented

Here, the application developer writes the object-oriented code for all of the application, including the objects to be mapped (except their persistence code). The mapper, usually aided by additional metadata provided by the user, then writes the persistence code for these objects. It may create the database in which the objects are to be persisted itself, or it may use an existing database or one created by the application developer.

The advantage here is flexibility, as this model enables the developer to write arbitrary logic into the classes that are to be mapped. This model also has the theoretical advantage that it allows application developers to focus on their domain of expertise—the object-oriented world—shielding them from the less familiar (and more distant from the business logic) world of databases. Due to these advantages, many, though by no means all, of the most popular ORM systems today are application-oriented.

An obvious requirement for a application-oriented mapper is that it be able to read the code. There are three ways of approaching this. First, the mapper can parse and interpret the raw code. Second, it can parse and interpret the compiler output (e.g., machine code or bytecode). Third, it can use reflection to access the structure of the code at runtime. This last method is the most sophisticated and the most resource-intensive.

2.1.3 Database-oriented

A final perspective for a mapper to take is that of the database. Here, the application developer builds the database, and the mapper probes the database and infers the class structure from it (again often assisted by metadata). It finally generates the code for the objects.

Just as application-oriented mapping has the advantage of speaking to developers in their own (object-oriented) language, database-oriented mapping has the perceived disadvantage of speaking to them in exactly the wrong language and denying them the flexibility to edit the mapped classes. On the other hand, this model may be well suited to applications in which the data structure and the relationships between different types of data are especially salient and the business logic is less important.

2.1.4 Problems with this classification

Not all mappers fit neatly into this taxonomy. Some mappers have more than one mode of operation. In some situations, application developers may actually build all three components themselves—code, metadata, and database—leaving the mapper with the sole task of adding persistence functionality to the existing code, using the provided metadata to reference the existing database. It is not clear which category such a system might best fit into.

2.2 Transparency

Different ORM systems, even ORM systems that adhere to the same mapping paradigm, often exhibit substantially different architectures. One property of an ORM system is *transparency*. The term *transparency* is usually used in reference to application-oriented systems. Such a system is transparent if classes do not need special infrastructure to persist their data. Some ORM systems require classes to implement mapper-specified interfaces or carry special attributive structures in order to be persistent. By contrast, a transparent system, "ordinary" classes can be persisted, without cluttering them with structure whose sole purpose is to satisfy the persistence layer. Indeed, in a transparent system, classes do not even know that they are persistent. Transparency is based on the notion that business classes should not need to know how, why, or even whether they persist their data; objects should behave the same whether they exist only in memory or whether they are stored permanently.

In a transparent system, auxiliary classes in the persistence layer carry out the mapping. These classes consume business classes and persist their data, often via existing database APIs such as ODBC and JDBC. In particular, there is often a *persistence manager* class that provides all persistence services for application objects: constructing and executing queries, controlling transactions, and managing the cache.

2.3 Inheritance mapping

Inheritance is one of the most fundamental and integral notions in object-oriented programming [4]. However, the concept is absent from the relational paradigm. Consequently, the issue of mapping inheritance hierarchies is one of the first deep problems that ORM systems must confront. Of course, the problem is not exclusive to ORM. Software engineers writing database connection code for classes by hand face the same problem. However, software engineers can intelligently decide inheritance mappings on a case-by-case basis; ORM software must adopt a consistent, systematic strategy for mapping inheritance hierarchies that works reasonably well all (or almost all) the time.

In this section, we consider several strategies for mapping inheritance hierarchies. To concretize our discussion, we will consider how these strategies would be applied to the simple inheritance hierarchy depicted in Figure 2.1.

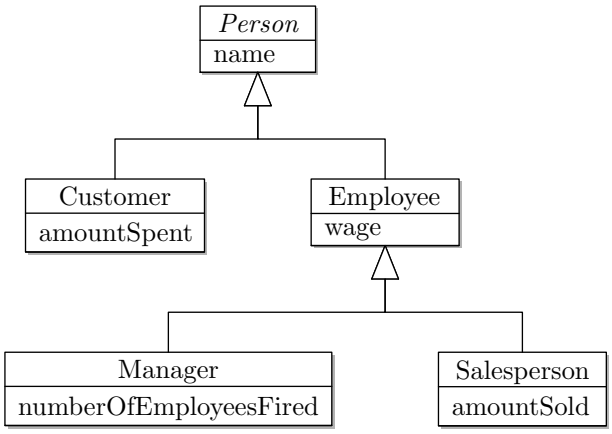

Figure 2.1: An inheritance hierarchy of the sort that an ORM system might encounter. The `Person` class is abstract (its name is in italics); all others are concrete.

Strictly speaking, this hierarchy suffers from a number of design problems that refactoring could solve. For one thing, it doesn't allow a person to be both a customer and an employee. Also, depending on system requirements, we would probably want to store more information than just, say, number of employees fired; we would probably want to know which employees a manager fired on which dates. Nonetheless, this example will reveal the basic issues that an ORM system faces in handling inheritance hierarchies.

There are three major approaches to inheritance mapping, described in [5, §3.6] and [1, §2]. The rest of this section covers each in turn.

2.3.1 Single table per hierarchy

Perhaps the simplest way to map an inheritance hierarchy to the relational model is to map it to a single table. This single table must contain the columns necessary to describe any instance of any class in the hierarchy fully. In our example, this procedure would produce a single table, `Person`, with a column for each data member of each subclass of the `Person` class (Fig. 2.2). There are also two new columns: a surrogate key (`PersonId`) and a field recording the subtype to which the instance belongs (`PersonType`). Figure 2.3 shows sample data for this table structure.

Besides being easy to understand and implement, this method also has the advantage of relatively high performance. The DBMS does not need to execute any joins or other resource-intensive operations across tables, and the job of the persistence layer is likewise straightforward.

A disadvantage of the method is that it maps large inheritance hierarchies to wide tables. Since each class in the hierarchy may have many data members,

Person
PersonId
PersonType
Name
AmountSpent
Wage
NumberOfEmployeesFired
AmountSold

Figure 2.2: The hierarchy of Figure 2.1 can be mapped to this single table.

Person						
Person-Id	PersonType	Name	Amount-Spent	Wage	NumberOf-Employees-Fired	Amount-Sold
1	Employee	Liz	NULL	$15	NULL	NULL
2	Customer	Pat	$200	NULL	NULL	NULL
3	Salesperson	Jim	NULL	$10	NULL	$1200
4	Customer	John	$500	NULL	NULL	NULL
5	Manager	Mary	NULL	$50	7	NULL

Figure 2.3: Sample data for the table structure of Figure 2.2

the number of fields in the resulting table could quickly become unmanageable.

2.3.2 Table for each class

The opposite strategy is to create a table for each class in the inheritance hierarchy. In our example hierarchy, this would result in five tables (Fig. 2.4). Now `PersonId` becomes the primary key for all five tables—and simultaneously a foreign key for all but the base class.

Although all three methods preserve all the *data* that passes through the persistence layer, this method is the only one that perserves the *structure* of the inheritance hierarchy in its entirety. In essence, it translates the inheritance structure of the object-oriented system directly into a relational structure in the database, confronting the object-relational impedance mismatch head-on. The output of this method is at a higher level of normalization (at least in an informal sense) than that of the other methods; it renders all the null values of the first method unnecessary by persisting the inheritance structure to the database.

This has a certain theoretical appeal, but the downside is a large quantity of tables and relationships and relatively low performance. The normalization that this method affords may not be worth these costs.

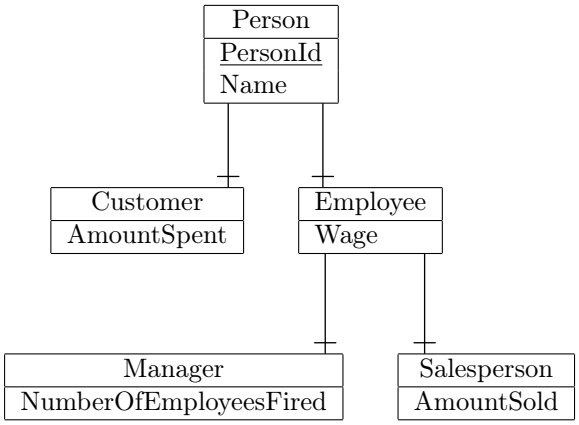

Figure 2.4: Here, we map each class in the hierarchy of Figure 2.1 to a table.

Person	
PersonId	Name
1	Liz
2	Pat
3	Jim
4	John
5	Mary

Customer	
PersonId	AmountSpent
2	$200
4	$500

Employee	
PersonId	Wage
1	$15
3	$10
5	$50

Manager	
PersonId	NumberOfEmployeesFired
5	7

Salesperson	
PersonId	AmountSold
3	$1200

Figure 2.5: Sample data for the table structure of Figure 2.4. This is the same data as Figure 2.3 but under a different mapping.

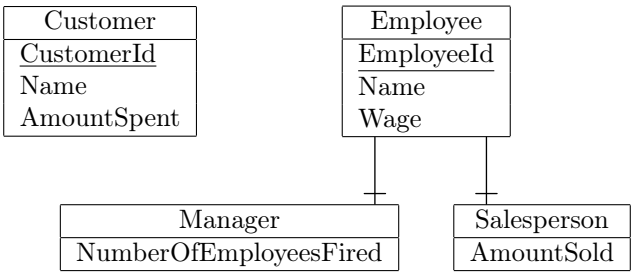

Figure 2.6: Here, we map each concrete class in the hierarchy of Figure 2.1 to a table.

Customer		
CustomerId	Name	AmountSpent
2	Pat	$200
4	John	$500

Employee		
EmployeeId	Name	Wage
1	Liz	$15
3	Jim	$10
5	Mary	$50

Manager	
EmployeeId	NumberOf-EmployeesFired
5	7

Salesperson	
EmployeeId	AmountSold
3	$1200

Figure 2.7: Sample data for the table structure of Figure 2.6. This is the same data as Tables 2.3 and 2.5 but under a different mapping.

2.3.3 Table for each concrete class

The final method is a slightly more pragmatic version of the technique just discussed. Here, once again, there is a table for each class, but this time abstract classes are excluded. (In our example, the `Person` class is the only abstract class.)

This compromise is problematic. First, it unappealingly requires duplication of semantically identical fields across tables. Here, the `Name` field occurs in two tables, and the problem would be worse if `Person` had more subclasses or if `Employee` were also abstract. If we wanted to modify this field, we would have to do so in several places. More importantly, this mapping makes polymorphic queries difficult. If we want to search for all people named Pat, we must first search the `Customer` table and then the `Employee` table. This is an inconvenience and a performance hit.

2.3.4 Existing support for inheritance mapping methods

Existing enterprise-quality ORM tools often provide software developers with a choice of inheritance mapping techniques, so that inheritance mapping can be optimized on a case-by-case (hierarchy-by-hierarchy) basis. For example, Hibernate provides `<subclass>`, `<joined-subclass>`, and `<class>` metadata elements to achieve table-per-hierarchy, table-per-class, and table-per-concrete-class mapping, respectively. The following snippet of Hibernate metadata XML would map our person example to a single table.

```
<hibernate-mapping>
   <class name="Person" table="Person" discriminator-value="P">
      <id name="id" column="PersonId" type="long">
         <generator class="native"/>
      </id>
      <discriminator column="PersonType" type="string"/>
      <property name="name" column="Name" type="string"/>
      <subclass name="Customer" discriminator-value="C">
         <property name="amountSpent" column="AmountSpent"/>
      </subclass>
      <subclass name="Employee" discriminator-value="E">
         <property name="wage" column="Wage"/>
         <subclass name="Manager" discriminator-value="M">
            <property name="numberOfEmployeesFired"
               column="NumberOfEmployeesFired"/>
         </subclass>
         <subclass name="Salesperson" discriminator-value="S">
            <property name="amountSold" column="AmountSold"/>
         </subclass>
      </subclass>
   </class>
</hibernate-mapping>
```

This code is rather clear. The only points that may need explanation are the "discriminator" element and attributes. These control the schema and content of the `PersonType` field, which distinguishes between different subtypes of `Person`. (Remember that in this model, all data goes into one table, so the only reliable way to determine which subtype a record belongs to is via this discriminator field.)

In conclusion, large and complex hierarchies are problematic for any inheritance mapping technique. Under a mapping strategy that maps an entire hierarchy to a single database table, large hierarchies can result in unmanageably wide tables. Under a strategy that maps each class to its own table, large hierarchies produce unmanageably numerous tables. However, large and complex inheritance hierarchies are a design problem independent of ORM. Skillful refactoring can reduce the complexity of an inheritance hierarchy without compromising its integrity, making it more maintainable and useable and

simultaneously obviating much of the problem of inheritance mapping.

2.4 Transaction processing

Traditionally, applications interact with DBMSs in *transactions*, or logical units of work. According to the classical definition [15], the system guarantees that it will execute the transaction exactly once (reliability); that if it executes one part of the transaction, it will execute all of it (atomicity); and that it will insulate the transaction from other database operations, so no such operation can see the data in an intermediate state (isolation). Additionally, the database must be in a legal state when the transaction begins and when it ends (consistency). In practice, modern transaction processing systems do not always meet all these criteria all the time, but they abide by them to the extent that pragmatic considerations allow.

Nowadays, transaction processing is a core functionality of any professional-grade DBMS. Most flavors of SQL contain commands such as `START TRANSACTION`, `COMMIT`, and `ROLLBACK` [19]. (In fact, transaction processing commands have been part of the SQL standard since 1992 [14].) To execute a database transaction, an application issues a `START TRANSACTION` command and then performs a sequence of database operations. At that point, it can `COMMIT` these changes, concluding the transaction and making its effects visible to all users. Alternatively, it can `ROLLBACK` the transaction, which returns the database to its previous state and terminates the transaction.

The classic example of a scenario requiring the protection of a transaction is a transfer between two bank accounts. Suppose that Angela wants to transfer $100 to Bob. We can view this transfer as two operations: withdrawing $100 from Angela's account and depositing $100 in Bob's account. We could naïvely complete such a transfer by first issuing a

UPDATE Account **SET** Balance = Balance − 100 **WHERE** Id = 1

statement against Angela's account. Subsequently we would issue a command

UPDATE Account **SET** Balance = Balance + 100 **WHERE** Id = 2

to Bob's account. (In practice we might want to insert checks to make sure that the balance does not fall below 0, but for simplicity we ignore such considerations in this example.) This works fine in ordinary circumstances. However, if the second command fails, the database is left in an inconsistent state; $100 has disappeared from the system. Moreover, even if both commands ultimately succeed, a query from some other user could be interposed between the two, and that query would see the database in an inconsistent state, possibly leading to incorrect or incoherent results. Transactions solve this problem.

START TRANSACTION
UPDATE Account **SET** Balance = Balance − 100 **WHERE** Id = 1
UPDATE Account **SET** Balance = Balance + 100 **WHERE** Id = 2
COMMIT

This ensures that no error or interruption will leave the database in an inconsistent state and that no other statements can become interposed and leave the database in an inconsistent state.

Database APIs often provide interfaces for database transactions that use the transaction functionality that the DBMS provides. This allows API users to conceive of transaction processing at a higher level of abstraction. For example, consider the following Java snippet, which uses JDBC to perform the bank transfer described above.

```
con.setAutoCommit(false);
PreparedStatement changeBalance = con.prepareStatement(
    "UPDATE Account SET Balance = Balance + ? WHERE Id = ?");

changeBalance.setBigDecimal(1, −100.00);
changeBalance.setInt(2, 1);
changeBalance.executeUpdate();

changeBalance.setBigDecimal(1, 100.00);
changeBalance.setInt(2, 2);
changeBalance.executeUpdate();

con.commit();
con.setAutoCommit(true);
```

The `setAutoCommit` method in the first and last lines refers to the auto-commit feature in JDBC, which automatically wraps each statement in a transaction. With this feature activated, as it is by default,

```
UPDATE Account SET Balance = Balance − 100 WHERE Id = 1
UPDATE Account SET Balance = Balance + 100 WHERE Id = 2
```

would become

```
START TRANSACTION
UPDATE Account SET Balance = Balance − 100 WHERE Id = 1
COMMIT
START TRANSACTION
UPDATE Account SET Balance = Balance + 100 WHERE Id = 2
COMMIT
```

Turning auto-commit off effectively begins a manually controlled transaction. Using a prepared statement for improved performance, we subtract $100 from account 1 and add it to account 2. Finally, we commit the transaction and restore auto-commit.

Transaction processing is too complex and domain-specific a functionality to be provided by object-relational mappers automatically. In general, there is no way for a mapper to tell when groups of commands need to be issued as a transaction and when they can be issued individually. The mapper, then, must

present the user with an interface for transactions that leverages the transaction processing infrastructure of the underlying DBMS (or API).

Hibernate provides a good example of this. Its API provides a `Transaction` interface with methods for enclosing a database transaction.

```
Session session = getSessionFactory().openSession();
Transaction tx = null;
try {
    tx = session.beginTransaction();

    Customer angela = (Customer) session.get(Customer.class, 1);
    angela.changeBalance(-100.00);

    Customer bob = (Customer) session.get(Customer.class, 2);
    bob.changeBalance(100.00);

    tx.commit();
}
catch (Exception e) {
    if (tx != null) tx.rollback();
    throw e;
}
finally {
    session.close();
}
```

Hibernate does not actually propagate any of the database actions until `Transaction.Commit` is called. This "flushes" the session, synchronizing it with the database. If an exception occurs in the meantime, this code rolls back the transaction. Regardless of what happens, exception or no exception, the code concludes by closing the session, which releases the JDBC connection to the pool.

2.4.1 Isolation levels and locking

In order to maintain transaction isolation, DBMSs rely on *locking*, which temporarily prevents concurrent access to particular data structures. Under a *pessimistic locking* policy, a database user locks an item of data as soon as it reads it and retains the lock until it completes the transaction. This guarantees isolation, but it can lead to database-level deadlocks, which limits the scalability of the policy. Consequently, many systems relax their demands on the isolation of their transactions, allowing for a more lenient *isolation level* than complete isolation. The possible isolation levels are defined by the ANSI/ISO SQL standard in terms of the isolation anomalies that they allow or prevent. There are three relevant types of isolation anomaly:

Dirty read This occurs when one transaction reads changes made by another

transaction that has not been committed. This becomes problematic if the other transaction is rolled back, invalidating the value read.

Non-repeatable read This occurs when a transaction reads the same record twice and receives different results. This would happen if a concurrent transaction had modified the record (and committed its modification) between the two reads.

Phantom read This occurs when a transaction executes a query twice and the second result set includes records that were absent from the first. This could happen if a concurrent transaction had inserted new records (and committed its insertion) between the two reads, or if it had modified existing records to satisfy the query.

