

NHibernate in Action

MEAP

Unedited Draft

Pierre Henri Kuate
Tobin Harris
Christian Bauer
Gavin King

 MANNING



Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Licensed to Jason Wilden <jasonwilden@internode.on.net>

Table of Contents

Chapter One: Object/Relational Persistence in .Net

Chapter Two: Hello NHibernate

Chapter Three: Writing and Mapping classes

Chapter Four: Working with persistent objects

Chapter Five: Transactions, concurrency, and caching

Chapter Six: Advanced mapping concepts

Chapter Seven: Retrieving objects efficiently

Chapter Eight: Developing NHibernate Applications

Chapter Nine: Writing Real World Domain Models

Chapter Ten: Advanced Persistent Techniques

Appendix A: SQL Fundamentals

Appendix B: More on NHibernate Mapping Attributes

Appendix C: Introducing ActiveRecord and MonoRail

Appendix D: Going Forward

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



MEAP Edition
Manning Early Access Program

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Licensed to Jason Wilden <jasonwilden@internode.on.net>

Object/Relational Persistence in .NET

This chapter covers

- .NET persistence and relational databases
- Layering .NET applications
- Approaches to implement persistence in .NET
- How NHibernate solves persistence of objects in relational databases
- Advanced persistence features

Software Development is an ever changing discipline where new techniques and technologies are constantly emerging. As software developers, we have an enormous array of tools and practices available, and picking the right ones can often make or break a project. One choice that is thought to be particularly critical is how to manage persistent data, or put more simply, how to store, load and save data.

There are endless options available to us. We can store data in simple binary or text files on a disk. We can chose different formats including CSV, XML, JSON, YAML, SOAP, or even invent our own format. Alternatively, we can send our data over the network to another application or service, such as a relational database, an Active Directory server, or a message queue. We might even need to store data in several places, or combine all these options within a single application.

As you may begin to realize, managing persistent data is considered a thorny topic. Of all the options, relational databases have proven to be extremely popular but there are still many choices, questions and options that confront us in our daily work. For example, should we use DataSets, or are DataReaders more suitable? Should we use stored procedures? Should we hand code our SQL, or let our tools dynamically generate it for us? Should we strongly type our DataSets? Or, should we actually build a hand-coded *domain model* containing classes? If so, how do we go about loading and saving the data to and from the database? Do we use code generation? The list of continues...

This topic is not just restricted to .NET. Then entire development community has been debating this topic, often fiercely, for many years.

Debate continues, but one approach that has gained widespread popularity is *object/relational mapping*, or ORM. Over the years, many libraries and tools have emerged to help the developer implement ORM in their applications. One such tool is NHibernate - a sophisticated and mature object/relational mapping tool for .NET.

NHibernate is a .NET port of the popular Java “Hibernate” library. NHibernate aims to be a complete solution to the problem of managing persistent data when working with relational databases and domain model classes. It strives to undertake the hard work of mediating between the application and the database, leaving the developer free to concentrate on the business problem at hand. This book covers both basic and advanced NHibernate usage. It also recommends best practices for developing new applications using NHibernate.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Before we can get started with NHibernate, it will be useful for you to understand what persistence is and the various ways it can be implemented using the .NET framework. This chapter will help you understand why tools like NHibernate are really needed.

First, we define the notion of persistence in the context of .NET applications. We then demonstrate how a classic .NET application is implemented, using the standard persistence tools available in the .NET framework. You will discover some common difficulties encountered when using relational databases with object-oriented frameworks such as .NET, and how popular persistence approaches try to solve these problems. Collectively, these problems are referred to as the *paradigm mismatch* between object oriented and database design. We then go on to introduce the approach taken by NHibernate and many of its advantages. Following that, we'll dig in to some complex persistence challenges that make a tool like NHibernate essential. Finally, we define what Object/Relational Mapping is and why you should use it. By the end of this chapter, you should have a clear idea of the great benefits you can reap by using NHibernate.

Do I really need to read all this background information?

No. If you want to try NHibernate right away, skip to chapter 2, where we jump in and start coding a (small) NHibernate application. You'll be able to understand chapter 2 without reading this chapter, but we recommend that you read this chapter if you are new to persistence in .NET. That way, you'll understand the advantages of NHibernate and know when to use it. You will also learn some important concepts like *unit of work*. So, if you are interested by this discussion, you might just as well continue on with chapter 1, get a broad idea of persistence in .NET and then move on.

1.1 What is Persistence?

Persistence is one of the fundamental concerns in application development. If you have some experience in software development, you have already dealt with it. Actually, almost all applications require persistent data. We use persistence to allow our data to be stored even when the programs that use it are not running.

To illustrate this, let's say that you want to create an application that allows users to store their company telephone numbers and contact details, and retrieve them whenever needed. Unless you want the user to leave the program running all the time, you'd soon realize that your application will need to somehow save the contacts somewhere. You are now faced with a persistence decision; you'll need to work out which *persistence mechanism* you want to use. You have the option of persisting your data in many places, the simplest being a text file. More often than not, you might choose a *relational database*, because it is widely understood and offers some great features for reliably storing and retrieving data.

1.1.1 Relational databases

You have probably already worked with a relational database such as Microsoft SQL Server, MySQL or Oracle. If it isn't the case, take a look at appendix A. Most of us use relational databases every day, they have widespread acceptance and are considered a robust and mature solution to modern data management challenges.

A relational database management system (RDBMS) isn't specific to .NET, and a relational database isn't necessarily specific to any one application. So, we can have several applications accessing a single database, some written in .NET, some written in Java or Ruby etc. Relational technology provides a way of sharing data between many different applications. In fact, even different components within a single application can independently access a relational database (a reporting engine and a logging component, for example). Essentially, relational technology is a common denominator of many unrelated systems and technology platforms. Hence, the relational data model is often the common enterprise-wide representation of *business*

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

objects. Or put more simply; a business usually needs to store information about various things such as Customers, Accounts and Products etc (the business objects), and the relational database is usually the chosen central place where these are both defined and stored. This makes the relational database a very important piece in the IT landscape.

Relational database management systems have SQL-based application programming interfaces; hence we call today's relational database products *SQL database management systems* or, when we're talking about particular systems, *SQL databases*.

1.1.2 Understanding SQL

As with any .NET database development, a solid understanding of relational databases and SQL is a prerequisite when using NHibernate. You'll need to use your knowledge of SQL to tune the performance of your NHibernate application. NHibernate will automate many repetitive coding tasks, but your knowledge of persistence technology must extend beyond NHibernate itself if you want take advantage of the full power of modern SQL databases. Remember that the underlying goal is robust, efficient management of persistent data.

If you feel you may need to improve your SQL skills, then pick up a copy of the excellent books *SQL Tuning* by Dan Tow [Tow 2003] and *SQL Cookbook* by Anthony Molinaro [Mol 2005]. Joe Celko also has some excellent books on advanced SQL techniques. For a more theoretical background, consider reading *An Introduction to Database Systems* [Date 2004].

1.1.3 Using SQL in .NET Applications

.NET offers us many tools and choices when it comes to making our applications work with SQL databases. We might lean on the Visual Studio IDE, taking advantage of its drag and drop capabilities where, in a series of mouse clicks, we can create database connections, execute queries and display editable data on-screen. We believe this is great for simple applications, but the approach doesn't scale well for larger, more complex applications.

Alternatively, we may use *SqlCommand* objects, and manually write and execute our SQL to build *DataSets*. This can quickly become rather tedious, and what we really want to do is work at a slightly higher level of abstraction so that we can focus on solving business problems rather than worrying about data access concerns. If you are interested in learning more about the wide range of tried and tested approaches to data access, then consider reading Martin Fowlers' "Patterns of Enterprise Application Architecture", where many techniques are catalogued explained in depth.

Of all the options, the approach that we want to take is to write actual classes – or business entities - that can be loaded and saved to and from the database. Unlike *DataSets*, these classes are not designed to mirror the structure of a relational database (such as rows and columns). Instead they are concerned with solving the business problem at hand. Together these classes typically represent the object oriented *domain model*.

1.1.4 Persistence in object oriented applications

In an object-oriented application, persistence allows an object to outlive the process or application that created it. The state of the object may be stored to disk and an object with the same state re-created at some point in the future.

This application isn't limited to single objects—entire graphs of interconnected objects may be made persistent and later re-created. Most objects aren't actually persistent; a *transient* object is one which has a limited lifetime that is bounded by the life of the process that instantiated it. A simple example is a web control

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

object which only exists in memory for a fraction of a second before it is rendered to screen and flushed from memory. Almost all .NET applications contain a mix of persistent and transient objects, and it makes good sense to have a subsystem that manages the persistent ones.

Modern relational databases provide a structured representation of persistent data, enabling sorting, searching, and grouping of data. Database management systems are responsible for managing things like concurrency and data integrity; they're responsible for sharing data between multiple users and multiple applications. A database management system also provides data-level security. When we discuss persistence in this book, we're thinking of all these things:

- Storage, organization, and retrieval of structured data
- Concurrency and data integrity
- Data sharing

In particular, we're thinking of these problems in the context of an object-oriented application that uses a domain model. An application with a domain model doesn't work directly with the tabular representation of the business entities (i.e. using DataSets); the application has its own, object-oriented model of the business entities. If the database has ITEM and BID tables, the .NET application would define Item and Bid classes rather than using DataTables for these.

Then, instead of directly working with the rows and columns of a DataTable, the business logic interacts with this object-oriented domain model and its runtime realization as a graph of interconnected objects. The business logic is never executed in the database (as an SQL stored procedure), it's implemented in .NET. This allows business logic to make use of sophisticated object-oriented concepts such as inheritance and polymorphism. For example, we could use well known design patterns such as Strategy, Mediator, and Composite [GOF 1995], all of which depend on polymorphic method calls. Now a caveat: Not all .NET applications are designed this way, nor should they be. Simple applications might be much better off without a domain model. SQL and the ADO.NET are perfectly serviceable for dealing with pure tabular data, and the DataSet makes CRUD operations even easier. Working with a tabular representation of persistent data is straightforward and well understood.

However, in the case of applications with nontrivial business logic, the domain model helps to improve code reuse and maintainability significantly. We focus on applications with a domain model in this book, since NHibernate and ORM in general are most relevant to this kind of application.

It will be useful to understand how this domain model fits into the "bigger picture" of an entire software system. To explain this, we take a step back and look at something called the layered architecture.

1.1.5 Persistence and the Layered Architecture

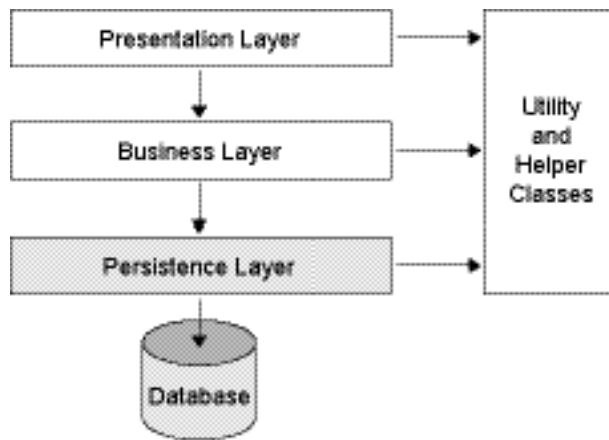
Many, if not most, systems today are designed with a layered architecture, and NHibernate works very well with that design. However, what exactly is a layered architecture?

A layered architecture splits a system in to several "groups", where each group contains code addressing a particular problem area. These groups are called layers. For example, a User Interface layer might contain all the application code for building web pages and processing user input. One major benefit to the layering approach is that changes to one layer can often be made without significant disruption to the other layers, thus making systems less fragile, and more maintainable. The practice of layering includes some basic rules:

- Layers communicate top to bottom. A layer is dependent only on the layer directly below it.
- Each layer is unaware of any other layers except for the layer just below it.

For business applications, there is a popular, proven, high-level application architecture that comprises of three layers: The presentation layer, the business logic layer and the persistence layer. See figure 1.1.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>



Layered architecture highlighting the persistence layer.

Let's take a closer look at the layers and elements in the diagram:

- *Presentation layer*—The user interface logic is topmost. In a web application this contains the code responsible for drawing pages or screens, collecting user input, and controlling navigation.
- *Business layer*—The exact form of this layer varies widely between applications. It's generally agreed, however, that the business layer is responsible for implementing any business rules or system requirements that would be understood by users as part of the problem domain. Most of the time, this layer shares the representation of the business domain entities with the persistence layer. We revisit this issue in chapter 3.
- *Persistence layer*—The persistence layer is a group of classes and components responsible for saving and retrieving application data to and from one or more data stores. This layer defines a mapping between the business domain entities and the database. It may not surprise you to hear that NHibernate would be used primarily in this layer.
- *Database*—The database exists outside the .NET application. It's the actual, persistent representation of the system state. If a SQL database is used, the database includes the relational schema and possibly stored procedures.
- *Helper/utility classes*—Every application has a set of infrastructural helper or utility classes that support the other layers. For example, UI widgets, messaging classes, **Exception** classes and logging utilities. These infrastructural elements aren't considered to be a layer, because they don't obey the rules for interlayer dependency in a layered architecture.

Remember that layers are particularly useful for breaking down large and complex applications, and are often overkill for the extremely simple .NET applications. For such simple programs, you may choose to put all your code in one place. So, instead of neatly separating business rules and database access functions into separate layers, you'd put them all in to your code-behind files (or Windows Forms classes). Tools like Visual Studio .NET make it very easy and painless to build this kind of simple application. However, be aware that this approach can quickly lead to a problematic code base; as the application grows, you have to add more and more code to each form or page, and things start to become increasingly difficult to work with. Moreover, changes made to the database may easily break your application, and finding and fixing the affected parts can be time consuming and downright painful!

Should all applications have three layers?

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Although a *three layers architecture* is common and advantageous in many cases, not all .NET applications are designed like that, nor should they be. Simple applications might be much better off without complex objects. SQL and the ADO.NET API are perfectly serviceable for dealing with pure tabular data, and the ADO.NET DataSet makes basic operations even easier. Working with a tabular representation of persistent data is straightforward and well understood.

1.2 Approaches to Persistence in .NET

We've discussed how, in any sizeable application, a persistence layer is needed to handle the loading and saving of data. There are many approaches available to us when building this persistence layer, each having its own advantages and disadvantages. Three popular choices are:

- DataSets
- Hand-coding
- NHibernate (or similar)

Despite the fact we highly recommend NHibernate, it's always wise to consider the alternatives. As you'll soon learn, building applications with NHibernate is quite straightforward, but that does not mean it is perfect for every project. In the following sections we will examine and compare these strategies in detail, discussing implications in database access and user interface.

1.2.1 Choice of Persistence Layer

In our applications, we often want to load, manipulate and save items to and from the database. Regardless of which persistence approach we use, at some point ADO.NET objects must be created and SQL commands must be executed. It would be tedious and unproductive to write all this SQL code each time we have to manipulate some data, so we can use a persistence layer to take care of these low-level steps.

Essentially, the persistence layer is the set of classes and utilities used to make life easier when it comes to saving and loading data. ADO.NET allows executing SQL commands which actually perform the persistence, but the complexity of this process require that we wrap these commands behind components that understand how our entities should be persisted. These components can also hide the specificities of the database, making our application less coupled to the database, hence easier to maintain. For example, when you use a SQL identifier containing spaces or reserved keywords, you must delimit this identifier. Databases like SQL Server use brackets for that, while MySQL use back-ticks. It is possible to hide this detail and let the persistence layer select the right delimiter.

Based on the approach used, the internals of the persistence layer widely differ.

DataSet-based persistence layer

Visual Studio lets us effortlessly generate our own persistence layer, which can then be extended with new functionality with few clicks. The classes generated by Visual Studio know how to access the database, and can be used to load and save the entities contained in the DataSet.

Again, a small amount of work is required to get started. However, we still have to fall back to hand-coding when more control is needed, which is usually inevitable as described in section 1.3.

Hand-coded persistence layer

Hand coding a persistence layer can ultimately involve a lot of work; it is common to first build a generic set of functions to handle database connections, execution of SQL commands, etc. Then, on top of this sub-layer,

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

we then have to build another set of functions that save, load and find our business entities. Things get much more involved if you need to introduce caching, business rule enforcement or handling of entity relationships.

Obviously, hand coding your persistence layer gives you the greatest degree of flexibility and control; you have ultimate design freedom and can easily exploit specialized database features. On the other hand, it can be a huge undertaking, and is often quite tedious and repetitive work, even when using *code generation*.

Persistence layer using NHibernate

NHibernate provides all the features required to quickly build an advanced persistence layer in code. It is capable of loading and saving entire graphs of interconnected objects whilst maintaining the relationships between them.

In the context of our auction application, we can easily save an **Item and its Bids** by implementing a method like:

```
public void Save(Item item) {
    OpenNHibernateSession();
    session.Save(item);
    CloseNHibernateSession();
}
```

Here, session is an object provided by NHibernate. Don't worry about understanding the code yet. For now, we just want you to see how simple the persistence layer is with NHibernate. We will start using NHibernate in Chapter 2, where you will discover that it is very straightforward to execute persistence operations. All you need to do is to write your entities and explain to NHibernate how to persist them.

1.2.2 Implementing the entities

Once we have chosen a persistence layer approach, we can focus on building our business objects, or entities, that the application will manipulate. Basically, they are classes representing the real-world elements that our application must manipulate. For an auction application, **User**, **Item** and **Bid** would be common examples. We now discuss how we can implement our business entities in each of the three approaches.

Entities in a DataSet

A DataSet represents a collection of database tables, and in turn these tables contain the data of the entities. Therefore, a DataSet stores data about business objects in a similar fashion to a database. A generated typed DataSet can be used to ease the manipulation of data, and it is also possible to insert some business logic and rules.

As long as we want to manipulate data, .NET and IDEs provide most features required to work with a DataSet. However, as soon as we think about business objects as *objects* in the sense of object-oriented design, we can hardly be satisfied by a DataSet (typed or not). After all, business objects represent real-world elements; and these elements have data and behavior. They may be linked by advanced relationships like inheritance; but this is not possible with DataSet. This level of freedom in the design of entities can only be achieved by hand-coding them.

Hand-coded entities

Returning to the example of an auction application, we considered having the entities: **User**, **Item** and **Bid**. Aside the data they contain, we would also expect there to be relationships between them. For example, an **Item** has a collection of bids and a **Bid** refers to an Item; in C# classes, this might be expressed using a

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

collection like `item.Bids` and a property like `bid.Item`. As you can see, the object oriented view is different to the relational view of things: Instead of having primary and foreign keys, we have associations. You'll find that object oriented design gives us some other powerful modeling concepts such as inheritance and polymorphism.

Hand-coded entities are free from any constraint; they are even free from the way they are persisted in the database. They can evolve (almost) independently and be shared by very different applications; this is a very important advantage when working in a complex environment.

However, they are difficult and tedious to code; think about the manual work that would be required to support the persistence of entities inheriting from other entities. It is common to use code generation or base classes (like `DataSet`) to add features with a minimal effort. These features may be related to the persistence, transfer or presentation of information. However, without a helpful framework, these features can be very time-consuming to implement.

Entities and NHibernate

NHibernate is *non-intrusive*; it is common to use it with hand-coded (or generated) entities. You must provide mapping information that tells how these entities should be loaded and saved. Therefore, you get the best of both worlds: Object-oriented entities easily persisted in a relational database.

There are many fundamental differences between objects and relational data. Trying to use them together reveals the paradigm mismatch (also called object/relational impedance mismatch). We explore this mismatch in section 1.3. By the end of this chapter, you will have a clear idea of the problems caused by the paradigm mismatch and how NHibernate solves these problems.

Once the entities are implemented, we must think about how they will be presented to the end-user.

1.2.3 Displaying entities in the user interface

Using NHibernate implies using entities, and using entities has some consequences on the way the User Interface (UI) is written. For the end-user, the UI is one of the most important elements. Whether it is a Web application (using `ASP.NET`) or a Windows application, it must satisfy the needs of the user. A deep discussion around the implementation of a UI is not in the scope of this book; but the way the persistence layer is implemented has a direct effect on the way the UI will be implemented.

In this book, we will refer to the UI as the *presentation layer*. .NET provides controls to display information. The simplicity of this operation depends on how the information is stored.

DataSet-based presentation layer

Microsoft has made the effort of adding support for data binding with `DataSet` in most .NET controls. It is easy to bind a `DataSet` to a control so that its information is displayed and any changed (done by the user) reverberated in the `DataSet`.

Using `DataSets` is probably the most productive way to implement a presentation layer. You may lose some control on how information is presented, but it is good enough in most cases. The situation is a little bit more complicated with hand-coded entities.

Presentation layer and entities

The difficulty in data binding information contained in hand-coded entities resides in the fact that there are so many possibilities to implement them. A `DataSet` is made of tables, columns and rows; but a hand-coded entity, a class without any constraint, contains fields and methods without any standardized way to access and display them. .NET controls can be data-bound to entities; in this case, they will only access public properties.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This is enough in simple cases; otherwise, we must fall back to hand-code the process of loading entities in the UI and copy back the changes to persist.

Although this approach is not as easy as when using DataSets, it is still quite simple. And you are free to present your entities exactly as you want. But there are some other situations where it is not that easy. One of them is *reporting*; tools like Crystal Reports provide a limited support for entities.

There are many libraries (like the open source project ObjectViews) which greatly simplify the data binding of entities. We suggest that you take a look at these libraries. And don't forget that you are always free to fallback to DataSets when dealing with edge cases like complex reporting where they are much easier to manipulate. We will discuss this issue in chapter 9.

Using persistence-able information affects the way the UI is designed. Basically, data should be loaded when opening a UI and saved when closing the UI. NHibernate proposes some patterns to deal with this process. You will learn more about these patterns in chapter 8.

Now, all layers are in place, we can work on performing actions.

1.2.4 Implementing CRUD operations

When working with persistent information, we are concerned with persisting and retrieving this information. CRUD stands for Create, Read, Update, Delete. These are primitive operations executed even in the simplest application. Most of the time, these operations are triggered by events raised in the presentation layer. For example, the user may click on a button to view an item. The persistence layer will be used to load this item which will then be bound to a form displaying its data.

No matter which approach you use, these primitive operations are well understood and simple to implement. Operations that are more complex are covered in the next section.

CRUD operations with DataSets

We already know that a good part of the persistence layer can be generated when using DataSets. This persistence layer contains classes to execute CRUD operations. And Visual Studio 2005 and .NET 2.0 come with more powerful classes called *table adapters*.

Not only do these classes support primitive CRUD operations, but they are also extensible. You can either add methods calling stored procedures or generate SQL commands by few clicks. But if you want to implement anything more complex, you must hand-code it; and we will see, in the next section, that there are some very useful features which are not easy to implement. And the structure of a DataSet may make them harder to implement.

Hand-coded CRUD operations

A hand-coded CRUD operation does exactly what you want because you write the SQL command to execute; on the other hand, it is a very repetitive and annoying work.

It is possible to implement a framework that generates these SQL commands. Once you understand that loading an entity implies executing a SELECT on its database row, you can automate primitive CRUD operations. However, much more work is required for complex queries and manipulating interconnected entities.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

CRUD operations using NHibernate

As soon as you provide to NHibernate the mapping information of your entities, you can execute a CRUD operation by a single method call. This is a fundamental feature of an Object/Relational Mapping (ORM) tool. Once it has all the information it needs, it can solve the *object/relational impedance mismatch* at each operation.

NHibernate is designed to efficiently execute CRUD operations. Experience and tests have helped discover many optimizations and best practices. For example, when manipulating entities, best performances are achieved by delaying persistence to the end of the *transaction*. At this point, a single connection is used to save all entities.

Now that we have covered all basic steps and operations in persistence, we can discover some advanced features that will clearly illustrate the advantages of NHibernate.

1.3 Why do we need NHibernate?

So far, we have talked about a very simple application. In the real-world; however, we rarely deal with simple applications. An enterprise application has many entities with complex business logic and design goals: Productivity, maintainability, and performance are all essential.

In this section, we will walk through some features indispensable to implement a successful application. First, we give some examples illustrating the fundamental differences between objects and relational database. You will also have an idea of how NHibernate helps create a bridge between these representations. Then, we turn to the persistence layer to discover how NHibernate deals with complex and numerous entities. You will learn the patterns and features that it provides to achieve the best performance. Finally, we cover complex queries; we will see that NHibernate can be used to write a powerful and complete search engine.

Let's start with the entities and their mapping to a relational database.

1.3.1 The paradigm mismatch

While a database is relational, we are using object-oriented languages. There is no direct way to persist an object as a database row. And persistence should not hinder our ability to design entities, which correctly represent what we are manipulating.

We call *paradigm mismatch* (or *object/relational impedance mismatch*) the fundamental incompatibilities between the design of objects and relational databases. Let's take a closer look at some of the problems created by the paradigm mismatch.

The problem of granularity

Granularity refers to the relative size of the objects you're working with. When we're talking about .NET objects and database tables, the granularity problem means persisting objects that can have various kinds of granularity to tables and columns that are inherently limited in granularity.

Let's take an example from the online auction described in section 1.1.4. Let's say we want to add an address to a **User** object, not as a string but as another object. How are we supposed to persist this user in a table? We may add an **ADDRESS** table, but it is generally not a good idea (for performance reasons). We may create a *user-defined column type* (UDT) but this option is not broadly supported and portable. Another option is to merge the address information into the user, but this is not a good object-oriented design and it is not much re-useable.

It turns out that the granularity problem isn't especially difficult to solve. Indeed, we probably wouldn't even list it, were it not for the fact that it's visible in so many approaches like DataSet. We describe the solution to this problem in chapter 3, section 3.6, "Fine-grained object models."

A much more difficult and interesting problem arises when we consider inheritance, a feature of object-oriented design that is commonly used.

The problem of inheritance and polymorphism

Object-oriented languages support the notion of *inheritance*. But this is not the case for relational databases. Let's say that, in our auction application, we can have many kind of items. We could create subclasses like `Furniture` and `Book`, each with its specific information. So how are we supposed to persist this hierarchy of entities in a relational database? You must also realize that a `Bid` can refer to any subclass of `Item`. It should be possible to run *polymorphic queries* like retrieving all bids on books. In chapter 3, section 3.7, "Mapping class inheritance," we discuss how object/relational mapping solutions such as NHibernate solve the problem of persisting a class hierarchy to a database table or tables.