The four isolation levels are defined by the anomalies they allow:

Serializable This level specifies full isolation and eliminates all of the anomalies described above. It behaves as if all transactions in the system were executed serially (although there can be concurrency in fact as long as the appearance of serial execution is preserved).

Repeatable read This level still prevents dirty and non-repeatable reads but allows phantom reads.

Read committed This allows phantom reads and non-repeatable reads but still prevents dirty reads.

Read uncommitted This level allows phantom, non-repeatable, and dirty reads. This is not equivalent to the absence of isolation, as read uncommitted locking still protects against some anomalies not described above, such as a *lost update*, which occurs when two transactions update a single row and the second one aborts, causing the first change to be lost as well.

These levels differ in the type and duration of locking they require, but the details are irrelevant.

The appropriate choice of isolation level varies from application to application. Developers must determine how to balance a desire for increased concurrency and better performance with a need to avoid subtle application bugs that are difficult to predict or reproduce. Read uncommitted isolation yields the best performance, but is dangerous to allow a transaction to use an uncommitted change from another transaction. Serializable isolation is the safest but does not scale well and is unnecessary for many applications. The factors that go into these decisions are again beyond the scope of this paper (see e.g. [28, pp. 181–183] for more information). What is important is that isolation level can vary on a per-application basis, so many persistence frameworks provide applications with a means of configuring the isolation level. Of course, database administrators can set a default isolation level on the database itself, but many database APIs and persistence frameworks allow this value to be reset by the application. In a Hibernate configuration file, for example, an application developer can write

```
hibernate.connection.isolation = 4
```

to set repeatable read isolation.

Some persistence layers also allow applications to set database locking on a per-item basis, affording them a finer level concurrency control. Let us continue to use Hibernate as an example, since it offers especially good concurrency control. The Hibernate API includes a `LockMode` class, which allows API users to demand a pessimistic lock on a particular item. (It has other features too, such as forcing Hibernate to bypass the cache or execute a version check.) To obtain a pessimistic lock on a customer, one would use an API call like

```
session.get(Customer.class, customerId, LockMode.UPGRADE)
```

If unspecified (as it has been in previous examples), the default lock mode is `LockMode.NONE`.

2.4.2 Application transactions

The transactions we have discussed—called *database transactions* or *system transactions*—so far are what are usually referred to when people speak of transactions. These are fine-grained transactions and must be restricted in duration, lest their isolation requirements prevent concurrency and reduce the scalability of the application. Besides these database or system transactions, though, we can also talk about coarser-grained units of work, transactions at the level of the application or business logic.

For example, consider a Web application that interacts with a user in a standard request-response paradigm. In a typical sequence of events, the (server-side) application might query data from the database and provide it to the client in a Web response. The user might indicate modifications to be made to the data via the Web browser and send these modifications to the server as a Web request. Finally, the server might make those modifications. This exchange might take many minutes. This is far too long for a database transaction. If we tried to wrap this entire sequence in a database transaction, the entire database system could be held up while waiting for the user to respond to the request (which might take a very long time or might never happen at all). This is clearly unacceptable in a system with more than one user. At the same time, this sort of interaction does raise concurrency issues. What happens if someone else modifies the data between the initial response to the client and the client's update request? The database can provide no help here, so it is up to the application to handle these *application transactions* or *user transactions*.

For the purpose of clarity, let us take an even more concrete example to use throughout this section: a wiki page. A wiki is a content-based Web site that allows any user to edit any page, often without registration. On popular wikis, scenarios like the following are common:

1. Anthony requests a wiki page. The server responds with the current version of the page.

2. Barbara requests the same wiki page. The server responds with the current version of the page.

3. Anthony submits edits to the page. The server updates the database to reflect the new content.

4. Barbara submits her edits to the page, unaware that a competing user (Anthony) has already submitted edits of his own.

How should the application handle such an occurrence? There are several possibilities:

Record locked during editing One option is for users to notify the application when they are about to make edits. The application would then prevent other users from attempting to make updates until the first user had completed the edit. In this example, Anthony would, in step 1, notify the application (as part of his request) that he was planning to edit the content. Then, when Barbara made a similar request in step 2, the application would deny her access until Anthony completed the request in step 3. This policy is extremely problematic, as it prevents concurrent data use entirely for minutes at a time—perhaps much longer, if Anthony disconnects from the Internet when the application is still awaiting an edit from him. It would also require substantial database infrastructure to support it, including a *lock table* for keeping track of application-level locks. We present this policy for completeness and because it is analogous to database-level pessimistic locking; it would seldom be appropriate for application-level transactions in a real, concurrent system.

Last commit wins Under this policy, both updates would succeed. Barbara's update would overwrite Anthony's, and neither user would receive any notification or error message. This is the easiest solution to implement (it requires no work at all on the part of the application developer or the persistence layer), but it is problematic. What if Anthony's edit was to correct an important error in the content of the page and Barbara's update was to correct a different but also significant error elsewhere in the page? Anthony's update would be lost, and no one would be the wiser for it (unless Anthony happened to check back later and notice the update missing).

First commit wins The opposite strategy is to allow Anthony's edit to prevail, presenting Barbara with an error message when she attempts to overwrite his edit. This solution, also called *optimistic locking*, is often sufficient, as Barbara can take appropriate action to deal with the conflict.

Merge updates A related strategy is to not only notify Barbara of the conflict, but also give her the opportunity to merge her changes with Anthony's selectively. This gives users a great deal of power to resolve conflicts in the best way possible—to assert that their own changes should overwrite

conflicting users' changes, to retract their own changes in the light of conflicting changes, or to merge changes so that the essence of all competing changes is preserved.

An even more sophisticated policy is to merge changes automatically whenever possible. For example, if Anthony corrects a typographical error in the first paragraph of a lengthy document and Barbara corrects an error in the final paragraph, the application might be intelligent enough to reconcile those changes without needing to consult any humans. Of course, users still need to be consulted when conflicts occur in proximity to each other. Automatic merging is not widespread in present-day wikis, but it is found in some version control systems such as CVS [8, §10.2].

In order two apply either of the last two policies, of course, an application needs to be able to detect conflicts. This is not as trivial as it might sound. If our wiki were implemented as naïvely as the preceding text suggests, the application would have no way of knowing whether Anthony had been editing the version of the page that Barbara submitted or whether he had been using an old copy (as, in fact, he was). To empower our application to recognize such situations, we must introduce some notion of versioning. Each wiki page must have some notion of the version it is currently on, or at least a time stamp indicating the date and time of the last edit. This version number or time stamp should be passed on to database users like Anthony and Barbara, who must indicate what version of the record they are editing when they submit their requests. In this case, both Anthony and Barbara would indicate that they were editing, say, version 4 of the page, or the version that took effect on February 18, 2007, at 21:42:32.591. When Anthony submitted his update, the application would increment the version number in the database record (or update the time stamp) at the same time as it made the changes he requested. Then, when Barbara submitted her update, the application would notice that the version number or time stamp she provided (4 or 2007-02-18 21:42:32.591) did not match that in the database (5 or the time of Anthony's update) and could handle the situation according to its conflict resolution policy.

This is where ORM comes in. An ORM tool can manage versioning for the application, saving the developer considerable work. In an ORM system that supports managed versioning, the developer typically adds an integral `Version` column (or something similar) to the database and indicates in the metadata that it should be used for managed versioning. In the Hibernate metadata format, for example, there is a `version` element that looks like this:

```
<version name="version" column="Version" />
```

(Hibernate also supports time stamps.) Hibernate modifies all its persistence commands accordingly, automatically incrementing the version column at each update and committing updates only when the version field holds the expected value. Instead of executing SQL like

```
UPDATE Customer SET Surname = "Johnson" WHERE Id = 3
```

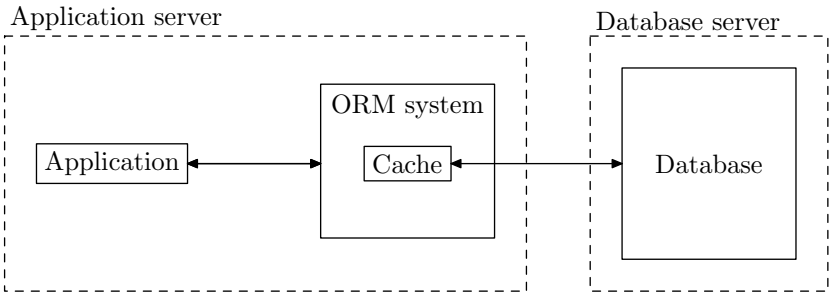

Figure 2.8: High-performance ORM systems insert a cache between the application and the database. The cache stores a subset of the information in the database—the subset that the mapper deems (at run time) most likely to be used soon—for faster access to these items, improving application performance.

Hibernate would execute

```
UPDATE Customer SET Surname = "Johnson", Version = 5
WHERE Id = 3 AND Version = 4
```

Hibernate throws a `StaleObjectStateException` if the update fails. The application can catch this exception and apply a first-commit-wins or merge-updates strategy appropriately.

2.5 Caching

One criticism of object-relational mapping has been founded on the assumption that mapping systems generally suffer from poor performance relative to hand-coded solutions. On its face, this assumption seems reasonable. Not only can hand-coded solutions exploit, in principle, domain-specific peculiarities to improve performance, but also many mapping systems rely on resource-intensive features such as reflection.

In fact, mapping systems can often *improve* on the performance of hand-coded solutions. One of the most important reasons for this is the incorporation of caching functionality into mapping systems. In many database-driven applications, database access (i.e., communication with a database server) is a bottleneck. By caching the results of queries locally, applications can often get away with less frequent communication with the database server. Of course, hand-coded persistence solutions can certainly employ caching, but the formidability of developing a caching solution for application objects on an individual basis often makes this practice uneconomical. By contrast, a mapping system can provide this functionality to applications freely and often transparently.

More explicitly, an object-relational mapping cache is a temporary store that resides between the application and the database (most often on the application server machine itself) and retains some content from the database so that the

application can access this content without a round trip to the database. This cache is part of the object-relational mapping system and is separate from any cache that might exist as part of the database management system (or as part of the application). It is conceptually analogous to any other computer cache, such as a CPU cache. However, the nature of an RDBMS-driven application persistence mechanism poses unique issues and challenges.

In their book on Hibernate, Bauer and King [5], §5.3.1 explicate a classification of ORM caches based on their scope. They identify the following cache scopes:

- Transaction scope. The cache is associated with a particular "unit of work," which may be a database transaction or a meaningful unit of work in an application.
- Process scope. The cache may be shared by many transactions.
- Cluster scope. The cache may be shared by multiple processes or multiple machines. A cache at this scope requires remote process communication for consistency, and data must be replicated to all nodes in the cluster.

Using any cache beyond the scope of a transaction can create problems with concurrent access. This is especially problematic when an application does not have exclusive access to the database. Suppose, for example, that our ORM-backed application shares a database with a legacy application that predates the ORM system. In general, barring the construction of some complex, custom, environment-specific solution using database triggers, our application has no way of knowing when the legacy system has changed the contents of the database.

If an application does have exclusive access to the database, though, higher-level caching can be helpful. Some classes are better candidates for high-level caching than others, of course. A class whose data is relatively static is a better candidate for process-level caching than one whose data changes frequently. Also, in some application domains, it may be acceptable to see slightly out-of-date data occasionally. For this reason, major mapping systems provide mechanisms for setting caching strategy on a class-by-class basis. To take a concrete example, iBATIS supports the following caching models [2, §3.8]:

- "Memory" cache. Cache data is stored in reference types, so the garbage collector effectively manages the cache.
- LRU cache. When the cache becomes full, the least recently used object is removed from the cache. According to the iBATIS documentation, this model is well suited for situations in which certain objects may be popular over long periods of time.
- FIFO cache. When the cache becomes full, the oldest object is removed from the cache. This model is good for situations in which an object is likely to be referenced several times in quick succession, but then perhaps not again afterwards.

2.6 Metadata

Some database-driven applications are so simple that a clever mapper can infer the relationship between the code and the database just by examining one or the other (or both). In virtually all non-trivial cases, though, a mapper will need some extra information to understand the relationship between the code and the data. At the least, it seems, a mapper ought to know which classes are to be mapped, since in most applications there are many classes that do not correspond to database relations. Hence there should be some way to mark specific classes for mapping (or perhaps instead mark specific classes that are not to be mapped). Most likely, though, the metadata will be substantially more elaborate than this. The mapper will need to know which classes correspond to which relations, which members correspond to which columns, the types to which specific fields should be mapped, and so on. These decisions can sometimes be inferred automatically, but in many cases this is impossible.

There are several ways in which mapping metadata can be created and stored. A popular medium is the Extensible Markup Language (XML). Hibernate, for example, is an important mapper that keeps its metadata in XML documents. The syntax and structure of a Hibernate mapping document are enforced by the rules of XML syntax and by a Hibernate-specific Document Type Definition, respectively. A Hibernate mapping document has a `<hibernate-mapping>` XML tag as its root. It typically has child tags such as `<class>`, which provides metadata for a specific Java class in the application. A `<class>` tag can in turn have children such as `<property>` and `<id>`. All these tags have attributes that provide the metadata (see §3.1.2). Many other mappers use essentially the same idea (see e.g. §3.2.2 and §3.4.2).

There is nothing that restricts us to XML. Other mappers use other textual formats for metadata. For example, the Enterprise Objects Framework, an object-relational mapper built into Apple's Web application server software, WebObjects, relies on custom-format files called model files to describe mappings between classes and tables, between class attributes and database columns, and between instances and database rows [3]. There is no fundamental difference between XML and other textual formats, so we regard them as conceptually the same. (There are, of course, practical differences. XML provides several advantages over custom formats. Perhaps most importantly, most developers are already familiar with XML, so the learning curve will be shallower for XML-based mappers. Additionally, a wide variety of tools exist for composing, editing, visualizing, and validating XML, which gives developers a great deal of flexibility in constructing and managing mapping documents. XML also makes programmatic creation and manipulation of mapping documents relatively easy, since many enterprise frameworks and class libraries have built-in XML functionality. On the other hand, custom formats may provide other advantages, such as terseness.)

Whether using XML or a custom format, mappers that keep their metadata in text files often provide graphical interfaces to allow developers to create

mapping documents more easily and intuitively. This saves them the trouble of remembering syntax rules and navigating lengthy, machine-readable text documents. In the Enterprise Objects Framework mentioned earlier, model files are practically always constructed using a visual tool such as EOModeler. Ordinary developers never even see the underlying text. Of course, the visual tool need not be developed by the developers of the mapper themselves. Since the text formats are (at least ideally) generally simple, well defined, and clearly documented, it is easy for third parties to develop graphical tools for manipulating mapping metadata files. Regardless, placing a graphical tool on top of a text format does not represent a fundamental change in conception. It is nothing more than a productivity aid for developers. These graphical interfaces may have substantial practical importance, but they are of little theoretical significance in this context.

A more fundamental distinction exists between mapping systems that keep their metadata in standalone text files, which we have just described, and systems that keep metadata inline, within the code itself. The Microsoft .NET languages, in particular, incorporate features that make the latter technique more elegant than it would be in other languages. Specifically, .NET introduced the idea of attribute-based programming. An attribute is a language construct that allows developers to impart metadata to code elements such as types to extend their functionality. This is done inline via a syntax designed for the purpose.

In languages without attributive syntax, such as Java, this technique is not quite as elegant. Most often, it is done through the use of specially marked comments. XDoclet is a code generation library that introduces attribute-based programming to Java via custom Javadoc tags. XDoclet can be configured to generate XML mapping documents for Hibernate from the inline Javadoc tags. For completeness, we should note that Java Specification Request 175 introduces "annotations" to Java [43], which are analogous to .NET attributes. JSR 175 annotations are likely to supplant XDoclet in the future. Still, the idea of simulating attribute-based programming via specially marked comments is not unique to Java. The same technique could be used to introduce attribute-based programming, and hence inline ORM metadata, to other languages.

2.7 Dirty checking

Dirty checking is a technique that exists in several ORM systems. To perform dirty checking, an ORM system must determine transparently whether and how a persistent object has been modified by the application since it was last synchronized with (i.e., retrieved from or stored to) the database. This allows the system to tell whether the application and the database need to be resynchronized (i.e., whether the database needs to be updated to reflect the changes that the object has undergone). To clarify the metaphor implicit in the name *dirty checking*, a system that supports dirty checking determines whether an object has been "dirtied"—modified—since it was last known to be in a "clean"—up-to-date—state.

Dirty checking is primarily a convenience for application developers. It is not by any means a necessity, nor is it an integral component of any ORM system, and many ORM products lack it. The alternative to dirty checking is to persist objects only when application developers explicitly request it (i.e., by expressly calling the persistence layer to operate on specific objects in the business layer). This is not so bad; it just means that application developers must remember to save their changes when they are done making them. In particular, if they persist an object and make subsequent changes to it after the save, they must remember to persist the object again before disposing of it, so that the changes will not be lost. Many ORM tools function perfectly well with this model.

However, although not an imperative, dirty checking is a convenience. A mapper that supports dirty checking can save the application developer the trouble of thinking about when and when not to save objects. The danger in failing to save objects when they have changed state is clear: the changes will be lost, and future operations that rely on the data that should have been persisted will be incorrect. However, it is worth noting that there is also a danger, albeit a lesser one, in saving too often: performance. Developers could take the naïve approach of saving after every change they make, but all these database interactions would result in severe performance penalties. Thus, dirty checking can improve performance by ensuring that objects are saved only when they need to be.

To clarify the change that dirty checking effects from the developer's point of view, consider the following sequence events in a system without dirty checking:

1. Via the application-provided user interface, the end user requests, in effect, that order 38 be associated with customer 92.

2. In response to the request, the application creates a transaction by calling the appropriate procedure in the persistence layer (i.e., the API provided by the ORM system).

3. Via the persistence layer, the application loads customer 92.

4. Via the persistence layer, the application loads order 38.

5. Via the business layer, the application associates the loaded order with the laoded customer. This would be done via the methods on the business objects that the application developers themselves have likely architected. The syntax might look something like `c.Events.Add(o)`, where `c` is the customer object and `o` is the order object. (This example is in C# or Visual Basic .NET. Here, `Events` is supposed to be a read-only collection property of the `Customer` class.)

6. The application saves the customer record. Note that, from the database point of view, what actually needs to be saved (in order for the change described above to persisted) is not any data in the `Customer` table at all. Instead, a relationship must be established between a `Customer` entity and an `Order` entity. Assuming the relationship between `Customer` and

`Order` is an ordinary, one-to-many relationship, this means that what actually needs to be saved is not the `Customer` table at all, but instead the `Order` entity in question needs a new foreign key value. An ORM system without dirty checking is not smart enough to realize this though. Consequently, either the application developer is left with the task of specifying exactly what part of the `Customer` object needs to be saved (in this case, one of its associated orders)—which is a burden for the developer—or the `Customer` object needs to save everything—not just the data in the `Customer` table itself, but also all data for all objects associated with `Customer`. If `Customer` has a large graph of objects associated with it, this could be a very expensive operation.

7. The application closes the transaction.

Dirty checking solves the problem presented by step 6 in a very clean way: by eliminating step 6, or rather making it implicit in step 7. In a system with dirty checking, all changes to the persistent objects that participate in the transaction are saved to the database.

The major question with dirty checking is that of how to detect when an object changes state. This is harder than it might sound; remember, in the object-oriented model, objects are black boxes and expose themselves only insofar as they wish to be exposed. They are not, in general, compelled to notify anyone when they change state, including the ORM system (which itself may be transparent to the object). There are a number of approaches to this problem.

2.7.1 Inheritance from a base class that monitors state change

The oldest, most naïve method of detecting object state changes is to generate an abstract base class that contains the logic for managing state changes. Usually, this base class is generated or otherwise provided by the ORM solution. The problems with this approach are obvious. First, it is a burden on developers. It is an inconvenience for business classes to have to inherit from a common base class (which the developers may have to go to some trouble to generate). But it is more than just an inconvenience. In languages and language systems that support only single inheritance, such as Java and the .NET Common Type System, inheriting from a particular base class prevents a class from extending any other class. This means that if a class is to be persistent, it cannot otherwise make use of inheritance as a language feature. A more theoretical problem is that this approach precludes transparency. Classes have to *know* that they are going to be persistent; they cannot be made persistent at will.

2.7.2 Manipulation of source code

Another approach is to insert a step into (or, rather, immediately before) the compilation process, a step that modifies the source code to insert state change management logic directly into each class that is to be persisted via the ORM

framework. That is, the ORM system directly rewrites the source code before it is compiled, adding functionality to it in the programming language itself.

Rewriting source code has a messy feel to it that makes experienced software engineers cringe, and with good reason. In order to rewrite source code, an ORM system has to first understand the existing code well enough to know where to insert what kind of code. This means that it must replicate much of the functionality of a compiler, such as lexical analysis, syntax analysis, and semantic analysis. It seems nonsensical for an object-relational mapper, of all things, to duplicate the functionality of a compiler. (At best, an object-relational mapper might be able to utilize existing compiler elements to parse and analyze the code, rather than using its own, custom-build parsing and analysis functionality, if the language provides compiling facilities that are transparent and accessible enough, but even this feels messy.) For one thing, this is a lot of work for the developers of the ORM software. For another, building parsing and analysis into an ORM system is prone to error. What if the authors of the ORM software fail to accommodate some little-known and little-used element of language syntax? Finally, code rewriting is very sensitive to language changes, such as the incremental modifications made to language specifications from subversion to subversion. An ORM system that works with version 2.1 of a language specification may not work with version 2.2.

There are more pragmatic considerations as well. If source code is rewritten before compilation, the error messages and other output produced by the compiler will be incorrect from the application developer's point of view. Line numbers will almost certainly be dramatically wrong.

Finally, it is hard to integrate source rewriting into the build process transparently. The sequence of operations in the build process varies greatly not only from language to language, but also from development environment to development environment (and between different developers using the same development environment, according to each developer's environment configuration). It is impossible to construct a general, reliable strategy for inserting a rewriting step before compilation that will work across many development environments. The problem is especially acute in development environments that support incremental compilation, a feature that effectively compiles as the developer types (or, rather, appears to do so from the developer's point of view), alerting the developer to both syntactic and semantic problems. Some systems, such as Microsoft Visual Studio 2005, even perform complete builds of projects on the fly to support requests for developer-architected services as they occur. It is difficult to see how source code rewriting could be reliably integrated into such a system.

2.7.3 Manipulation of bytecode

Many modern language implementations compile into a *bytecode* or *intermediate language*, which is in turn interpreted by a *virtual machine* that translates bytecode into platform-specific machine code (usually on the fly at run time) and executes the machine code. Java, .NET, Python, Tcl, and Perl (since version 6)

are all compiled into bytecode and executed by a virtual machine, and Ruby plans to adopt this model in version 1.9.1, due for release at the end of 2007 [27]. This model presents an alternative to rewriting source code: rewrite the bytecode instead. This carries a number of advantages over rewriting source code. Bytecode, though hard for a human to read, is easier for a machine to read (after all, it is designed for consumption by virtual machines, not people), which means that ORM technology can more easily understand and rewrite it.

Bytecode is also, to some degree, language-independent. To develop an ORM solution for the .NET Framework that used source rewriting, for example, ORM software engineers would need to support not just C#, but also Visual Basic .NET, C++/CLI, J#, and any other popular language that compiles into the Common Intermediate Language (CIL), the language interpreted by the .NET virtual machine, the Common Language Runtime. (This list is practically unbounded, since one of the selling points of the .NET Framework is that it is easy to write languages that compile into CIL.) Instead, ORM software engineers can directly manipulate the CIL, which allows them to support any of these languages, even .NET languages that are not widely used (like Boo and Mondrian for .NET) or do not even exist yet (Simula.NET, perhaps).

Bytecode processing is also more transparent to application developers than source rewriting. Indeed, application developers seldom have cause to even think about the bytecode that the compiler generates, let alone any modifications that ORM software might make to the bytecode after it is compiled. Of course, bytecode processing certainly won't cause problems of the sort one encounters with source rewriting—incorrect compiler output and the like.

Finally, bytecode processing is at least somewhat easier to incorporate into the build process than source rewriting. Bytecode processing happens after the rest of the build cycle is entirely completed (but of course before the bytecode is ever seen by a virtual machine). It is somewhat less problematic to insert a new step after the end of the build process than to insert one at its onset. Even so, depending on (intermediate) language and development environment, some complications arise in automating this step.

2.7.4 Reflection at run time

Many modern programming languages, including Java, the .NET languages, Objective-C, Smalltalk, Tcl, Eiffel, and most modern scripting languages, are *reflective*. A reflective programming language is one that allows an application to access and manipulate information about itself in the same way that it can access and manipulate information about its application domain. For example, object-oriented, reflective programming languages and platforms have classes that represent classes, classes that represent methods, classes that represent data members, and so on. Consider the following Java example. (In Java, reflection is provided by the java.lang.reflect package.)

```
Class c = Class.forName("Car");
Method m = c.getMethod("FillWithGasoline", null);
```

```
m.invoke(c.newInstance(), null);
```

Here, we instantiate the `Class` class; the new instance represents the entire `Car` class (not any particular instance). Then, we instantiate the `Method` class; this instance represents the `FillWithGasoline` method of the `Car` class. Finally, we invoke this method on a new instance of the class. This code is equivalent to the following, non-reflective Java code.

```
Car car = new Car();
car.FillWithGasoline();
```

Of course, this code is much simpler and cleaner. One would never write reflective code when non-reflective code would do equally well. However, the purpose of this example is to illustrate that it is possible for an application to reason about itself in the same terms that it can reason about its application domain; one can manipulate classes and methods in the same way that one manipulates cars and customers. Reflection has many practical applications, such as dynamically optimizing an application (perhaps even via self-modification) or adapting to diverse execution contexts at run time. Here, we will see another.

Reflection has one major problem: performance. In most platforms, including Java and the .NET Framework, reflection is quite expensive, due to the complexity of constructing memory structures that mirror the structure of the application itself. Indeed, even theorists who emphasize the importance of code maintainability over performance often shy away from reflection; reflection is dramatically more expensive than most ordinary operations. Fortunately, this will be slightly less of a problem for us. As it turns out, database operations are incredibly expensive themselves; a single use of reflection doesn't approach the time necessary to establish a database connection and transfer data across it. This is even more true if the database server is a different machine from the application, requiring communication across a network, which is even more expensive. Of course, reflection can still pose a performance problem if overused enough (especially when used to traverse large object graphs), but in general, moderate use of reflection will be much less expensive than the database operations that ORM systems, by their nature, must carry out frequently.

Reflection allows us to hook up state change monitoring logic at run time (rather than before or during the build). This affords a tremendous amount of transparency and flexibility. As far as the developer is concerned, business classes can appear entirely independent from the ORM framework that will support their persistence—and not only at design time, but even when the software has been built. Only at run time does the ORM software attach itself to the business layer. Provided that we can accept a performance hit, reflection provides a transparent, simple, and clean way for an ORM framework to monitor the classes it is to persist.

Chapter 3

Examples of ORM systems

It would be impossible to list, let alone examine, every ORM system in existence. There are hundreds or thousands of object-relational mappers that can be downloaded for free on the Internet or purchased through middleware vendors. Beyond that, an incalculable multitude of proprietary ORM software exists for private, internal use at individual software development companies. Here, we present a tiny, nonrepresentative subset of ORM systems that see widespread use in industry. We aim to exhibit systems that demonstrate the concepts underlying ORM exceptionally well, systems that are of especial theoretical interest, systems that present innovative solutions to the ORM problem, systems that test the limits of the applicability of ORM, systems that are particularly mature and well developed, and systems that are very widespread. Of course, these aims often overlap.

To emphasize that different ORM systems face the same sorts of questions, even though their answers to those questions may differ drastically, the sections in this chapter have parallel structure. We begin each section with an overview of the ORM system at hand, disclosing the basic facts about it, in many cases providing a simple example of its use, and perhaps briefly describing its history. In the first subsection, we look at the architecture of the system—not only whether it is, say, application-oriented or database-oriented, but also the structure of the ORM system itself. Then, we examine how the system handles the problem of metadata—how it gets the information it needs in order to wire up the persistence mechanism (see Section 2.6). Next, we tackle querying—the interface the system provides the application to express questions to ask the database. The next subsection is on caching, whose primary purpose is performance (see Section 2.5). Finally, we look at how the system handles transactions and concurrency in general (see Section 2.4).

3.1 Hibernate

Hibernate (http://hibernate.org/) is one of the most popular ORM systems in existence. Hibernate has been a non-commercial, open-source project since development began in 2001. Originally developed for Java, it was ported to .NET under the name NHibernate in 2005. Hibernate is not the oldest ORM system, but it is an excellent example to begin with, because it is a mature system that adopts a fairly traditional view of ORM while at the same time introducing a number of innovations to streamline the mapping.

To be persisted via Hibernate, objects need not implement any Hibernate-specific interfaces nor be placed in special wrapper objects. Hibernate can "manage" (make persistent) instances of ordinary classes with no special members at all. Instead of special interfaces or attributive devices, Hibernate uses mapping metadata in the form of XML files. Suppose we have a class called `Customer` like the one in Section 1.3. We could use a mapping document such as the following to orchestrate the mapping.

```
<hibernate-mapping>
   <class name="Customer" table="Customer" discriminator-value="P">
      <id name="id" column="CustomerId" type="long">
         <generator class="native"/>
      </id>
      <property name="name" column="Name" />
      <property name="streetAddress" column="StreetAddress" />
      <property name="postalCode" column="PostalCode" />
      <property name="country" column="Country" />
      <property name="telephoneNumber" column="TelephoneNumber" />
   </class>
</hibernate-mapping>
```

An instance `c` of this class can be saved to the database using code like the following:

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
session.save(c);
tx.commit();
session.close();
```

`Session` and `Transaction` are Hibernate interfaces that manage persistence. Specifically, they represent transactional units of database interaction. The `Session` object is an inexpensive, lightweight unit that serves as the interface for persistence operations. The `Transaction` API serves to abstract application code from underlying transaction implementation. The ultimate effect of these five lines of code is the execution of an `INSERT` statement against the `Customer` table in the database. Similarly, one could retrieve customer data from the database using code such as the following:

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
List customers = newSession.find("from Customer");
tx.commit();
tx.close();
```

The string "from Customer" is actually not SQL code to be inserted directly into the database query. Instead it is in Hibernate Query Language (HQL), which is based on SQL; Hibernate parses this text and uses it to compose the query to the database.

3.1.1 Architecture

In the Hibernate API architecture, the `Session` and `Transaction` classes, with some others such as `Query`, form the persistence layer of the application. Classes in the business layer, including the classes written by the application developer, such as `Customer` in this example, achieve persistence via this persistence layer. The database layer underlies the persistence layer; the Hibernate API interacts with the database via database APIs such as JDBC. This sort of architecture typifies many ORM solutions; developer-authored classes in the business layer interact with ORM classes in the persistence layer, which in turn use standard database-layer technology to achieve persistence. Of course, there are many interesting differences in the way that business-layer classes interact with the ORM technology in the persistence layer. In the case of Hibernate, for example, the persistence layer is called upon to service the business layer using independent metadata. In other ORM solutions, the business classes may be structurally related to ORM-provided structures. For example, some ORM solutions require business classes to implement interfaces provided by the ORM solution.

Objects in Hibernate follow a *persistence lifecycle*. The persistence lifecycle describes the states an object goes through in the course of being persisted to and retrieved from a data source. Objects begin their lives as *transient*, meaning that they are not associated with any record in the database. Transient objects are stored only in memory and become inaccessible (and available for garbage collection) as soon as they go out of scope or are disposed. The transactional architecture does not apply to transient objects; in particular, they do not receive rollback functionality.

Objects become *persistent* when they become associated with a record in the database—that is, when they have an identifier value that corresponds to a primary key value in some table. This means that an object becomes persistent when it is saved to the database or when it is created via a load operation on the database. Persistent objects participate in transactions and support rollback functionality.

Finally, when a transaction finishes, the persistent objects from the transaction that remain left over become *detached* from the database. This means Hibernate no longer guarantees their state to be synchronized with the database; the data may be stale. These instances can become persistent again if they are

associated with a new persistence manager.

3.1.2 Metadata

Developers using Hibernate for persistence provide metadata in XML files according to a format specified by Hibernate. The Hibernate metadata format aims for readability and ease of use. It provides many default values and reflects on the application to fill in details that metadata authors omit. The following simple example exhibits the Hibernate metadata format while omitting complexities such as foreign keys and inheritance.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
   PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
   "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
   <class name="org.hibernate.shopping.model.Customer"
      table="Customer">
      <id name="id" column="CustomerId"
         type="long">
         <generator class="native"/>
      </id>
      <property name="name" column="Name" type="string"/>
   </class>
</hibernate-mapping>
```

Unlike some ORM solutions, Hibernate furnishes a Document Type Definition mandating the format of its XML metadata documents. This ensures that the required metadata format is published in an unambiguous, machine-interpretable way.

Every Hibernate mapping document has a root `hibernate-mapping` element. Classes are specified by `class` elements, which are children of the `hibernate-mapping` element. (As a matter of convention, each class usually gets its own metadata document.) Metadata authors specify the qualified name of the Java class and the name of the corresponding table. The `class` element has `property` children, which specify the persistent data of the class and the names of their corresponding database columns. As the example above illustrates, the `id` element is used to mark primary keys.

Some of the attributes in this example can be omitted. For example, `string` is the default `type`, so we could omit that attribute from the `property` element. Also, if no column name is specified, Hibernate identifies the property with the like-named (case notwithstanding) database column, so we could simply write

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

and let the Hibernate defaults handle the rest.

One interesting feature of Hibernate metadata is run-time metadata manipulation. The Hibernate API provides methods that allow applications to change

metadata directly and programmatically. The following code dynamically adds an `address` property to the metadata.

```
PersistentClass mapping = cfg.getClassMapping(Customer.class);

Column column = new Column();
column.setType(Hibernate.STRING);
column.setName("Address");
mapping.getTable().addColumn(column);

SimpleValue value = new SimpleValue();
value.setTable(mapping.getTable());
value.addColumn(column);
value.setType(Hibernate.STRING);

Property property = new Property();
property.setValue(value);
property.setName("Address");
mapping.addProperty(property);

// Build a new session factory that reflects this domain model change.
// After a session factory is created, its mappings cannot be changed.
SessionFactory factory = cfg.buildSessionFactory();
```

Of course, this is considerably more verbose than the equivalent element in an XML metadata file, but the capability is useful for applications that need to change their metadata dynamically.

3.1.3 Querying

Hibernate provides three ways to query: by HQL, by criteria, and by example. (A fourth method is to drop down to raw, DBMS-specific SQL, but this is only necessary in rare cases where the application must leverage some unusual DBMS-specific feature that Hibernate does not support.)

HQL, for Hibernate Query Language, is an SQL-like language that Hibernate defines to allow the application to make queries on its own terms, referring to structures by their application names, not their database names. It is close enough to SQL that developers who know SQL should be able to write HQL with no problem. An application creates an HQL query by calling the `createQuery` method of the session, which returns an instance of the `Query` class.

```
Query query = session.createQuery(hqlQueryString);
```

The simplest sort of HQL query is one that retrieves all records from a given table. This sort of query would be created with code like

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

Of course, HQL supports queries with restriction clauses, like

```
from Customer c where c.firstName = 'Mary'
```

Unlike SQL (but like Java), this cannot be abbreviated to something like

```
from Customer where firstName = 'Mary'
```

HQL also supports the same comparison, string matching, and logical operators as SQL, and it supports ordering result sets. Once it creates a query, the application can execute it and retrieve its results as a `List` by calling its `list()` method.

```
List result = query.list();
```

If the query is guaranteed to return only a single result, we can use the `uniqueResult()` method instead of `list()` to obtain a single object.

HQL does support named parameters; one would seldom actually write a Hibernate query with a condition like `where c.firstName = 'Mary'`. Instead, one would use named parameters as in the following example

```
String queryString =
    "from Customer c where c.firstName = :searchString";
Query query = session.createQuery(queryString)
    .setString("searchString", "Mary");
List result = query.list();
```

(Obviously, *Mary* would likely be, say, user-supplied input, not just a literal in the code as in this example.) In addition to protecting against SQL injection attacks, named parameters can improve performance by allowing the database to precompile the query. It is also possible to use positional parameters instead of named parameters, as in

```
from Customer c where c.firstName = ? and c.creationDate >= ?
```

However, using named parameters makes the code more readable and more resistant to change.

Hibernate queries can be placed in the metadata as named queries—for example,

```
<query name="findRecentCustomersByFirstName"><![CDATA[
    from Customer c
    where c.firstName like :firstName
    and c.creationDate >= :creationDate
]]></query>
```

The application can retrieve these queries by calling the `getNamedQuery()` method of a session.