The problem of identity

The identity of a database row is commonly expressed as the *primary key* value. As you'll see in chapter 3, section 3.5, "Understanding object identity," .NET *object identity* is not naturally equivalent to the primary key value. With relational databases, it is recommended to use a *surrogate key*, that is a primary key column with no meaning to the user; but .NET objects have an intrinsic identity which is either based on their memory location or on an user-defined convention (by using the implementation of the `Equals()` method).

Based on this problem, how are we supposed to represent associations? Let's look at that next.

Problems relating to associations

In our object model, associations represent the relationships between objects. For instance, a bid has a relationship with an item. This association is done using object references. In the relational world, an association is represented as a *foreign key column*, with copies of key values in several tables. There are subtle differences between the two representations.

Object references are inherently directional; the association is from one object to the other. If an association between objects should be navigable in both directions, you must define the association *twice*, once in each of the associated classes.

On the other hand, foreign key associations aren't by nature directional. In fact, navigation has no meaning for a relational data model, because you can create arbitrary data associations with table joins and projection. We'll discuss association mappings in great detail in chapters 3 and 6.

If you think about DataSet in all these problems, you will realize how rigid its structure is. The information in a DataSet is presented exactly like in the database. In order to navigate from one row to another, you must manually resolve their relationship; that is use a *foreign key* to find the referred row in the related table. Let's move from the representation of the entities to how they can be efficiently manipulated.

1.3.2 Unit of Work and Conversations

When a user works on an application, he performs distinct unitary operations. These operations can be referred as *conversations* (or *business transactions* or *application transactions*). For example, placing a bid on an item is a conversation. Seasoned programmers know how hard it can be to make sure that many related operations

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

performed by the user are treated as if they were a single bigger business transaction (a *unit*). You will learn in this section that NHibernate makes it much easier to achieve. Let's take another example to better illustrate this concept.

Popular media players allow you to rate the songs you hear and later sort them based on your rating. This means that your ratings are persisted. When you open a list of songs, you listen and rate them one by one. When is persistence supposed to take place?

The first solution, which may come to your mind, is to persist the rating when it is entered by the user. But this is very inefficient: The user may change it many times and this persistence will be done separately for each song. Though, this approach is certainly safer if you expect this application to crash at any moment.

What we can do is let the user rate all songs of the list, and when he closes it, then we persists all ratings. The process of rating these songs can be called a conversation.

Let see how it works and what its benefits are.

The Unit of Work pattern

When working with a relational database, we tend to think of commands: saving or loading. But an application can perform operations involving many entities. When these entities are loaded or saved depends on the context.

For example, if you want to load the last item created by a user, you must first save this user (and his collection of items), then you can run a query retrieving this item. If you forget to save the user, you will start getting hardly-detectable bugs.

The Identity Map pattern

A pattern called *Identity Map* is used by NHibernate to make sure that this item's user is the same object as the user you had before loading this item (as long as you are working in the same transaction). You will learn more about the concept of identity in chapter 3, section 3.5, "Understanding object identity".

Now, imagine that you are involved in a very complex *conversation*. Manually tracking the entities to save or delete, and making sure that you load each entity only once can be a nightmare.

A Unit of Work is a pattern followed by NHibernate to solve this problem and ease the implementation of conversations. Note that, we cover conversations in chapter 5 and we implement them in chapter 10.

You can create entities and associate them to NHibernate, and then it keeps track of all loading and may save any change only when required. At the end of the transaction, it figures out and applies all changes in their correct order.

Transparent persistence and lazy loading

Because NHibernate keeps track of all entities, it can greatly simplify your application and increase its performance. Here are two simple examples:

When working on an item of our auction application, a user can add, modify or delete its bids. It would be painful to manually track these changes one by one. Instead, we can use the feature of NHibernate called *transparent persistence*: You can ask NHibernate to save all changes in the collection of bids when the item is persisted. It will automatically figure out which CRUD operations must be executed.

Now, if you want to modify a user, you will load, change and persist it. But what about the collection of items that this user has? Should you load these items or leave the collection un-initialized? Loading the items would be inefficient, but leaving the collection un-initialized will limit our ability to manipulate the user.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

NHibernate support a feature called *lazy loading* to solve this problem. Basically, when loading the user, you can decide between loading the items or not. If you choose not to do so, this collection will be transparently initialized when you need it.

There are many implications in using these features; we will progressively cover them in this book.

Caching

Tracking entities implies keeping their references in some place. NHibernate uses what is called a *cache*. This cache is indispensable to implement the Unit of Work pattern. However, it can also make applications more efficient. We deeply cover the concept of caching in chapter 5, section 5.3 “Caching theory and practice”.

A cache is used by the identity map of NHibernate to avoid loading an entity many times. And this cache can be shared by transactions and even applications.

Let’s say that we build a website for our auction application. Our visitors may be interested by some items. Without a cache, these items will be loaded from the database each time a visitor wants to see it. With few lines, we can ask NHibernate to cache these items and enjoy the performance gain.

1.3.3 Complex queries and ADO.NET Entity Framework

This is the last but not the least feature related to persistence. In section 1.2.5, we talked about CRUD operations. We have discovered some features related to Create, Update and Delete (all related to the Unit of Work pattern). Now, we are going to talk about Retrieve operations; that is searching and loading information.

We know that we can easily generate code to load an entity using its identifier (primary key in the context of relational database). But in real-world applications, users rarely deal with identifiers; instead they use some criteria to run a search and then pick the information they want.

Implementing a query engine

If you are familiar with SQL, you know that you can write very complex queries using the `SELECT ... FROM ... WHERE ...` construct. But if you work with business objects, you would then have to transform the results of your SQL queries into entities. We already advertised the benefits of working with entities, therefore it might make more sense to take advantage of those benefits even when querying the database .

Based on the fact that NHibernate can load and save entities, we can deduce that it knows how each entity is mapped to the database. When we ask for an entity by it’s identifier, NHibernate knows how to find it. So, we should be able to express a query using entity names and properties, and then NHibernate should be able to convert that into a corresponding SQL query understood by the relational database.

NHibernate provides two query APIs: The first is called the *Hibernate Query Language*. HQL is similar to SQL in many ways, but also has some useful object-oriented features. Note that you can query NHibernate using plain old SQL, but as you will learn, using HQL offers several advantages.

The second query API is called the *Query by Criteria API* (QBC). It provides a set of type-safe classes to build queries in your chosen .NET language. This means that if you’re using Visual Studio, you’ll benefit from the inline error reporting and Intellisense™!

To give you a foretaste of the power of these APIs, we are going to build three simple queries. Firstly, here is some HQL. It finds all bids for items, where the sellers name starts with the letter K:

```
from Bid bid
where bid.Item.Seller.Name like 'K%'
```

As you can see, it’s very easy to understand. If you want to write some SQL to do the same thing, you would need something a little more verbose, along the lines of:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

select B.*
from BID B
inner join ITEM I on B.ITEM_ID = I.ITEM_ID
inner join USER U on I.AUTHOR_ID = U.USER_ID
where U.NAME like 'K%'

```

To illustrate the power of the Query by Criteria API, we will use an example derived from that shown in chapter 8, section 8.5.1. This shows a method that allows us to find and load all users who are similar to an example user, and who also have a bid item similar to a given example item:

```

public IList<User> FindUsersWithSimilarBidItem(User u, Item i) {
    Example exampleUser =
        Example.Create(u).EnableLike(MatchMode.Anywhere);
    Example exampleItem =
        Example.Create(i).EnableLike(MatchMode.Anywhere);
    return GetSession().CreateCriteria(typeof(User))
        .Add( exampleUser )
        .CreateCriteria("Items")
        .Add( exampleItem )
        .List<User>();
}

```

Essentially, this method allows the developer to pass objects that represent the kind of Users we want NHibernate to find and load. It creates two NHibernate **Example** objects and uses the Query by Criteria API to run the query and retrieve a list of users. The notion of example entity (here, example User and example Item) is both powerful and elegant, as demonstrated below:

```

User u = new User();
Item I = new Item();
u.Name = "K";
i.State = ItemState.Active;
i.ApprovedBy = administratorUser;
List<User> result = FindUsersWithSimilarBidItem(u, i);

```

Here, we are using our **FindUsersWithSimilarBidItem** method to retrieve users whose names contain 'K' and who are selling an active bid Item, which has also been approved by the administrator. Quite a feat for such little code! If you are new to this approach, you should find it somewhat unbelievable. Don't even try to implement this query using hand-coded SQL.

You can learn more about queries in Chapters 5 and 7. If you aren't fully satisfied by these APIs, you may also want to watch new upcoming developments that allow LINQ to be used with NHibernate.

ADO.NET Entity Framework

At the time of this writing, Microsoft is working on its *Next-Generation Data Access* technology which introduces a number of interesting and exciting innovations. You may feel that this technology will soon replace NHibernate, but this is very unlikely. Let's see why.

Perhaps the most exciting new feature is a powerful query framework code-named LINQ. LINQ extends your favorite .NET language so that you can run queries against various types of data source, without having to embed query strings in your code. So, when querying a relational database, you can do something like this:

```

//C# LINQ example, look ma no strings!
IEnumerable users = from u in Users
    where u.Forename.StartsWith("K")
    order by user.Forename descending
    select u;

```

As you can see, the queries are type-safe and allow you to take advantage of many .NET language features. One key aspect of LINQ is that it gives us a declarative way of working with data, so we can express what we want in simple terms rather than typing lots of for-each loops. Furthermore, we can benefit from helpful IDE

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

capabilities such as auto-completion and parameter assistance. This is a big win for everybody. Because LINQ is designed to be extensible, other tools such as NHibernate are able to integrate with this technology and benefit from it greatly. At the time of publishing, there is some very promising work in progress for a LINQ to NHibernate project. At the time of writing, Manning are working on the “LINQ in Action” book, it sounds *very* interesting!

Microsoft is also working on a framework currently called LINQ to Entities, which aims to provide developers with an object-relational mapping framework not completely unlike NHibernate. This is a good step forward because Microsoft will promote the DataSet less often, and also start educating and promoting the benefits of object-relational mapping tools. Another project called LINQ over DataSet greatly improves the query capabilities of DataSet, but it doesn't solve many other issues discussed in this chapter.

All these technologies will take a little time to mature. There are still many unanswered questions, such as how extensible this framework will be? Will it support most popular RDBMSs or just SQL Server? Will it be easy to work with legacy database schemas? The fact is, no framework can provide all features, so it must be extendable to let you integrate your own features.

Note that if you're particular projects require you to work with legacy databases, you can read Chapter 10, section 10.2 to learn about the features NHibernate gives us to work with more exotic data structures.

Let's now dig in the methodology behind NHibernate

1.4 Object/Relational Mapping

You already have an idea of how NHibernate provides object/relational persistence. However, you may still be unable to tell what object/relational mapping (ORM) is. We will try to answer this question now. After that, we will discuss some non-technical reasons to use ORM.

What Is ORM?

Time has proved that relational databases provide a good means of storing data, and that object-oriented programming is a good approach to building complex applications. With object/relational mapping, it is possible to create a translation layer that can easily transform objects into relational data and back again. As this bridge will manipulate objects, it can provide many of the features we need (like caching, transaction, concurrency control). All we really have to do is provide information on how to map objects to tables.

Briefly, object/relational mapping is the automated (and possibly transparent) persistence of objects in an application to the tables in a relational database, using metadata that describes the mapping between the objects and the database. ORM, in essence, works by transforming data from one representation to another.

Isn't ORM a Visio plug-in?

The acronym ORM can also mean Object Role Modeling, and this term was invented before object/relational mapping became relevant. It describes a method for information analysis, used in database modeling, and is primarily supported by Microsoft Visio, a graphical modeling tool. Database specialists use it as a replacement or as an addition to the more popular entity-relationship modeling. However, if you talk to .NET developers about ORM, it's usually in the context of object/relational mapping.

We learnt in section 1.3.1 that there are many problems to solve when using ORM. We refer to these problems as the paradigm mismatch. Let's discuss, from a non-technical point of view, why we should face this mismatch and use an ORM tool like NHibernate.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

1.4.1 Why ORM?

The overall solution for these mismatch problems can require a significant outlay of time and effort. In our experience, the main purpose of up to 30 percent of the .NET application code written is to handle the tedious SQL/ADO.NET and the manual bridging of the object/relational paradigm mismatch. Despite all this effort, the end-result still doesn't feel quite right. We have seen projects nearly sink due to the complexity and inflexibility of their database abstraction layers.

Modeling mismatch

One of the major costs is in the area of modeling. The relational and object models must both encompass the same business entities. However, an object-oriented purist will model these entities in a very different way than an experienced relational data modeler. You learnt some details of this problem in section 1.3.1. The usual solution to this problem is to bend and twist the object model until it matches the underlying relational technology.

This can be done successfully, but only at the cost of losing some of the advantages of object orientation. Keep in mind that relational modeling is underpinned by relational theory. Object orientation has no such rigorous mathematical definition or body of theoretical work. There is no elegant transformation waiting to be discovered. (Doing away with .NET and SQL and starting from scratch isn't considered elegant.)

Productivity and maintainability

The domain modeling mismatch problem isn't the only problem solved by ORM. A tool like NHibernate makes you more productive. It eliminates much of the grunt work (more than you'd expect) and lets you concentrate on the business problem. No matter which application development strategy you prefer—top-down, starting with a domain model; or bottom-up, starting with an existing database schema—NHibernate used together with the appropriate tools will significantly reduce development time.

Fewer lines of code makes the system more understandable since it emphasizes business logic rather than plumbing. Most important, a system with less code is easier to refactor. NHibernate substantially improves maintainability. Not only because it reduces the number of lines of code, but also because it provides a buffer between the object model and the relational representation. It allows a more elegant use of object orientation on the .NET side, and it insulates each model from minor changes to the other.

Performance

A common claim is that hand-coded persistence can always be at least as fast, and can often be faster, than automated persistence. This is true in the same sense that it's true that assembly code can always be at least as fast as .NET code—in other words, it's beside the point.

The unspoken implication of the claim is that hand-coded persistence will perform at least as well in an actual application. But this implication will be true only if the effort required to implement at-least-as-fast hand-coded persistence is similar to the amount of effort involved in utilizing an automated solution. The really interesting question is, what happens when we consider time and budget constraints?

The best way to address this question is to define means to measure performance and thresholds of acceptability. Then, you can find out if the performance cost of an ORM is unacceptable. Experience has proved that a good ORM has a minimal impact on performance. It can even perform better than classic ADO.NET when correctly used, due to features like *caching* and *batching*. NHibernate is based on a mature architecture that enables you to take advantage of many performance optimizations with a minimal effort.

Database independence

NHibernate abstracts your application away from the underlying SQL database and SQL dialect. The fact that it supports a number of different databases confers a certain level of portability on your application.

You shouldn't necessarily aim to write totally database-independent applications, since the capabilities of databases differ and achieving full portability would require sacrificing some of the strength of the more powerful platforms. Nevertheless, an ORM can help mitigate some of the risks associated with vendor lock-in. In addition, database independence helps in development scenarios where developers use a lightweight local database but deploy for production on a different database.

1.5 Summary

In this chapter, we have discussed the concept of object persistence and the importance of NHibernate as an implementation technique. Object persistence means that individual objects can outlive the application process; they can be saved to a data store and be re-created at a later point in time. We have walked through the layered architecture of a .NET application and the implementation of persistence, exploring three approaches.

We understand the productivity of DataSet, but we also realize how limited and rigid it is. We have discovered many useful features that would be painful to hand-code. In addition, we know how NHibernate solves the object/relational mismatch.

This mismatch comes into play when the data store is a SQL-based relational database management system (RDBMS). For instance, a graph of richly typed objects can't simply be saved to a database table; it must be disassembled and persisted to columns of portable SQL data types.

We have glanced at the powerful query APIs of NHibernate. After you have started using them, you will never want to come back to SQL. You will also discover that NHibernate is designed to support esoteric mapping and to be extensible.

Finally, we learned what object/relational mapping (ORM) is and we discussed, from a non-technical point of view, the advantages of using this approach.

ORM isn't a silver bullet for all persistence tasks; its job is to relieve the developer of 95 percent of object persistence work, such as writing complex SQL statements with many table joins and copying values from ADO.NET result sets to objects or graphs of objects. A full-featured ORM middleware like NHibernate provides database portability, certain optimization techniques like caching, and other viable functions that aren't easy to hand-code in a limited time with SQL and ADO.NET.

It's likely that a better solution than ORM will exist some day. We (and many others) may have to rethink everything we know about SQL, persistence API standards, and application integration. The evolution of today's systems into true relational database systems with seamless object-oriented integration remains pure speculation. However, we can't wait, and there is no sign that any of these issues will improve soon (a multibillion-dollar industry isn't very agile). ORM is the best solution currently available, and it's a timesaver for developers facing the object/relational mismatch every day.

We've hopefully given you a great background in the reasons behind OR/M, the critical issues that must be addressed, and the tools and approaches available for .NET for addressing them. We've explained that NHibernate is a fantastic OR/M tool that allows you to combine the benefits of both object orientation relational databases simultaneously. The next step is to give you a much more hands-on look at NHibernate, so you can see how it might be used in your own projects. That's where chapter 2 comes in.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Hello NHibernate!

This chapter covers

- NHibernate in action with a “Hello World” application
- How to architecture a NHibernate application
- Writing and mapping a simple entity
- Configuring NHibernate
- Implementation of primitive CRUD operations

It’s good to understand the need for object/relational mapping in .NET applications, but you’re probably eager to see NHibernate in action. We’ll start by showing you a simple example that demonstrates some of its power.

As you’re probably aware, it’s traditional for a programming book to start with a “Hello World” example. In this chapter, we follow that tradition by introducing NHibernate with a relatively simple “Hello World” program. However, simply printing a message to a console window won’t be enough to really demonstrate NHibernate. Instead, our program will store newly created objects in the database, update them, and perform queries to retrieve them from the database.

This chapter will form the basis for the subsequent chapters. In addition to the canonical “Hello World” example, we introduce the core NHibernate APIs and explain how to configure NHibernate in various runtime environments, such as ASP.NET applications and stand-alone WinForms applications.

2.1 “Hello World” with NHibernate

NHibernate applications define persistent classes that are “mapped” to database tables. Our “Hello World” example consists of one class and one mapping file. Let’s see what a simple persistent class looks like, how the mapping is specified, and some of the things we can do with instances of the persistent class using NHibernate.

2.1.1 Installing NHibernate

Before we can start coding our “Hello World” application, we first need to install NHibernate. We then need to create a new Visual Studio solution to contain our sample application.

You can download NHibernate installer from <http://www.nhibernate.org>. The Java Hibernate project is also hosted at this web site, so you will need to locate the correct file to download. At the time of writing, the latest version of the Nhibernate can be installed by downloading and running a file called NHibernate1.2.1.GA.msi.

Once you have downloaded and installed NHibernate, you are ready to create a new solution and start using it.

2.1.2 Create a new Visual Studio Project

For our example application you should create a new blank project with Visual Studio. This is a simple application, so the easiest thing to create is a “# Console Application”. Name your project HelloWorldNHibernate or similar. Note that you can also use NHibernate with VB.NET projects, but in this book we’ve chosen to use # examples.

Our application will need to make use of the NHibernate library, so our next step is to reference it in our new project. To do this, right click on the project and select “Add reference...” You should then click the Browse tab and navigate to the folder where NHibernate installed. By default, NHibernate lives in the `C:\Program Files\NHibernate\bin\net2.0\` folder. Once you have found the assemblies, select the “NHibernate.dll” and “NHibernate.Mapping.Attributes.dll”. Clicking ok will add these references to your solution.

By default, the Console Application should have added a file called Program.cs to your solution. In console applications, this will be the first thing that is run when you execute the program.

We need to make sure that we reference the NHibernate at the top of the Program files with the `using NHibernate` and `using NHibernate.Cfg` statements, as shown below.

```
using System;
using System.Collections.Generic;
using System.Reflection;
using NHibernate;
using NHibernate.Cfg;

namespace HelloWorldNHibernate
{
    public class Program
    {
        static void Main()
        {
        }
    }
}
```

Now that we have our solution set up, we’re ready to start writing our first NHibernate application.

2.1.3 Creating the Employee Class

The objective of our sample application is to store an `Employee` record in a database and to later retrieve it for display. Our application therefore needs a simple persistent class, `Employee`, which will represent a person who is employed by a company.

In Visual Studio, add a new class file to your application, and name it Employee.cs when prompted. Then, type the following code for our Employee entity. This is shown in listing 2.1.

Listing 2.1 Employee.cs: A simple persistent class

```
namespace HelloWorldNHibernate
{
    class Employee
    {
        public int id;
        public string name;
        public Employee manager;
    }
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    public string SayHello()
    {
        return string.Format(
            "Hello World!", said {0}.", name);
    }
}

```

Our `Employee` class has three fields: the identifier field, the name of the employee, and a reference to the employee's manager. The identifier field allows the application to access the database identity—the primary key value—of a persistent object. If two instances of `Employee` have the same identifier value, they represent the same row in the database. We've chosen `int` for the type of our identifier field, but this isn't a requirement. NHibernate allows virtually anything for the identifier type, as you'll see later.

Note that I've used public *fields* here rather than properties. This is purely to make the sample code shorter, and is not always considered good practice.

Instances of the `Employee` class may be managed (made persistent) by NHibernate, but they don't have to be. Since the `Employee` object doesn't implement any NHibernate-specific classes or interfaces, we can use it like any other .NET class.

```

Employee fred = new Employee();
fred.name = "Fred Bloggs";
Console.WriteLine( fred.SayHello() );

```

This code fragment does exactly what we've come to expect from "Hello World" applications: It prints "Hello World, said Fred Bloggs" to the console. It might look like we're trying to be cute here; in fact, we're demonstrating an important feature that distinguishes NHibernate from some other persistence solutions. Our persistent class can be used with or without NHibernate —no special requirements are needed. Of course, you came here to see NHibernate itself, so let's first set up the database, and then demonstrate using NHibernate to save a new `Employee` to it.

2.1.4 Setting up the database

We need to have a database set up so that NHibernate has somewhere to save entities. Setting up a database for this program should only take a minute. NHibernate can work with many databases, but for this example we'll use Microsoft SQL Server 2005.

Your first step will be to open Microsoft SQL Server Management Studio, connect to your database server and open a new query window. Type the following in the SQL window to quickly create a new database.

```

CREATE DATABASE HelloNHibernate
GO

```

Run this SQL to create the database. The next step is to switch to that database, and create a table to hold our `Employee` data. To do that, delete the above SQL and replace it with the following.

```

USE HelloNHibernate
GO
CREATE TABLE Employee (
    id int identity primary key,

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
        name varchar(50),
        manager int )
GO
```

Run this code, and you have now created a place to store your Employee entities. We are now ready to see NHibernate in Action!

Note that, in chapter 9, we'll show you how to use NHibernate to *automatically* create the tables your application needs using just the information in the mapping files. There's some more SQL you won't need to write by hand!

2.1.3 Creating an Employee and saving to the Database

The code required to create an Employee and save them to the database is shown below. This comprises of two functions: `SaveEmployeeToDatabase` and `OpenSession`. You can type these functions that into your `Program.cs` file, below the `static void Main(...)` function in the Program class.

```
static void CreateEmployeeAndSaveToDatabase()
{
    Employee tobin = new Employee();
    tobin.name = "Tobin Harris";

    using (ISession session = OpenSession())
    {
        using( ITransaction transaction = session.BeginTransaction() )
        {
            session.Save(tobin);
            transaction.Commit();
        }
        Console.WriteLine("Saved Tobin to the database");
    }
}

static ISession OpenSession()
{
    Configuration c = new Configuration();
    c.AddAssembly(Assembly.GetCallingAssembly());
    ISessionFactory f = c.BuildSessionFactory();
    return f.OpenSession();
}
```

The `CreateEmployeeAndSaveToDatabase` function calls the NHibernate `Session` and `Transaction` interfaces. (We'll get to that `OpenSession()` call soon.) We're not quite ready to run the code just yet, but to give you an idea of what would happen, running the `CreateEmployeeAndSaveToDatabase` function would result in some SQL being executed behind the scenes by NHibernate:

```
insert into Employees (name, manager)
values ('Tobin Harris', null)
```

Hold on—the `Id` column is not being initialized here. We didn't set the `id` field of message anywhere, so how can we expect it to get a value? Actually, the `id` property is special: It's an identifier property—it holds a unique value generated by the database. This generated value is assigned to the `Employee` instance by NHibernate during the call to the `save()` method.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

We won't discuss the `OpenSession` function in depth, but essentially it configures NHibernate and returns a `session` object that we can use to save, load and search objects in our database (and much more!). Don't use this `OpenSession` function in your production projects, as you will learn more economical approaches throughout this book.

We want "Hello World" program to print the message to the console. Now that we have an `Employee` in the database, we're ready to demonstrate this next.

1.2.4 Loading an Employee from the Database

Having written some code to create an `Employee` and save them to the database, we now demonstrate some code that can retrieve all `Employees` from the database, in alphabetical order. Type this code in below the previous `OpenSession()` function.

```
static void LoadEmployeesFromDatabase()
{
    using (ISession nhibernateSession = OpenSession())
    {
        IQuery query = nhibernateSession.CreateQuery(
            "from Employee as emp order by emp.name asc");

        IList<Employee> foundEmployees = query.List<Employee>();

        Console.WriteLine("\n{0} employees found:", foundEmployees.Count);

        foreach( Employee employee in foundEmployees )
        {
            Console.WriteLine(employee.SayHello());
        }
    }
}
```

The literal string `"from Employee as emp order by emp.name asc"` is a NHibernate query, expressed in NHibernate's own object-oriented Hibernate Query Language (HQL). This query is internally translated into the following SQL when `query.List()` is called:

```
select e.id, e.name, e.manager
from Employee e
order by e.name asc
```

If you've never used an ORM tool like NHibernate before, you were probably expecting to see the SQL statements somewhere in the code or metadata. They aren't there. All SQL is generated at runtime (actually at startup where possible).

So far we've defined our `Employee` entity, set up the database, and written some code to create a new employee. NHibernate has barely entered the picture yet, so next we will tell NHibernate about our `Employee` entity and how we want them saved in the database.

1.2.5 Creating a Mapping File

To allow NHibernate to do its magic in any of the above code, it first needs more information about how the `Employee` class should be made persistent. This information is usually provided in an XML mapping document. The mapping document defines, among other things, how properties of the `Employee` class map to columns of the `Employees` table. Let's look at the mapping document in listing 2.2.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

To add a mapping document to your solution, you can simply add a new XML document and call it `Employee.hbm.xml`. Then, highlight the file in solution explorer, and look for the property named “Build Action” in the properties pane. Change this to “**Embedded Resource**”. This is an important step and shouldn’t be missed, as it allows NHibernate to find the mapping information.

Listing 2.2 ??? A simple Hibernate XML mapping

```
<?xml version="1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2" auto-import="true">
  <class name="HelloNHibernate.Employee, HelloNHibernate" lazy="false">
    <id name="id" access="field">
      <generator class="native" />
    </id>
    <property name="name" access="field" column="name"/>
    <many-to-one access="field" name="manager" column="manager" cascade="all"/>
  </class>
</hibernate-mapping>
```

The mapping document tells NHibernate that the `Employee` class is to be persisted to the `Employees` table, that the `id` field maps to a column named `id`, that the `name` field maps to a column named `name`, and that the property named `manager` is an association with many-to-one multiplicity that maps to a column named `ManagerId`. (Don’t worry about the other details for now.)

As you can see, the XML document isn’t difficult to understand. You can easily write and maintain it by hand. In chapter 3, we discuss a way of generating the XML file from comments embedded in the source code. Whichever method you choose, NHibernate has enough information to completely generate all the SQL statements that would be needed to insert, update, delete, and retrieve instances of the `Employee` class. You no longer need to write these SQL statements by hand.

NOTE

NHibernate has sensible defaults that minimize typing and a mature document type definition that can be used for auto-completion or validation in editors, including Visual Studio. You can even automatically generate metadata with various tools.

Whilst we’re on the subject of XML, now would be a good time to show you how to configure NHibernate.

1.2.6 Configuring Your Application

If you’ve created .NET applications that use `DataSets` or `DataReaders` to connect to a database, you may also be familiar with the concept of storing a `ConnectionString` in your `web.config` or `app.config` files. Configuring NHibernate is similar; we just add some connection information to the config file.

Start by right-clicking your `HelloNHibernate` project in the Solution Explorer, and selecting `Add -> New Item...` Then select “Application Configuration File” from the options. Hit ok, and this will add an `app.config` file to the project.

You should then copy the following XML into your file.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="nhibernate"
      type="System.Configuration.NameValueSectionHandler, System,
Version=1.0.3300.0,Culture=neutral, PublicKeyToken=b77a5c561934e089" />
  </configSections>
  <nhibernate />
</configuration>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        />
</configSections>
<nhibernate>
  <add key="hibernate.show_sql"
    value="false" />
  <add key="hibernate.connection.provider"
    value="NHibernate.Connection.DriverConnectionProvider" />
  <add key="hibernate.dialect"
    value="NHibernate.Dialect.MsSql2000Dialect" />
  <add key="hibernate.connection.driver_class"
    value="NHibernate.Driver.SqlClientDriver" />
  <add key="hibernate.connection.connection_string"
    value="Data Source=127.0.0.1;Initial Catalog=HelloNHibernate;Integrated
Security=SSPI;" />
</nhibernate>
</configuration>

```

There's quite a lot of it! But, remember NHibernate is *very* flexible and can be configured in many ways. Note that you may need to change the `hibernate.connection.connection_string` key at the bottom of the XML to connect to the database server on your development computer.

1.2.7 Updating an Employee

Before we run any code, we'll add one more function to demonstrate how NHibernate can update existing entities. We'll write some code to update our first `Employee` and, while we're at it, create a new `Employee` to be the manager of the the first, as shown in listing 2.3. Again, you should type this below the other functions in `Program.cs`.

Listing 2.3 Updating an employee

```

static void UpdateTobinAndAssignPierreAsManager()
{
    using (ISession session = OpenSession())
    {
        using (ITransaction transaction = session.BeginTransaction())
        {
            IQuery q = session.CreateQuery(
                "from Employee where name = 'Tobin Harris'");

            Employee tobin = q.List<Employee>()[0];
            tobin.name = "Tobin David Harris";

            Employee pierre = new Employee();
            pierre.name = "Pierre Henri Kuate";

            tobin.manager = pierre;
            session.Flush();
            transaction.Commit();

            Console.WriteLine("Updated Tobin and added Pierre");
        }
    }
}

```

Behind the scenes, NHibernate would run four SQL statements inside the same transaction:

```

select e.id, e.name, e.manager
from Employee e
where e.id = 1

insert into Employees (name, manager)

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

values ('Pierre Henri Kuate', null)

declare @newId int
select @newId = scope_identity()

update Employees
set name = 'Tobin David Harris', manager = @newId
where id = 1

```

Notice how NHibernate detected the modification to the `name` and `manager` properties of the first `Employee` (Fred) and automatically updated the database. We've taken advantage of a NHibernate feature called *automatic dirty checking*: This feature saves us the effort of explicitly asking NHibernate to update the database when we modify the state of an object. Similarly, you can see that the new `Employee` (Bill) was saved when it was associated with the first `Employee`. This feature is called *cascading save*: It saves us the effort of explicitly making the new object persistent by calling `Save()`, as long as it's reachable by an already persistent object (Bill). Also notice that the ordering of the SQL statements isn't the same as the order in which we set fields of the object. NHibernate uses a sophisticated algorithm to determine an efficient ordering that avoids database foreign key constraint violations but is still sufficiently predictable to the user. This feature is called *transactional write-behind*.

1.2.8 – Running the Program

Before finally running our example, we need to write some code to run all these functions in the right order. Modify your `Program.cs` Main method to look like this:

```

static void Main()
{
    CreateEmployeeAndSaveToDatabase();
    UpdateTobinAndAssignPierreAsManager();
    LoadEmployeesFromDatabase();

    Console.WriteLine("Press any key to exit...");
    Console.ReadKey();
}

```

If we run “Hello World”, it prints

```

Saved Tobin to the database
Updated Tobin and added Pierre

2 employees found:
'Hello World!', said Pierre Henri Kuate.
'Hello World!', said Tobin David Harris.
Press any key to exit...

```

This is as far as we'll take the “Hello World” application. Now that we finally have some code under our belt, we'll take a step back and present an overview of Hibernate's main APIs.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.2 Understanding the Architecture

The programming interfaces are the first thing you have to learn about NHibernate in order to use it in the persistence layer of your application. A major objective of API design is to keep the interfaces between software components as narrow as possible. In practice, however, ORM APIs aren't especially small. Don't worry, though; you don't have to understand all the NHibernate interfaces at once.

Figure 2.1 illustrates the roles of the most important NHibernate interfaces in the business and persistence layers. We show the business layer above the persistence layer, since the business layer acts as a client of the persistence layer in a traditionally layered application. Note that some simple applications might not cleanly separate business logic from persistence logic; that is okay—it merely simplifies the diagram.

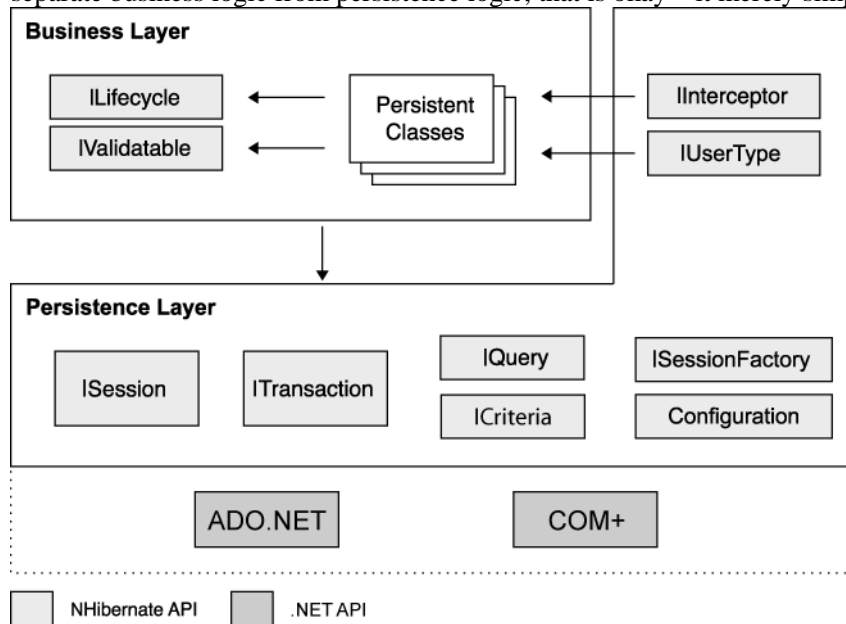


Figure 2.1 High-level overview of the NHibernate API in a layered architecture

The NHibernate interfaces shown in figure 2.1 may be approximately classified as follows:

- Interfaces called by applications to perform basic CRUD and querying operations (that is: Create, Retrieve, Update and Delete). These interfaces are the main point of dependency of application business/control logic on NHibernate. They include `ISession`, `ITransaction`, `IQuery` and `ICriteria`.
- Interfaces called by application infrastructure code to configure NHibernate, most importantly the `Configuration` class.
- Callback interfaces that allow the application to react to events occurring inside NHibernate, such as `IInterceptor`, `ILifecycle`, and `IValidatable`.
- Interfaces that allow extension of NHibernate's powerful mapping functionality, such as `IUserType`, `ICompositeUserType`, and `IIdentifierGenerator`. These interfaces are implemented by application infrastructure code (if necessary).

NHibernate makes use of existing .NET APIs, including ADO.NET and its `ITransaction` API. ADO.NET provides a rudimentary level of abstraction of functionality common to relational databases, allowing almost any database with an ADO.NET driver to be supported by NHibernate.

In this section, we don't cover the detailed semantics of NHibernate API methods, just the role of each of the primary interfaces. We will progressively cover API methods in the next chapters. You can find a complete

and succinct description of these interfaces in NHibernate's reference documentation. Let's take a brief look at each interface in turn.

2.2.1 The core interfaces

The five core interfaces are listed below and used in just about every NHibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

ISession interface

The `ISession` interface is the primary interface used by NHibernate applications, it exposes NHibernate's methods for finding, saving, updating and deleting objects. An instance of `ISession` is lightweight and is inexpensive to create and destroy. This is important because your application will need to create and destroy sessions all the time, perhaps on every ASP.NET page request. NHibernate sessions are not thread safe and should by design be used by only one thread at a time. This is discussed in further details in future chapters.

The NHibernate notion of a *session* is something between *connection* and *transaction*. It may be easier to think of a session as a cache or collection of loaded objects relating to a single unit of work. NHibernate can detect changes to the objects in this unit of work. We sometimes call the `ISession` a *persistence manager* because it's also the interface for persistence-related operations such as storing and retrieving objects. Note that a NHibernate session has nothing to do with an ASP.NET session. When we use the word *session* in this book, we mean the NHibernate session.

We describe the `ISession` interface in detail in chapter 4, section 4.2, "The persistence manager."

ISessionFactory interface

The application obtains `ISession` instances from an `ISessionFactory`. Compared to the `ISession` interface, this object is much less exciting.

The `ISessionFactory` is certainly not lightweight! It's intended to be shared among many application threads. There is typically a single instance of `ISessionFactory` for the whole application—created during application initialization, for example. However, if your application accesses multiple databases using NHibernate, you'll need a `SessionFactory` for each database.

The `SessionFactory` caches generated SQL statements and other mapping metadata that NHibernate uses at runtime. It can also hold cached data that has been read in one unit of work, and which may be reused in a future unit of work or session. This is possible if you configure class and collection mappings to use the *second-level cache*.

Configuration interface

The `Configuration` object is used to configure NHibernate. The application uses a `Configuration` instance to specify the location of mapping documents, and to set NHibernate-specific properties before creating the `ISessionFactory`.

Even though the `Configuration` interface plays a relatively small part in the total scope of a NHibernate application, it's the first object you'll meet when you begin using NHibernate. Section 2.2 covers the problem of configuring NHibernate in some detail.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

ITransaction interface

The `ITransaction` interface is shown in figure 2.1, next to the `ISession` interface. The `ITransaction` interface is an optional API. NHibernate applications may choose not to use this interface, instead managing transactions in their own infrastructure code. A NHibernate `ITransaction` abstracts application code from the underlying transaction implementation—which might be an ADO.NET transaction or any kind of manual transaction—allowing the application to control transaction boundaries via a consistent API. This helps to keep NHibernate applications portable between different kinds of execution environments and containers.

We use the NHibernate `ITransaction` API throughout this book. Transactions and the `ITransaction` interface are explained in chapter 5.

IQuery and ICriteria interfaces

The `IQuery` interface gives you powerful ways of performing queries against the database, whilst also controlling how the query is executed. It is the basic interface used for fetching data using NHibernate. Queries are written in HQL or in the native SQL dialect of your database. An `IQuery` instance is lightweight and can't be used outside the `ISession` that created it. It is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

The `ICriteria` interface is very similar; it allows you to create and execute object-oriented criteria queries.

To help make application code less verbose, NHibernate provides some helpful shortcut methods on the `ISession` interface that let you invoke a query in one line of code. Behind the scenes it is using the `IQuery` interface. We won't use these shortcuts in the book; instead, we'll always use the `IQuery` interface.

We describe the features of the `IQuery` interface in chapter 7, where you'll learn how to use it in your applications. Now that we've introduced you to the main APIs needed to write real-world NHibernate applications, the next section introduces some more advanced features. After that we'll dive into how NHibernate is configured, and also how you can set up logging to view what NHibernate is doing behind the scenes (a great way of seeing NHibernate in action, if you'll excuse the pun!)

2.2.2 Callback Interfaces

Callback interfaces allow the application to receive a notification when something interesting happens to an object—for example, when an object is loaded, saved, or deleted. NHibernate applications don't need to implement these callbacks, but they're useful for implementing certain kinds of generic functionality, such as creating audit records.

The `ILifecycle` and `IValidatable` interfaces allow a persistent object to react to events relating to its own *persistence lifecycle*. The persistence lifecycle is encompassed by an object's CRUD operations (when it is created, retrieved, updated or deleted).

Note: The Hibernate team was heavily influenced by other ORM solutions that have similar callback interfaces. Later, they realized that having the persistent classes implement Hibernate-specific interfaces probably isn't a good idea, because doing so pollutes our persistent classes with non-portable code. Since these interfaces are deprecated, we don't discuss them in this book.

The `IInterceptor` interface was introduced to allow the application to process callbacks without forcing the persistent classes to implement NHibernate-specific APIs. Implementations of the `IInterceptor` interface are passed to the persistent instances as parameters. We'll discuss an example in chapter 8.

2.2.3 Types

A fundamental and very powerful element of the architecture is NHibernate's notion of a `Type`. A NHibernate `Type` object maps a .NET type to a database column type (actually, the type may span multiple columns). All persistent properties of persistent classes, including associations, have a corresponding NHibernate type. This design makes NHibernate extremely flexible and extensible as each RDBMS has a different set of mapping to .NET types.

There is a rich range of built-in types, covering all .NET primitives and many CLR classes, including types for `System.DateTime`, `System.Enum`, `byte[]`, and `Serializable` classes.

Even better, NHibernate supports user-defined *custom types*. The interfaces `IUserType`, `ICompositeUserType` and `IParameterizedType` are provided to allow you to add your own types and `IUserCollectionType` for your collection types. You can use this feature to allow commonly used application classes such as `Address`, `Name`, or `MonetaryAmount` to be handled conveniently and elegantly. Custom types are considered a central feature of NHibernate, and you're encouraged to put them to new and creative uses!

We explain NHibernate types and user-defined types in chapter 6, section 6.1, "Understanding the NHibernate type system." We'll now go on to list some of the more low-level interfaces. You may not need to use or understand all of these, but knowing they exist might give you extra flexibility when it comes to designing your applications.

2.2.4 Extension Interfaces

Much of the functionality that NHibernate provides is configurable, allowing you to choose between certain built-in strategies. When the built-in strategies are insufficient, NHibernate will usually let you plug in your own custom implementation by implementing an interface. Extension points include:

- Primary key generation (`IIdentifierGenerator` interface)
- SQL dialect support (`Dialect` abstract class)
- Caching strategies (`ICache` and `ICacheProvider` interfaces)
- ADO.NET connection management (`IConnectionProvider` interface)
- Transaction management (`ITransactionFactory` and `ITransaction` interfaces)
- ORM strategies (`IClassPersister` interface hierarchy)
- Property access strategies (`IPropertyAccessor` interface)
- Proxy creation (`IProxyFactory` interface)

NHibernate ships with at least one implementation of each of the listed interfaces, so you don't usually need to start from scratch if you wish to extend the built-in functionality. The source code is available for you to use as an example for your own implementation.

You should now have an awareness of the various APIs and interfaces that NHibernate gives us. Luckily you won't need them all! In fact, for simple applications you might only need the Configuration and `ISession` interfaces, as shown in our "Hello World" example. However, before you can start to use NHibernate in your own applications, you'll need to have some understanding of how NHibernate is configured. That is what we discuss next.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.3 Basic configuration

NHibernate can be configured to run in almost any .NET application and development environment. Generally, NHibernate is used in two- and three-tiered client/server applications, with NHibernate deployed only on the server. The client application is usually a web browser, but Windows client applications aren't uncommon either. Although we concentrate on multi-tiered web applications in this book, we will cover Windows applications when needed.

The first thing you must do is start NHibernate. In practice, doing so is very easy: You have to create an `ISessionFactory` instance from a `Configuration`.

2.3.1 Creating a SessionFactory

In order to create an `ISessionFactory` instance, you first create a single instance of `Configuration` during application initialization and use it to set the database access and mapping information. Once configured, the `Configuration` instance is used to create the `SessionFactory`. After the `SessionFactory` is created, you can discard the `Configuration` class.

In our previous samples, we used a `MySessionFactory` static property to create `ISession` instances. Here is its implementation:

```
private static ISessionFactory sessionFactory = null;
public static ISessionFactory MySessionFactory
{
    get
    {
        if(sessionFactory == null) |1
        {
            Configuration cfg = new Configuration();
            cfg.Configure();
            cfg.AddInputStream(
                HbmSerializer.Default.Serialize(typeof(Employee)) ); |2
            // OR: cfg.AddXmlFile("Employee.hbm.xml"); |3
            sessionFactory = cfg.BuildSessionFactory();
        }
        return sessionFactory;
    }
}
```

#1 Done only once (at the first access)
#2 When using `NHibernate.Mapping.Attributes`
#3 When using a XML mapping file

The location of the mapping file, `Employee.hbm.xml`, is relative to the application current directory. In this example, we also use a XML file to set all other configuration options (which might have been set before by application code or in the application configuration file).

Method chaining

Method chaining is a programming style supported by many NHibernate interfaces (they are also called *fluent interfaces*). This style is more popular in Smalltalk than in .NET and is considered by some people to be less readable and more difficult to debug than the more accepted .NET style. However, it's very convenient in most cases.

Most .NET developers declare setter or adder methods to be of type `void`, meaning they return no value. In Smalltalk, which has no `void` type, setter or adder methods usually return the receiving object. This would allow us to rewrite the previous code example as follows:

Following code is part of the sidebar

```
ISessionFactory sessionFactory = new Configuration()
    .Configure()
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
.AddXmlFile( "Employee.hbm.xml" )
.BuildSessionFactory();
```

Notice that we didn't need to declare a local variable for the `Configuration`. We use this style in some code examples; but if you don't like it, you don't need to use it yourself. If you *do* use this coding style, it's better to write each method invocation on a different line. Otherwise, it might be difficult to step through the code in your debugger.

By convention, NHibernate XML mapping files are named with the `.hbm.xml` extension. Another convention is to have one mapping file per class, rather than have all your mappings listed in one file (which is possible but considered bad style). Our "Hello World" example had only one persistent class. But let's assume we have multiple persistent classes, with an XML mapping file for each. Where should we put these mapping files?

The NHibernate documentation recommends that the mapping file for each persistent class is placed in the same directory as that class file. For instance, the mapping file for the `Employee` class would be placed in a file named `Employee.hbm.xml` in the same directory as the file `Employee.cs`. If we had another persistent class, it would be defined in its own mapping file. We suggest that you follow this practice and that you load multiple mapping files by calling `AddXmlFile()`.

It is even possible to embed XML mapping files inside assemblies (your compiled.dll or .exe files). You just have to tell to the compiler that each of these files is an `Embedded Resource`; most IDEs allow you to specify this option. Then, you can use the method `AddClass()`, passing a the class's type as the parameter:

```
ISessionFactory sessionFactory = new Configuration()
    .Configure()
    .AddClass( typeof(Model.Item) )
    .AddClass( typeof(Model.User) )
    .AddClass( typeof(Model.Bid) )
    .BuildSessionFactory();
```

The `AddClass()` method assumes that the name of the mapping file ends with the `.hbm.xml` extension and is embedded in the same assembly as the mapped class file.

If you want to add all mapped classes (with .NET attributes) in an assembly, you can use an overload of the method `HbmSerializer.Serialize()`; or if you want to add all mapping files embedded in an assembly, you can use the method `AddAssembly()`:

```
ISessionFactory sessionFactory = new Configuration()
    .Configure()
    .AddInputStream( // .NET Attributes
        HbmSerializer.Default.Serialize(typeof(Model.Item).Assembly) )
    .AddAssembly( typeof(Model.Item).Assembly ) // XML
    .BuildSessionFactory();
```

Note that it is error-prone to use assemblies' names (like `"NHibernate.Auction"`). That's why we use one class's type to retrieve directly the assembly containing the embedded mapping files.

Why NHibernate says that it doesn't know my class?

A common issue when starting to use NHibernate is making sure that all your mappings are sent to NHibernate; if you miss one, you will get an exception. When building the session factory, it will be a `MappingException` with a comment containing "... refers to an unmapped class: YourClass". When executing a query, it will be a `QueryException` with a comment like "possibly an invalid or unmapped class name was used in the query".

To solve this issue, the first step is to set `log4net` to INFO level (you will learn how to do that in section 3.3.2). Then, read the log to make sure that NHibernate read your mappings; you should find a message like `"Mapping class: Namespace.YourClass -> YourClass"`. If it is not the case, then check your initialization code to make sure that you really send the mappings.

If you use `AddAssembly()`, make sure that they are really embedded in your assembly.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

On the other hand, you may get a `DuplicateMappingException` if you add a mapping many times. For example, avoid adding both the XML and the attributes-based mapping.

We've demonstrated the creation of a single `SessionFactory`, which is all that most applications need. If another `ISessionFactory` instance is needed—in the case of multiple databases, for example—repeat the process. Each `SessionFactory` is then available for one database and ready to produce `ISession` instances to work with that particular database and a set of class mappings. Once we have our `SessionFactory`, we can go on to create sessions, and start loading and saving objects.

Of course, there is more to configuring NHibernate than just pointing to mapping documents. You also need to specify how database connections are to be obtained, along with various other settings that affect the behavior of NHibernate at runtime. The multitude of configuration properties may appear overwhelming (a complete list appears in the NHibernate documentation), but don't worry; most define reasonable default values, and only a handful are commonly required.

To specify configuration options, you may use any of the following techniques:

- Pass an instance of `System.Collections.IDictionary` to `Configuration.SetProperties()` or use `Configuration.SetProperty()` for each property (or manipulate the collection `Configuration.Properties` directly).
- Set all properties in the application configuration file (that is `App.config` or `Web.config`).
- Include `<property>` elements in a XML file called `hibernate.cfg.xml` in the current directory.

The first option is rarely used except for quick testing and prototypes, but most applications need a fixed configuration file. Both the application configuration file and the `hibernate.cfg.xml` files provide the same function: to configure NHibernate. Which file you choose to use depends on your syntax preference. `hibernate.cfg.xml` is the file name chosen by convention. Actually, you can use any file name (like `NHibernate.config`, as `.config` files are automatically protected by ASP.NET when deployed) and provide this file name to the method `Configure()`. It's even possible to mix both options and have different settings for development and deployment.

A rarely used alternative option is to allow the application to provide an ADO.NET `IDbConnection` when it opens an NHibernate `ISession` from the `SessionFactory` (for example, by calling `sessionFactory.OpenSession(myConnection)`). Using this option means that you don't have to specify any database connection properties (the other properties are still required). We don't recommend this approach for new applications that can be configured to use the environment's database connection infrastructure.

Of all the configuration options, database connection settings are the most important because, without them, NHibernate won't know how to correctly talk to the database.

2.3.2 Configuration of the ADO.NET database access

Most of the time, the application is responsible for obtaining ADO.NET connections. NHibernate is part of the application, so it's responsible for getting these connections. You tell NHibernate how to get (or create new) ADO.NET connections.

Figure 2.2 shows how .NET applications interact with ADO.NET. Without NHibernate, the application code usually receives an ADO.NET connection from the connection pool (which is configured transparently) and uses it to execute SQL statements.

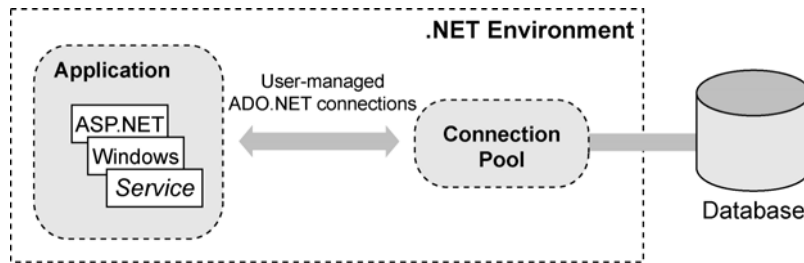


Figure 2.2 Direct access to ADO.NET connections

With NHibernate, the picture changes: It acts as a client of the ADO.NET and its connection pool, as shown in figure 2.3. The application code uses the NHibernate `ISession` and `IQuery` APIs for persistence operations and only has to manage database transactions, ideally using the NHibernate `ITransaction` API.