```
Query query = session
    .getNamedQuery("findRecentCustomersByFirstName")
    .setString("firstName", firstName)
    .setDate("creationDate", creationDate);
```

This is considered better than scattering HQL queries throughout the application code, because it places all the queries in one, convenient place, and it separates the domain-level application concerns (performing particular persistence tasks) from the details of the persistence operation.

The second method of querying in Hibernate is by criteria. To use this method, the application creates a `Criteria` object based on a class and sets constraints on it. The following example illustrates this:

```
Criteria criteria = session.createCriteria(Customer.class);
Criterion namedMary = Expression.eq("firstName", "Mary");
criteria.add(namedMary);
List result = criteria.list();
```

The first step is to create a "root entity" based on the appropriate class, in this case `Customer`. Then we add `Criterion` instances to it; in this case the only criterion is that the first name of the customer be *Mary*. Then, we execute the query by calling `list()` or `uniqueResult()`.

The final method of querying in Hibernate is by example. This is the least powerful method, but it can be convenient for some applications. Here, to query the database, the application exhibits an example of the sort of result it would like to receive. That is, it provides an actual instance of the type it is querying in order to receive results that match that type. The following example also queries for customers with the first name *Mary*:

```
Customer exemplar = new Customer();
examplar.setFirstName("Mary");
Criteria criteria = session.createCriteria(Customer.class);
criteria.add(Example.create(exemplar));
List result = criteria.list();
```

Here again, we use the `Criteria` class.

3.1.4 Caching

Hibernate has a sophisticated, two-level cache architecture. The first-level cache is essentially compulsory and can't be deactivated; it is the Hibernate session. A session has a lifetime that may span only a single database transaction, or it may span several database transactions, effectively forming a coarser, application-level transaction, but it is not generally longer-lived than this. It provides the application with a cache at the scope of a (database or application) transaction. This cache ensures that within the scope of a transaction, an object is unique in referring to a particular database record. In particular, when an application requests the same object twice via a single session, it gets back two references to the same Java object, not two copies of it. This prevents conflicting representations of a single record from arising in the context of a transaction, ensuring that changes made to an object in a session are always and immediately visible to other code that accesses the same record via that session.

The second-level cache can be scoped to the process or cluster level (usually, it has the scope of a Hibernate session factory) and can be configured per class and per association. In particular, not only can it be turned on and off at this level of granularity, but caching policy details can be configured, including concurrency strategy, cache expiration policy, and cache format. These items can be configured in the metadata documents. Persistent objects are not stored intact as whole Java objects in the second-level cache; instead, they are "disassembled" into a compact form that preserves state and returns it by value. The second-level cache should be active only for read-mostly data, not data that is modified frequently by the application.

Hibernate provides four concurrency strategies for the second-level cache, which correspond roughly to the isolation levels described in Section 2.4.1.

Transactional Maintains repeatable-read isolation. This strategy is appropriate when it is important to avoid stale data in case of an update.

Read-write Maintains read-committed isolation. This is appropriate when it is important to avoid stale data in case of an update.

Nonstrict read-write Does not guarantee consistency between the cache and database. This strategy is appropriate if data rarely changes (at least hours or days between changes) and stale data will not cause catastrophic problems.

Read-only This strategy is for use with data that never changes.

3.1.5 Transactions

The `Transaction` interface in Hibernate provides methods to declare the boundaries of a database transaction. An example of the use of the `Transaction` interface appears on page 48, so we will not reproduce it here. The usage is fairly clear: begin a database transaction with `Session.beginTransaction()`, carry out some persistence tasks, and then call `commit()` on the transaction to commit it, or `rollback()` if there is a problem.

Hibernate also provides some functionality to support application transactions, namely managed versioning for optimistic locking. This functionality is described in Section 2.4.2, with examples from Hibernate.

3.2 iBATIS

iBATIS (http://ibatis.apache.org/) is most notable for its approach to querying, although it is otherwise a standard ORM solution cut from the same cloth as Hibernate and other mappers. It consists of two components (which need not necessarily be used in conjunction), the iBATIS Data Mapper and iBATIS Data Access Objects (DAOs). The Data Mapper, also called SQL Maps, is responsible for actually persisting data—shuffling data back and forth between applications

and relational databases. DAOs is an abstraction layer that provides a data access API to the application.

The iBATIS project was born in 2001, initiated by the software developer Clinton Begin. Originally focused on the development of cryptographic software, iBATIS soon changed direction to focus on Internet technologies. Initially iBATIS was developed for Java, but it has since been ported to the Microsoft .NET Framework and to Ruby.

3.2.1 Architecture

The designers of iBATIS do not think of it as a true object-relational mapper, but instead as a data mapper. The term *data mapper* refers to the data mapper pattern described in Fowler's book on enterprise patterns [13, pp. 165–181]. Like an object-relational mapper, a data mapper like iBATIS is a persistence layer that mediates between an application and a database, but iBATIS allows a looser coupling between application structure and database structure than a strict object-relational mapper. (We adopt a broad definition of the term *ORM* in this paper, so we consider iBATIS to be an object-relational mapper for our purposes.) Instead of directly linking classes to tables and properties to columns, iBATIS maps the input to and output from the database to application entities, adding a layer of indirection between the application and the database. This mapping is described by SQL queries provided by the application developer. (Where traditional ORM solutions hide the SQL from the developers, iBATIS places them in full control of it.) Because of this loose coupling, iBATIS accommodates systems in which the application design and the database design may be drastically mismatched, which is especially advantageous for dealing with legacy and shared databases as well as handling databases whose designs change over time.

3.2.2 Metadata

The most important metadata for iBATIS are SQL mapping descriptors, files that contain SQL to realize the relationship between the application and the database. The following example is a simple SQL mapping descriptor file.

```
<select id="getCustomer"
    parameterClass="int"
    resultClass="Customer">
    SELECT
        Id as id,
        Name as name,
        Address as address,
        TelephoneNumber as telephoneNumber
    FROM Customer
    WHERE Id = #id#
</select>
```

Note that this is real, valid SQL that would run against the database (once an integer is substituted for the `#id#` placeholder, obviously), not some non-SQL query language to be parsed by the mapper (like Hibernate Query Language). iBATIS will actually run this SQL against the database in the course of persistence operations. This reverses the traditional ORM idea that SQL should be generated by the mapper and never seen by developers. Here, developer-authored SQL is at the heart of the mapping.

With this metadata in place, it is trivial to retrieve a record from the database:

```
Customer customer = (Customer) sqlMap.queryForObject(
    "getCustomer", new Integer(2));
```

There are other mapped statement types for other types of command. Mapping descriptor files allow not only `<select>` elements, but also `<insert>`, `<update>`, and `<delete>` elements (for `INSERT`, `UPDATE`, and `DELETE` SQL commands, obviously). There is also a `<procedure>` element for calling stored procedures and a seldom used `<statement>` element that supports arbitrary statements.

Also of interest are the `<sql>` and `<include>` elements, which allow mapped statement authors to build complex queries out of small, reusable building blocks of SQL, as the following simple example illustrates

```
<sql id="select-min-price">
    SELECT MIN(Price) AS value FROM Product
</sql>

<sql id="select-max-price">
    SELECT MAX(Price) AS value FROM Product
</sql>

<sql id="where-product-recent">
    <![CDATA[
        WHERE creationDate > #value:DATE#
    ]]>
</sql>

<select id="getMinPriceOfRecentProducts"
    resultClass="java.math.BigDecimal">
    <include refid="select-min-price" />
    <include refid="where-product-recent" />
</select>

<select id="getMaxPriceOfRecentProducts"
    resultClass="java.math.BigDecimal">
    <include refid="select-max-price" />
    <include refid="where-product-recent" />
```

```
</select>
```

In this example, we define three "building blocks" and assemble them in two different ways to obtain complete queries. (Here the gains in maintainability and terseness are less than impressive, but this example should serve to demonstrate the purpose and usage of the `<sql>` and `<include>` elements.)

3.2.3 Querying

Given the SQL-heavy approach that iBATIS takes to automated persistence, it should not be surprising to learn that queries are very important to iBATIS architecturally. As we saw in the previous section, retrievals in iBATIS are defined by `SELECT` SQL queries enclosed in `<select>` elements in mapping descriptor files. As a consequence, though, there is not much to say about the querying capabilities of iBATIS. It suffices to say that iBATIS allows developers to specify and execute essentially any SQL statement that the DBMS will accept.

It is worth making explicit what happens when the application makes a call like

```
Customer customer = (Customer) sqlMap.queryForObject(
    "getCustomer", id);
```

When this line is executed, the iBATIS API looks up the `getCustomer` mapped statement and transforms the inline parameters into prepared statement parameters, producing SQL like

```
SELECT Id AS id, Name AS name, Address AS address,
    TelephoneNumber AS telephoneNumber
FROM Customer WHERE Id = ?
```

iBATIS passes this prepared statement to the JDBC API and sets the value of the parameter appropriately (in this case to the value of the `id` variable that was passed to it). Finally, it executes the prepared statement, maps the resulting row to an object, and returns it.

3.2.4 Caching

We have already seen that iBATIS takes a different approach to persistence than traditional ORM frameworks like Hibernate. Instead of directly mapping tables to classes, iBATIS maps SQL to objects. To accord with its distinctive architecture, iBATIS also takes an exceptional approach to caching. Traditional ORM caches are concerned with the issue of identity. Most systems avoid having the same database record appear twice in the cache (or having two objects in the cache representing the same database record), as this can lead to inconsistency within the cache. (Also, the redundancy can consume memory needlessly.) However, iBATIS challenges this idea, largely ignoring the issue of identity and instead caching results returned by queries regardless of whether identical objects exist elsewhere in the cache.

From the application developer's perspective, enabling caching for iBATIS entails the creation of "cache models" (i.e., cache configurations) that are suited to the purpose at hand. Once a cache model is defined, it may be associated at will with queries in the mapping descriptor files, causing those queries to use the cache model. That is, different queries can use different cache models. A cache model is defined by placing a `<cacheModel>` element in a mapping descriptor file. The `<cacheModel>` element has two required attributes: `id`, which specifies a unique string by which query mapped statements can refer to the cache model, and `type`, which specifies a cache implementation.

There are four cache types built into iBATIS:

`MEMORY` In this mode, the contents of the cache are stored in memory until the garbage collector disposes of them. iBATIS uses *reference objects* to manage garbage collection of the memory cache. In Java, reference objects, described in detail in [35], are objects that effectively give applications a greater degree of control over garbage collection policies. Without reference objects, an object is either (strongly) reachable, meaning that it is accessible via the object graph, or it is unreachable, meaning that it is ready for garbage collection. Reference objects allow for three additional states vis-à-vis garbage collection: softly reachable, weakly reachable, and phantomly reachable. The details of these states are not worth describing here. What is important is that the `MEMORY` cache model allows descriptor file authors to specify `WEAK`, `SOFT`, or `STRONG` reference types, which affects how readily cached objects are discarded. For example, if the descriptor file specifies a `STRONG` cache reference type, the cache will always retain objects until the next flush interval, regardless of memory constraints. Under the `WEAK` type, the cache will discard objects quickly, freeing up memory but requiring more database hits.

`LRU` This model uses a least-recently-used (LRU) management policy for the cache. The cache keeps track of the order in which objects have been most recently used and discards least recently used objects when the cache exceeds its size constraint. (The cache size constraint in iBATIS is expressed in number of objects instead of amount of memory, so an application could run into problems when the cache is filled with large objects.) LRU is a good strategy in situations with certain sets of data that are used for extended periods of time and then neglected.

`FIFO` This model uses a first-in-first-out (FIFO) policy. The cache keeps track of the order in which objects enter the cache. When the cache exceeds its size constraint, it discards old objects. FIFO is a good strategy for situations in which objects are most relevant for brief periods of time soon after they are first used and then become less relevant as time passes.

`OSCACHE` This model uses the third-party product OSCache, a high-performance Java caching framework [31].

It is also possible to create custom cache models [6, §9.5.5].

3.2.5 Transactions

All iBATIS persistence occurs within the scope of a transaction. There is no direct support for the auto-commit mode of JDBC. However, transactions need not be explicitly demarcated; statement executions are automatically wrapped in transactions. Automatic transactions are transparent to iBATIS users but still offer some measure of concurrency protection to the statements they enclose.

Of course, transactions can also be controlled by the application through `SqlMapClient`:

```
// Precondition: We have obtained a SqlMapClient called sqlMapClient.
try {
    sqlMapClient.startTransaction();
    // Query some objects; persist some data.
    sqlMapClient.commitTransaction();
} finally {
    sqlMapClient.endTransaction();
}
```

iBATIS also provides impressive support for "global transactions," transactions which may involve multiple databases and even multiple applications. There is also support for custom transaction handling, either by defining a new transaction manager using the iBATIS interfaces or by assuming control of the JDBC `Connection` object that iBATIS is using.

3.3 TopLink

Oracle TopLink (http://oracle.com/technology/products/ias/toplink/) differs from many of the other ORM solutions discussed in this chapter in that it is proprietary (and expensive—the current price of a production license is $5,000 per CPU, although a full-featured version of TopLink can be freely downloaded for evaluation and development [34]). TopLink was born at an early Smalltalk consulting firm, The Object People (hence the *TOP* in *TopLink*), in the early 1990s. In 1996, the TopLink team rebuilt TopLink from the ground up in Java (instead of Smalltalk). In 2000, TopLink was acquired by WebGain. TopLink has been an Oracle product since Oracle Corporation acquired it in 2002 [39].

In addition to the core TopLink ORM product, Oracle offers TopLink Essentials, an open-source reference implementation of the Java Persistence API intended to compete directly with open-source offerings such as Hibernate, and TopLink Object-XML, a product for object-XML mapping. We will not discuss either of these products further; TopLink Essentials is essentially a watered-down version of TopLink, and object-XML mapping is, strictly speaking, beyond the scope of this paper.

3.3.1 Architecture

TopLink comprises a session front end and a data access back end, which use mapping, querying, caching, and transaction components. Client applications access TopLink—in particular, the query framework and the transaction functionality—via the session. The query and transaction components take advantage of the cache to minimize communication with the data source. The data access component on the back end accesses the data source using JDBC.

One feature of TopLink is that it accommodates a range of architectural decision at the application level. That is, it is designed to work with many application architectures. For example, the TopLink developer's guide [33] describes support for application architectures including the following:

Three-tier Probably the most common architecture for TopLink projects is a three-tier architecture, in which clients connect to an application server, which in turn connects to a database. A typical example is a Web application, in which many end users interface remotely with a Web server that uses a database for persistence. In a three-tier architecture, TopLink runs on the application server.

EJB session bean façade This is an extension of the three-tier architecture, with Enterprise JavaBeans session beans wrapping access to the application tier. In this architecture, TopLink shares a server session between session beans. When a session bean needs a TopLink session, it obtains a client session from the shared server session. Transactions are thus encapsulated in session beans.

EJB 3.0 with JPA The EJB 3.0 specification includes the Java Persistence API, a standardized framework for Java persistence, which I describe briefly in Section 4.2.2. TopLink provides supports developing applications thorugh the JPA.

EJB entity beans with CMP Container-Managed Persistence is part of the J2EE component model and provides an object persistence service for EJB containers. TopLink CMP extends the TopLink framework to integrate EJB containers of application servers.

EJB entity beans with BMP Bean-Managed Persistence is also part of the J2EE component model, providing persistence for beans. TopLink BMP extends the TopLink framework to combine the advantages of TopLink with BMP.

Web services A Web services application is similar to a three-tier application except that it encapsulates business logic in a Web service. Clients communicate with the application using SOAP (a protocol for exchanging XML messages over networks). In the Web services architecture, TopLink can map the object model to an XML schema for use with the Web service.

Two-tier Conceptually, the two-tier architecture is the simplest, but it is seldom used due to scalability limitations. In the two-tier model, clients access the database directly. Consequently, each client application requires its own session.

3.3.2 Metadata

TopLink provides graphical mapping tools, called the TopLink Workbench, that allow developers to establish object-relational mappings without writing metadata documents or code to reference a persistence API. This carries a number of advantages, including tempering the learning curve. Nonetheless, some developers prefer the control that hand coding affords, so TopLink does allow developers to bypass the graphical interface. We are more concerned with the underlying mapping metadata documents, not because of any qualms with the idea of graphical ORM tools (nor with these particular tools), but because our purpose at the moment is not to achieve an object-relational mapping, but to examine the architecture and structure of TopLink as a solution to the ORM problem.

The most important TopLink metadata file is called (by default) project.xml and describes all the mappings for the application, as well as named queries. The format of the metadata file is not especially important, since few developers ever see it. (The great majority of developers who use TopLink use the graphical interface to manage the metadata. Many of those who do not generate it programmatically. Editing project.xml manually is generally frowned upon, because it compromises the synchronization of the file with the classes and the data source, but it can be done.) However, it is worth pointing out that TopLink, too, uses XML for metadata; the metadata format is defined by an XML Schema Definition. Other XML documents contain other persistence-related metadata such as metadata relating to sessions.

The TopLink metadata architecture emhasizes the independence of the object model and database schema from the metadata, allowing metadata to adapt to the object model and database schema rather than vice versa. TopLink Workbench is flexible, supporting several approaches to establishing an ORM. It allows:

- Importing classes and tables and establishing a mapping between them
- Importing classes and generating tables and mappings
- Importing tables and generating classes and mappings
- Defining and generating both classes and mappings

This effectively gives TopLink the flexibility to operate in any of the three mapping paradigms we described in Section 2.1. However, it is most common for the application developers to create both the classes and the tables and then use TopLink Workbench only to establish a mapping between them.

At the core of the TopLink metadata architecture are mappings. A mapping associates a single data member of a domain object with its data source representation and defines the conversion between the two. TopLink supports four kinds of mappings, including XML mappings and EIS mappings (for non-relational enterprise information systems), but we are primarily interested in relational mappings. There are many types of relational mapping, including:

Direct to field Direct-to-field mappings map a primitive data members to database fields.

One to one One-to-one mappings represent a single reference between two objects.

One to many One-to-many mappings represent the relationship between a source object and a collection of target objects that it references.

Many to many Many-to-many mapping represents the relationships between a collection of source objects and a collection of target objects.

There are seven other types of relational mapping, including aggregate collection mappings, direct collection mappings, direct map mappings, and aggregate object mappings, but further detail would risk belaboring the point.

3.3.3 Querying

TopLink supports nine types of queries. The four most basic are:

Session queries A session query is constructed and executed by a session object and is capable of performing the most basic persistence actions on an object.

Database queries A database query is an isntance of class `DatabaseQuery` constructed exclusively to represent a query. It can perform any persistence action on objects or data.

Named queries A named query is an instance of `DatabaseQuery` stored by name in a session object or descriptor. It is constructed and prepared once and can be executed many times.

Call queries Call queries are used to execute custom SQL and stored procedures.

Developers can build queries through the TopLink Workbench or the TopLink API. Here, we describe the query API. (The TopLink Workbench does not support all types of queries. For example, it does not support session or database queries.)

Session objects support simple query operations, namely reading, writing, or deleting one or multiple objects. For example, to find all airplanes with a wingspan greater than 30, one could write

```
Vector results = session.readAllObjects(Airplane.class,
    new ExpressionBuilder.get("wingspan").greaterThan(30));
```

To create a new record, one could write

```
Airplane a = new Airplane();
a.setWingspan(30);
// Initialize other instance variables.
session.insertObject(a);
```

Database queries are more flexible but also more verbose. For example, using a database query to find all airplanes with a wingspan greater than 30 would require several lines of code.