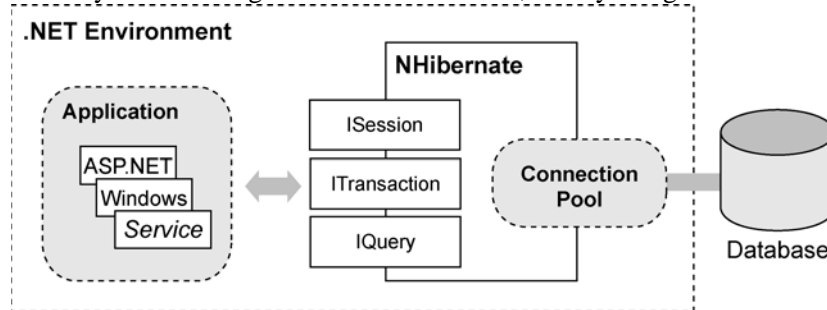


Figure 2.3 NHibernate managing the database access

Configuration of NHibernate using `hibernate.cfg.xml`

In listing 2.1, we use a file named `hibernate.cfg.xml` to configure NHibernate to access a Microsoft SQL Server 2000 database.

Listing 2.1 Using `hibernate.cfg.xml` to configure Nhibernate

```
<?xml version="1.0" ?>
<hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
  <session-factory>
    <property name="connection.provider">
      NHibernate.Connection.DriverConnectionProvider
    </property>
    <property name="dialect">
      NHibernate.Dialect.MsSql2000Dialect
    </property>
    <property name="connection.driver_class">
      NHibernate.Driver.SqlClientDriver
    </property>
    <property name="connection.connection_string">
      Data Source=(local); Initial Catalog=nhibernate;
      Integrated Security=SSPI
    </property>
  </session-factory>
</hibernate-configuration>
```

This code's lines specify the following information:

- The name of the .NET class implementing the `IConnectionProvider` for NHibernate; here, we are using the default one.
- The name of the .NET class implementing the `Dialect` which enables certain platform dependent features. Despite the ANSI standardization effort, SQL is implemented differently by various databases vendors. So, you must specify a `Dialect`. NHibernate includes built-in support for most popular SQL databases, and new dialects may be defined easily.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

- The name of the .NET class implementing the ADO.NET `Driver`. Note that, since NHibernate 1.2, when using the partial name of a driver which is in the global assembly cache (GAC), you have to add a `<qualifyAssembly>` element in the application configuration file to specify its fully qualified name so that NHibernate can successfully load it.
- The `ConnectionString`: the string used to create a database connection as defined by ADO.NET.

Note that these names (except the `ConnectionString`) should be fully qualified type names; they aren't here because they are implemented in `NHibernate.dll` library which is where the .NET framework will look for non-fully qualified types when NHibernate will try to load them.

Starting NHibernate

How do you start NHibernate with these properties? You declared the properties in a file named `hibernate.cfg.xml`, so you need only place this file in the application current directory. It will be automatically detected and read when you create a `Configuration` object and call its `Configure()` method.

Let's summarize the configuration steps you've learned so far (this is a good time to download and install NHibernate):

1. If the ADO.NET data provider of your database is not yet installed, download and install it; it is usually available from the database vendor web site. If you're using SQL Server, then you can skip this step.
2. Add `log4net.dll` as reference to your project. This is optional but recommended.
3. Decide which database access properties will be needed NHibernate.
4. Let the `Configuration` know about these properties by placing them in a `hibernate.cfg.xml` file in the current directory.
5. Create an instance of `Configuration` in your application, call the method `Configure()`, load the mapped classes (with .NET attributes) using `HbmSerializer.Default.Serialize()` and `AddInputStream()`, load the XML mapping files using either `AddAssembly()`, `AddClass()` or `AddXmlFile()`. Build an `ISessionFactory` instance from the `Configuration` by calling `BuildSessionFactory()`.
6. Remember to close the instance of `ISessionFactory` (using `MySessionFactory.Close()`) when you are done using NHibernate. Most of the time, you will do it while closing your application.

There are a few more steps when using COM+ Enterprise Services; we will learn more about them in chapter 6. Don't worry, NHibernate code can be easily integrated to COM+ as only few additions are required.

You should now have a running NHibernate system. Create and compile a persistent class (the initial `Employee`, for example), add a reference to `NHibernate.dll` and other required libraries to your project, put a `hibernate.cfg.xml` file in the application current directory, and build an `ISessionFactory` instance.

The next section covers advanced NHibernate configuration options. Some of them are recommended, such as logging executed SQL statements for debugging or using the convenient XML configuration file instead of plain properties. However, you may safely skip this section and come back later once you have read more about persistent classes in chapter 3.

2.4 Advanced configuration settings

When you finally have a NHibernate application running, it's well worth getting to know all the NHibernate configuration parameters. These parameters let you optimize the runtime behavior of NHibernate, especially by tuning the ADO.NET interaction (for example, using ADO.NET batch updates).

We won't bore you with these details now; the best source of information about configuration options is the NHibernate reference documentation. In the previous section, we showed you the options you'll need to get started.

However, there is one parameter that we *must* emphasize at this point. You'll need it continually whenever you develop software with NHibernate. Setting the property `show_sql` to the value `true` enables logging of all generated SQL to the console. You'll use it for troubleshooting, performance tuning, and just to see what's going on. It pays to be aware of what your ORM layer is doing—that's why ORM doesn't hide SQL from developers.

So far, we've assumed that you specify configuration parameters using a `hibernate.cfg.xml` file or programmatically using the collection `Configuration.Properties`. You may also specify these parameters using the application configuration file (`web.config`, `app.config` etc).

2.4.1 Using the application configuration file

You can use the application configuration file (as demonstrated in listings 2.2 and 2.3) to fully configure an `ISessionFactory` instance. It can either contain configuration parameters using a `<nhibernate>` section, or the same content as `hibernate.cfg.xml` using a `<hibernate-configuration>` section. Many users prefer to centralize the configuration of NHibernate in this way instead of adding parameters to the `Configuration` in application code.

Listing 2.2 App.config configuration file using <nhibernate>

```
<?xml version="1.0" ?>
<configuration>
  <configSections>
    <section name="nhibernate" |1
      type="System.Configuration.NameValueSectionHandler, |1
      System, Version=1.0.5000.0, Culture=neutral, |1
      PublicKeyToken=b77a5c561934e089" /> |1
    <section name="log4net" |1
      type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
  </configSections>

  <nhibernate>
    <add |2
      key="hibernate.connection.provider" |2
      value="NHibernate.Connection.DriverConnectionProvider" |2
    /> |2
    <add |2
      key="hibernate.dialect" |2
      value="NHibernate.Dialect.MsSql2000Dialect" |2
    /> |2
    <add |2
      key="hibernate.connection.driver_class" |2
      value="NHibernate.Driver.SqlClientDriver" |2
    /> |2
    <add |2
      key="hibernate.connection.connection_string" |2
      value="initial catalog=nhibernate;Integrated Security=SSPI" |2
    /> |2
  </nhibernate>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    <!-- log4net configuration settings here... --> | 3
</configuration>
#1 NHibernate section declaration
#2 Property specifications
#3 Log4net settings should be there

```

NHibernate section is declared in #1 as a series of key/value entries. In #2, the key is the name of the property to set. We will learn about log4net #3 in the next section.

However, it is recommended to use a `<hibernate-configuration>` section:

Listing 2.x App.config configuration file using `<hibernate-configuration>`

```

<?xml version="1.0" ?>
<configuration>
  <configSections>
    <section name="hibernate-configuration" | 1
    type="NHibernate.Cfg.ConfigurationSectionHandler,NHibernate" /> | 1
    <section name="log4net"
    type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
  </configSections>

  <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory> | 2
      <property name="connection.provider"> | 2
        NHibernate.Connection.DriverConnectionProvider | 2
      </property> | 2
      <property name="dialect"> | 2
        NHibernate.Dialect.MsSql2000Dialect | 2
      </property> | 2
      <property name="connection.driver_class"> | 2
        NHibernate.Driver.SqlClientDriver | 2
      </property> | 2
      <property name="connection.connection_string"> | 2
        Initial Catalog=nhibernate;Integrated Security=SSPI | 2
      </property> | 2
    </session-factory> | 2
  </hibernate-configuration> | 2

  <!-- log4net configuration settings here... --> | 3
</configuration>
#1 NHibernate section declaration
#2 Property specifications
#3 Log4net settings should be there

```

Now, #1 declares a `<hibernate-configuration>`, like in `hibernate.cfg.xml`, which is based on the schema `nhibernate-configuration.xsd`. In #2, the value is inside the tag `<property>`.

This way is far more elegant and powerful as you can also specify assemblies/mapping documents; and you can configure an IDE like Visual Studio to provide IntelliSense inside the `<hibernate-configuration>` section: Just copy the configuration schema file (`nhibernate-configuration.xsd`) in the sub-directory `\Common7\Packages\schemas\xml\` of Visual Studio installation directory. You can also copy the mapping schema file (`nhibernate-mapping.xsd`) to have IntelliSense when editing mapping files. You can find these files in NHibernate's source code.

Note that you can use a `<connectionStrings>` configuration file element to define a connection string and then give its name to NHibernate using the property `hibernate.connection.connection_string_name`.

Now you can initialize NHibernate using:

```

ISessionFactory sessionFactory = new Configuration()
    .Configure()
    .BuildSessionFactory();

```

Wait—how did NHibernate know where the configuration file was located?

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

When `Configure()` was called, NHibernate first searched for these information in the application configuration file, then in a file named `hibernate.cfg.xml` in the current directory. If you wish to use a different filename or have NHibernate look in a subdirectory, you must pass a path to the `Configure()` method:

```
ISessionFactory sessionFactory = new Configuration()  
    .Configure("NHibernate.config")  
    .BuildSessionFactory();
```

Using an XML configuration file is certainly more comfortable than using programmatic configuration. The fact that you can have the class mapping files externalized from the application's source (even if it would be only in a startup helper class) is a major benefit of this approach. You can, for example, use different sets of mapping files (and different configuration options), depending on your database and environment (development or production), and switch them programmatically.

If you have both an application configuration file and `hibernate.cfg.xml` in the current directory, the settings of the application configuration file will be used.

Note that the `ISessionFactory` can be given a name. This name is specified as an attribute like this: `<session-factory name="MySessionFactory">`. NHibernate uses this name to identify this instance after creation. You can use the static method: `NHibernate.Impl.SessionFactoryObjectFactory.GetNamedInstance()` to retrieve it. This feature may be useful when sharing a `SessionFactory` between loosely coupled components. However, it is seldom used because, most of the time, it is better to hide NHibernate behind the persistence layer.

Now that we have a functional NHibernate application, we will start encountering runtime errors. To ease the debug process, we need to log NHibernate operations.

2.4.2 Logging

NHibernate (and many other ORM implementations) defers the execution of SQL statements. An `INSERT` statement isn't usually executed when the application calls `ISession.Save()`; an `UPDATE` isn't immediately issued when the application calls `Item.AddBid()`. Instead, the SQL statements are usually issued at the end of a transaction. This behavior is called *write-behind*, as we mentioned earlier.

This fact is evidence that tracing and debugging ORM code is sometimes nontrivial. In theory, it's possible for the application to treat NHibernate as a black box and ignore this behavior. Certainly, the NHibernate application cannot detect this write-behind (at least, not without resorting to direct ADO.NET calls).

However, when you find yourself troubleshooting a difficult problem, you need to be able to see *exactly* what's going on inside NHibernate. Since NHibernate is open source, you can easily step into the NHibernate code. Occasionally, doing so helps a great deal! However, especially in the face of write-behind behavior, debugging NHibernate can quickly get you lost. You can use logging to get a view of NHibernate's internals.

We've already mentioned the `show_sql` configuration parameter, which is usually the first port of call when troubleshooting. Sometimes the SQL alone is insufficient; in that case, you must dig a little deeper.

NHibernate logs all interesting events using the open source library log4net. To see any output from log4net, you'll need to add some information in your application configuration file. The example in listing 2.4 directs all log messages to the console:

Listing 2.4 Basic configuration of log4net

```
<?xml version="1.0" ?>  
<configuration>  
  <configSections>  
    <section
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        name="log4net"
        type="log4net.Config.Log4NetConfigurationSectionHandler,log4net"
    />
</configSections>

<log4net>
  <appender name="ConsoleAppender"
    type="log4net.Appender.ConsoleAppender, log4net">
    <layout type="log4net.Layout.PatternLayout, log4net">
      <param name="ConversionPattern" value="%m" />
    </layout>
  </appender>
  <root>
    <priority value="WARN" />
    <appender-ref ref="ConsoleAppender" />
  </root>
</log4net>
</configuration>

```

You can easily merge this file in the listing 2.3. With this configuration, you won't see many log messages at runtime. Replacing the priority value from **WARN** to **INFO** or **DEBUG** will reveal the inner workings of NHibernate. Make sure you don't do this in a production environment—writing the log will be much slower than the actual database access. We will not give more details about log4net configuration. Feel free to read its documentation.

In this section, we talked about database access configuration. This configuration is useless if NHibernate doesn't know how to manipulate our entities. The next section covers NHibernate mapping.

2.5 Summary

In this chapter, we took a high-level look at NHibernate and its architecture after running a simple "Hello World" example. You also saw how to configure NHibernate in various environments and with various techniques.

The Configuration and SessionFactory interfaces are the entry points to NHibernate for applications running in both WinForms and ASP.NET environments.

Hibernate can be integrated into almost every .NET environment, be it a Console application, an ASP.NET application, or a fully managed three-tiered client/server application. The most important elements of a NHibernate configuration are the database resources (connection configuration), the transaction strategies, and, of course, the XML-based mapping metadata.

NHibernate's configuration interfaces have been designed to cover as many usage scenarios as possible while still being easy to understand. Usually, a few modifications to your .config file and one line of code are enough to get NHibernate up and running.

None of this is much use without some persistent classes and their XML mapping documents. The next chapter is dedicated to writing and mapping persistent classes. You'll soon be able to store and retrieve persistent objects in a real application with a nontrivial object/relational mapping.

Writing and Mapping classes

This chapter covers

- POCO basics for rich domain models
- The concept of object identity and its mapping
- Mapping class inheritance
- Associations and collections mappings

The “Hello World” example in chapter 2 introduced you to NHibernate; however, it isn’t very useful for understanding the requirements of real-world applications with complex data models. For the rest of the book, we’ll use a much more sophisticated example application—an online auction system—to demonstrate NHibernate.

In this chapter, we start our discussion of the application by introducing a programming model for persistent classes. Designing and implementing the persistent classes is a multi-step process that we’ll examine in detail.

This chapter is quite long as we cover many interconnected topics. First, you’ll learn how to identify the *business objects* (or *entities*) of a problem domain. We create a conceptual model of these entities and their attributes, called a *domain model*. We implement this domain model in C# by creating a persistent class for each entity, and we’ll spend some time exploring exactly what these .NET classes should look like.

We then define *mapping metadata* to tell NHibernate how these classes and their properties relate to database tables and columns. We already covered the basis of this step in chapter 2. In this chapter, we give an in-depth presentation of the mapping techniques for fine-grained classes, object identity, inheritance, and associations. This chapter therefore provides the beginnings of a solution to the first generic problems of ORM listed in chapter 1, section 1.3.1 For example, how do we map our fine-grained objects to our simple tables, or how do we map inheritance hierarchies to our tables?

We’ll start by introducing the example application.

3.1 The CaveatEmptor application

The CaveatEmptor online auction application demonstrates ORM techniques and NHibernate functionality; you can download the source code for the entire working application from the web site <http://caveatemptor.hibernate.org/>. The application will have a console-based user interface. We won’t pay much attention to the user interface; we’ll concentrate on the data access code. In chapter 8, we discuss the changes that would be necessary if we were to perform all business logic and data access from a separate business-tier. And in chapter 10, we will discuss many solutions to common issues which arise when integrating NHibernate in Windows and Web applications.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

But, let's start at the beginning. In order to understand the design issues involved in ORM, let's pretend the CaveatEmptor application doesn't yet exist, and that we're building it from scratch. Our first task would be *analysis*.

3.1.1 Analyzing the business domain

A software development effort begins with analysis of the problem domain (assuming that no legacy code or legacy database already exists).

At this stage, you, with the help of problem domain experts, identify the main *entities* that are relevant to the software system. Entities are usually notions understood by users of the system: **Payment**, **Customer**, **Order**, **Item**, **Bid**, and so forth. Some entities might be abstractions of less concrete things the user thinks about (for example, **PricingAlgorithm**), but even these would usually be understandable to the user. All these entities are found in the conceptual view of the business, which we sometimes call a *business model*. Developers of object-oriented software analyze the business model and create an object model, still at the conceptual level (no C# code). This object model may be as simple as a mental image existing only in the mind of the developer, or it may be as elaborate as a UML class diagram (as in figure 3.1) created by a CASE (Computer-Aided Software Engineering) tool like Microsoft Visio, Sparx Systems Enterprise Architect or UMLet.

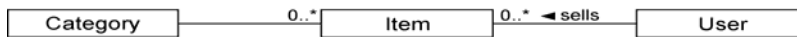


Figure 3.1 A class diagram of a typical online auction object model

This simple model contains entities that you're bound to find in any typical auction system: **Category**, **Item**, and **User**. The entities and their relationships (and perhaps their attributes) are all represented by this model of the problem domain. We call this kind of model—an object-oriented model of entities from the problem domain, encompassing only those entities that are of interest to the user—a *domain model*. It's an abstract view of the real world. We refer to this model when we implement our persistent .NET classes.

Let's examine the outcome of our analysis of the problem domain of the CaveatEmptor application.

3.1.2 The CaveatEmptor domain model

The CaveatEmptor site auctions many different kinds of items, from electronic equipment to airline tickets. Auctions proceed according to the "English auction" model: Users continue to place bids on an item until the bid period for that item expires, and the highest bidder wins.

In any store, goods are categorized by type and grouped with similar goods into sections and onto shelves. Clearly, our auction catalog requires some kind of hierarchy of item categories. A buyer may browse these categories or arbitrarily search by category and item attributes. Lists of items appear in the category browser and search result screens. Selecting an item from a list will take the buyer to an item detail view.

An auction consists of a sequence of bids. One particular bid is the winning bid. User details include name, login, address, email address, and billing information.

A *web of trust* is an essential feature of an online auction site. The web of trust allows users to build a reputation for trustworthiness (or untrustworthiness). Buyers may create comments about sellers (and vice versa), and the comments are visible to all other users.

A high-level overview of our domain model is shown in figure 3.2. Let's briefly discuss some interesting features of this model.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

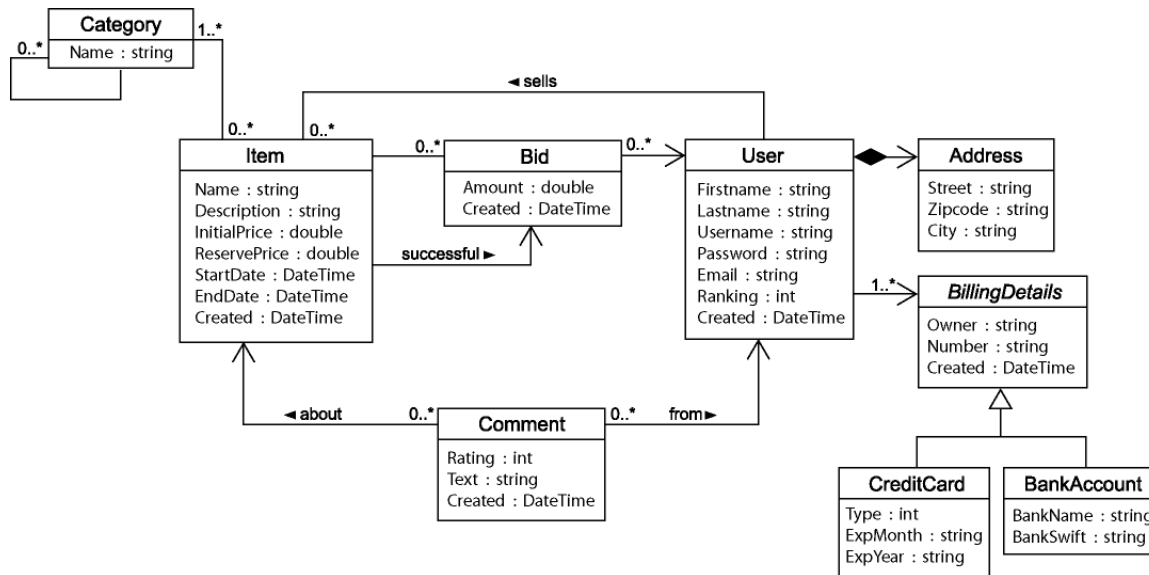


Figure 3.2 Persistent classes of the CaveatEmptor object model and their relationships

Each item may be auctioned only once, so we don't need to make *Item* distinct from the *Auction* entities. Instead, we have a single auction item entity named *Item*. Thus, *Bid* is associated directly with *Item*. Users can write *Comments* about other users only in the context of an auction; hence the association between *Item* and *Comment*. The *Address* information of a *User* is modeled as a separate class, even though the *User* may have only one *Address*. We do allow the user to have multiple *BillingDetails*. The various billing strategies are represented as subclasses of an abstract class (allowing future extension).

A *Category* might be nested inside another *Category*. This is expressed by a *recursive* association, from the *Category* entity to itself. Note that a single *Category* may have multiple child categories but at most one parent category. Each *Item* belongs to at least one *Category*.

The entities in a domain model should encapsulate state and behavior. For example, the *User* entity should define the name and address of a customer and the logic required to calculate the shipping costs for items (to this particular customer). Our domain model is a *rich* object model, with complex associations, interactions, and inheritance relationships. An interesting and detailed discussion of object-oriented techniques for working with domain models can be found in *Patterns of Enterprise Application Architecture* [Fowler 2003] or in *Domain-Driven Design* [Evans 2004].

However, in this book, we won't have much to say about business rules or about the *behavior* of our domain model. This is certainly not because we consider this an unimportant concern; rather, this concern is mostly orthogonal to the problem of persistence. It's the *state* of our entities that is persistent. So, we concentrate our discussion on how to best represent state in our domain model, not on how to represent behavior. For example, in this book, we aren't interested in how tax for sold items is calculated or how the system might approve a new user account. We're more interested in how the relationship between users and the items they sell is represented and made persistent.

Can you use ORM without a domain model?

We stress that object persistence with full ORM is most suitable for applications based on a rich domain model. If your application doesn't implement complex business rules or complex interactions between entities (or if you have few entities), you may not need a domain model. Many simple and some not-so-simple problems are perfectly suited to table-oriented solutions, where the application is designed around the database data model instead of around an object-oriented domain model, often

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

with logic executed in the database (stored procedures). However, the more complex and expressive your domain model, the more you will benefit from using NHibernate; it shines when dealing with the full complexity of object/relational persistence.

Now that we have a domain model, our next step is to implement it in C#. Let's look at some of the things we need to consider.

3.2 Implementing the domain model

Several issues typically must be addressed when you implement a domain model. For instance, how do you separate the business concerns from the cross-cutting concerns (such as transactions and even persistence)? What kind of persistence is needed: Do you need *automated* or *transparent* persistence? Do you have to use a specific programming model to achieve this? In this section, we examine these types of issues and how to address them in a typical NHibernate application.

Let's start with an issue that any implementation must deal with: the separation of concerns. The domain model implementation is usually a central, organizing component; it's reused heavily whenever you implement new application functionality. For this reason, you should be prepared to go to some lengths to ensure that concerns other than business aspects don't leak into the domain model implementation.

3.2.1 Addressing leakage of concerns

The domain model implementation is such an important piece of code that it shouldn't depend on other .NET APIs. For example, code in the domain model shouldn't perform Input/Output operations or call the database via the ADO.NET API. This allows you to reuse the domain model implementation virtually anywhere. Most importantly, it makes it easy to *unit test* the domain model (in NUnit, for example) outside of any application server or other managed environment.

We say that the domain model should be "concerned" only with modeling the business domain. However, there are other concerns, such as persistence, transaction management, and authorization. You shouldn't put code that addresses these *cross-cutting concerns* in the classes that implement the domain model. When these concerns start to appear in the domain model classes, we call this an example of *leakage of concerns*.

DataSet doesn't address this problem. It can't be regarded as a Domain Model mainly because it is not designed to include business rules.

Much discussion has gone into the topic of persistence, and both NHibernate and DataSets take care of that concern. However, NHibernate offers something that DataSets don't: *transparent persistence*.

3.2.2 Transparent and automated persistence

A DataSet allows you to extract the changes performed on it in order to persist them. NHibernate provides a different feature which is very sophisticated and powerful: It can automatically persist your changes in a way that is *transparent* to your domain model.

We use *transparent* to mean a complete separation of concerns between the persistent classes of the domain model and the persistence logic itself, where the persistent classes are unaware of—and have no dependency to—the persistence mechanism.

Our `Item` class, for example, will not have any code-level dependency to any NHibernate API. Furthermore:

- NHibernate doesn't require that any special base classes or interfaces be inherited or implemented by persistent classes. Nor are any special classes used to implement properties or associations. Thus, transparent persistence improves code readability, as you'll soon see.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

- Persistent classes may be reused outside the context of persistence, in unit tests or in the user interface (UI) tier, for example. Testability is a basic requirement for applications with rich domain models.
- In a system with transparent persistence, objects aren't aware of the underlying data store; they need not even be aware that they are being persisted or retrieved. Persistence concerns are externalized to a generic *persistence manager* interface—in the case of NHibernate, the `ISession` and `IQuery` interfaces.

Transparent persistence fosters a degree of portability; without special interfaces, the persistent classes are decoupled from any particular persistence solution. Our business logic is fully reusable in any other application context. We could easily change to another transparent persistence mechanism.

By this definition of transparent persistence, you see that certain non-automated persistence layers are transparent (for example, the DAO pattern) because they decouple the persistence-related code with abstract programming interfaces. Only plain .NET classes without dependencies are exposed to the business logic. Conversely, some automated persistence layers (like many ORM solutions) are non-transparent, because they require special interfaces or intrusive programming models.

We regard transparency as required. In fact, transparent persistence should be one of the primary goals of any ORM solution. However, no automated persistence solution is completely transparent: Every automated persistence layer, including NHibernate, imposes *some* requirements on the persistent classes. For example, NHibernate requires that collection-valued properties be typed to an interface such as `IList` or `IDictionary` (or their .NET 2.0 generic versions) and not to an actual implementation such as `ArrayList` (this is a good practice anyway). (We discuss the reasons for this requirement in appendix B, “ORM implementation strategies.”)

You now know why the persistence mechanism should have minimal impact on how you implement a domain model and that transparent and automated persistence are required. `DataSet` isn't suitable here, so what kind of programming model should you use? Do you need a special programming model at all? In theory, no; in practice, you should adopt a disciplined, consistent programming model that is well accepted by the .NET community. Let's discuss this programming model and see how it works with NHibernate.