```
ReadAllQuery query = new ReadAllQuery(Airplane.class);
ExpressionBuilder builder = query.getExpressionBuilder();
query.setSelectionCriteria(builder.get("wingspan").greaterThan(30));
Vector results = (Vector) session.executeQuery(query);
```

However, database queries are necessary for more complex operations, such as querying by example. In the following example, we find all cities in Jefferson County, New York.

```
City city = new City();
city.setCountyName("Jefferson");
city.setState("New York");
ReadObjectQuery query = new ReadAllQuery(City.class);
query.setExampleObject(city);
Vector results = (Vector) session.executeQuery(query);
```

TopLink's query-by-example system is much more flexible than most. For example, it accommodates comparisons, string operations, and even keyword searches. The following query by example finds all airplanes with a wingspan over 30.

```
Airplane airplane = new Airplane();
airplane.setWingspan(30);
ReadObjectQuery query = new ReadAllQuery(Airplane.class);
query.setExampleObject(airplane);

QueryByExamplePolicy policy = new QueryByExamplePolicy();
policy.addSpecialOperation(Integer.class, "greaterThan");
query.setQueryByExamplePolicy(policy);

Vector results = (Vector) session.executeQuery(query);
```

Of course, database queries have many features and options beyond the scope of the present paper.

Named queries are like database queries except that they are associated with a name and prepared in advance. The simplest syntax is

```
result = session.executeQuery("findLargeAirplanes");
```

Named queries can also accept arguments.

Finally, call queries are the most flexible means of querying. They allow developers to run SQL statements against the database directly. To use a call query, a developer creates an `SQLCall`, `StoredProcedureCall`, or `StoredFunctionCall` object and passes it to the session.

3.3.4 Caching

In TopLink, there are two types of cache: session caches and unit-of-work caches. The former keep track of objects retrieved from and written to the database; the latter store objects participating in transactions. A unit-of-work cache is not independent of the session cache. In the typical pattern, database reads will enter the session cache. When the application registers an object with a unit of work, that object enters the unit-of-work cache from the session cache. When the application commits the transaction, TopLink writes data from the unit-of-work cache to the database and the session cache.

Caching brings up a number of issues, one of which is object identity. TopLink preserves object identity in the cache by comparing the primary keys of retrieved and saved records. There are several choices for managing identity, which can be configured on a class-by-class basis.

Full identity map This choice provides full caching. Objects remain in memory until deletion. This becomes problematically memory-intensive for batch operations, which read many rows.

Weak identity map This option is like the full identity map, except the map holds weak references to cached objects, allowing garbage collection but still guaranteeing identity.

Soft-and-hard–cache weak identity map This option is like the weak identity map, except with a subcache that contains the most frequently used items. The subcache manages soft and hard references to the items to ensure that they are garbage-collected only if the system is low on memory.

No identity map This choice turns off caching. The system does not preserve object identity.

Another problem of caching is stale data. The primary mechanism TopLink provides to circumvent this problem is locking. By locking a cached object, processes can momentarily limit the ability of competing processes to read or write to the object.

A feature of TopLink is cache invalidation, which allows application developers to specify when cache entries should become invalid. By default, items remain in the cache until they are explicitly deleted, but developers can specify invalidation policies. For example, one can specify that an object should be flagged invalid at a particular time of day, or that an object should become invalid when some length of time has elapsed since the last read. Developers can configure invalidation policies at the project, class, or query level.

3.3.5 Transactions

Transactions are represented by `UnitOfWork` objects in TopLink. A unit of work can be acquired by calling the `acquireUnitOfWork()` method of a session object. The unit of work remains valid until and only until its `commit()` or `release()` method is called. Internally, the unit of work makes changes to clones of objects in a cache internal to the unit of work. Upon a successful commit, the changes to the clone are integrated into the database (and the session cache). `UnitOfWork` derives from `Session` and so supports the same methods as `Session`.

TopLink supports optimistic and pessimistic locking. With optimistic locking, all processes have read access to data. When a process attempts to effect a change, the persistence layer verifies that the data has not changed since that process last read the data. With pessimistic locking, the first user who accesses the data to update it locks it until the update is finished.

One exceptional feature that TopLink supports is external transaction control. In an application architecture with many clients connecting to an application server, the application server can provide a transaction service that manages transactions globally.

3.4 Neo

Not suprisingly, the ORM playing field in the .NET world is not as mature as that in the Java world. Indeed, many of the most popular ORM solutions for the .NET Framework are ports of Java solution, such as NHibernate (the .NET port of Hibernate) and iBATIS.NET. Nonetheless, many ORM solutions have been developed exclusively for .NET, and some of them are widespread enough and of sufficiently compelling theoretical interest that they merit discussion here.

One such solution is Neo (an acronym for *.NET Entity Objects*), an open-source tool developed by the software consultant Erik Dörnenburg of Thoughtworks (http://neo.codehaus.org/). Neo diverges dramatically from the classical ORM solutions covered so far, and language is only a relatively superficial difference. One of the most important differences is that Neo manages the domain model. In Hibernate and TopLink, application developers create plain objects to suit the domain; then, the persistence layer takes these objects and operates on them to persist their state. In Neo, software architects define the domain model in an abstract format (in an XML *model file*), and Neo generates the classes. This approach carries both advantages and disadvantages. On the one hand, it seems to deprive developers of a degree of freedom, preventing them from enlivening domain objects with structure and behavior that cannot be expressed via the metadata. However, this problem is largely invalidated by the fact that developers can create subclasses of these Neo-generated base classes with whatever additional functionality they want. The application can then treat these hand-crafted derived classes as the domain objects, ignoring the automatically generated base classes that underlie them. Of course, problems can occur when architects change the metadata in a way that introduces breaking

changes into the generated base classes, but these breaks can almost always be detected by the compiler rather than in run-time testing. The great advantage of the Neo approach is that it solves problems such as maintaining consistency in object graphs. To clarify this, consider a domain in which each customer may have a number of orders, each of which is uniquely associated to that single customer. For convenience, we want to be able to ask of each customer object what its orders are, and we want to be able to ask of each order object who its customer is. That is, the customer-order link in the object graph should be bidirectionally navigable. Now, suppose we have a customer `c` and we have created a new order `o` which we want to associate to `c`. In a hand-coded system, we would likely have to write code like the following to associate the two:

```
o.Customer = c;
c.Orders.Add(o);
```

If a developer carelessly omitted either of these lines (or perhaps worse, mistyped a line so that exactly one of the properties held an incorrect value), the link between the objects would become asymmetrical, likely precipitating hard-to-predict and hard-to-explain failures at run time. There is no easy way to solve this. One could add logic to the classes to ensure that the two directions of the relationship remain symmetrical, but this would require the customer and order classes to know a lot about one another, compromising encapsulation. (This is an abstract problem, but one with real consequences.) If the order and customer classes have been generated by Neo, though, one can simply write

```
c.Orders.Add(o);
```

and the generated code will ensure that that `Customer` property of `o` remains correct. Of course, the generated customer and order classes will know a lot about one another, but since developers (ideally) never need to read, let alone maintain, the generated code, this is not a problem.

3.4.1 Architecture

Figure 3.1 depicts the architecture of a typical Neo project. An `ObjectContext` object is a transactional package that may essentially encompass the entire Neo system (although it is certainly possible to have operate multiple `ObjectContext` simultaneously, provided that they have independent data sources). The `EntityObject` class is the base class for all persistent objects in the domain. This class is provided by the Neo API and includes basic, abstract persistence functionality. Each `EntityObject` corresponds to a record in the database, and in fact is associated with an ADO.NET `DataRow`, a .NET object that represents a row of data in a relational database. The `ObjectContext` object, incidentally, is explicitly associated with an ADO.NET `DataSet`, which is a collection of `DataTable`s.

The classes that Neo generates to accommodate domain objects all derive from `EntityObject`, adding a layer of specificity to refine the abstract functionality provided by the `EntityObject` class. In the figure, these classes are called

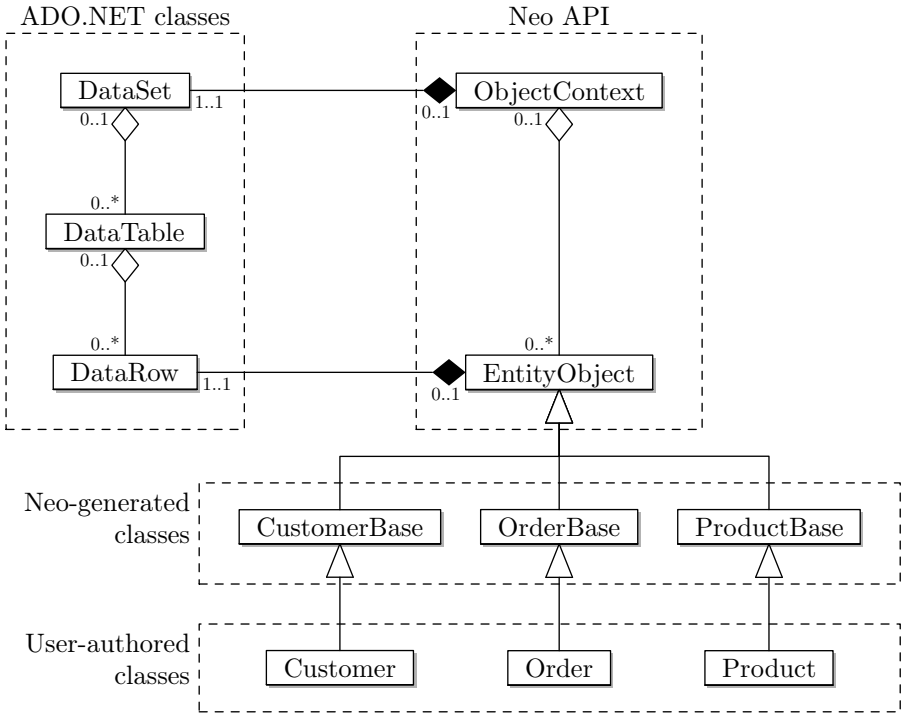

Figure 3.1: The architecture of a typical Neo project. Adapted from Dörnenburg [12], p. 2.

`CustomerBase` and `OrderBase`. Typically, application developers create one subclass for each of these generated classes, so that they can add their own functionality to that provided automatically. In the figure, these are called `Customer` and `Order`.

To put it more concisely, the classes that the application uses inherit from tool-generated base classes, which in turn inherit from a single `EntityObject` class, which uses ADO.NET for persistence, and all these `EntityObject` objects are collected into one `ObjectContext` object corresponding to an ADO.NET `DataSet`.

It is actually the `ObjectContext` itself that handles persistence to the database; the methods provided by the `EntityObject` class and its subclasses are façades. The `ObjectContext` does not persist data changes to the database as they happen; instead, it tracks changes via the ADO.NET architecture that backs it (i.e., via its associated `DataSet`) and provides methods for synchronizing with the database explicitly (`ObjectContext.SaveChanges` and `ObjectContext.GetChanges`). In fact, as we will note in Section 3.4.4, `ObjectContexts` can even work in disconnected mode, without a database, using the `DataSet` alone for temporary data storage and manipulation.

Domain objects are created in a specific `ObjectContext` by factories associated to that context. For example, to create new customer objects in an `ObjectContext` called `oc`, one would write

```
f = new CustomerFactory(oc);
```

3.4.2 Metadata

In the classification of ORM solutions given in Section 2.1, Neo is a metadata-oriented mapper. This lends metadata especial importance for Neo, relative to the mappers we have studied so far in this chapter. Nonetheless, the metadata documents themselves do not look so different from those in other systems. The metadata format of Neo borrows heavily from Apache Torque, an earlier (and still extant—http://db.apache.org/torque/) ORM tool for Java, although it is relatively easy to extend Neo to accommodate other metadata formats.

In Neo, a single XML file represents the *domain model* of the system. Neo uses this model to generate code (and a database schema); it is not part of the compiled system and is not used at run time. The format for the domain model is based on the entity-relationship paradigm (see Section 1.2.2); entities are defined in terms of their attributive structure, and relationships are defined between them. In the database, entities are tables and relationships are represented via foreign keys. In the application, classes represent entities and properties represent attributes (if they are value types) or relationships (if they are references to other classes). The following XML file is a Neo domain model.

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE database SYSTEM "norque.dtd">
<?neo path="..\..\build"?>
<database name="Customers" package="Example.Shopping"
    defaultIdMethod="native">
    <table name="customer">
        <column name="customer_id" primaryKey="true"
            hidden="true" type="INTEGER"/>
        <column name="name" required="true"
            type="VARCHAR" size="50"/>
        <iforeign-key foreignTable="Order" name="Orders"
            onDelete="cascade">
            <ireference local="order_id" foreign="CustomerId"/>
        </iforeign-key>
    </table>
    <table name="order">
        <column name="order_id" primaryKey="true"
            hidden="true" type="INTEGER"/>
        <!-- Other order attributes -->
    </table>
</database>
```

This short example avoids complexities such as inheritance, revealing only the basics of the domain model syntax. Note that enough information is available to generate both the code and the database schema. For example, the domain model author specifies the lengths of string fields in the database.

The doctype specified here, norque.dtd, is for Norque, the Neo adaptation of the Torque format, as described above. The `<?neo` processing directive directs Neo to the template files. The root `database` element indicates the namespace in which the generated classes should be placed, in this case Example.Shopping, and the method used to create primary keys ("native" means that an identity column will be used). The `database` element has several `table` children, each of which represents an entity. The `name` attribute of the `table` and `column` entities defines the names of the objects in the database. In this case, the names of the constructs in the generated code are determined automatically by converting underscore-delimited names like `customer_id` to camel-case names like `CustomerId`. (Metadata authors can override this convention by explicitly specifying a `javaName` for the table or column.) The rest of the metadata is largely self-explanatory, except for the `hidden="true"` attribute on the identifier columns. This excludes the identifier columns from the generated code and hence the object model; hidden columns are not of interest to the application, but are instead of use only in persistence.

Once the metadata file is constructed, the next step is to use it to generate a database schema. To do this, the metadata author runs the neo command-line tool against the metadata document with the appropriate parameters. This generates a data definition script that, when run against the database, creates the desired objects.

Next, Neo can generate the code files. Usually, developers use a Visual Studio plug-in to generate the classes, but it is also possible to use the command line. For each `table` in the metadata, Neo generates two classes: a base class, which developers are not to modify (because it will overwritten in subsequent generations) and a concrete subclass into which developers may insert custom functionality, as discussed above. The concrete subclass, which is essentially empty initially, is never overwritten; it is generated only the first time Neo encounters a new `table`, not on subsequent runs. In addition, Neo generates several other utility classes such as factories. Usually, the concrete subclass is the only class the application developers ever look at, let alone modify. The base classes and the utility classes are all stored together in a single, very large code file.

3.4.3 Querying

Queries are generally made through factories. The simplest query is to fetch all objects of a given type, which is accomplished by calling the `FindAllObjects()` method of the relevant factory. There is also a `Find()` method that takes parameters to perform a more refined query.

Internally, the `Find()` and `FindAllObjects()` methods create a `FetchSpecification` tailored to the requested operation and pass it to an overload of the

`Find()` method that accepts a `FetchSpecification` as its single parameter. To construct complex queries, developers can create `FetchSpecification` objects directly.

Creating fetch specifications explicitly can also benefit performance. For example, suppose we want to find information on all cities, including their mayors. (Mayors are stored in a separate `PoliticalFigure` table to which `City` has a foreign key relation.) The most straightforward approach is to call the `FindAllObjects()` method of the city factory and iterate over the result set, relying on lazy fetching to retrieve information on the mayors. However, this causes a serious performance problem, as it requires one SQL query to retrieve each mayor. It would be better to retrieve all this information at once, with a single join. Fortunately, the `FetchSpecification` class provides a `Spans` property, which specifies related objects to fetch. In this example, we could write

```
FetchSpecification spec = new FetchSpecification();
spec.Spans = new string[] {"Mayor"};
CityList results = cityFactory.Find(spec);
```

Now, instead of many SQL statements, only two are executed, one to fetch the cities and one to fetch the mayors.

3.4.4 Caching

Caching is intrinsic to the architecture of Neo. Rather than being an afterthought added for performance, the cache (although it is not called a cache in practice) takes center stage in all data manipulation carried out via the persistence mechanism. Recall from Section 3.4.1 that persistent (generated) classes use ADO.NET data tables for all data manipulation—selections, insertions, updates, and deletions. ADO.NET data tables are data structures (in memory) that replicate much of the functionality of DBMS relations, allowing for the storage, retrieval, and organization of data in an RDBMS-like way, without the expense of a database connection. For Neo, this is essentially a powerful cache.

Of course, data must be saved to and retrieved from the database eventually. For this, Neo provides explicit synchronization commands, most importantly the `SaveChanges` method of the `ObjectContext` object. Data is persisted to the permanent store only when such explicit synchronization commands are given. In the absence of explicit persistence, the data remains in memory.

The cache-focused architecture of Neo opens up an interesting (albeit applicable only in special situations) possibility: treating memory as the primary memory store of an application, and going to the persistent store only when necessary—perhaps not even at all. Indeed, the ability to operate in "disconnected mode" is a prized feature of Neo. It may not be immediately obvious why this is desirable. In fact, there are many contexts in which it is impractical to persist changes frequently. For example, consider a mobile device that runs continuously but is in only sporadic contact with the remote database server to which it saves data. Unlike ORM tools that rely heavily on the querying

capabilities of the database server itself, Neo would be well suited to such a situation. The mobile device could use the disconnected mode of Neo to carry out all its normal database operations—loading, saving, querying—as if it were actually connected to the database server, but in reality these changes would be occurring in memory, using the ADO.NET architecture. It could persist its changes to the database server whenever it is able to make contact with it. (Of course, for this plan to work, the mobile device had better have a considerable amount of memory.)

When would it be useful for Neo not to persist its data at all, not even infrequently—to work in disconnected mode 100 percent of the time? Surely this defeats the purpose of ORM, which is to achieve long-term persistence of data. However, there are many situations in which applications may want to have the power of database queries and temporary, relational-style storage in which it is unnecessary to persist data across sessions and impractical to use an actual DBMS. Good examples are consumer applications that run locally on consumer machines, which seldom run DBMSs and often lack the resources to support them. Nonetheless, consumer applications may want transparent, relational-style querying for certain types of data. Neo would be perfect for such an application.

The cache-heavy architecture of Neo would seem to carry a heavy memory cost alongside the improved time performance that it yields. This is true to a certain extent, but there are mitigating factors. For example, there is little duplication of data. Persistent do not keep their own copies of their member data as variables; instead, they provide accessors that reference the underlying ADO.NET data tables. Thus, persistent data is stored only once, in the data set, not in every persistent object or collection in which it occurs.

A disadvantage of this architecture is that it does not play nice with other users of the permanent data store. Legacy or other applications that must share the database with Neo will have a very hard time defending their data from being overwritten inadvertently by Neo, let alone knowing whether the data they use is up to date. Developers who use Neo and expect to share their data store with another application must be extremely careful to avoid problems. Neo is unlikely to be the best tool for the job if the job involves multiple simultaneous database users.

3.4.5 Transactions

You will find little mention of either caching or transactions in the Neo documentation, but their absences are for entirely different reasons. Caching is not discussed as such because what I call the cache is such an integral part of the Neo architecture that it is no longer merely a volatile data repository whose purpose is to aid performance, but instead the most important data store in the architecture, more important even then the database itself. As such, the Neo documentation seldom refers to these ADO.NET data sets as caches. By contrast, transactions go unmentioned because they are genuinely unimportant to the Neo model.

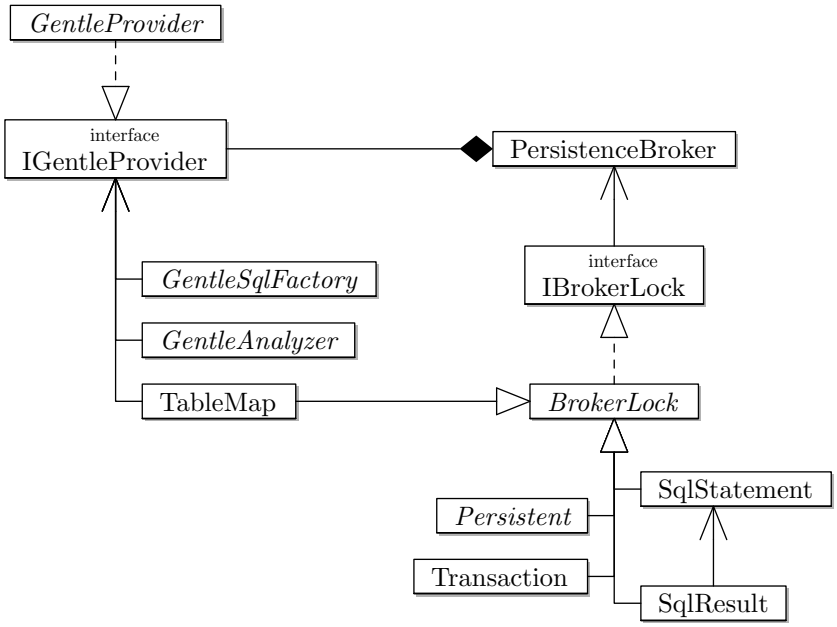

Figure 3.2: An architectural diagram of the most important classes and interfaces of the Gentle API. The abstract classes `GentleProvider`, `GentleSqlFactory`, and `GentleAnalyzer` concretize on a per-DBMS basis, yielding inheritors like `OracleProvider` and `SQLServerAnalyzer`. Liu [20] presents an alternative depiction of the Gentle architecture.

From the point of view of a developer authoring an application that uses Neo, the concept of a transaction is more or less absent.

3.5 Gentle

Gentle (http://www.mertner.com/confluence/x/Ag) is another persistence tool designed specifically for the .NET Framework. It exhibits several important architectural differences from the mappers we have discussed so far.

3.5.1 Architecture

At the heart of Gentle is the `PersistenceBroker`, which manages all access to a particular data source. (Applications that need to connect to only a single database may use the static `Broker` class instead of `PersistenceBroker`.) To carry out a persistence task on an object to be persisted, an application developer passes the object to the appropriate method of the `PersistenceBroker`, such as `Insert` or `Update`. Data retrievals also occur via the `PersistenceBroker`,

namely via methods such as `Retrieve`.

Each persistence broker is uniquely associated to one *provider*, which implements the `IGentleProvider` interface. A provider provides services associated with a specific DBMS. Gentle comes with several providers, such as `SQLServerProvider` and `OracleProvider`, which all inherit from a common abstract class `GentleProvider`. A provider relies on a number of helper classes to assist it in providing services. One such class is `GentleSqlFactory`, which generates DBMS-appropriate SQL for the factory. (Actually `GentleSqlFactory` is an abstract class with subclasses `SQLServerFactory`, `OracleFactory`, etc.) Another is `GentleAnalyzer` (once again an abstract class concretized on a per-DBMS basis), which obtains metadata by directly examining the structure of the database, obviating the need to replicate this information in the code. A final example is `TableMap` (a concrete class), a utility class for maintaining metadata. There is a `ProviderFactory` class for dynamically instantiating providers.

A `PersistenceBroker` may have a number of `BrokerLock`s attached to it. `BrokerLock` is an abstract base class for all classes that are to be locked to a `PersistenceBroker` instance. The subclasses are quite varied. In fact, we have already met one: `TableMap`. (A `TableMap` may be associated with either a provider or a broker.) The most important subclasses of `BrokerLock` are `Persistent`, `Transaction`, `SqlStatement`, and `SqlResult`. `Persistent` is a base class from which persistent classes in the application may optionally derive; it provides some useful persistence-related logic and interfaces. `Transaction` provides transaction protection and will be discussed in Section 3.5.5. `SqlStatement` models a database-specific SQL statement, and `SqlResult` holds query results.

3.5.2 Metadata

Unlike all the mappers we have discussed so far, which use XML mapping documents to describe application-specific metadata, Gentle uses attribute-based programming to record metadata. This programming language feature was discussed in detail in Section 2.6. Briefly, though, an attribute is a language construct that allows developers to impart metadata to classes and other types via a purpose-designed syntax. In Gentle, instead of storing the metadata in an isolated document in a different language from either the code or the database queries, developers insert the metadata directly into the code. The following customer class is tagged with metadata so that Gentle can persist it.

```
[TableName("Customer")]
public class Customer
{
    private int id;
    private string name;

    [TableColumn, PrimaryKey]
    public int Id
```

```
    {
        get { return id; }
        set { id = value; }
    }

    [TableColumn(NotNull=true)]
    public string Name
    {
        get { return name; }
        set { name = value; }
    }

    // Methods (including constructors) would go here.
}
```

To be clear, the bracketed attributes are not preprocessor directives to be parsed and stripped before the code is compiled, nor are they some sort of add-on to the ordinary syntax; they are a bona fide feature of the C# language.

All classes that are to be persisted via Gentle must be tagged with the `TableName` attribute, which designates the database table to which the class corresponds. (Gentle presumes a one-to-one class-to-table mapping.) The properties of the class that are to be persisted must be tagged with the `TableColumn` attribute, which has a number of properties that can be set to provide more specific metadata. For example, the `NotNull` property illustrated in the attribute of the `Name` property above indicates whether null values are permissible. There is also a `Name` property that indicates the name of the corresponding column in the database (by default, the column name matches the property name in the class), a `Size` property that indicates the length of the database field, an `IsReadOnly` property for properties that should not be set on insert and update (intended for use on columns set by the database itself), and several others. The code sample above also illustrates the `PrimaryKey` attribute; there is also a `ForeignKey` attribute and many others.

For application developers, attribute-based metadata has the advantage all the information about a class is contained in a single place: in the code of the class itself. This can make it easier for developers to understand the relationship between the application and the database. Of course, this can be a disadvantage too; it can be nice to separate the structure and behavior of a class from information about the way it persists its state, since the persistence mechanism is largely irrelevant to the way a class conducts its business.

Relatedly, attribute-based metadata destroys the transparency of an ORM solution. No longer can arbitrary, unadorned classes be persisted to a database at will; instead, classes must be decorated appropriately with attributes in order to become persistent. To put it differently, classes need to *know* they are persistent in order for Gentle to accommodate them, while in transparent solutions such as Hibernate classes can be ignorant of this, concerning themselves only with their domain behavior.

Optionally, persistent classes can inherit from the `Persistent` class or implement the `IPersistent` interface, which provide standard methods for persisting and retrieving objects. However, this compromises the transparency of the solution even further, forcing application architects to decide whether to trade off transparency for functionality or vice versa. Inheriting from the `Persistent` class provides the greatest functionality, but is especially invasive because it precludes persistent classes from inheriting from other classes that might otherwise be appropriate. Implementing the `IPersistent` interface is less invasive, because .NET supports multiple interface implementation, but still compromises transparency.

3.5.3 Querying

As described in Section 3.5.1, a `PersistenceBroker` manages all access to a data source; it often serves as a single point of access for the application to persist its objects. (This is not always the case. For example, persistent objects that implement `IPersistent` can be asked to persist themselves to a database. However, under the hood the `IPersistent` object just calls the `PersistenceBroker` to persist it.) Among the tasks that the `PersistenceBroker` administers is data retrieval, including querying. For example, to query for a single object of a given type, one uses the `PersistenceBroker.RetrieveInstance` method, whose signature is

```
public object RetrieveInstance(Type type, Key key);
```

The caller specifies the type of object to retrieve and the value of its primary key, and `RetrieveInstance` returns the requested object. `RetrieveInstance` throws an exception if no such object exists, a `TryRetrieveInstance` method with the same signature is available in case the caller is not sure that the object exists. For retrieving a list of objects, there is a `RetrieveList` method

```
public IList RetrieveList(Type type, Key key);
```

Here `key` is not (necessarily) a primary key, but simply a list of constraints. (The `Key` class, which is defined by the Gentle API, represents a hash table whose keys are column names—not, incidentally, property names—and whose values impose equality constraints on the fields.) By omitting the `key` argument the caller obtains a list of all members of a given persistent type.

This approach to querying is relatively simple, but it works only for the simplest queries—namely, those whose only constraints are equality constraints. It cannot support queries such as "Find everyone born after 1980" or "Find everyone whose name starts with *B*." The Gentle API provides a `SqlStatement` class to accommodate more complex queries. In general, though, application developers do not directly state queries in SQL or another query language; instead, they construct `SqlStatement`s using the `SqlBuilder` class. A `SqlBuilder` may be associated with a provider or a persistence broker. One of its constructors has the signature

```
public SqlBuilder(PersistenceBroker broker, StatementType stmtType,
    Type type, LogicalOperator logicalOperator);
```

`StatementType` is an enumeration with members including `Select`, `Count`, `Insert`, `Update`, and so on. The `type` argument indicates the class (and hence table) being queried. `LogicalOperator` is an enumeration with members `And` and `Or`, which selects the logical operator to be used with the `SqlBuilder`'s constraints.

After a `SqlBuilder` is created, a SQL statement can be assembled. `SqlBuilder` includes many methods for constructing complex queries, but the most important is the `AddConstraint()` method, one of whose overloads has the signature

```
public void AddConstraint(Operator op, string field, object value);
```

This adds a constraint on the field specified by the `field` parameter (which may be the name of either a property or a column). The `op` parameter indicates the type of constraint (`Equals`, `LessThan`, etc.). The `value` parameter indicates the value to which the field will be compared.

Finally, after the statement is assembled by adding constraints and using the other construction facilities that `SqlBuilder` provides, the user asks the `SqlBuilder` to convert itself into an `SqlStatement`, which may then be executed. The following example queries a `Person` table to find each person who was born after after 1980 or whose name starts with *B*.

```
// Assume we are using a PersistenceBroker called persistenceBroker.
SqlBuilder builder = new SqlBuilder(persistenceBroker,
    StatementType.Select, typeof(Person), LogicalOperator.Or);
builder.AddConstraint(Operator.GreaterThan, "BirthYear", 1980);
builder.AddConstraint(Operator.Like, "Name", "B%");
SqlStatement statement = builder.GetStatement();
persistenceBroker.Execute(statement);
```

Even `SqlBuilder` has its limits. It is not able to express every query that an application developer might want to execute. Consequently, developers sometimes have to express constraints or even entire queries directly in SQL. Gentle provides API functions that accept SQL statements, such as an overload of the `SqlBuilder.AddConstraint` method that takes a single string, an SQL constraint clause. (This can be used for complex constraints such as subselects that `SqlBuilder` does not otherwise accommodate.)

The Gentle querying system has a number of weaknesses. First, it is verbose. The snippet of code above could be expressed in SQL very succinctly:

```
SELECT * FROM Person
WHERE BirthYear > 1980 AND Name LIKE 'B%'
```

With Gentle, assembling this query requires the creation of a `SqlBuilder` instance associated with the correct persistence broker and type; the population of the `SqlBuilder` with constraints, one at a time; the conversion of the `SqlBuilder` to

an `SqlStatement`; and the execution of the `SqlStatement`. For complex queries with sophisticated constraints, this can become tedious.

Gentle querying also suffers from a transparency problem. Several methods in the Gentle API require table and column names from the database—not class and property names. This means that developers must think about not only the application structure, but also the database architecture, in order to use the API. In a way, the attributive metadata placement mitigates this problem, because it makes it trivial for developers to find out the name of the database object that corresponds to an application abstraction; they just have to look at its attributes. However, it does force developers to think about the underlying database, when really they would prefer to think about the application objects it persists. A more substantial problem is that it is not always easy to tell, without looking at the documentation, whether a given API function expects a string naming a class or property or one naming a database object, or whether it will accept either. This makes life problematic for developers.

Despite these problems, Gentle handles simple queries relatively elegantly.

3.5.4 Caching

Gentle provides transparent caching to improve performance (although some of its caching capability is turned off by default). Gentle caches three categories of information: metadata, statements, and objects. Metadata is always cached, but statement caching and object caching are configurable.

Statement caching is active by default, unless turned off in the configuration file. Once a SQL statement is created by the `SqlBuilder` class, Gentle caches it for future use. Besides yielding a performance improvement in itself, this also allows Gentle to use prepared queries, storying a query plan in the database back end. This can result in a substantial performance gain, equivalent to the performance gain that results from using stored procedures instead of ad hoc SQL.

Object caching, on the other hand, is turned off by default and must be switched on manually in the configuration file in order to become active. When object caching is turned on, objects are cached as soon as they are retrieved from the database (in fact, as they are constructed). Each cache entry is identified by type and primary key. This caching policy does not result in duplication of data; caching an object does not copy it to a new location, but instead merely keeps a reference to it. This not only saves memory, but also ensures that the cached version of an object will reflect changes made by the application to the object itself.

When both statement caching and object caching are enabled in the configuration file, Gentle can also be instructed to use query skipping. When this option is enabled, Gentle caches associations between executed queries and their result sets. When a previously executed query is executed again, Gentle reconstructs the result using the cached data (as long as it remains available). This bypasses the database entirely, resulting in substantial gains.

A feature closely related to the cache is "object uniqing," which ensures that there are not multiple objects in memory representing the same database row. This feature uses the object cache, so object caching must be enabled for uniqing to work. (In fact, object caching affords no performance benefit on its own; it is useful only in conjunction with query skipping or object uniqing.) The most important function of object uniqing is to prevent data inconsistency, but of course it also reduces the memory footprint of the application.

A final configuration setting worth noting is the `DefaultStrategy` setting. Gentle provides three caching strategies: `Never`, `Temporary`, and `Permanent`. `Temporary` is the default setting; it allows garbage collection of cached objects. `Permanent` protects cached objects from garbage collection, but this setting is problematic because objects will pile up in the cache indefinitely (until the application explicitly directs the cache manager to clear them). `Never` turns off caching.

3.5.5 Transactions

The vehicle for transaction control in Gentle is the `Transaction` class provided in the API. There is nothing too incredible about this class; it is primarily a vehicle for accessing the transaction control mechanism provided by SQL. `Transaction` inherits from `BrokerLock`, and consequently a transaction is attached to a persistence broker at construction. In fact, the persistence methods that the `Transaction` class provides are really just calls to the `PersistenceBroker` under the hood.

When it is initialized, `Transaction` begins a database transaction (using ADO.NET, which in turn uses the transaction functionality of the underlying database). Subsequently, the application calls the persistence methods of the `Transaction`, which mirror the persistence methods of the `PersistenceBroker` class: `Transaction.Insert`, `Transaction.Update`, `Transaction.Remove`, and so on. (As implied above, these methods actually just call the persistence broker.) Finally, `Transaction` includes `Commit` and `Rollback` methods for ending the transaction. (These just call the `Commit` and `Rollback` methods of the underlying ADO.NET `IDbTransaction`.) A simple transaction might look something like the following outline:

```
Transaction t = new Transaction(persistenceBroker);
try {
    t.Insert(something);
    t.Update(somethingElse);
    t.Remove(anotherThing);
    t.Commit();
}
catch {
    t.Rollback();
    throw;
}
```

```
finally {
    t.Dispose();
}
```

Besides transactions, Gentle provides another means of concurrency control: automated revision numbering [23, §3.7]. To support this, the application developer adds an integer field to a persistent class and tags it with the `Concurrency` attribute.

```
[TableColumn, Concurrency]
private int revisionNumber;
```

This is a concurrency control field for Gentle use. Since it is not part of the domain model, we can leave it as a private field rather than creating a public accessor. This field keeps track of a revision number for the current instance. When the object is first persisted to the database, it is assigned a revision number of 0. Every time an update occurs, the revision number is incremented by 1. The revision counter is then used as an equality criterion in every single-row update or delete against the database. That is, with concurrency control enabled, instead of executing SQL such as

```
UPDATE TableName SET Column1 = 'value1', Column2 = 'value2'
WHERE IdentityColumn = 5
```

Gentle executes SQL like

```
UPDATE TableName SET Column1 = 'value1', Column2 = 'value2'
WHERE Identity Column = 5 AND ConcurrencyControl = 8
```

(if the current revision number is 8). Consequently, if the application tries to persist out-of-date data to the database, the update will fail. In fact, not only does no update occur, but Gentle also throws an exception to alert the application that its state is out of date. At that point the application can call the `Refresh` method of the persistence broker to update its state and then reapply the changes and try again to persist.