3.2.3 Writing POCOs

Developers have found `DataSets` to be unnatural for representing business objects in many situations. The opposite of a heavy model like `DataSet` is the *Plain Old CLR Object* (POCO). It is a back-to-basics approach that essentially consists of using unbound classes in the business layer. ¹

When using NHibernate, entities are implemented as POCOs. The few requirements that NHibernate imposes on your entities are also best practices for the POCO programming model. So, most POCOs are NHibernate-compatible without any changes. The programming model we'll introduce is a non-intrusive mix of POCO best practices, and NHibernate requirements. A POCO declares *business methods*, which define behavior, and *properties*, which represent state. Some properties represent associations to other POCOs.

Listing 3.1 shows a simple POCO class; it's an implementation of the `User` entity of our domain model.

Listing 3.1 POCO implementation of the `User` class

```
[Serializable] public class User { |1
```

¹The term POCO has been derived from the Java term POJO which means *Plain Old Java Object*. It is sometimes also written as Plain *Ordinary* Java Objects; this term was coined in 2002 by Martin Fowler, Rebecca Parsons, and Josh Mackenzie. As an alternative to POCO, it is also common to use the term PONO which stands for *Plain Old .NET Object*.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

private string username;
private Address address;
public User() {} |2
public string Username { |3
    get { return username; } |3
    set { username = value; } |3
} |3
public Address Address { |3
    get { return address; } |3
    set { address = value; } |3
} |3
public MonetaryAmount CalcShipCosts(Address from) { |4
    // ...
}
}
#1 Serializable class
#2 Class constructor
#3 Properties
#4 Business method

```

Some comments on the code above:

NHibernate doesn't require persistent classes to be serializable (as this class is, shown in #1). However, it is commonly needed, mainly when using .NET remoting.

NHibernate requires a "default" parameterless constructor for every persistent class (shown in #2). The constructor may be non-public, but it should be at least protected if runtime-generated proxies will be used for performance optimization (see chapter 4). Note that .NET automatically adds a public parameterless constructor to classes if you haven't written one in the code.

The properties of the POCO implement the attributes of our business entities as shown in #3. For example, the `User`'s name `UserName` provides access to the private `userName` instance variable (same for `Address`). NHibernate doesn't require that properties be declared public; it can easily use private ones too. Some properties do something more sophisticated than simple instance variables access (validation, for example). Trivial properties are common, however.

In #4, This POCO also defines a business method that calculates the cost of shipping an item to a particular user (we left out the implementation of this method).

Now that you understand the value of using POCO persistent classes as the programming model, let's see how you handle the associations between those classes.

3.2.4 Implementing POCO associations

You use properties to express associations between POCO classes, and you use accessor methods to navigate the object graph at runtime. Let's consider the associations defined by the `Category` class. The first association is shown in figure 3.3.

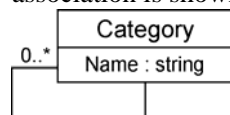


Figure 3.3 Diagram of the `Category` class with an association

As with all our diagrams, we left out the association-related attributes (`parentCategory` and `childCategories`) because they would clutter the illustration. These attributes and the methods that manipulate their values are called *scaffolding code*.

Let's implement the scaffolding code for the *one-to-many* self-association of `Category`:

```

public class Category : ISerializable {
    private string name;

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

private Category parentCategory;
private ISet childCategories = new HashSet();
public Category() { }
...
}

```

Note that we could use .NET 2.0 generics here by writing `ISet<Category> childCategories`. And no other change would be required (even in the mapping).

To allow bidirectional navigation of the association, we require two attributes. The `parentCategory` attribute implements the *single-valued end* of the association and is declared to be of type `Category`. The *many-valued end*, implemented by the `childCategories` attribute, must be of collection type. We choose an `ISet`, and initialize the instance variable to a new instance of `HashSet`.

NHibernate requires interfaces for collection-typed attributes. You must, for example, use `ISet` rather than `HashSet`. At runtime, NHibernate wraps the collection instance with an instance of one of NHibernate's own classes. (This special class isn't visible to the application code). It is good practice to program to collection interfaces, rather than concrete implementations, so this restriction shouldn't bother you.

Used external library: lesi.Collections

In Java, there is a kind of collection called `Set` which allows storing items without duplication (that is, you can't add the same object many times). But .NET doesn't provide an equivalent to this collection. So, NHibernate uses a library called `lesi.Collections` which includes the interface `ISet` and many implementations (like `HashSet`). Their behavior is similar to the one of the `ICollection` so you should be able to easily use them. We will frequently use `Sets` because their semantic fits with the requirement of our classes.

We now have some private instance variables but no public interface to allow access from business code or property management by NHibernate. Let's add some properties to the `Category` class:

```

public string Name {
    get { return name; }
    set { name = value; }
}
public ISet ChildCategories {
    get { return childCategories; }
    set { childCategories = value; }
}
public Category ParentCategory {
    get { return parentCategory; }
    set { parentCategory = value; }
}

```

Again, these properties need to be declared `public` only if they're part of the external interface of the persistent class, the public interface used by the application logic.

The basic procedure for adding a child `Category` to a parent `Category` looks like this:

```

Category aParent = new Category();
Category aChild = new Category();
aChild.ParentCategory = aParent;
aParent.ChildCategories.Add(aChild);

```

Whenever an association is created between a parent `Category` and a child `Category`, two actions are required:

- The `parentCategory` of the child must be set, effectively breaking the association between the child and its old parent (there can be only one parent for any child).
- The child must be added to the `childCategories` collection of the new parent `Category`.

Managed relationships in NHibernate

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

NHibernate doesn't "manage" persistent associations. If you want to manipulate an association, you must write exactly the same code you would write without NHibernate. If an association is bidirectional, both sides of the relationship must be considered. Anyway, this is required if you want to use your objects without NHibernate (for testing or with the UI).

If you ever have problems understanding the behavior of associations in NHibernate, just ask yourself, "What would I do *without* NHibernate?" NHibernate doesn't change the usual .NET semantics.

It's a good idea to add a convenience method to the `Category` class that groups these operations, allowing reuse and helping ensure correctness:

```
public void AddChildCategory(Category childCategory) {
    if (childCategory.ParentCategory != null)
        childCategory.ParentCategory.ChildCategories
            .Remove(childCategory);
    childCategory.ParentCategory = this;
    childCategories.Add(childCategory);
}
```

The `AddChildCategory()` method not only reduces the lines of code when dealing with `Category` objects, but also enforces the cardinality of the association. Errors that arise from leaving out one of the two required actions are avoided. This kind of *grouping of operations* should always be provided for associations, if possible.

Because we would like the `AddChildCategory()` to be the only externally visible mutator method for the child categories, we make the `ChildCategories` property private; we may add more methods to access to `ChildCategories` if required. NHibernate doesn't care if properties are private or public, so we can focus on good API design.

A different kind of relationship exists between `Category` and the `Item`: a bidirectional *many-to-many association* (see figure 3.4).

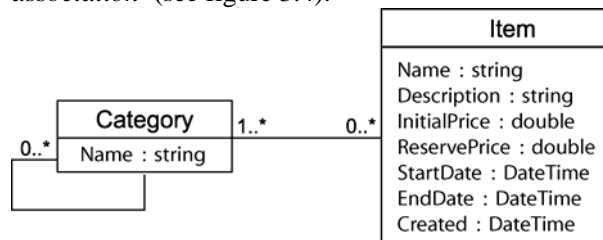


Figure 3.4 `Category` and the associated `Item`

In the case of a many-to-many association, both sides are implemented with collection-valued attributes. Let's **add** the new attributes and methods to access the `Item` class to our `Category` class, as shown in listing 3.2.

Listing 3.2 `Category` to `Item` scaffolding code

```
public class Category {
    ...
    private ISet items = new HashSet();
    ...
    public ISet Items {
        get { return items; }
        set { items = value; }
    }
}
```

The code for the `Item` class (the other end of the many-to-many association) is similar to the code for the `Category` class. We add the **collection** attribute, the standard properties, and a method that simplifies relationship management (you can also add this to the `Category` class, see listing 3.3).

Listing 3.3 `Item` to `Category` scaffolding code

```
public class Item {
    private string name;
    private string description;
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

...
private ISet categories = new HashSet();
...
public ISet Categories() {
    get { return categories; }
    set { categories = value; }
}
public void AddCategory(Category category) {
    category.Items.Add(this);
    categories.Add(category);
}
}

```

The `AddCategory()` of the `Item` method is similar to the `AddChildCategory` convenience method of the `Category` class. It's used by a client to manipulate the relationship between `Item` and a `Category`. For the sake of readability, we won't show convenience methods in future code samples and assume you'll add them according to your own taste.

You should now understand how to create classes to form your domain model that can be persisted by NHibernate. Also, you should be able to create associations between these classes, using convenience methods where necessary to improve the domain model. The next step is to further enrich the domain model by adding business logic. We'll start by looking at how you can add logic to your properties.

3.2.5 Adding logic to properties

One of the reasons we like to use properties is that they provide encapsulation: The hidden internal implementation of a property can be changed without any changes to the public interface. This allows you to abstract the internal data structure of a class—the instance variables—from the design of the database.

For example, if your database stores a name of the user as a single `NAME` column, but your `User` class has `firstname` and `lastname` properties, you can add the following persistent `name` property to your class:

```

public class User {
    private string firstname;
    private string lastname;
    ...
    public string Name {
        get { return firstname + ' ' + lastname; }
        set {
            string[] names = value.Split(' ');
            firstname = names[0];
            lastname = names[1];
        }
    }
}
...
}

```

Later, you'll see that a NHibernate *custom type* is probably a better way to handle many of these kinds of situations. However, it helps to have several options.

Properties can also perform validation. For instance, in the following example, the `FirstName` property's setter verifies that the name is capitalized:

```

public class User {
    private string firstname;
    ...
    public string Firstname {
        get { return firstname; }
        set {

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        if ( !StringUtil.IsCapitalizedName(firstname) )
            throw new InvalidNameException(value);
        firstname = value;
    }
    ...
}

```

However, NHibernate will later use our properties to populate the state of an object when loading the object from the database. Sometimes we would prefer that this validation *not* occur when NHibernate is initializing a newly loaded object. In that case, it might make sense to tell NHibernate to directly access the instance variables (we will later see that we can do so by mapping the property with `access="field"` in NHibernate metadata), forcing NHibernate to bypass the property and access the instance variable directly. Another issue to consider is *dirty checking*. NHibernate automatically detects object state changes in order to synchronize the updated state with the database. It's usually completely safe to return a different object from the get accessor to the object passed by NHibernate to the set accessor. NHibernate will compare the objects by value—not by object identity—to determine if the property's persistent state needs to be updated. For example, the following get accessor won't result in unnecessary SQL UPDATES:

```

public string Firstname {
    get { return new string(firstname); }
}

```

However, there is one very important exception. Collections are compared by identity!

For a property mapped as a persistent collection, you should return *exactly* the same collection instance from the get accessor as NHibernate passed to the set accessor. If you don't, NHibernate will update the database, even if no update is necessary, *every time* the session synchronizes state held in memory with the database. This kind of code should almost always be avoided in properties:

```

public IList Names {
    get { return new ArrayList(names); }
    set { names = new string[value.Count]; value.CopyTo(names, 0); }
}

```

You can see that NHibernate doesn't unnecessarily restrict the POCO programming model. You're free to implement whatever logic you need in properties (as long as you keep the same collection instance in both get and set accessors). Note that collections should not have a setter at all.

If absolutely necessary, you can tell NHibernate to use a different access strategy to read and set the state of a property (for example, direct instance field access), as you'll see later. This kind of transparency guarantees an independent and reusable domain model implementation.

At this point we have defined a number of classes for our domain model and set up some associations between them. We've also added some convenience methods to make working with the model easier, and added some business logic. Our next goal is to be able to load and save objects in the domain model to and from a relational database. So, we now need to look at setting up the necessary pieces to let NHibernate persist our objects, or more specifically, the Object Relational Mapping.

3.3 Defining the mapping metadata

ORM tools require a metadata format for the application to specify the mapping between classes and tables, properties and columns, associations and foreign keys, .NET types and SQL types. This information is called the object/relational mapping *metadata*. It defines the transformation between the different data type systems and relationship representations.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

It's our job as developers to define and maintain this metadata. There are two different ways to do this: attributes and metadata. In this section, you will learn how to write mapping using these two ways and we will compare them so that you can decide which one to use. Let's start with the mapping you are already familiar with: The mapping using .NET XML files.

3.3.1 Mapping using XML

NHibernate provides a mapping format based on the popular XML. Mapping documents written in and with XML are lightweight, are human readable, are easily hand-editable, are easily manipulated by version-control systems and text editors, and may be customized at deployment time (or even at runtime, with programmatic XML generation).

However, is XML-based metadata really a viable approach? A certain backlash against the overuse of XML can be seen in the developer community. Every framework and service seems to require its own XML descriptors.

In our view, there are three main reasons for this backlash:

- Many existing metadata formats weren't designed to be readable and easy to edit by hand. In particular, a major cause of pain is the lack of sensible defaults for attribute and element values, requiring significantly more typing than should be necessary.
- Metadata-based solutions were often used inappropriately. Metadata is not, by nature, more flexible or maintainable than plain C# code.
- Good XML editors, especially in IDEs, aren't as common as good .NET coding environments. Worst, and most easily fixable, a XML Schema Definition (XSD) often isn't provided, preventing auto-completion and validation. Another problem are XSDs that are too generic, where every declaration is wrapped in a generic "extension" of "meta" element (like the key/value approach).

There is no getting around the need for text-based metadata in ORM. However, NHibernate was designed with full awareness of the typical metadata problems. The metadata format is extremely readable and defines useful default values. When some values are missing, NHibernate uses reflection on the mapped class to help determine the defaults. NHibernate comes with a documented and complete XSD. Finally, IDE support for XML has improved lately, and modern IDEs provide dynamic XML validation and even an auto-complete feature. If that's not enough for you, in chapter 9 we demonstrate some tools that may be used to generate NHibernate XML mappings.

Let's look at the way you can use XML metadata in NHibernate. We introduced the mapping of the class `Category` in previous section; now we are going to give more details about the structure of its XML mapping document that is reproduced in listing 3.4.

Listing 3.4 NHibernate XML mapping of the `Category` class

```
<?xml version="1.0"?>
<hibernate-mapping                                | 1
  xmlns="urn:hibernate-mapping-2.2"              | 2
  auto-import="true">

  <class                                          | 3
    name="CaveatEmptor.Model.Category, CaveatEmptor"
    lazy="false">

    <id name="Id">                                | 4
      <generator class="native" />
    </id>

    <property                                    | 5
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        name="Name"
        column="name" />

    <many-to-one
        name="ParentCategory"
        cascade="all" />
</class>
</hibernate-mapping>

```

- #1 Mapping declaration
- #2 XSD declaration (optional)
- #3 Category class mapping
- #4 Identifier mapping
- #5 Name property mapping
- #6 Reference to another employee

As you can see, a XML mapping document can be divided in many parts:

- Mappings are declared inside a `<hibernate-mapping>` element. You can include as many class mappings as you like, along with certain other special declarations that we'll mention later in the book.
- The NHibernate mapping XSD is declared in order to provide syntactic validation of the XML and many XML editors use it for auto-completion. But it is not recommended to use the online copy of this file for performance reasons.
- The class `Category` (in the assembly `CaveatEmptor.Model`) is mapped to the table of the same name (`Category`). Every row in this table represents one instance of type `Category`.
- We haven't much discussed the concept of *object identity*. This complex topic is covered in section 3.3. To understand this mapping, it's sufficient to know that every record in the `Category` table will have a primary key value that matches the object identity of the instance in memory. The `<id>` mapping element is used to define the details of object identity.
- The property `Name` is mapped to a database column of the same name (`Name`). NHibernate will use .NET reflection to discover the type of this property and deduce how to map it to the SQL column, assuming they have compatible types. Note that it is possible to explicitly specify the *mapping data type* that NHibernate should use. We take a close look at these types in chapter 7, section 7.1, "Understanding the NHibernate type system".
- We use an association to link a `Category` to another. Here, it is a *many-to-one* association. In the database, the table `Category` contains a column `ParentCategory` which is a foreign key to another row in the same table. Association mappings are more complex, so we'll return to them in chapter 4, section 4.6.

Although it's possible to declare mappings for multiple classes in one mapping file by using multiple `<class>` elements, the recommended practice (and the practice expected by some NHibernate tools) is to use one mapping file per persistent class. The convention is to give the file the same name as the mapped class, appending an `hbm` suffix: for example, `Category.hbm.xml`.

Sometimes you may want to use .NET attributes rather than XML files for defining your mappings, and so we briefly explain how this can be done next. After that, we'll go on to look more closely at the nature of the class and property mappings described here in this section.

3.3.2 Attribute-oriented programming

One way to define the mapping metadata is to use .NET attributes. Since its first release, .NET provides support for class/member attributes. We have introduced, in chapter 2, the `NHibernate.Mapping.Attributes` library. It uses attributes directly embedded in the .NET source code to provide all the information NHibernate

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

needs to map classes. All we have to do is to mark up the .NET source code of our persistent classes with custom .NET attributes, as shown in listing 3.5.

Listing 3.5 Mapping with NHibernate.Mapping.Attributes

```
using NHibernate.Mapping.Attributes;

[Class(Lazy=false)]
public class Category {
    ...
    [Id(Name="Id")]
    [Generator(1, Class="native")]
    public long Id {
        ...
    }
    ...
    [Property]
    public string Name {
        ...
    }
    ...
}
```

It is very easy to use this mapping with NHibernate:

```
cfg.AddInputStream(
    NHibernate.Mapping.Attributes.HbmSerializer.Default.Serialize(
        typeof(Category) ) );
```

Here, NHibernate.Mapping.Attributes generates a XML stream from the mapping in the class `Category` and this stream is sent to NHibernate configuration. It is also possible to write this mapping information in external XML documents.

XML Mapping or .NET Attributes?

We have introduced mapping using XML mapping files and using `NHibernate.Mapping.Attributes`. Although you can use both at the same time, it is more common (and homogenous) to use only one technique. Your choice is based on the way you develop your application. You can read more details about development processes in chapter 8.

For now, you have already realized that .NET attributes are much more convenient and reduce the lines of metadata significantly. They are also type-safe, support auto-completion in your IDE as you type (like any other C# type), and make refactoring of classes and properties easier. `NHibernate.Mapping.Attributes` is usually used when starting a new project. Arguably, attribute mappings are less configurable at deployment time. However, nothing is stopping you from hand-editing the generated XML before deployment, so this probably isn't a significant objection.

On the other hand, XML mapping documents are external; this means that they can evolve independently of your domain model; they are also easier to manipulate for very complex mapping and they can contain some useful information (not directly related to the mapping of the classes). It is common to use XML mapping files when the classes already exist and aren't under our control.

Note that, in few cases, it is better to write XML mapping; for example, when dealing with a highly customized component or collection mapping. In these cases, you can use the attribute `[RawXml]` to insert this XML in your attribute mapping.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

You should now grasped the basic idea of how both XML mappings and attribute mappings work. Next we will look more closely at the types of mappings in more detail, starting with property and class mappings.

3.4 Basic property and class mappings

In this section, you will learn a number of features and tips to write better mappings. NHibernate can “guess” some information in order to make your mappings shorter. It is also possible to configure it to access your entities in a specific way.

Let’s start with a deeper review of the mapping of simple properties.

3.4.1 Property mapping overview

A typical NHibernate property mapping defines a property name, a database column name, and the name of a NHibernate type. It maps a .NET property to a table column. The basic declaration provides many variations and optional settings, for example, it is often possible to omit the type name. So, if `Description` is a property of (.NET) type `String`, NHibernate will use the NHibernate type `String` by default (we discuss the NHibernate type system in chapter 7). NHibernate uses reflection to determine the .NET type of the property. Thus, the following mappings are equivalent, as long as they are on the property `Description`:

```
[Property(Name="Description", Column="DESCRIPTION", Type="String")]
[Property(Column="DESCRIPTION")]
public string Description { ... }
```

These mapping can be written using XML; the following mappings are equivalent:

```
<property name="Description" column="DESCRIPTION" type="String"/>
<property name="Description" column="DESCRIPTION"/>
```

As you already know, you can omit the column name if it’s the same as the property name, ignoring case. (This is one of the sensible defaults we mentioned earlier.)

In some cases, you might need to tell NHibernate more about the database column than just it’s name. For this you can use the `<column>` element instead of the `column` attribute. The `<column>` element provides more flexibility; it has more optional attributes and may appear more than once. The following two property mappings are equivalent:

```
[Property]
[Column(1, Name="DESCRIPTION")]
public string Description { ... }
```

Using XML, you can write:

```
<property name="Description" type="String">
  <column name="DESCRIPTION"/>
</property>
```

Because .NET attributes are not ordered, you sometimes need to specify their position. Here, `[Column]` comes after `[Property]`, so its position is 1; the position of `[Property]` is 0 (the default value).

`NHibernate.Mapping.Attributes` mimics XML mapping so if you can write one, you can deduce how to write the other. The main difference is that you don’t need to specify names with attribute mappings because `NHibernate.Mapping.Attributes` can guess it based on where the attribute mapping is in the code. The exception is `[Id]`; you must specify the identifier’s name when it has one, because it is optional.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The `<property>` element (and especially the `<column>` element) also defines certain attributes that apply mainly to automatic database schema generation. If you aren't using the `hbm2ddl` tool (see section 10.1.1) to automatically generate the database schema, you can safely omit these. However, it's still preferable to include at least the `not-null` attribute, since NHibernate will then be able to report illegal null property values without going to the database:

```
<property name="InitialPrice" column="INITIAL_PRICE" not-null="true"/>
```

Detection of illegal null values is mainly useful for providing sensible exceptions at development time. It isn't intended for true data validation, which is outside the scope of NHibernate.

Some properties don't map to a column at all. In particular, a *derived* property takes its value from an SQL expression.

3.4.2 Using derived properties

The value of a derived property is calculated at runtime by evaluation of an expression. You define the expression using the `formula` attribute. For example, for a `ShoppingCart` class we might map a `TotalIncludingTax` property, and because it is a formula there is no column to store that value in the database:

```
<property name="TotalIncludingTax"
  formula="TOTAL + TAX_RATE * TOTAL"
  type="Double"/>
```

The given SQL formula is evaluated every time the entity is retrieved from the database. So, the database does the calculation rather than the .NET object. The property doesn't have a `column` attribute (or sub-element) and never appears in an SQL INSERT or UPDATE, only in SELECTs. Formulas may refer to columns of the database table, call SQL functions, and include SQL subselects.

This example, mapping a derived property of `Item`, uses a correlated subselect to calculate the average amount of all bids for an item:

```
<property
  name="AverageBidAmount"
  formula="( select AVG(b.AMOUNT) from BID b
    where b.ITEM_ID = ITEM_ID )"
  type="Double"/>
```

Notice that unqualified column names (in this case, those not preceded by a `b.`) refer to table columns of the class to which the derived property belongs.

As we mentioned earlier, NHibernate doesn't require properties on entities if you define a new property access strategy. The next section will explain the various strategies and when you should use them in your mapping.

3.4.3 Property access strategies

The `access` attribute allows you to specify how NHibernate should access values of the entity. The default strategy, `property`, uses the property accessors – these being the getters and setters you declare in your classes. In your mapping XML file, mapping a class property getter and setter to a column is quite simple:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


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

When NHibernate loads or saves an object, it will always use the defined getter and setter to access the data in the object.

In our “Hello World” example in Chapter 2, you might remember that we used the `field` access strategy in the XML mapping file for our `Employee` entity. The `field` strategy is useful for times when you haven’t defined property getters and setters for your classes. Behind the scenes, it uses reflection to access the instance class field directly. For example, the following “`property`” mapping doesn’t require a getter/setter pair in the class because it is using `field` access strategy:

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

Using the `field` access strategy can be useful at times, but access through property getters and setters is considered best practice by the NHibernate community; they give you an extra level of abstraction between the .NET domain model and the data model, beyond that already provided by NHibernate. Properties are also more flexible than fields; for example, property definitions may be overridden by persistent subclasses.

NHibernate gives you additional flexibility when working with properties. For example, what if your property setters contain business logic? Often we only want this logic to be executed when our client code sets the property, not during load time. If a class is mapped using a property setter, NHibernate will run the code as it loads the object. Thankfully, there are ways to deal with this situation. NHibernate gives us a special access strategy called the `nosetter.*` strategy. Using this in your mapping tells NHibernate to use the property getter when reading data from the object, but to use the underlying field whilst writing data to it.

If you find you need even more flexibility than this, you can learn about other access strategies available in the NHibernate reference documentation online. As a taster, if you wanted NHibernate to not use the getter if you use the standard way of naming fields in C#, that is camel case prefixed by an underscore (such as `_firstName`), you can map it like this, using `NHibernate.Mapping.Attributes`:

```
private string _firstName;

[Property(Access="field.camelcase-underscore")]
public string FirstName {
    get { return _firstName; }
}
```

The equivalent XML mapping is:

```
<property name="FirstName" access="field.camelcase-underscore" />
```

The nice side-effect of this example is that, when writing NHibernate HQL queries, you use the more readable property name rather than ugly field names. Behind the scenes, NHibernate knows to bypass the property and instead use the field when loading and saving objects. Because we’re using a field, the property is effectively ignored –it doesn’t even have to exist in the code! As a useful extra, if you wanted to do this for all properties on a class, you can specify this access strategy at class-level by using `<hibernate-mapping default-access="...">` or the property `HbmSerializer.HbmDefaultAccess` when using `NHibernate.Mapping.Attributes`.

If you still need more flexibility, you can define your own customized property access strategy by implementing the interface `NHibernate.Property.IPropertyAccessor` and name it in the `access` attribute (using its fully qualified name). Using `NHibernate.Mapping.Attributes`, you have the alternative of using:

```
[Property(AccessType=typeof(MyPropertyAccessor))]
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This facility (adding `"Type"` at the end of an element to provide a .NET type instead of a string) is available in many other places.