Transactions and revision numbering can be used in conjunction. They are both solutions to concurrency problems, but they fill different roles. For example, consider a situation in which a thread wants to atomically update a couple of objects that it has been using, a and b. The requirement of atomicity means that the thread should use a transaction to make its updates.

Now suppose that just before the thread starts its transaction, some other thread updates the database copy of b. The sequence of events would look like this:

1. Thread 1 loads a and b from the database and manipulates them in memory.
2. Thread 2 updates b in the database.
3. Thread 1 begins a transaction.
4. Thread 1 updates a.

5. Thread 1 updates b.
6. Thread 1 commits the transaction.

The changes that thread 1 makes to b overwrite those made by thread 2, although neither thread is ever made aware of the other. This is a concurrency problem that transactions do not solve. However, revision numbering can solve this problem. If b has a revision number field, the sequence of events will look like this:

1. Thread 1 loads a and b from the database and manipulates them in memory.
2. Thread 2 updates b in the database.
3. Thread 1 begins a transaction.
4. Thread 1 updates a.
5. Thread 1 tries to update b, but the revision numbers do not match. Gentle throws an exception.
6. Thread 1 receives the exception and can handle it as it deems appropriate. It will probably roll back the transaction, refresh its copy of b, and decide whether to reapply its changes to b or let the changes made by thread 2 stand.

The conflict is an inconvenience, but at least now thread 1 is made aware of the conflict so that it can resolve it as best it can under the circumstances.

Gentle concurrency controls can also help the application share the database with other users—even if those users do not employ the same persistence layer. Gentle transactions are effective at the level of the database, not just at the level of the persistence layer, so reliability, atomicity, isolation, and consistency are guaranteed for Gentle transactions even if other applications are sharing the database.

Revision numbering takes a little more work to function with other applications. In particular, other database users must respect the rules for revision numbering; they must initialize new rows with the correct initial revision number, increment the revision number when they update data, and check that they do not persist out-of-date data to the database. This is OK if the competing database users are developed concurrently with or subsequent to the Gentle-powered application, although it can create difficulties if Gentle needs to share a database with a legacy application. In such a case, the legacy application would need to be updated to incorporate revision numbering, or the application architects would have to decide that a certain amount of overwriting is acceptable.

3.6 Active Record

This Ruby ORM tool is named after the active record design pattern described by Fowler [13], pp. 160–164. It is therefore worth briefly describing this design

pattern here. The active record pattern is used by many developers who architect non-automated persistence solutions—by many developers, in fact, who are unaware that they are following what Fowler calls the active record pattern. An active record, according to Fowler, is a domain model responsible for its own persistence logic. Structurally, it is closely tied to one particular table in the database, and it contains all the logic necessary to save itself to and load itself from the database. The following Java snippet shows some functionality we might expect of an active record `Customer`.

```
Customer c = new Customer();
c.set_FirstName("Jane");
// ...
c.save();

c = Customer.Load(5);
c.set_LastName("Dennis");
c.save();

c.reportHistory(); // Suppose this to be a piece of business
                   // logic that does not use the database.
```

In this example, we see that we can create an active record from scratch, initialize its values, and insert it into the database. We can load an existing record from the database, modify it, and update the database with our modifications. An active record may also include business logic unrelated to its persistence functions, such as `reportHistory` in this example.

Some of the ORM systems we have already seen effectively simulate the active record pattern. In Gentle, for example, the application developer can ask a persistent object to persist itself, provided the class of that object inherits from `Persistent`. (In Gentle, the persistence methods of the `Persistent` class are just calls to the persistence broker under the hood. The persistence broker performs all actual persistence.)

Ruby's Active Record aims to solve two problems of the active record pattern: associations and inheritance. The former is solved by meta-programming macros, and the latter is solved by integrating the single table inheritance pattern, also described by Fowler [13], pp. 278–284. We will examine these solutions shortly. First, some background on Ruby and Rails is appropriate for readers unfamiliar with these technologies.

Ruby is an object-oriented programming language prized for its expressive power and ease of use. Currently, Ruby is interpreted, which makes its performance inferior to that of most compiled languages [40], but Ruby is to become bytecode-compiled (like Java and .NET) in version 1.9.1, due for release at the end of 2007 [27]. Ruby is well known for its reflection and meta-programming features, and it also includes other modern language features such as exception handling, operator overloading, garbage collection, and portability. Rails (http://rubyonrails.org/) is a Web application framework for Ruby. Released in 2004, Rails has become quite popular in a relatively short time. For con-

text, Ruby on Rails competes with other Web application frameworks such as ASP.NET and PHP. Rails follows Ruby's goal of expressivity and succinctness. Active Record is available as a stand-alone ORM solution for Ruby (http://rubyforge.org/projects/activerecord/), but it is most commonly used with Rails, as it forms part of the Rails standard library.

Active Record makes heavy use of both reflection and database analysis, to a much greater extent than the other ORM solutions examined in this paper. This is in line with Ruby's pro-reflection design philosophies. For example, if a database already exists for the application, creating a class corresponding to a particular table takes two lines of Ruby code.

```
class Customer < ActiveRecord::Base
end
```

Using behavior inherited from the `ActiveRecord::Base` class, this `Customer` class reflects on itself to find its own name, queries the database to see that it contains a table of the same name, analyzes the structure of that table, and modifies its own structure and behavior to match the database. All of this happens at run time.

Now, a customer object can be created, populated with data, and persisted very easily:

```
customer = Customer.new
customer.name = "Mary Smith"
customer.street_address = "123 Main St."
customer.postal_code = "12345"
customer.save
```

3.6.1 Architecture

From the preceding example, it should be clear that Ruby Active Record operates in a database-oriented paradigm (see Section 2.1.3), in contrast to the application-oriented and metadata-oriented systems we have seen heretofore. Application architects create the database first, and then the objects corresponding to the database tables are created to match it. Note that although Active Record for Ruby is very dynamic, reflective, and execution-oriented, it is worth observing incidentally these traits are by no means inherent (nor exclusive) to the database-oriented paradigm. On the contrary, a database-oriented mapper could carry out the mapping before compile time.

In Active Record, on the other hand, the mapping is accomplished almost entirely at run time. All that the software architects must specify in the application code is the existence of the classes corresponding to the database tables, as in the example above. (Of course, they may specify additional functionality for these classes beyond what is provided by the default mapping infrastructure, but anything beyond the class definition is optional.)

All persistent objects in Active Record inherit from `ActiveRecord::Base`, which provides powerful persistence functionality for its inheritors. This `Base`

class is by far the most important (and complex) part of the Active Record infrastructure. It includes the logic necessary to establish and destroy connections to the database, query for data, persist new objects and update existing ones, and address complex mapping problems such as inheritance and associations. The internal operations of the `Base` class are largely irrelevant to the present section; what is important is that the core persistence functionality of Active Record is consolidated into a single abstract class that serves as the base for all persistent objects.

`Base` is by no means the only component in the `ActiveRecord` framework, though. Also of architectural importance is the `ConnectionAdapters` module. For each DBMS that Active Record supports there is a connection adapter. For example, there are `MysqlAdapter` and `PostgreSQLAdapter` classes. All these adapters inherit from a common base class, `AbstractAdapter`. These adapters accommodate DBMS-specific peculiarities. For example, since the syntax to rename a table differs from DBMS to DBMS, each adapter has a `rename_table` method. In the `MySqlAdapter` class, it is defined by

```
def rename_table(name, new_name)
  execute "RENAME TABLE #{name} TO #{new_name}"
end
```

since MySQL includes a `RENAME TABLE` command. However, in PostgreSQL tables are renamed via the `ALTER TABLE` command, so in the `PostgreSQLAdapter` class the `rename_table` method is defined by

```
def rename_table(name, new_name)
  execute "ALTER TABLE #{name} RENAME TO #{new_name}"
end
```

Besides the `Base` class and the connection adapters, the Active Record framework includes a number of other classes and modules, but these provide peripheral functionalities that, though important, are not part of core persistence operations. Among them are modules such as `ActiveRecord::Validations`, which provides validation for active records, and `ActiveRecord::Locking`, which allows for optimistic or pessimistic locking.

3.6.2 Metadata

Active Record's extensive analysis of the database obviates much of the need for metadata. It can see for itself the names of the database tables and columns and it assumes that the application classes and properties should have related names. In particular, Active Record makes certain assumptions about table, field, and class names and translates freely between the two. For example, it assumes that tables are named in lowercase with plural nouns, with words delimited by underscores (`customers`, `system_operators`, etc.). It translates these names into camel-case class names like `Customer` and `SystemOperator`. It is fairly clever about this, translating `sentries` into `Sentry` and even `people` into `Person`. There is a similar operation for field names. Some of the assumptions that Active Record

makes are configurable. For example, to reverse the assumption that table names are plural, one would set

```
ActiveRecord::Base.pluralize_table_names = false
```

Of course, even a fairly clever system such as this must accommodate exceptions to its rules. In Ruby, this is not done through some external metadata file, but instead directly in the code of the relevant class. For example, if we are architecting a biological database that includes a `classes` table to accommodate taxonomic classes, but we are concerned about conflicts with the reflective `Class` class built into Ruby, we can name our corresponding persistent class `TaxonomicClass` as follows.

```
class TaxonomicClass < ActiveRecord::Base
  set_table_name "classes"
end
```

(Of course, in this case we might want to consider renaming the database table to `taxonomic_classes` to avoid confusion caused by a class-table naming mismatch, but this is just an example.)

3.6.3 Querying

Querying in Active Record is relatively simple. Persistent classes that inherit from `ActiveRecord::Base` have a `find` method, which provides three retrieval approaches. First, we can find by ID.

```
Customer.find(1)
```

returns the customer whose identifier is 1. We can also pass several IDs into `find` to retrieve several records at a time:

```
Customer.find(1,2,3)
```

Second, we can find all.

```
Customer.find(:all)
```

returns an array of all customers. We can refine our query to match only certain conditions, as in

```
Customer.find(:all, :conditions => "first_name = 'Mary'")
```

which finds all customers with the first name *Mary*. Besides `:conditions:`, the `find` method accommodates many other option hashes, such as `:group:` for `GROUP BY` clauses, `:include:` to produce outer joins, and `:order:` to specify result orderings.

The third retrieval approach is to find first, which returns the first result only.

```
Customer.find(:first)
```

Of course, this can be refined by option hashes too.

To make querying even nicer, Active Record provides dynamic attribute-based finders, which automatically provide new methods like `find_by_first_name` and `find_all_by_first_name`, which return the first record or all records that match the first name provided. Thus

```
Customer.find(:all, :conditions => "first_name = 'Mary'")
```

could be abbreviated to

```
Customer.find_all_by_first_name("Mary")
```

In some of the other mappers we have seen, one problem with querying is that it is sometimes unclear when clients should use the name of a database field and when they should instead supply the name of the class property. In Active Record, this is not a problem, as property names are identical to field names. (Of course, this is a limitation in itself, but in everyday situations it is not a problematic one.)

Like any querying framework, the Active Record infrastructure has limits to the queries it can accept. For queries that stray from these bounds, persistent objects provide the `find_by_sql` method, which takes an SQL command as its parameter. This command is run directly against the database, unedited. This means that it must use the names of the tables in the database, not the names of the corresponding classes. However, this is less problematic in a database-oriented solution like Active Record than in an application-oriented solution.

3.6.4 Caching

Active Record does not include any caching to speak of. This, of course, hampers performance, but it is not without any advantage. The lack of object caching is in line with Active Record's database-oriented philosophy. With object caching, Active Record could hardly be as dynamic as it is in fact. Active Record can respond to changes in the database—even changes to the structure of the database—instantaneously at run time. If it relied on its cache, it would not notice such changes immediately, and even once it did, it would be hard-pressed to reconcile its cache with the new information from the database. Indeed, none of the other ORM tools we have seen are nearly so responsive to database changes.

A second and related advantage is that Active Record is good at sharing databases with other applications. Without an object cache, Active Record has no problem with changes that other applications make to the database. Of course, this assumes that the other database users share politely, too. For example, they should not use caching either, and they should use transactions to ensure that the database is never visible to Active Record in an inconsistent state.

The performance benefit that caching affords, however, is not negligible. For stable applications that have sole control over a database and that have heavy

performance requirements, Active Record's refusal to cache can be frustrating. There is at least one tool that overcomes this limitation, a plug-in that adds caching to Ruby [44].

3.6.5 Transactions

Active Record does include support for transactions, and the syntax is simple. For example, if we wanted to transfer money from one bank account to another by withdrawing it from one account and depositing it in another, we might write

```
transaction do
  account1.withdraw(100)
  account2.deposit(100)
end
```

The persistence framework commits the transaction at the end of the block. If an exception occurs in the block, the framework rolls back the transaction. The framework still propagates the exception outward after rolling back the transaction so that the application can catch it.

Chapter 4

ORM: Present and future

4.1 Obstacles to the adoption of ORM

Despite a vast number of increasingly advanced and readily available ORM solutions and the maturity of ORM technology, ORM has yet to see widespread adoption in everyday industrial operations. One consultant called ORM "the Vietnam of Computer Science" [30], articulating the skepticism that many developers feel toward ORM. In this section, we present a number of architectural, technical, and cultural factors that seem to have undermined wider acceptance and use of ORM. In a later section, we make suggestions for abating these barriers.

4.1.1 The learning curve and other up-front costs

Every developer who has ever written a database application in .NET knows what ADO.NET is and how to use it; likewise for Java and JDBC, or Visual Basic and ActiveX Data Objects. Not every developer knows how to use Hibernate, iBATIS, or TopLink. From the perspective of a software development company or team, this presents a serious obstacle to ORM adoption. Regardless of which ORM solution the project manager or team leader settles on, it is unlikely that more than a small fraction of the software development staff will have more than, at best, a passing familiarity with it. Consequently, the organization will incur a substantial cost in training its developers to use the ORM solution in question (or, for that matter, even to gain an appreciation of the fundamentals of ORM, since many entry-level and mid-level developers scarcely know what ORM is). Unfortunately, this is not just a one-time cost. New developers hired subsequently to the adoption of ORM technology will likewise have to receive training (or otherwise the company will have to seek out developers who already have ORM experience, a search likely to be difficult and expensive).

In addition to these developer training costs, there may also be bureaucratic obstacles to the adoption of new technologies, including ORM, from the ranks

of managers and executive officers who have an "if it isn't broke, don't fix it" mentality.

Finally, besides personnel costs, there are technological costs to implementing an ORM solution. Just as every .NET developer knows ADO.NET, every .NET installation comes with the ADO.NET architecture; likewise for Java and JDBC. By contrast, setting up and installing ORM software on all the development, testing, staging, and production machines in an organization can be a substantial undertaking.

In many contexts, the long-term benefits afforded by ORM can outweigh these one-time personnel and technological costs. However, it is not always easy to see in advance that this is the case, and it may be even more difficult to convince management that it is so.

4.1.2 Actual and perceived performance limitations

Many developers and managers have a perception that ORM is slower than hand-crafted persistence code that directly interacts with a database API such as JDBC or ADO.NET. This perception has a kernel of truth. A naïvely architected ORM solution will indeed be slower than hand-crafted persistence code that does the same thing, because it introduces overhead: the overhead of reading metadata, reflecting on classes (if necessary), generating queries, and so on. In a hand-crafted solution, these tasks would normally be completed at design time by the developer and would not affect application performance. Indeed, the early ORM systems did suffer from poor performance, as do casually jerry-built ORM systems today.

However, modern, popular ORM systems like Hibernate and iBATIS are not naïvely architected. Instead, they introduce performance innovations like lazy checking and automatic caching that tend to far outweigh the relatively modest overhead that automated persistence layers incur by nature. Of course, features like caching and lazy checking can be implemented by a hand-crafted solution in principle, but this is a laborious process that developers must carry out on a class-by-class basis, so in applications with hand-crafted persistence code these niceties seldom appear.

Regardless of the relative performance merits of ORM and hand-crafted solutions, the perception of ORM solutions as slow persists. This perception has a chilling effect on adoption of ORM.

4.1.3 Sensitivity to architectural revisions

ORM would work best if, once the mapping was completed, no architectural changes were made to the application, database, or metadata ever again. Unfortunately, this is an unrealistic assumption. In reality, application developers, database administrators, and project managers frequently want to make architectural changes late in the development cycle. Sometimes these changes are modest in scope—adding a field (attribute) to an existing table (object)—and sometimes they are grand—reforming a collection of previously structurally

unrelated objects into an elaborate inheritance hierarchy. To accommodate such a change, all three components of the persistence system—application, database, and metadata—must change. The application and database must be restructured in parallel, and the metadata must be rewritten to reflect the new relationships between the objects and the tables, but in addition any existing data in the database must be shuffled around to fit into the new structure.

These operations are very difficult to automate. Consequently, most ORM systems have unsatisfactory support for revisions to the architecture of a system after a system has been mapped. In fairness to ORM, these operations are painful in a hand-mapped solution as well. However, ORM arguably exacerbates the problem, because in a system that uses ORM, architectural revisions require developers to think carefully about details that had previously been largely hidden from them.

4.1.4 Accommodation of legacy systems

A related problem involves the use of legacy databases. Often, new applications need to use data from an existing data source—a database lying around from a previous incarnation of the application, or a data source provided by a client or third party. In this case, neither the application developers nor the ORM solution have any say into the architecture of the data source; they have to work with what is available. In hand-coded persistence layers, this is merely an inconvenience; developers have to write their code to persist data to the legacy data source, even if the legacy data source has a questionable structure (e.g., poor normalization). In insufficiently flexible, automated persistence layers, this can be problematic. If the metadata language is not especially inexpressive, it may be difficult or even impossible to represent the mapping to the legacy database.

A similar problem occurs when a second application uses a database that is managed by an ORM system for an object-oriented application—in other words, when two applications, one using a persistence layer provided by an ORM product and one using a different persistence solution, need to use the same database. This is problematic regardless of whether the database was originally created for use with the application that uses the ORM solution, the application that does not, or even if it was created with both in mind. The reason is that many ORM products expect to have full ownership of the database to which they persist. In particular, they may implement caching, transactional, and delayed persistence strategies that make this assumption. To take a concrete example, suppose the ORM-enabled application retrieves some data and, unaware that it shares the database with another user, caches it. Then the other application incidentally changes this data. Later, when the persistence layer of the ORM-enabled application is summoned to retrieve this data, it will draw upon its cache, not realizing that another database user has inadvertently made its cache obsolete.

4.1.5 Limitations in expressing queries

ORM does a good job of accounting for the structural relationship between objects and tables and of shuffling data between objects and the database. It is not so good at capturing the behavioral aspects of data. In particular, many ORM systems fail to provide an elegant method for querying databases, beyond elementary operations. Neward [30] calls this "the data retrieval mechanism concern" and identifies three major approaches that ORM solutions can take to supporting queries: query by example, query by API, and query by language.

To query by example, an application creates an object of the type it wants to query to exemplify the sort of results it wants. For example, to find all persons named Smith, we might use code such as the following C# snippet:

```
Person p = new Person();
p.Surname = "Smith";
IList results = PersistenceManager.Execute(p);
```

This approach has serious defects. Most significantly, it does not readily accommodate queries of any complexity, even simple queries like "Find all cities whose population is less than 100,000" or "Find all persons not named Smith." That is, it doesn't support comparisons other than equality. Second, it requires all persistent classes to support both default constructors and null values in all fields, which may be a violation of business domain rules to which the classes would otherwise adhere. This compromises transparency.

In the query-by-API model, the ORM solution provides an API to allow applications to construct and execute query objects. The following C# snippet exemplifies this approach.

```
Query q = new Query(typeof(Person));
q.Filters.Add(new PropertyStringEqualityFilter("Surname", "Smith"));
IList results = PersistenceManager.Execute(q);
```

In this imaginary API, there is a `Query` object whose constructor takes an argument indicating which class or table is being queried. Here, the argument is of type `System.Type`; that is, the caller specifies the actual object-oriented class that is being queried. Presumably the `Query` object will then use metadata or reflection to learn what it needs to know about this class and its corresponding database table. A reasonable alternative approach would be to instead pass a string identifying the table in the database—as in `new Query("Person")`.

The `Query` class in this example includes a `Filters` property, which accesses a collection of filters that are to be conjoined and will ultimately appear in the `WHERE` clause of a `SELECT` statement against the database. Here we add a filter of type `PropertyStringEqualityFilter`, which compares the `Surname` property and the string "Smith" for equality. Finally, the persistence manager is called to execute the query. An alternative architecture would allow the query to execute itself: `q.Execute(PersistenceManager)` or something similar.

The obvious disadvantage of this approach is that it requires application developers to learn a new API, an API that may be quite elaborate, as the

above example suggests. The `PropertyStringEqualityFilter` class for example, hints at a great family of other, related filter classes that must exist: `PropertyPropertyEqualityFilter` classes (and other equality filter classes along these lines), inequality filter classes, less-than filter classes, less-than-or-equal-to filter classes, and so on. Furthermore, queries expressed through an API like this tend to be far more verbose than queries expressed in a query language like SQL, and complex queries such as those involving sophisticated joins may be difficult if not impossible to express through the API. Finally, this approach requires the developer to have a degree of familiarity with the underlying mapping—enough to know, for instance, that there is a `Surname` column in the person table that corresponds to the `Surname` property of the `Person` class. We can take advantage of the type system's reflection framework to overcome this to some degree. For example, by having the `Query` constructor in the example above accept a class instead of a string identifying a table, we eliminate the need for the developer to remember details about the mapping. However, this precipitates a performance penalty and exacerbates the problem of verbosity.

The final querying approach is for the ORM system to define a new query language similar to (the data retrieval portion of) SQL but better suited to the ORM context. This approach affords both expressive power and brevity. Query languages can easily be constructed to support the same sorts of complex operations that SQL supports (aggregates, joins, etc.). The following C# snippet makes the same query as the previous two examples. The ORM API and query language in this example are NHibernate and the Hibernate Query Language (HQL), respectively.

```
IQuery q = session.CreateQuery(
    "FROM Person p WHERE p.Surname = :surname");
q.SetString("surname", "Smith");
IList results = q.List();
```

Note that in HQL, names like `Person` and `Surname` actually refer to structures in the application code (here, the `Person` class and its `Surname` property), not to database structures such as tables and their columns. One complaint about this approach is that query languages provided by ORM systems are sometimes less expressive than true SQL. This objection held water in the past, but the query languages provided by the most mature ORM systems are increasingly capable. HQL, for example, supports subqueries, joins, aggregate functions, and even polymorphic queries [17, §11].

A more abstract objection is that using an SQL-style language to query the database compromises one of the central supposed benefits of ORM: that it enables application developers to understand data from a purely object-oriented perspective. A query language, it is said, compromises the purity of this notion by forcing developers to write RDBMS-style query code.

None of the three approaches to querying present here is perfect. Querying by example is inadequate for all but the most basic of queries. Querying by API is verbose and tedious. Querying by language is generally the best solution, but

arguably compromises the promise of ORM to enable to developers to think only in terms of objects.

4.2 The state of the art

ORM is not a very old technology, but it has been around for long enough for substantial innovations to have been made. The earliest automated persistence technologies were clumsy and slow. The popular ORM solutions of today, by contrast, provide impressive arrays of features that save time for application developers and improve performance and normalization. We discussed many of these features in depth and from a theoretical standpoint in Chapter 2. Here, we take a higher-level view of a few of the most important recent developments in ORM technology, both technical and conceptual.

4.2.1 Performance enhancements

Some of the most significant and impressive improvements to ORM systems have been in the area of performance. An early, and initially justified, criticism of ORM was that it could not match the performance of hand-tailored solutions that directly called a database API. This was generally true, because, in the absence of performance-enhancing features, ORM systems add overhead (extra layers of abstraction) to code and have no way of mitigating the resulting performance deterioration.

This concern is seldom valid anymore. Modern ORM systems use a variety of tricks to improve performance: caching, lazy fetching, dirty checking, and so on. These tricks reduce the frequency with which the application has to connect to and communicate with the database. Since database transactions are often the most expensive operations in an application (especially when executed across a network, as they often are), these performance improvements tend to far outweigh the overhead of the persistence layer (even if it uses expensive language features like reflection).

4.2.2 Standardization

The Java community has recently witnessed a powerful movement in the direction of standardization of persistence solutions. Version 5 of the Java Platform, Enterprise Edition [43, JSR 244], released in 2006, includes the totally new Java Persistence API, an ORM specification based on existing persistence technologies like Hibernate, TopLink, and Java Data Objects (JDO). (JDO itself was also a specification developed under the Java Community Process [43, JSR 12].) The Java Persistence API was developed as part of the Enterprise JavaBeans 3.0 standard [43, JSR 220], but its use is no longer limited to Enterprise JavaBeans components, nor even to the Enterprise Edition of the Java Platform. The Java Persistence API standard includes the definition of the API, which is the

javax.persistence package; the definition of the Java Persistence Query Language; and provision for metadata.

The Java Persistence API was developed not in competition with existing persistence solution vendors, but on the contrary with their cooperation and participation. Consequently, Hibernate and TopLink are moving into conformity with the standard, giving them the status of implementations of the standard. New products are also emerging to implement the standard, including Apache OpenJPA (http://incubator.apache.org/openjpa/).

Currently this standardization activity is largely limited to the Java community, but it is possible that persistence APIs will appear for other platforms in the future. A distinct but related topic is the integration of persistence tools into languages and enterprise platforms. For example, in the past there has been speculation that future versions of the .NET Framework will support ORM natively, although there has been no indication from Microsoft in this direction. A less speculative example is Ruby on Rails. Although Active Record is a stand-alone ORM package for Ruby, it is incorporated into the Rails platform. It seems likely that better examples of languages with integrated persistence will appear in the future.

4.2.3 Multi-platform tools

Different programming languages and enterprise platforms often have profound differences between them, presenting a formidable obstacle to porting tools from language to language. One would think that this would be particularly true of ORM tools, which are not only syntactically but architecturally tied to the language or platform in which they operate. Nonetheless, some cross-platform ports of ORM tools have been surprisingly successful. The two most notable cases are Hibernate, a Java persistence solution ported to the .NET Framework, and iBATIS, also a Java persistence solution, which has been ported not only to .NET, but also to Ruby. (Indeed, the iBATIS team has considered additional ports, including PHP and Python [6, §2.6.4].)

It is not surprising that the most successful ports have been between Java and .NET; Java and .NET bear important conceptual and architectural similarities, so it makes sense that successful solutions to a Java problem would also be successful when applied to the same problem in .NET. The port of iBATIS to Ruby is rather more impressive.

4.3 Future directions

Here we consider what the future of ORM might look like. The comments in this section are intended primarily as recommendations of what future ORM systems ought to look like (either to improve developer productivity, system performance, or the popularity of ORM), but they may also be regarded as predictions. Since the areas in which ORM tools might best improve are those areas in wihch it is weakest, and since future development is likely to follow the course suggested

by present development, the items below parallel several of the subsections in Sections 4.1 and 4.2.

Improve awareness of ORM and ORM tools. Unfamiliarity with and distrust of ORM is a major problem. Despite the great number of persistent applications being produced, many entry-level and mid-level developers, as well as project managers and executive officers, remain unaware of even the existence of automated persistence. Even among those engineers who are familiar with ORM, many do not know the scope of its application or the features that modern ORM solutions provide. In particular, some developers have outdated notions of the capabilities of ORM, believing it to be slow and incapable of supporting sophisticated database operations. Consequently, improving awareness of ORM and the facilities that modern ORM tools provide is an important, non-technical step to more widespread adoption of the technology.

Continue to improve performance. Despite advances in ORM technology that can make it perform better than naïvely hand-authored solutions that directly call database APIs, some developers continue to identify performance as an important weakness of ORM. It is certainly true that there is room for improvement in this area. The largest improvements are likely to be attained by reducing the frequency of communications with the DBMS (e.g., by making caching plans smarter) and by improving the performance of expensive procedures such as navigating object graphs.

Standardize persistence solutions. It is still too early to see the results of the standardization of Java persistence APIs, but it promises a number of benefits. Once the community comes to embrace the standard, developers of persistent Java applications will no longer need to worry about different ORM solutions with different persistence APIs. Instead, Java developers can learn a single persistence API and apply this knowledge to all persistent Java applications that use it—which should before long be a solid majority of applications that use automated persistence technology. As the standard API gains popularity, developers have an increasing incentive to learn it (since it will have increasingly broad applicability), so eventually many persistent-application developers will be familiar with the API. Software development teams then have a greater incentive to adopt ORM solutions, since (regardless of which implementation of the Java Persistence API they choose) more developers will be available to support it. This creates a sort of virtuous circle reinforcing the proliferation of the Java Persistence API. Another consequence of standardization is that software projects will be able to more easily switch between competing ORM solutions (e.g., to migrate from TopLink to Hibernate), because standard API calls should remain largely unchanged and because developers need not be retrained. Other language communities might do well to follow the example that the Java Persistence API provides.

Integrate transparent persistence into object-oriented languages. Relatedly, future languages and enterprise platforms may integrate ORM into their class libraries. This would be especially appropriate in platforms, such as the .NET Framework, which are based on large, capable class libraries supporting diverse application demands. Currently, no mainstream, object-oriented language supports ORM natively, but ActiveRecord, an ORM utility, is built into Ruby on Rails.

Improve automated support for architectural revisions. Application developers often have to restructure their objects and the relationships between them and make corresponding changes to the database, and in practice these changes can often occur unplanned very late in the development cycle (or in post-release maintenance). Automating these changes is a difficult task, and current persistence solutions provide little support to the developers who have to make them. Future ORM solutions should work to provide better support for automating architectural revisions.

Better accommodate legacy systems. It would be nice (for ORM) if every application used exactly one database, but in fact there are many situations in which multiple applications must share the same database simultaneously. For example, often new projects must work alongside legacy applications, sharing a database that already exists. This requires developers to think it some depth about the operation of the persistence layer, since sharing a database precludes the use of certain ORM features such as caching. It should be easier to configure future object-relational mappers to work alongside legacy and other applications that must share the database.

Support expressive, powerful queries. Queries are a technical and conceptual problem for many ORM solutions, which must balance the need to present solution users with the full expressive power and concision of SQL with a theoretically motivated desire to uphold the objects-only pattern in ORM. Future ORM technology must work to reconcile these goals. Additionally, future querying solutions should hide the database structure from the application developer as much as possible. Even if the ORM API provides a query language that resembles SQL syntactically, it should reference objects and classes, not records and tables. Since they reference the application code instead of the database, ORM query languages should also support object-oriented notions such as inheritance and polymorphism. Current query languages support these concepts to some extent, but more work can be done in this direction.

Bibliography

[1] Scott W. Ambler. Mapping objects to relational databases: O/R mapping in detail, 2006. URL http://agiledata.org/essays/mappingObjects.html.

[2] *iBATIS.NET: DataMapper Application Framework (v1.5.1)*. Apache, Forest Hill, MD, July 2006. URL http://ibatis.apache.org/docs/dotnet/datamapper/.

[3] *EOModeler User Guide*. Apple, Cupertino, CA, May 2006. URL http://developer.apple.com/documentation/WebObjects/UsingEOModeler/.

[4] Deborah J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49:123–128, 2006.

[5] Christian Bauer and Gavin King. *Hibernate in Action*. Manning, Greenwich, CT, 2005.

[6] Clinton Begin, Brandon Goodin, and Larry Meadors. *iBATIS in Action*. Manning, Greenwich, CT, 2007.

[7] John Carlis and Joseph Maguire. *Mastering Data Modeling: A User-Driven Approach*. Addison-Wesley, Boston, 2000.

[8] Per Cederqvist. *CVS—Concurrent Version Systems v1.11.22*, June 2006. URL http://ximbiot.com/cvs/manual/cvs-1.11.22/cvs.html.

[9] Peter Pin-Shan Chen. The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems*, 1:9–36, 1976.

[10] E. F. Codd. Is your DBMS really relational? *Computerworld*, October 14, 2006.

[11] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13:377–387, 1970.

[12] Erik Dörnenburg. *.NET Entity Objects: Architecture and Implementation*, 2004. URL http://neo.codehaus.org/docs/Neo_Architecture.ppt.

[13] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, Boston, 2003.

[14] Michael M. Gorman. Is SQL a real standard anymore? *The Data Administration Newsletter*, 16, April 2001. URL http://www.tdan.com/i016hy01.htm.

[15] Jim Gray. A transaction model. In *Automata, Languages, and Programming: Seventh Colloquium (Noordwijkerhout, the Netherlands, July 14–18, 1980)*, volume 85 of *Lecture Notes in Computer Science*, Berlin, 1980. Springer. URL http://research.microsoft.com/~gray/papers/A%20Transaction%20Model%20RJ%202895.pdf.

[16] Peter Gulutzan. Standard SQL. *DBAzine*, May 3, 2005. URL http://dbazine.com/db2/db2-disarticles/gulutzan3.

[17] *NHibernate: Relational Persistence for Idiomatic .NET (v1.0.2)*. JBoss, Raleigh, 2006. URL http://hibernate.org/hib_docs/nhibernate/html/.

[18] Alan Kay. Re: Clarification of "object-oriented". E-mail to Stefan Ram, 2003. URL http://www.purl.org/stefan_ram/pub/doc_kay_oop_en.

[19] Kevin Kline and Daniel Kline. *SQL in a Nutshell*. O'Reilly, Sebastopol, CA, 2001.

[20] Robin Liu. Gentle.Net's architecture, 2004. URL http://www.mertner.com/confluence/x/6AE.

[21] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, Wokingham, England, 1993.

[22] Jim Melton. *Advanced SQL:1999: Understanding Object-Relational and Other Advanced Features*. Morgan Kaufmann, San Francisco, 2002.

[23] Morten Mertner and Clayton Harbour. *Gentle.NET Users Guide*, 2006. URL http://www.mertner.com/confluence/x/Pg.

[24] *What's New in the .NET Framework Version 2.0*. Microsoft, Redmond, WA, 2005. URL http://msdn2.microsoft.com/en-us/library/t357fb32.aspx.

[25] *Transact-SQL Reference*. Microsoft, Redmond, WA, 2005. URL http://msdn2.microsoft.com/en-us/library/ms189826.aspx.

[26] *What's New and Enhanced in Transact-SQL*. Microsoft, Redmond, WA, 2005. URL http://msdn2.microsoft.com/en-us/library/ms189465.aspx.

[27] Daigo Moriwaki. Day one of RubyKaigi2006. *RedHanded*, June 10, 2006. URL http://redhanded.hobix.com/cult/rubyKaigi2006.html.

[28] Craig S. Mullins. *Database Administration: The Complete Guide to Practices and Procedures*. Addison-Wesley, Boston, 2002.

[29] *MySQL 5.1 Reference Manual*. MySQL AB, 2007. URL http://mysql.com/doc/refman/5.1/en/.

[30] Ted Neward. The Vietnam of Computer Science. *The Blog Ride*, June 26, 2006. URL http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx.

[31] *OSCache*. OpenSymphony, 2005. URL http://www.opensymphony.com/oscache/.

[32] *PL/SQL User's Guide and Reference (10g Release 1)*. Oracle Corporation, Redwood Shores, CA, 2003. URL http://download-east.oracle.com/docs/cd/B14117_01/appdev.101/b10807/toc.htm.

[33] *Oracle TopLink Developer's Guide 10g*. Oracle Corporation, Redwood Shores, CA, 2006. URL http://oracle.com/technology/products/ias/toplink/doc/10131/main/_html/toc.htm.

[34] *Oracle TopLink Frequently Asked Questions*. Oracle Corporation, Redwood Shores, CA, 2006. URL http://oracle.com/technology/products/ias/toplink/technical/tl10g_faq.htm.

[35] Monica Pawlan. Reference objects and garbage collection, 1998. URL http://java.sun.com/developer/technicalArticles/ALT/RefObj/.

[36] Hamid Pirahesh. Object-oriented features of DB2 Client/Server. In *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, page 483, 1994.

[37] *PostgreSQL 8.2.3 Documentation*. The PostgreSQL Global Development Group, 2006. URL http://postgresql.org/docs/8.2/.

[38] Jonathan Rees. JAR on object-oriented. E-mail to Paul Graham. URL http://mumble.net/~jar/articles/oo.html.

[39] Donald Smith. A brief history of TopLink, 2004. URL http://oracle.com/technology/tech/java/newsletter/articles/toplink/history_of_toplink.html.

[40] Joel Spolsky. Ruby performance revisited. *Joel on Software*, September 12, 2006. URL http://joelonsoftware.com/items/2006/09/12.html.

[41] Potter Stewart. Concurring opinion, *Jacobellis v. Ohio*, 378 U.S. 184, 1964.

[42] Sun Microsystems. Sun ships JDK 1.1: JavaBeans included. Press release, February 1997. URL http://www.sun.com/smi/Press/sunflash/1997-02/sunflash.970219.0001.xml.

[43] *Java Specification Requests*. Sun Microsystems, Santa Clara, 2007. URL http://jcp.org/en/jsr/all.

[44] Chris Wanstrath. Scaling Rails with memcached. Presentation originally given at the San Francisco Ruby Meetup, September 12, 2006. URL http://errtheblog.com/static/pdfs/memcached.pdf.