So far we have now learnt how to build the classes for our domain model, and then how we can define mapping metadata to tell NHibernate how to persist these classes and their members. You can see that NHibernate gives us a wealth of features and flexibility, but essentially we're talking about fairly straightforward ORM capabilities. Next we'll look at some of the deeper "under the bonnet" aspects of NHibernate, including the ability to disable its optimizer to assist with debugging, the ability to enforce that objects are immutable by preventing NHibernate from inserting and updating them, and a few other handy tricks that will help you tackle other thorny scenarios.

3.4.4 Taking advantage of the reflection optimizer

We mentioned that NHibernate can use reflection to get and set properties of an entity at runtime. Reflection can be slow, so NHibernate goes a step further and uses an optimizer to speed up this process. The optimizer is enabled by default, and goes to work as you create your session factories. Because of this, you suffer a small startup cost, but it's usually worth it.

Depending on the version of .NET you're using, NHibernate takes different approaches to optimizing reflection.

Under .NET 1.1, NHibernate uses a `CodeDom` provider. This provider generates special classes at runtime that know about your business entities, and which can access them without using reflection. A small caveat is that it only works for public properties, therefore you must use the default access strategy `"property"` in your mapping files to get optimal results. Another restriction is that quoted SQL identifiers (section 3.5.6) are not supported.

NHibernate 1.2 introduces another provider which only works (and is used by default) under .NET 2.0. This provider injects dynamic methods in to your business entities at startup, and it is more powerful because it is not restricted to public properties.

It's rarely necessary, but you can disable this reflection optimizer by updating your configuration file:

```
<property name="hibernate.use_reflection_optimizer">false</property>
```

Or, at runtime, using:

```
Environment.UseReflectionOptimizer = false;
```

This may be helpful when debugging your application, because runtime generated classes are harder to trace. You must set this property before actually instantiating the Configuration. So, you can not use the `<hibernate-configuration>` section in your config file (`hibernate.cfg.xml`, `web.config` etc) because this is read *after* the Configuration object is created (during the call to your Configuration objects `Configure()` method).

You can select the `CodeDom` provider using:

```
<property name="hibernate.bytecode.provider">codedom</property>
```

The value `codedom` can be replaced by `null` (to disable the optimizer) or `lcg` (on .NET 2.0 or later only). Note that `codedom` may not work properly with generic types.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

You must set this property in the `<nhibernate>` section of your application configuration file. At runtime, before building the session factory, you can set the property `Environment.BytecodeProvider` to the value returned by the static method `Environment.BuildBytecodeProvider()` or to an instance of your own provider which implements the interface `NHibernate.Bytecode.IBytecodeProvider`.

The next interesting capability we'll look at is the ability to control database inserts and updates for classes and their members. This level of control comes in very useful when you want to create immutable objects, or when you want to disable updates on a per-property basis.

3.4.5 Controlling insertion and updates

For properties that map to columns, you can control whether they appear in the `INSERT` statement by using the `insert` attribute and whether they appear in the `UPDATE` statement by using the `update` attribute.

The following property is never written to the database:

```
<property name="Name"
  column="NAME"
  type="String"
  insert="false"
  update="false"/>
```

The property `Name` of the entity is therefore immutable; it can be read from the database but not modified in any way. If the complete class is immutable, set the `mutable="false"` in the class mapping. If you're unfamiliar with immutable classes, they're basically just a class where you've decided they should never be updated after they've been created. An example might be a financial transaction record.

In addition, we also have the `dynamic-insert` and `dynamic-update` attributes that tell NHibernate whether to include unmodified property values during SQL INSERTS and UPDATES,

```
<class name="NHibernate.Auction.Model.User, NHibernate.Auction"
  dynamic-insert="true"
  dynamic-update="true">
  ...
</class>
```

These are both class-level settings that are off by default, so that when NHibernate generates INSERT and UPDATE SQL for an object, it does so for all properties on the object regardless if they've actually changed since the object was loaded. Enabling either of these settings will cause NHibernate to generate some SQL at runtime, instead of using the SQL cached at startup time. The performance and memory cost of doing this is usually small. Furthermore, leaving out columns in an insert (and especially in an update) can occasionally improve performance if your tables define many/large columns.

3.4.6 Using quoted SQL identifiers

By default, NHibernate doesn't quote table and column names in the generated SQL. This makes the SQL slightly more readable and also allows us to take advantage of the fact that most SQL databases are case insensitive when comparing unquoted identifiers. From time to time, especially in legacy databases, you'll encounter identifiers with strange characters or whitespace, or you may wish to force case-sensitivity.

If you quote a table or column name with backticks in the mapping document, NHibernate will always quote this identifier in the generated SQL. The following property declaration forces NHibernate to generate SQL with the quoted column name `"Item Description"`. NHibernate will also know that Microsoft SQL Server needs the variation `[Item Description]` and that MySQL requires ``Item Description``.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

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

There is no way, apart from quoting all table and column names in backticks, to force NHibernate to use quoted identifiers everywhere.

NHibernate gives us further control when mapping between our domain model and the database schema, by also letting us control naming conventions. We discussed this next.

3.4.7 Naming conventions

Development teams often have strict conventions for table and column names in their databases. NHibernate provides a feature that allows you to enforce naming standards automatically.

Suppose that all table names in CaveatEmptor should follow the pattern `CE_<table name>`. One solution is to manually specify a `table` attribute on all `<class>` and collection elements in our mapping files. This approach is time-consuming and easily forgotten. Instead, we can implement NHibernate's `INamingStrategy` interface, as in listing 3.6.

Listing 3.6 `INamingStrategy` implementation

```
public class CENamingStrategy : INamingStrategy {
    public string ClassToTableName(string className) {
        return TableName(
            StringHelper.Unqualify(className).ToUpper() );
    }
    public string PropertyToColumnName(string propertyName) {
        return propertyName.ToUpper ();
    }
    public string TableName(string tableName) {
        return "CE_" + tableName;
    }
    public string ColumnName(string columnName) {
        return columnName;
    }
    public string PropertyToTableName(string className,
        string propertyName) {
        return ClassToTableName(className) + '_' +
            PropertyToColumnName(propertyName);
    }
}
```

The `ClassToTableName()` method is called only if a `<class>` mapping doesn't specify an explicit `table` name. The `PropertyToColumnName()` method is called if a property has no explicit `column` name. The `TableName()` and `ColumnName()` methods are called when an explicit name is declared.

If we enable our `CENamingStrategy`, this class mapping declaration

```
<class name="BankAccount">
```

will result in `CE_BANKACCOUNT` as the name of the table. The `ClassToTableName()` method was called with the fully qualified class name as the argument.

However, if a table name is specified

```
<class name="BankAccount" table="BANK_ACCOUNT">
```

then `CE_BANK_ACCOUNT` will be the name of the table. In this case, `BANK_ACCOUNT` was passed to the `TableName()` method.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The best feature of the `INamingStrategy` is the potential for dynamic behavior. To activate a specific naming strategy, we can pass an instance to the NHibernate `Configuration` at runtime:

```
Configuration cfg = new Configuration();
cfg.NamingStrategy = new CENamingStrategy();
ISessionFactory sessionFactory =
    cfg.configure().BuildSessionFactory();
```

This will allow us to have multiple `ISessionFactory` instances based on the same mapping documents, each using a different `INamingStrategy`. This is extremely useful in a multi-client installation where unique table names (but the same data model) are required for each client.

However, a better way to handle this kind of requirement is to use the concept of an SQL *schema* (a kind of namespace).

3.4.8 SQL schemas

SQL Schemas are a feature available in many databases, including SQL Server 2005 and MySQL. They let you organize your database objects into meaningful groups. For example, the AdventureWorks sample database that comes with Microsoft SQL Server 2005 defines five schemas: Human Resources, Person, Production, Purchasing and Sales. All these schemas live in a single database, and each has its own tables, views and other database objects.

Many databases are designed with only one schema, and therefore you can specify a default schema using the `hibernate.default_schema` configuration option; there are some small performance benefits in doing so.

Alternatively, if your database is like the AdventureWorks one, and has many schemas, you can specify the schema for a particular mapping document, or even a particular class or collection mapping:

```
<hibernate-mapping>
  <class
    name="NHibernateInAction.HelloWorld.Message,
      NHibernateInAction.HelloWorld"
    schema="HelloWorld">
    ...
  </class>
</hibernate-mapping>
```

It can even be declared for the whole document:

```
<hibernate-mapping
  schema="HelloWorld">
  ..
</hibernate-mapping>
```

Next we'll discuss another useful thing you can do with the `<hibernate-mapping>` element, which is to specify a default namespace for your classes to reduce duplication.

3.4.9 Declaring class names

In this chapter, we introduced the CaveatEmptor application. All the persistent classes of the application are declared in the namespace `NHibernate.Auction.Model` and are compiled in the `NHibernate.Auction` assembly. It would become tedious to specify this fully qualified name every time we named a class in our mapping documents.

Let's reconsider our mapping for the `User` class (the file `User.hbm.xml`):

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

<?xml version="1.0"?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:nhibernate-mapping-2.2
http://nhibernate.sourceforge.net/schemas/nhibernate-mapping.xsd">
  <class
    name="NHibernate.Auction.Model.User, NHibernate.Auction">
    ...
  </class>
</hibernate-mapping>

```

We don't want to repeat the fully qualified name whenever this or any other class is named in an association, subclass, or component mapping. Therefore, we'll instead specify a **namespace** and an **assembly**:

```

<hibernate-mapping
  namespace="NHibernate.Auction.Model"
  assembly="NHibernate.Auction">
  <class
    name="User">
    ...
  </class>
</hibernate-mapping>

```

Now all unqualified class names that appear in this mapping document will be prefixed with the declared package name. We assume this setting in all mapping examples in this book. However, this setting is mostly useless when using attribute mapping because we can specify the .NET type and NHibernate.Mapping.Attributes will write its fully qualified name.

You may also use the methods `SetDefaultNamespace()` and `SetDefaultAssembly()` of the class `Configuration` to achieve the same result for the whole application.

Note: We will no longer write the XSD information as it clutters these examples.

Both approaches we have described so far, XML and .NET attributes, assume that all mapping information is known at deployment time. Suppose that some information isn't known before the application starts. Can you programmatically manipulate the mapping metadata at runtime?

3.4.10 Manipulating metadata at runtime

It's sometimes useful for an application to browse, manipulate, or build new mappings at runtime. You can safely skip this section and come back to it later when you need to. .NET provides XML APIs that allow direct runtime manipulation of XML documents. Therefore, you could create or manipulate an XML document at runtime, before feeding it to the `Configuration` object.

However, NHibernate also exposes a configuration-time metamodel. The metamodel contains all the information declared in your XML mapping documents. Direct programmatic manipulation of this metamodel is sometimes useful, especially for applications that allow for extension by user-written code.

For example, the following code adds a new property, `Motto`, to the `User` class mapping:

```

PersistentClass userMapping = cfg.GetClassMapping(typeof(User));      | 1

Column column = new Column(new StringType(), 0);                    | 2
column.Name = "MOTTO";                                             | 2
column.IsNullable = false;                                         | 2
column.IsUnique = true;                                            | 2
userMapping.Table.AddColumn(column);                                 | 2

SimpleValue value = new SimpleValue();                              | 3

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

value.Table = userMapping.Table;           |3
value.AddColumn(column);                   |3
value.Type = new StringType();              |3

Property prop = new Property();            |4
prop.Value = value;                        |4
prop.Name = "Motto";                       |4
userMapping.AddProperty(prop);             |4

ISessionFactory sf = cfg.BuildSessionFactory(); |5

```

1. Get mapping information for User from the configuration
2. Define a new column for the USER table
3. Wrap the column in a value
4. Define a new property of the User class
5. Build a new session factory, using the new mapping

A `PersistentClass` object represents the metamodel for a single persistent class; we retrieve it from the `Configuration`. `Column`, `SimpleValue`, and `Property` are all classes of the NHibernate metamodel and are available in the namespace `NHibernate.Mapping`; the class `StringType` is in the namespace `NHibernate.Type`. Keep in mind that adding a property to an *existing* persistent class mapping as shown here is easy, but programmatically creating a new mapping for a previously unmapped class is quite a bit more involved.

Once an `ISessionFactory` is created, its mappings are immutable. In fact, the `ISessionFactory` uses a different metamodel internally than the one used at configuration time. There is no way to get back to the original `Configuration` from the `ISessionFactory` or `ISession`. However, the application may read the `ISessionFactory`'s metamodel by calling `GetClassMetadata()` or `GetCollectionMetadata()`. For example:

```

User user = ...;
ClassMetadata meta = sessionFactory.GetClassMetadata(typeof(User));
string[] metaPropertyNames = meta.GetPropertyNames();
object[] propertyValues = meta.GetPropertyValues(user);

```

This code snippet retrieves the names of persistent properties of the `User` class and the values of those properties for a particular instance. This helps you write generic code. For example, you might use this feature to label UI components or improve log output.

Now let's turn to a special mapping element you've seen in most of our previous examples, the *identifier property mapping*. We'll begin by discussing the notion of *object identity*.

3.5 Understanding object identity

It's vital to understand the difference between *object identity* and *object equality* before we discuss terms like *database identity* and how NHibernate manages identity. We need these concepts if we want to finish mapping our `CaveatEmptor` persistent classes and their associations with NHibernate.

3.5.1 Identity versus equality

.NET developers understand the difference between .NET object *identity* and *equality*. Object identity, `object.ReferenceEquals()`, is a notion defined by the CLR environment. Two object references are identical if they point to the same memory location.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

On the other hand, object equality is a notion defined by classes that implement the `Equals()` method (or the operator `==`), sometimes also referred to as *equivalence*. Equivalence means that two different (non-identical) objects have the same value. Two different instances of `string` are equal if they represent the same sequence of characters, even though they each have their own location in the memory space of the virtual machine. (We admit that this is not entirely true for `strings`, but you get the idea.)

Persistence complicates this picture. With object/relational persistence, a persistent object is an in-memory representation of a particular row of a database table. So, along with .NET identity (memory location) and object equality, we pick up *database identity* (location in the persistent data store). We now have three methods for identifying objects:

- *Object identity*—Objects are identical if they occupy the same memory location. This can be checked by using `object.ReferenceEquals()`.
- *Object equality*—Objects are equal if they have the same value, as defined by the `Equals(object o)` method. Classes that don't explicitly override this method inherit the implementation defined by `System.Object`, which compares object identity.
- *Database identity*—Objects stored in a relational database are identical if they represent the same row or, equivalently, share the same table and primary key value.

You need to understand how database identity relates to object identity in NHibernate.

3.5.2 Database Identity with NHibernate

NHibernate exposes database identity to the application in two ways:

- The value of the *identifier property* of a persistent instance
- The value returned by `ISession.GetIdentifier(object o)`

The identifier property is special: Its value is the primary key value of the database row represented by the persistent instance. We don't usually show the identifier property in our domain model—it's a persistence-related concern, not part of our business problem. In our examples, the identifier property is always named `id`. So if `myCategory` is an instance of `Category`, calling `myCategory.Id` returns the primary key value of the row represented by `myCategory` in the database.

Should you make the property for the identifier private scope or public? Well, database identifiers are often used by the application as a convenient handle to a particular instance, even outside the persistence layer. For example, web applications often display the results of a search screen to the user as a list of summary information. When the user selects a particular element, the application might need to retrieve the selected object. It's common to use a lookup by identifier for this purpose—you've probably already used identifiers this way, even in applications using direct ADO.NET. It's therefore usually appropriate to fully expose the database identity with a public identifier property.

On the other hand, we usually don't implement a `set` accessor for the identifier (in this case, NHibernate uses .NET reflection to modify the identifier field). And we also usually let NHibernate generate the identifier value. The exceptions to this rule are classes with natural keys, where the value of the identifier is assigned by the application before the object is made persistent, instead of being generated by NHibernate. (We discuss natural keys in the next section.) NHibernate doesn't allow you to change the identifier value of a persistent instance after it's first assigned. Remember, part of the definition of a primary key is that its value should never change. Let's implement an identifier property for the `Category` class and map it with .NET attributes:

```
[Class(Table="CATEGORY")]
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

public class Category {
    private long id;
    ...
    [Id(Name="Id", Column="CATEGORY_ID", Access="nosetter.camelcase")]
    [Generator(1, Class="native")]
    public long Id {
        get { return this.id; }
    }
    ...
}

```

The property type depends on the primary key type of the `CATEGORY` table and the NHibernate mapping type. This information is determined by the `<id>` element in the mapping document. Here is the XML mapping:

```

<class name="Category" table="CATEGORY">
    <id name="Id" column="CATEGORY_ID" access="nosetter.camelcase">
        <generator class="native"/>
    </id>
    ...
</class>

```

The identifier property is mapped to the primary key column `CATEGORY_ID` of the table `CATEGORY`. The NHibernate type for this property is `long`, which maps to a `BIGINT` column type in most databases and which has also been chosen to match the type of the identity value produced by the `native` identifier generator. (We discuss identifier generation strategies in the next section.) The access strategy used here: `access="nosetter.camelcase"` tells to NHibernate that there is no set accessor and that it should use the camelCase transformation to deduce the name of the identity field using the property name. If possible, NHibernate will use a reflection optimizer to avoid reflection costs (explained later in this chapter).

So, in addition to operations for testing .NET object identity (`object.ReferenceEquals(a,b)`) and object equality (`a.Equals(b)`), you may now use `a.Id==b.Id` to test database identity.

An alternative approach to handling database identity is to not implement any identifier property, and let NHibernate manage database identity internally. In this case, you omit the `name` attribute in the mapping declaration:

```

<id column="CATEGORY_ID">
    <generator class="native"/>
</id>

```

NHibernate will now manage the identifier values internally. You may obtain the identifier value of a persistent instance as follows:

```

long catId = (long) session.GetIdentifier(category);

```

This technique has a serious drawback: You can no longer use NHibernate to manipulate *detached objects* effectively (see chapter 4, section 4.1.5, "Outside the identity scope"). So, you should always use identifier properties in NHibernate. If you don't like them being visible to the rest of your application, make the property protected or private.

Using database identifiers in NHibernate is easy and straightforward. Choosing a good primary key (and key generation strategy) might be more difficult. We discuss these issues next.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.5.3 Choosing primary keys

You have to tell NHibernate about your preferred primary key generation strategy. But first, let's define *primary key*.

The *candidate key* is a column or set of columns that uniquely identifies a specific row of the table. A candidate key must satisfy the following properties:

- The value or values are never null.
- Each row has a unique value or values.
- The value or values of a particular row never change.

For a given table, several columns or combinations of columns might satisfy these properties. If a table has only one identifying attribute, it is by definition the *primary key*. If there are multiple candidate keys, you need to choose between them (candidate keys not chosen as the primary key should be declared as unique keys in the database). If there are *no* unique columns or unique combinations of columns, and hence no candidate keys, then the table is by definition not a relation as defined by the relational model (it permits duplicate rows), and you should rethink your data model.

Many legacy SQL data models use *natural* primary keys. A natural key is a key with business meaning: an attribute or combination of attributes that is unique by virtue of its business semantics. Examples of natural keys might be a U.S. Social Security Number or Australian Tax File Number. Distinguishing natural keys is simple: If a candidate key attribute has meaning outside the database context, it's a natural key, whether or not it's automatically generated.

Experience has shown that natural keys almost always cause problems in the long run. A good primary key must be unique, constant, and required (never null or unknown). Very few entity attributes satisfy these requirements, and some that do aren't efficiently indexable by SQL databases. In addition, you should make absolutely certain that a candidate key definition could never change throughout the lifetime of the database before promoting it to a primary key. Changing the definition of a primary key and all foreign keys that refer to it is a frustrating task.

For these reasons, we strongly recommend that new applications use synthetic identifiers (also called *surrogate keys*). Surrogate keys have no business meaning—they are unique values generated by the database or application. There are a number of well-known approaches to surrogate key generation.

NHibernate has several built-in identifier generation strategies. We list the most useful options in table 3.1.

Table 3.1 NHibernate's built-in identifier generator modules

Generator name	Description
native	The native identity generator picks other identity generators like identity, sequence, or hilo depending on the capabilities of the underlying database.
identity	This generator supports identity columns in DB2, MySQL, MS SQL Server, Sybase, and Informix. The identifier returned by the database is converted to the property type using <code>Convert.ChangeType</code> . Any integral property type is thus supported.
sequence	A sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi, Firebird is used. The identifier returned by the database is converted to the property type using <code>Convert.ChangeType</code> . Any integral property type is thus supported.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

increment	At NHibernate startup, this generator reads the maximum primary key column value of the table and increments the value by one each time a new row is inserted. The generated identifier can be of any integral type. This generator is especially efficient if the single-server NHibernate application has exclusive access to the database but shouldn't be used in any other scenario (like in clusters).
hilo	A high/low algorithm is an efficient way to generate identifiers of any integral type, given a table and column (by default hibernate_unique_key and next_hi, respectively) as a source of hi values. The high/low algorithm generates identifiers that are unique only for a particular database. See [Ambler 2002] for more information about the high/low approach to unique identifiers. Do not use this generator with a user-supplied connection.
uuid.hex	This generator uses System.Guid and its ToString(string format) method to generate identifiers of type string. The length of the string returned depends on the configured format. This generation strategy isn't popular, since CHAR primary keys consume more database space than numeric keys and are marginally slower.
guid	This generator is used when the identifier's type is Guid. The identifier must have Guid.Empty as default value. When saving the entity, this generator will assign it a new value using Guid.NewGuid().
guid.comb	This generator is similar to the previous one. However, it uses another algorithm that makes it almost as fast as when using integers (especially when saving in a SQL Server database). Furthermore, the generated values are ordered; you can use a part of these values as reference numbers, for example.

You aren't limited to these built-in strategies; there are some others that you can discover by reading NHibernate's reference documentation. You may also create your own identifier generator by implementing NHibernate's `IIdentifierGenerator` interface. It's even possible to mix identifier generators for persistent classes in a single domain model, but for non-legacy data we recommend using the same generator for all classes.

The special `assigned` identifier generator strategy is most useful for entities with natural primary keys. This strategy lets the application assign identifier values by setting the identifier property before making the object persistent by calling `Save()`. This strategy has some serious disadvantages when you're working with detached objects and transitive persistence (both of these concepts are discussed in the next chapter). Don't use `assigned` identifiers if you can avoid them; it's much easier to use a surrogate primary key generated by one of the strategies listed in table 3.1.

For legacy data, the picture is more complicated. In this case, we're often stuck with natural keys and especially *composite keys* (natural keys composed of multiple table columns). Because composite identifiers can be more difficult to work with, we only discuss them in the context of chapter 9, section 9.2, "Legacy schemas and composite keys."

The next step is to add identifier properties to the classes of the CaveatEmptor application. Do *all* persistent classes have their own database identity? To answer this question, we must explore the distinction between *entities* and *value types* in NHibernate. These concepts are required for fine-grained object modeling.

3.6 Fine-grained object models

A major objective of the NHibernate project is support for *fine-grained* object models, which we isolated as the most important requirement for a rich domain model. It's one reason we've chosen POCOs.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

In crude terms, *fine-grained* means “more classes than tables.” For example, a user might have both a billing address and a home address. In the database, we might have a single `USER` table with the columns `BILLING_STREET`, `BILLING_CITY`, and `BILLING_ZIPCODE` along with `HOME_STREET`, `HOME_CITY`, and `HOME_ZIPCODE`. There are good reasons to use this somewhat denormalized relational model (performance, for one).

In our object model, we could use the same approach, representing the two addresses as six string-valued properties of the `User` class. But we would much rather model this using an `Address` class, where `User` has the `billingAddress` and `homeAddress` properties.

This object model achieves improved cohesion and greater code reuse and is more understandable. In the past, many ORM solutions haven’t provided good support for this kind of mapping.

NHibernate emphasizes the usefulness of fine-grained classes for implementing type-safety and behavior. For example, many people would model an email address as a string-valued property of `User`. We suggest that a more sophisticated approach is to define an actual `EmailAddress` class that could add higher level semantics and behavior. For example, it might provide a `SendEmail()` method.

3.6.1 Entity and value types

This leads us to a distinction of central importance in ORM. In .NET, all classes are of equal standing: All objects have their own identity and lifecycle, and all class instances are passed by reference. Only primitive types are passed by value.

We’re advocating a design in which there are more persistent classes than tables. One row represents multiple objects. Because database identity is implemented by primary key value, some persistent objects won’t have their own identity. In effect, the persistence mechanism implements pass-by-value semantics for some classes. One of the objects represented in the row has its own identity, and others depend on that.

NHibernate makes the following essential distinction:

- An object of entity type has its own database identity (primary key value). An object reference to an entity is persisted as a reference in the database (a foreign key value). An entity has its own lifecycle; it may exist independently of any other entity.
- An object of value type has no database identity; it belongs to an entity, and its persistent state is embedded in the table row of the owning entity (except in the case of collections, which are also considered value types, as you’ll see in chapter 6). Value types don’t have identifiers or identifier properties. The lifespan of a value-type instance is bounded by the lifespan of the owning entity.

The most obvious value types are simple objects like `Strings` and `Integers`. NHibernate also lets you treat a user-defined class as a value type, as you’ll see next. (We also come back to this important concept in chapter 6, section 6.1, “Understanding the NHibernate type system.”)

3.6.2 Using components

So far, the classes of our object model have all been entity classes with their own lifecycle and identity. The `User` class, however, has a special kind of association with the `Address` class, as shown in figure 3.5.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

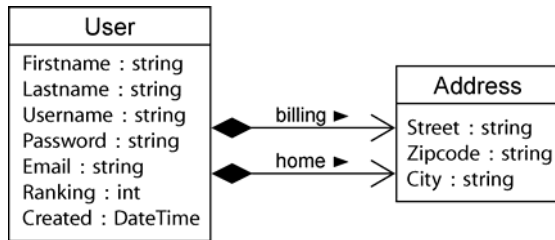


Figure 3.5 Relationships between `User` and `Address` using composition

In object modeling terms, this association is a kind of *aggregation*—a “part of” relationship. Aggregation is a strong form of association: It has additional semantics with regard to the lifecycle of objects. In our case, we have an even stronger form, *composition*, where the lifecycle of the part is dependent on the lifecycle of the whole.

Object modeling experts and UML designers will claim that there is no difference between this composition and other weaker styles of association when it comes to the .NET implementation. But in the context of ORM, there is a big difference: a composed class is often a candidate value type.

We now map `Address` as a value type and `User` as an entity. Does this affect the implementation of our POCO classes?

.NET itself has no concept of composition—a class or attribute can’t be marked as a component or composition. The only difference is the object identifier: A component has no identity; hence the persistent component class requires no identifier property or identifier mapping. The composition between `User` and `Address` is a metadata-level notion; we only have to tell NHibernate that the `Address` is a value type in the mapping document.

NHibernate uses the term *component* for a user-defined class that is persisted to the same table as the owning entity, as shown in listing 3.7. (The use of the word *component* here has nothing to do with the architecture-level concept, as in *software component*.)

Listing 3.7 Mapping the `User` class with a component `Address`

```

<class
  name="User"
  table="USER">
  <id
    name="Id"
    column="USER_ID"
    type="Int64">
    <generator class="native"/>
  </id>
  <property
    name="Username"
    column="USERNAME"
    type="String"/>
  <component
    name="HomeAddress"
    class="Address" >1
    <property name="Street"
      type="String"
      column="HOME_STREET"
      not-null="true"/>
    <property name="City"
      type="String"
      column="HOME_CITY"
      not-null="true"/>
    <property name="Zipcode"
      type="Int16"
      column="HOME_ZIPCODE"
      not-null="true"/>
  </component>
</component>

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

name="BillingAddress"
class="Address">
<property name="Street"
    type="String"
    column="BILLING_STREET"
    not-null="true"/>
<property name="City"
    type="String"
    column="BILLING_CITY"
    not-null="true"/>
<property name="Zipcode"
    type="Int16"
    column="BILLING_ZIPCODE"
    not-null="true"/>
</component>
...
</class>

```

#1 Declare persistent attributes
#2 Reuse component class

In #1, we declare the persistent attributes of `Address` inside the `<component>` element. The property of the `User` class is named `HomeAddress`. In #2, we reuse the same component class to map another property of this type to the same table.

Figure 3.6 shows how the attributes of the `Address` class are persisted to the same table as the `User` entity.

<<Table>> USER	
USER_ID <<PK>>	
USERNAME	

BILLING_STREET	Billing Address Component
BILLING_ZIPCODE	
BILLING_CITY	
HOME_STREET	Home Address Component
HOME_ZIPCODE	
HOME_CITY	

Figure 3.6 Table attributes of `User` with `Address` component

Components may be harder to map with `NHibernate.Mapping.Attributes`. When using a component in many classes with the identical mapping; it is very easy to do (far easier than with XML mapping):

```

[Component]
public class Address {
    [Property(NotNull=true)]
    public string Street { ... }
    [Property(NotNull=true)]
    public string City { ... }
    [Property(NotNull=true)]
    public short Zipcode { ... }
}
[Class]
class User {
    ...
    [ComponentProperty]
    public Address HomeAddress { ... }
}
[Class]
class House {
    ...
    [ComponentProperty]

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

    public Address Location { ... }
}

```

But the class `User` has two addresses, each mapped to different columns. There are many ways to map them (you can discover them in the appendix B). Here is one solution:

```

[Class]
class User {
    ...
    [Component(Name="HomeAddress", ClassType=typeof(Address))]
    protected class HomeAddressMapping {
        [Property(Column="HOME_STREET", NotNull=true)]
        public string Street { ... }
        [Property(Column="HOME_CITY", NotNull=true)]
        public string City { ... }
        [Property(Column="HOME_ZIPCODE", NotNull=true)]
        public short Zipcode { ... }
    }
    public Address HomeAddress { ... }
    [Component(Name="BillingAddress", ClassType=typeof(Address))]
    protected class BillingAddressMapping {
        [Property(Column="BILLING_STREET", NotNull=true)]
        public string Street { ... }
        [Property(Column="BILLING_CITY", NotNull=true)]
        public string City { ... }
        [Property(Column="BILLING_ZIPCODE", NotNull=true)]
        public short Zipcode { ... }
    }
    public Address BillingAddress { ... }
}

```

We simulate the hierarchy of the XML mapping using the classes `HomeAddressMapping` and `BillingAddressMapping` whose sole purpose is to provide the mapping. Note that `NHibernate.Mapping.Attributes` will automatically pick them because they belong to the class `User`. This solution is not very elegant. Hopefully, you will not have to deal with this kind of mapping very often.

Whenever you think that XML mapping would be easier to use than attributes, you can use the `[RawXml]` attribute to integrate this XML inside your attributes. This is probably the case here; so we can include the XML mapping of the components in listing 3.7 like this:

```

[Class]
class User {
    ...
    [RawXml( After=typeof(ComponentAttribute), Content=@"
        <component name=""HomeAddress"">
            ...
        </component>" )]
    public Address HomeAddress { ... }
    ...
}

```

The `[RawXml]` attribute has two properties: `After` tells after which kind of mapping the XML should be inserted; most of the time, it is the type of the attribute which is defined in the XML. This property is optional, in which case the XML is inserted on the top of the mapping. The second property is `Content`. It is the string containing the XML to include.

Notice that in this example, we have modeled the composition association as *unidirectional*. We can't navigate from `Address` to `User`. `NHibernate` supports both unidirectional and bidirectional compositions; however, unidirectional composition is far more common. Here's an example of a bidirectional mapping:

```

<component
  name="HomeAddress"
  class="Address">
  <parent name="Owner"/>
  <property name="Street" type="String" column="HOME_STREET"/>
  <property name="City" type="String" column="HOME_CITY"/>

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
<property name="Zipcode" type="short" column="HOME_ZIPCODE" />
</component>
```

The `<parent>` element maps a property of type `User` to the owning entity, in this example, the property is named `Owner`. We then call `Address.Owner` to navigate in the other direction.

A NHibernate component may own other components and even associations to other entities. This flexibility is the foundation of NHibernate's support for fine-grained object models. (We'll discuss various component mappings in chapter 6.)

However, there are two important limitations to classes mapped as components:

- Shared references aren't possible. The component `Address` doesn't have its own database identity (primary key) and so a particular `Address` object can't be referred to by any object other than the containing instance of `User`.
- There is no elegant way to represent a null reference to an `Address`. In lieu of an elegant approach, NHibernate represents null components as null values in all mapped columns of the component. This means that if you store a component object with all null property values, NHibernate will return a null component when the owning entity object is retrieved from the database.

Finally, it is also possible to make a component immutable using:

```
<component ... insert="false", update="false" />
```

Support for fine-grained classes isn't the only ingredient of a rich domain model. Class inheritance and polymorphism are defining features of object-oriented models.

3.7 Mapping class inheritance

A simple strategy for mapping classes to database tables might be “one table for every class.” This approach sounds simple, and it works well until you encounter inheritance.

Inheritance is the most visible feature of the structural mismatch between the object-oriented and relational worlds. Object-oriented systems model both “is a” and “has a” relationships. SQL-based models provide only “has a” relationships between entities.

There are three different approaches to representing an inheritance hierarchy. These were catalogued by Scott Ambler [Ambler 2002] in his widely read paper “Mapping Objects to Relational Databases”:

- *Table per concrete class*—Discard polymorphism and inheritance relationships completely from the relational model
- *Table per class hierarchy*—Enable polymorphism by denormalizing the relational model and using a type discriminator column to hold type information
- *Table per subclass*—Represent “is a” (inheritance) relationships as “has a” (foreign key) relationships

This section takes a *top down* approach; it assumes that we're starting with a domain model and trying to derive a new SQL schema. However, the mapping strategies described are just as relevant if we're working *bottom up*, starting with existing database tables.

3.7.1 Table per concrete class

Suppose we stick with the simplest approach: We could use exactly one table for each (non-abstract) class. All properties of a class, including inherited properties, could be mapped to columns of this table, as shown in figure 3.7.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

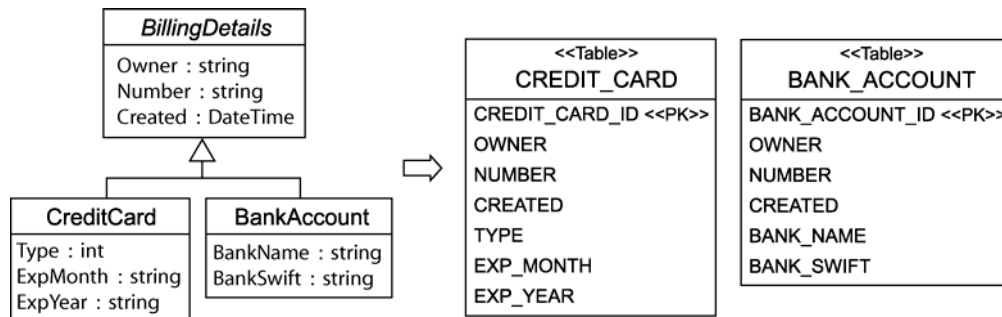


Figure 3.7 Mapping a composition bidirectional

The main problem with this approach is that it doesn't support polymorphic associations very well. In the database, associations are usually represented as foreign key relationships. In figure 3.7, if the subclasses are all mapped to different tables, a polymorphic association to their base class (abstract `BillingDetails` in this example) can't be represented as a simple foreign key relationship. This would be problematic in our domain model, because `BillingDetails` is associated with `User`; hence both tables would need a foreign key reference to the `USER` table.

Polymorphic queries (queries that return objects of all classes that match the interface of the queried class) are also problematic. A query against the base class must be executed as several SQL `SELECT`s, one for each concrete subclass. We might be able to use an SQL `UNION` to improve performance by avoiding multiple round trips to the database. However, unions are somewhat nonportable and otherwise difficult to work with. NHibernate doesn't support the use of unions at the time of writing, and will always use multiple SQL queries. For a query against the `BillingDetails` class (for example, restricting to a certain date of creation), NHibernate would use the following SQL:

```
select CREDIT_CARD_ID, OWNER, NUMBER, CREATED, TYPE, ...
from CREDIT_CARD
where CREATED = ?
select BANK_ACCOUNT_ID, OWNER, NUMBER, CREATED, BANK_NAME, ...
from BANK_ACCOUNT
where CREATED = ?
```

Notice that a separate query is needed for each concrete subclass.

On the other hand, queries against the concrete classes are trivial and perform well:

```
select CREDIT_CARD_ID, TYPE, EXP_MONTH, EXP_YEAR
from CREDIT_CARD where CREATED = ?
```

Note that here, and in other places in this book, we show SQL that is *conceptually* identical to the SQL executed by NHibernate. The actual SQL might look superficially different.

A further conceptual problem with this mapping strategy is that several different columns of different tables share the same semantics. This makes schema evolution more complex. For example, a change to a base class property type results in changes to multiple columns. It also makes it much more difficult to implement database integrity constraints that apply to all subclasses.

This mapping strategy doesn't require any special NHibernate mapping declaration: Simply create a new `<class>` declaration for each concrete class, specifying a different `table` attribute for each. We recommend this approach (only) for the top level of your class hierarchy, where polymorphism isn't usually required.

3.7.2 Table per class hierarchy

Alternatively, an entire class hierarchy could be mapped to a single table. This table would include columns for all properties of all classes in the hierarchy. The concrete subclass represented by a particular row is identified by the value of a *type discriminator* column. This approach is shown in figure 3.8.

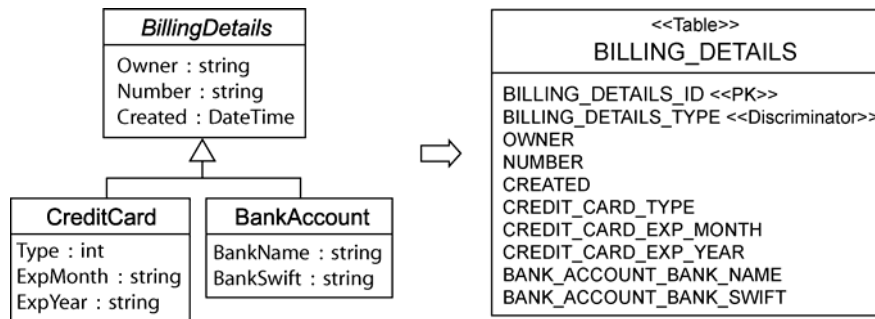


Figure 3.8 Table per class hierarchy mapping

This mapping strategy is a winner in terms of both performance and simplicity. It's the best-performing way to represent polymorphism—both polymorphic and nonpolymorphic queries perform well—and it's even easy to implement by hand. Ad hoc reporting is possible without complex joins or unions, and schema evolution is straightforward.

There is one major problem: Columns for properties declared by subclasses must be declared to be nullable. If your subclasses each define several non-nullable properties, the loss of **NOT NULL** constraints could be a serious problem from the point of view of data integrity.

In NHibernate, we use the `<subclass>` element to indicate a table-per-class hierarchy mapping, as in listing 3.8.

Listing 3.8 NHibernate `<subclass>` mapping

```

<hibernate-mapping>
  <class name="BillingDetails" table="BILLING_DETAILS" discriminator-value="BD"> |1
    <id name="Id" column="BILLING_DETAILS_ID" type="Int64">
      <generator class="native" />
    </id>
    <discriminator column="BILLING_DETAILS_TYPE" type="String" /> |2
    <property name="Name" column="OWNER" type="String" /> |3
    ...
    <subclass name="CreditCard" discriminator-value="CC"> |4
      <property name="Type" column="CREDIT_CARD_TYPE" />
      ...
    </subclass>
  </class>
</hibernate-mapping>

```

#1 Root class, mapped to table

#2 Discriminator column

#3 Property mappings

#4 CreditCard subclass

#1 The root class **BillingDetails** of the inheritance hierarchy is mapped to the table **BILLING_DETAILS**.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

#2 We have to use a special column to distinguish between persistent classes: the discriminator. This isn't a property of the persistent class; it's used internally by NHibernate. The column name is `BILLING_DETAILS_TYPE`, and the values will be strings—in this case, "CC" or "BA". NHibernate will automatically set and retrieve the discriminator values.

#3 Properties of the base class are mapped as always, with a `<property>` element.

#4 Every subclass has its own `<subclass>` element. Properties of a subclass are mapped to columns in the `BILLING_DETAILS` table. Remember that `not-null` constraints aren't allowed, because a `CreditCard` instance won't have a `BankSwift` property and the `BANK_ACCOUNT_BANK_SWIFT` field must be null for that row.

The `<subclass>` element can in turn contain other `<subclass>` elements, until the whole hierarchy is mapped to the table. A `<subclass>` element can't contain a `<joined-subclass>` element. (The `<joined-subclass>` element is used in the specification of the third mapping option: one table per subclass. This option is discussed in the next section.) The mapping strategy can't be switched anymore at this point.

Here are the classes mapped using `NHibernate.Mapping.Attributes`:

```
[Class(Table="BILLING_DETAILS", DiscriminatorValue="BD")]
public class BillingDetails {
    [Id(Name="Id", Column="BILLING_DETAILS_ID")]
    [Generator(1, Class="native")]
    [Discriminator(2, Column="BILLING_DETAILS_TYPE",
        TypeType=typeof(string))]
    public long Id { ... }
    [Property(Column="OWNER")]
    public string Name { ... }

    [Subclass(DiscriminatorValue="CC")]
    public class CreditCard : BillingDetails {
        [Property(Column="CREDIT_CARD_TYPE")]
        public CreditCardType Type { ... }
    }
    ...
}
```

Remember that when we want to specify a class in the mapping, we can add "Type" to the element's name; for the attribute `[Discriminator]`, we used `TypeType`. Note that this attribute can be written before any property as it isn't linked to any field/property of the class (if there is more than this attribute on the property, make sure that it comes after the `[Id]` and before the other attributes).

NHibernate would use the following SQL when querying the `BillingDetails` class:

```
select BILLING_DETAILS_ID, BILLING_DETAILS_TYPE,
       OWNER, ..., CREDIT_CARD_TYPE,
from BILLING_DETAILS
where CREATED = ?
```

To query the `CreditCard` subclass, NHibernate would use a condition on the discriminator:

```
select BILLING_DETAILS_ID,
       CREDIT_CARD_TYPE, CREDIT_CARD_EXP_MONTH, ...
from BILLING_DETAILS
where BILLING_DETAILS_TYPE='CC' and CREATED = ?
```

How could it be any simpler than that?

Instead of having a discriminator field, it is possible to use an arbitrary SQL formula. For example:

```
<discriminator type="String"
  formula="case when CREDIT_CARD_TYPE is null then 'BD' else 'CC' end"
/>
```

Here, we use the column `CREDIT_CARD_TYPE` to evaluate the type.

Now, let's discover the alternative to table-per-class-hierarchy.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.7.3 Table per subclass

The third option is to represent inheritance relationships as relational foreign key associations. *Every* subclass that declares persistent properties—including abstract classes and even interfaces—has its own table.

Unlike the strategy that uses a table per concrete class, the table here contains columns only for each *non-inherited* property (each property declared by the subclass itself) along with a primary key that is also a foreign key of the base class table. This approach is shown in figure 3.9.

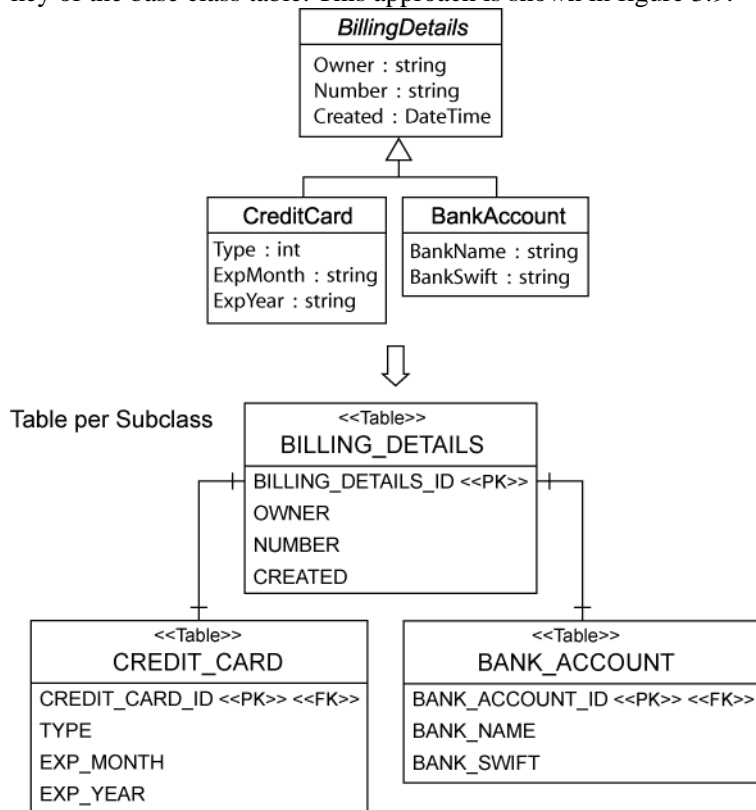


Figure 3.9 Table per subclass mapping

If an instance of the `CreditCard` subclass is made persistent, the values of properties declared by the `BillingDetails` base class are persisted to a new row of the `BILLING_DETAILS` table. Only the values of properties declared by the subclass are persisted to the new row of the `CREDIT_CARD` table. The two rows are linked together by their shared primary key value. Later, the subclass instance may be retrieved from the database by joining the subclass table with the base class table.

The primary advantage of this strategy is that the relational model is completely normalized. Schema evolution and integrity constraint definition are straightforward. A polymorphic association to a particular subclass may be represented as a foreign key pointing to the table of that subclass.

In NHibernate, we use the `<joined-subclass>` element to indicate a table-per-subclass mapping (see listing 3.9).

Listing 3.9 NHibernate `<joined-subclass>` mapping

```
<?xml version="1.0"?>
<hibernate-mapping>
  <class name="BillingDetails" table="BILLING_DETAILS">
    <id name="Id" />
  </class>
  <joined-subclass name="CreditCard" table="CREDIT_CARD">
    <id name="CreditCardId" />
  </joined-subclass>
  <joined-subclass name="BankAccount" table="BANK_ACCOUNT">
    <id name="BankAccountId" />
  </joined-subclass>
</hibernate-mapping>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        column="BILLING_DETAILS_ID"
        type="Int64">
        <generator class="native"/>
    </id>
    <property
        name="Owner"
        column="OWNER"
        type="String"/>
        ...
    <joined-subclass                                | 2
        name="CreditCard"
        table="CREDIT_CARD">
        <key column="CREDIT_CARD_ID">                | 3
        <property
            name="Type"
            column="TYPE"/>
        ...
    </joined-subclass>
    ...
</class>
</hibernate-mapping>

```

#1 BillingDetails root class, mapped to BILLING_DETAILS table

#2 <joined-subclass> element

#3 Primary/foreign key

#1 Again, the root class `BillingDetails` is mapped to the table `BILLING_DETAILS`. Note that no discriminator is required with this strategy.

#2 The new `<joined-subclass>` element is used to map a subclass to a new table (in this example, `CREDIT_CARD`). All properties declared in the joined subclass will be mapped to this table. Note that we intentionally left out the mapping example for `BankAccount`, which is similar to `CreditCard`.

#3 A primary key is required for the `CREDIT_CARD` table; it will also have a foreign key constraint to the primary key of the `BILLING_DETAILS` table. A `CreditCard` object lookup will require a join of both tables.

A `<joined-subclass>` element may contain other `<joined-subclass>` elements but not a `<subclass>` element. NHibernate doesn't support mixing of these two mapping strategies.

NHibernate will use an outer join when querying the `BillingDetails` class:

```

select BD.BILLING_DETAILS_ID, BD.OWNER, BD.NUMER, BD.CREATED,
       CC.TYPE, ..., BA.BANK_SWIFT, ...
case
  when CC.CREDIT_CARD_ID is not null then 1
  when BA.BANK_ACCOUNT_ID is not null then 2
  when BD.BILLING_DETAILS_ID is not null then 0
end as TYPE
from BILLING_DETAILS BD
left join CREDIT_CARD CC on
  BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
left join BANK_ACCOUNT BA on
  BD.BILLING_DETAILS_ID = BA.BANK_ACCOUNT_ID
where BD.CREATED = ?

```

The SQL `case` statement uses the existence (or nonexistence) of rows in the subclass tables `CREDIT_CARD` and `BANK_ACCOUNT` to determine the concrete subclass for a particular row of the `BILLING_DETAILS` table.

To narrow the query to the subclass, NHibernate uses an inner join instead:

```

select BD.BILLING_DETAILS_ID, BD.OWNER, BD.CREATED, CC.TYPE, ...
from CREDIT_CARD CC
inner join BILLING_DETAILS BD on
  BD.BILLING_DETAILS_ID = CC.CREDIT_CARD_ID
where CC.CREATED = ?

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

As you can see, this mapping strategy is more difficult to implement by hand—even ad hoc reporting will be more complex. This is an important consideration if you plan to mix NHibernate code with handwritten SQL/ADO.NET. (For ad hoc reporting, database views provide a way to offset the complexity of the table-per-subclass strategy. A view may be used to transform the table-per-subclass model into the much simpler table-per-hierarchy model.)

Furthermore, even though this mapping strategy is deceptively simple, our experience is that performance may be unacceptable for complex class hierarchies. Queries always require either a join across many tables or many sequential reads. Our problem should be recast as how to choose an appropriate *combination* of mapping strategies for our application’s class hierarchies. A typical domain model design has a mix of interfaces and abstract classes.

3.7.4 Choosing a strategy

You can apply all mapping strategies to abstract classes and interfaces. Interfaces may have no state but may contain property declarations, so they can be treated like abstract classes. You can map an interface using `<class>`, `<subclass>`, or `<joined-subclass>`; and you can map any declared or inherited property using `<property>`. NHibernate won’t try to instantiate an abstract class, however, even if you query or load it.

Here are some rules of thumb:

- If you don’t require polymorphic associations or queries, lean toward the table-per-concrete-class strategy. If you require polymorphic associations (an association to a base class, hence to all classes in the hierarchy with dynamic resolution of the concrete class at runtime) or queries, and subclasses declare relatively few properties (particularly if the main difference between subclasses is in their behavior), lean toward the table-per-class-hierarchy model.
- If you require polymorphic associations or queries, and subclasses declare many properties (subclasses differ mainly by the data they hold), lean toward the table-per-subclass approach.

By default, choose table-per-class-hierarchy for simple problems. For more complex cases (or when you’re overruled by a data modeler insisting upon the importance of nullability constraints), you should consider the table-per-subclass strategy. But at that point, ask yourself whether it might be better to remodel inheritance as delegation in the object model. Complex inheritance is often best avoided for all sorts of reasons unrelated to persistence or ORM. NHibernate acts as a buffer between the object and relational models, but that doesn’t mean you can completely ignore persistence concerns when designing your object model.

Note that you may also use `<subclass>` and `<joined-subclass>` mapping elements in a separate mapping file (as a top-level element, instead of `<class>`). You then have to declare the class that is extended (for example, `<subclass name="CreditCard" extends="BillingDetails">`), and the base class mapping must be loaded before the subclass mapping file. This technique allows you to extend a class hierarchy without modifying the mapping file of the base class. Using `NHibernate.Mapping.Attributes`, you can move the implementation of `CreditCard` in another file and map it like this:

```
[Subclass(ExtendsType=typeof(BillingDetails), DiscriminatorValue="CC")]
public class CreditCard : BillingDetails {
    ...
}
```

You have now seen the intricacies of mapping an entity in isolation. In the next section, we turn to the problem of mapping associations between entities, which is another major issue arising from the object/relational paradigm mismatch.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.8 Introducing associations

Managing the associations between classes and the relationships between tables is the soul of ORM. Most of the difficult problems involved in implementing an ORM solution relate to association management.

The NHibernate association model is extremely rich but is not without pitfalls, especially for new users. In this section, we won't try to cover all the possible combinations. What we'll do is examine certain cases that are extremely common. We return to the subject of association mappings in chapter 7, for a more complete treatment.

But first, there's something we need to explain up front.

3.8.1 Unidirectional associations

When using (typed) DataSets, associations are represented like in database. To link two entities, you have to set the foreign key in one entity to the primary key of the other. There is not the notion of collection; so you can't add an entity to a collection and get the association created.

Transparent POCO-oriented persistence implementations such as NHibernate provide support for collections. But it is important to understand that, NHibernate associations are all inherently *unidirectional*. As far as NHibernate is concerned, the association from `Bid` to `Item` is a *different association* than the association from `Item` to `Bid`. This means that `bid.Item=item` and `item.Bids.Add(bid)` are two unrelated operations.

To some people, this seems strange; to others, it feels completely natural. After all, associations at the language level are always unidirectional—and NHibernate claims to implement persistence for plain .NET objects. We'll merely observe that this decision was made because NHibernate objects are not bound to any context. In NHibernate applications, the behavior of a non-persistent instance is the same as the behavior of a persistent instance.

Because associations are so important, we need a very precise language for classifying them.

3.8.2 Multiplicity

In describing and classifying associations, we'll almost always use the association *multiplicity*. Look at figure 3.10.

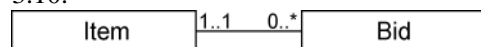


Figure 3.10 Relationship between `Item` and `Bid`

For us, the multiplicity is just two bits of information:

- Can there be more than one `Bid` for a particular `Item`?
- Can there be more than one `Item` for a particular `Bid`?

After glancing at the object model, we conclude that the association from `Bid` to `Item` is a *many-to-one* association. Recalling that associations are directional, we would also call the inverse association from `Item` to `Bid` a *one-to-many* association.

(Clearly, there are two more possibilities: *many-to-many* and *one-to-one*; we'll get back to these possibilities in chapter 6.)

In the context of object persistence, we aren't interested in whether "many" really means "two" or "maximum of five" or "unrestricted."

3.8.3 The simplest possible association

The association from `Bid` to `Item` is an example of the simplest possible kind of association in ORM. The object reference returned by `bid.Item` is easily mapped to a foreign key column in the `BID` table. First, here's the C# class implementation of `Bid` mapped using .NET attributes:

```
[Class(Table="BID")]
public class Bid {
    ...
    private Item item;
    [ManyToOne(Column="ITEM_ID", NotNull=true)]
    public Item Item {
        get { return item; }
        set { item = value; }
    }
    ...
}
```

Next, here's the corresponding NHibernate mapping for this association:

```
<class
  name="Bid"
  table="BID">
  ...
  <many-to-one
    name="Item"
    column="ITEM_ID"
    class="Item"
    not-null="true" />
</class>
```

This mapping is called a *unidirectional many-to-one association*. The column `ITEM_ID` in the `BID` table is a foreign key to the primary key of the `ITEM` table.

We have explicitly specified the class `Item`, which the association refers to. This specification is usually optional, since NHibernate can determine this using reflection.

We specified the `not-null` attribute because we can't have a bid without an item. The `not-null` attribute doesn't affect the runtime behavior of NHibernate in this case; it exists mainly to control automatic data definition language (DDL) generation (see chapter 10).

In some legacy databases, it may happen that a many-to-one association points to a nonexistent entity. The property `not-found` allows you to define how NHibernate should react to this situation.

```
<many-to-one ... not-found="ignore|exception" />
```

Using `not-found="exception"` (the default value), NHibernate will throw an exception. And `not-found="ignore"` will make NHibernate ignore this association (leaving it null).

3.8.4 Making the association bidirectional

So far so good. But we also need to be able to easily fetch all the bids for a particular item. We need a bidirectional association here, so we have to add scaffolding code to the `Item` class:

```
public class Item {
    ...
    private ISet bids = new HashSet();
    public ISet Bids {
        get { return bids; }
        set { bids = value; }
    }
    public void AddBid(Bid bid) {
        bid.Item = this;
        bids.Add(bid);
    }
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
    ...  
}
```

You can think of the code in `AddBid()` (a convenience method) as implementing a strong bidirectional association in the object model.

A basic mapping for this *one-to-many association* would look like this:

```
[Set]  
    [Key(1, Column="ITEM_ID")]  
    [OneToMany(2, ClassType=typeof(Bid))]  
public ISet Bids { ... }
```

Here is the equivalent XML wrapped in its class mapping:

```
<class  
    name="Item"  
    table="ITEM">  
    ...  
    <set name="Bids">  
        <key column="ITEM_ID"/>  
        <one-to-many class="Bid"/>  
    </set>  
</class>
```

The column mapping defined by the `<key>` element is a foreign key column of the associated `BID` table. Notice that we specify the same foreign key column in this collection mapping that we specified in the mapping for the many-to-one association. The table structure for this association mapping is shown in figure 3.11.

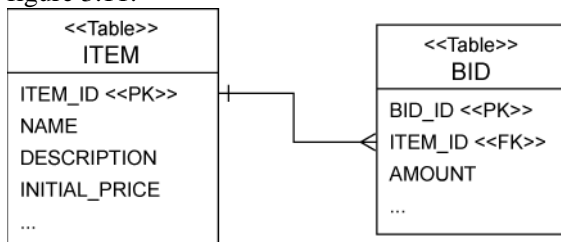


Figure 3.11 Table relationships and keys for a one-to-many/many-to-one mapping

Now we have two different unidirectional associations mapped to the same foreign key, which poses a problem. At runtime, there are two different in-memory representations of the same foreign key value: the `item` property of `Bid` and an element of the `bids` collection held by an `Item`. Suppose our application modifies the association by, for example, adding a bid to an item in this fragment of the `AddBid()` method:

```
bid.Item = this;  
bids.Add(bid);
```

This code is fine, but in this situation, NHibernate detects two different changes to the in-memory persistent instances. From the point of view of the database, just one value must be updated to reflect these changes: the `ITEM_ID` column of the `BID` table. *NHibernate doesn't transparently detect the fact that the two changes refer to the same database column, since at this point we've done nothing to indicate that this is a bidirectional association.*

We need one more thing in our association mapping to tell NHibernate to treat this as a bidirectional association: The `inverse` attribute tells NHibernate that the collection is a mirror image of the many-to-one association on the other side:

```
<class  
    name="Item"  
    table="ITEM">  
    ...  
    <set  
        name="bids"  
        inverse="true">  
            <key column="ITEM_ID"/>  
            <one-to-many class="Bid"/>  
        </set>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
</class>
```

Without the `inverse` attribute, NHibernate would try to execute two different SQL statements, both updating the same foreign key column, when we manipulate the association between the two instances. By specifying `inverse="true"`, we explicitly tell NHibernate which end of the association it should synchronize with the database. In this example, we tell NHibernate that it should propagate changes made at the `Bid` end of the association to the database, ignoring changes made only to the `bids` collection. Thus if we only call `item.Bids.Add(bid)`, no changes will be made persistent. This is consistent with the behavior in .NET without NHibernate: If an association is bidirectional, you have to create the link on two sides, not just one.

We now have a working *bidirectional many-to-one association* (which could also be called a bidirectional one-to-many association, of course).

One final piece is missing. We explore the notion of *transitive persistence* in much greater detail in the next chapter. For now, we'll introduce the concepts of *cascading save* and *cascading delete*, which we need in order to finish our mapping of this association.

When we instantiate a new `Bid` and add it to an `Item`, the bid should become persistent immediately. We would like to avoid the need to explicitly make a `Bid` persistent by calling `Save()` on the `ISession` interface.

We make one final tweak to the mapping document to enable cascading save:

```
<class
  name="Item"
  table="ITEM">
  ...
  <set
    name="Bids"
    inverse="true"
    cascade="save-update">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
  </set>
</class>
```

The `cascade` attribute tells NHibernate to make any new `Bid` instance persistent (that is, save it in the database) if the `Bid` is referenced by a persistent `Item`.

The `cascade` attribute is directional: It applies to only one end of the association. We could also specify `cascade="save-update"` for the many-to-one association declared in the mapping for `Bid`, but doing so would make no sense in this case because `Bids` are created after `Items`.

Are we finished? Not quite. We still need to define the lifecycle for both entities in our association.

3.8.5 A parent/child relationship

With the previous mapping, the association between `Bid` and `Item` is fairly loose. We would use this mapping in a real system if both entities had their own lifecycle and were created and removed in unrelated business processes. Certain associations are much stronger than this; some entities are bound together so that their lifecycles aren't truly independent. In our example, it seems reasonable that deletion of an item implies deletion of all bids for the item. A particular bid instance references only one item instance for its entire lifetime. In this case, cascading both saves and deletions makes sense.

If we enable cascading delete, the association between `Item` and `Bid` is called a *parent/child relationship*. In a parent/child relationship, the parent entity is responsible for the lifecycle of its associated child entities. This is the same semantics as a composition (using NHibernate components), but in this case only entities are involved; `Bid` isn't a value type. The advantage of using a parent/child relationship is that the child may be

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

loaded individually or referenced directly by another entity. A bid, for example, may be loaded and manipulated without retrieving the owning item. It may be stored without storing the owning item at the same time. Furthermore, we reference the same `Bid` instance in a second property of `Item`, the single `SuccessfulBid` (see figure 3.2). Objects of value type can't be shared.

To remodel the `Item` to `Bid` association as a parent/child relationship, the only change we need to make is to the `cascade` attribute:

```
<class
  name="Item"
  table="ITEM">
  ...
  <set
    name="Bids"
    inverse="true"
    cascade="all-delete-orphan">
    <key column="ITEM_ID" />
    <one-to-many class="Bid" />
  </set>
</class>
```

We used `cascade="all-delete-orphan"` to indicate the following:

- Any newly instantiated `Bid` becomes persistent if the `Bid` is referenced by a persistent `Item` (as was also the case with `cascade="save-update"`). Any persistent `Bid` should be deleted if it's referenced by an `Item` when the item is deleted.
- Any persistent `Bid` should be deleted if it's removed from the `bids` collection of a persistent `Item`. (NHibernate will assume that it was only referenced by this item and consider it an orphan.)

We have achieved the following with this mapping: A `Bid` is removed from the database if it's removed from the collection of `Bids` of the `Item` (or it's removed if the `Item` itself is removed).

The cascading of operations to associated entities is NHibernate's implementation of *transitive persistence*. We look more closely at this concept in chapter 4, section 4.3, "Using transitive persistence in NHibernate."

We have covered only a tiny subset of the association options available in NHibernate. However, you already have enough knowledge to be able to build entire applications. The remaining options are either rare or are variations of the associations we have described.

We recommend keeping your association mappings simple, using NHibernate queries for more complex tasks.

3.9 Summary

In this chapter, we focused on the structural aspect of the object/relational paradigm mismatch and discussed the first four generic ORM problems. We discussed the programming model for persistent classes and the NHibernate ORM metadata for fine-grained classes, object identity, inheritance, and associations.

You now understand that persistent classes in a domain model should be free of cross-cutting concerns such as transactions and security. Even persistence-related concerns shouldn't leak into the domain model. We no longer entertain the use of restrictive programming models such as `DataSets` for our domain model. Instead, we use transparent persistence, together with the unrestrictive `POCO` programming model—which is really a set of best practices for the creation of properly encapsulated `.NET` types.

You also learned about the important differences between *entities* and *value-typed* objects in NHibernate. Entities have their own identity and lifecycle, whereas value-typed objects are dependent on an entity and are persisted with by-value semantics. NHibernate allows fine-grained object models with fewer tables than persistent classes.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Finally, we have introduced the three well-known inheritance-mapping strategies in NHibernate. We have also covered associations and collections mapping and we have implemented and mapped our first parent/child association between persistent classes, using database foreign key fields and the cascading of operations.

With this understanding, you should find you can experiment with NHibernate and handle most common mapping scenarios, perhaps some of the thornier ones too.

As you become familiar with creating domain models and persisting them with NHibernate, you may face other architectural challenges. We next investigate the dynamic aspects of the object/relational mismatch, including a much deeper study of the cascaded operations we introduced and the lifecycle of persistent objects.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Working with persistent objects

1.1 What is Persistence?	6
1.1.1 Relational databases.....	6
1.1.2 Understanding SQL.....	7
1.1.3 Using SQL in .NET Applications.....	7
1.1.4 Persistence in object oriented applications.....	7
1.1.5 Persistence and the Layered Architecture.....	8
1.2 Approaches to Persistence in .NET.....	10
1.2.1 Choice of Persistence Layer.....	10
DataSet-based persistence layer.....	10
Hand-coded persistence layer.....	10
Persistence layer using NHibernate.....	11
1.2.2 Implementing the entities.....	11
Entities in a DataSet.....	11
Hand-coded entities.....	11
Entities and NHibernate.....	12
1.2.3 Displaying entities in the user interface.....	12
DataSet-based presentation layer.....	12
Presentation layer and entities.....	12
1.2.4 Implementing CRUD operations.....	13
CRUD operations with DataSets.....	13
Hand-coded CRUD operations.....	13
CRUD operations using NHibernate.....	14
1.3 Why do we need NHibernate?.....	14
1.3.1 The paradigm mismatch.....	14
The problem of granularity.....	14
The problem of inheritance and polymorphism.....	15
The problem of identity.....	15
Problems relating to associations.....	15
1.3.2 Unit of Work and Conversations.....	15
The Unit of Work pattern.....	16
Transparent persistence and lazy loading.....	16
Caching.....	17
1.3.3 Complex queries and ADO.NET Entity Framework.....	17
Implementing a query engine.....	17
ADO.NET Entity Framework.....	18
1.4 Object/Relational Mapping.....	19
What is ORM?.....	19
1.4.1 Why ORM?.....	20
Modeling mismatch.....	20
Productivity and maintainability.....	20
Performance.....	20
Database independence.....	21
1.5 Summary.....	21
2.1 “Hello World” with NHibernate.....	22
2.1.1 Installing NHibernate.....	22
2.1.2 Create a new Visual Studio Project.....	23

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

2.1.3	Creating the Employee Class	23
2.1.4	Setting up the database	24
2.1.3	Creating an Employee and saving to the Database	25
1.2.4	Loading an Employee from the Database	26
1.2.5	Creating a Mapping File	26
1.2.6	Configuring Your Application	27
1.2.7	Updating an Employee	28
1.2.8	– Running the Program	29
2.2	Understanding the Architecture	30
2.2.1	The core interfaces	31
I	Session interface	31
I	SessionFactory interface	31
C	onfiguration interface	31
I	Transaction interface	32
I	Query and ICriteria interfaces	32
2.2.2	Callback interfaces	32
2.2.3	Types	33
2.2.4	Extension interfaces	33
2.3	Basic configuration	34
2.3.1	Creating a SessionFactory	34
2.3.2	Configuration of the ADO.NET database access	36
C	onfiguration of NHibernate using hibernate.cfg.xml	37
S	tarting NHibernate	38
2.4	Advanced configuration settings	39
2.4.1	Using the application configuration file	39
2.4.2	Logging	41
2.5	Summary	42
3.1	The CaveatEmptor application	43
3.1.1	Analyzing the business domain	44
3.1.2	The CaveatEmptor domain model	44
3.2	Implementing the domain model	46
3.2.1	Addressing leakage of concerns	46
3.2.2	Transparent and automated persistence	46
3.2.3	Writing POCOs	47
3.2.4	Implementing POCO associations	48
3.2.5	Adding logic to properties	51
3.3	Defining the mapping metadata	52
3.3.1	Mapping using XML	53
3.3.2	Attribute-oriented programming	54
3.4	Basic property and class mappings	56
3.4.1	Property mapping overview	56
3.4.2	Using derived properties	57
3.4.3	Property access strategies	57
3.4.4	Taking advantage of the reflection optimizer	59
3.4.5	Controlling insertion and updates	60
3.4.6	Using quoted SQL identifiers	60
3.4.7	Naming conventions	61
3.4.8	SQL schemas	62
3.4.9	Declaring class names	62
3.4.10	Manipulating metadata at runtime	63
3.5	Understanding object identity	64
3.5.1	Identity versus equality	64
3.5.2	Database identity with NHibernate	65
3.5.3	Choosing primary keys	67

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

3.6 Fine-grained object models.....	68
3.6.1 Entity and value types.....	69
3.6.2 Using components.....	69
3.7 Mapping class inheritance.....	73
3.7.1 Table per concrete class.....	73
3.7.2 Table per class hierarchy.....	74
3.7.3 Table per subclass.....	77
3.7.4 Choosing a strategy.....	79
3.8 Introducing associations.....	80
3.8.1 Unidirectional associations.....	80
3.8.2 Multiplicity.....	80
3.8.3 The simplest possible association.....	81
3.8.4 Making the association bidirectional.....	81
3.8.5 A parent/child relationship.....	83
3.9 Summary.....	84
4.1 The persistence lifecycle.....	89
4.1.1 Transient objects.....	90
4.1.2 Persistent objects.....	90
4.1.3 Detached objects.....	91
4.1.4 The scope of object identity.....	92
4.1.5 Outside the identity scope.....	93
4.1.6 Implementing Equals() and GetHashCode().....	94
Using database identifier equality.....	94
Comparing by value.....	95
Using business key equality.....	96
4.2 The persistence manager.....	97
4.2.1 Making an object persistent.....	97
4.2.2 Updating the persistent state of a detached instance.....	98
4.2.3 Retrieving a persistent object.....	99
4.2.4 Updating a persistent object.....	99
4.2.5 Making an object transient.....	100
4.3 Using transitive persistence in NHibernate.....	100
4.3.1 Persistence by reachability.....	101
4.3.2 Cascading persistence with NHibernate.....	102
4.3.3 Managing auction categories.....	102
4.3.4 Distinguishing between transient and detached instances.....	105
4.4 Retrieving objects.....	106
4.4.1 Retrieving objects by identifier.....	106
4.4.2 Introducing the Hibernate Query Language.....	107
4.4.3 Query by criteria.....	108
4.4.4 Query by example.....	108
4.4.5 Fetching strategies.....	109
Immediate fetching.....	110
Lazy fetching.....	110
Eager (outer join) fetching.....	110
Batch fetching.....	110
4.4.6 Selecting a fetching strategy in mappings.....	110
Single point associations.....	111
Collections.....	112
Setting the fetch depth.....	113
Initializing lazy associations.....	114
4.4.7 Tuning object retrieval.....	114
4.5 Summary.....	115

This chapter covers

- The lifecycle of objects in a NHibernate application

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

- Using the session persistence manager
- Transitive persistence
- Efficient fetching strategy

You now have an understanding of how NHibernate and ORM solve the static aspects of the object/relational mismatch. With what you know so far, it's possible to solve the structural mismatch problem, but an efficient solution to the problem requires something more. We must investigate strategies for runtime data access, since they're crucial to the performance of our applications. You need to learn how to efficiently store and load objects.

This chapter covers the *behavioral* aspect of the object/relational mismatch, listed in chapter 1 as the four O/R mapping problems described in section 1.3.1. We consider these problems to be at least as important as the structural problems discussed in chapter 3. In our experience, many developers are only aware of the structural mismatch and rarely pay attention to the more dynamic behavioral aspects.

In this chapter, we discuss the lifecycle of objects—how an object becomes persistent, and how it stops being considered persistent—and the method calls and other actions that trigger these transitions. The NHibernate persistence manager, the `ISession`, is responsible for managing object state, so you'll learn how to use this important API.

Retrieving object graphs efficiently is another central concern, so we introduce the basic strategies in this chapter. NHibernate provides several ways to specify queries that return objects without losing much of the power inherent to SQL. Because network latency caused by remote access to the database can be an important limiting factor in the overall performance of .NET applications, you must learn how to retrieve a graph of objects with a minimal number of database hits.

Let's start by discussing objects, their lifecycle, and the events that trigger a change of persistent state. These basics will give you the background you need when working with your object graph, so you'll know when and how to load and save your objects. The material might be formal, but a solid understanding of the *persistence lifecycle* is essential.

4.1 The persistence lifecycle

Since NHibernate is a transparent persistence mechanism, classes are unaware of their own persistence capability. It's therefore possible to write application logic that is unaware of whether the objects it operates on represent persistent state or temporary state that exists only in memory. The application shouldn't necessarily need to care that an object is persistent when invoking its methods.

However, in any application with persistent state, the application must interact with the persistence layer whenever it needs to transmit state held in memory to the database (or vice versa). To do this, you call NHibernate's persistence manager and query interfaces. When interacting with the persistence mechanism that way, it's necessary for the application to concern itself with the state and lifecycle of an object with respect to persistence. We'll refer to this as the *persistence lifecycle*.

Different ORM implementations use different terminology and define different states and state transitions for the persistence lifecycle. Moreover, the object states used internally might be different from those exposed to the client application. NHibernate defines only three states, hiding the complexity of its internal implementation from the client code. In this section, we explain these three states: *transient*, *persistent*, and *detached*.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Let's look at these states and their transitions in a state chart, shown in figure 4.1. You can also see the method calls to the persistence manager that trigger transitions. We discuss this chart in this section; refer to it later whenever you need an overview.

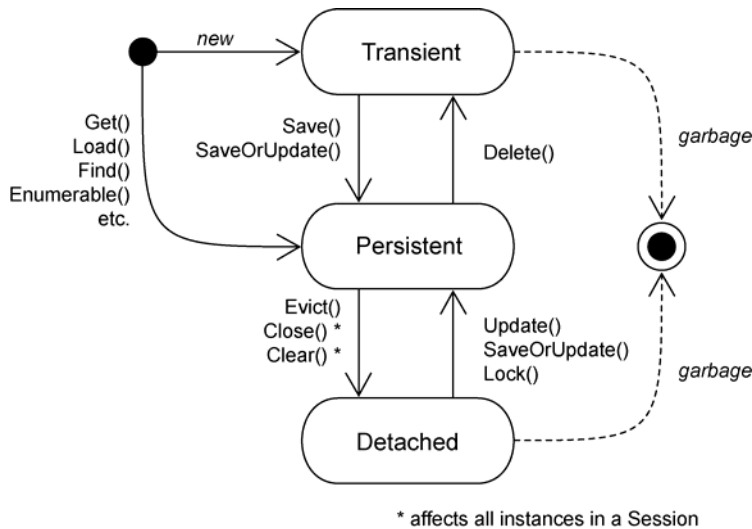


Figure 4.1 States of an object and transitions in a NHibernate application

In its lifecycle, an object can transition from a transient object to a persistent object to a detached object. Let's take a closer look at each of these states.

4.1.1 Transient objects

In NHibernate, objects instantiated using the `new` operator aren't immediately persistent. Their state is *transient*, which means they aren't associated with any database table row, and so they are just like any other object in a .NET application. More specifically, their state is lost as soon as they're dereferenced (no longer referenced by any other object) by the application. These objects have a lifespan that effectively ends at that time, and they become inaccessible and available for garbage collection.

NHibernate considers all transient instances to be nontransactional; a modification to the state of a transient instance isn't made in the context of any transaction. This means NHibernate doesn't provide any rollback functionality for transient objects. In fact, NHibernate doesn't roll back any object changes, as you'll see later.

Objects that are referenced only by other transient instances are, by default, also transient. An instance can transition from transient to persistent state in two ways. One is to `Save()` it using the persistence manager, another way is to create a reference to it from an already persistent instance.

4.1.2 Persistent objects

A persistent instance is any instance with a *database identity*, as defined in chapter 3, section 3.5, "Understanding object identity." That means a persistent instance has a primary key value set as its database identifier.

Persistent instances might be objects instantiated by the application and then made persistent by calling the `Save()` method of the persistence manager (the NHibernate `ISession`, discussed in more detail later in this chapter). Persistent instances are then associated with the persistence manager. They might even be objects that became persistent when a reference was created from another persistent object already associated with a

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

persistence manager. Alternatively, a persistent instance might be an instance retrieved from the database by execution of a query, by an identifier lookup, or by navigating the object graph starting from another persistent instance. In other words, persistent instances are always associated with an `ISession` and are *transactional*.

Persistent instances participate in transactions—their state is synchronized with the database at the end of the transaction. When a transaction commits, state held in memory is propagated to the database by the execution of SQL `INSERT`, `UPDATE`, and `DELETE` statements. This procedure might also occur at other times. For example, NHibernate might synchronize with the database before execution of a query. This ensures that queries will be aware of changes made earlier during the transaction.

We call a persistent instance *new* if it has been allocated a primary key value but has not yet been inserted into the database. The new persistent instance will remain “new” until synchronization occurs.

Of course, NHibernate doesn’t have to update the database row of every persistent object in memory at the end of the transaction. Saving objects that haven’t changed would be time-consuming and unnecessary. ORM software must therefore have a strategy for detecting which persistent objects have been modified by the application in the transaction. We call this *automatic dirty checking* (an object with modifications that haven’t yet been propagated to the database is considered *dirty*). Again, this state isn’t visible to the application. We call this feature *transparent transaction-level write-behind*, meaning that NHibernate propagates state changes to the database as late as possible but hides this detail from the application.

NHibernate can detect exactly which attributes have been modified, so it’s possible to include only the columns that need updating in the SQL `UPDATE` statement. This might bring performance gains, particularly with certain databases. However, it isn’t usually a significant difference, and, in theory, it could harm performance in some environments. So, by default, NHibernate includes all columns in the SQL `UPDATE` statement. Hence, NHibernate can generate and cache this basic SQL once at startup, rather than on-the-fly each time an object is saved. If you only want to update modified columns, you can enable dynamic SQL generation by setting `dynamic-update="true"` in a class mapping. Note that this feature is extremely difficult and time-consuming to implement in a handcoded persistence layer. We talk about NHibernate’s transaction semantics and the synchronization process, or *flushing*, in more detail in the next chapter.

Finally, a persistent instance may be made transient via a `Delete()` call to the persistence manager API, resulting in deletion of the corresponding row of the database table.

4.1.3 Detached objects

When a transaction completes and data is written to the database, the persistent instances associated with the persistence manager still exist in memory. If the transaction were successful, the state of these instances will have been synchronized with the database. In ORM implementations with *process-scoped identity* (see the following sections), the instances retain their association to the persistence manager and are still considered persistent.

In the case of NHibernate, however, these instances lose their association with the persistence manager when you `Close()` the `ISession`. Because they are no longer associated our persistence manager, we refer to these objects as *detached*. Detached instances may no longer be guaranteed to be synchronized with database state; they’re no longer under the management of NHibernate. However, they still contain persistent data. It’s possible, and common, for the application to retain a reference and update a detached object outside of a transaction, and therefore without NHibernate tracking the changes. Fortunately, NHibernate lets you use these instances in a new transaction by reassociating them with a new persistence manager. After reassociation, they’re considered persistent again. This feature has a deep impact on how multitiered applications may be designed. The ability to return objects from one transaction to the presentation layer and later reuse them in a

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

new transaction is one of NHibernate's main selling points. We discuss this usage in the next chapter as an implementation technique for long-running *application transactions*. We also show you how to avoid the DTO (anti-) pattern by using detached objects in chapter 10, section 10.3.1.

NHibernate also provides an explicit way of detaching instances: the `Evict()` method of the `ISession`. However, this method is typically used only for cache management (a performance consideration). It's *not* normal to perform detachment explicitly. Rather, all objects retrieved in a transaction become detached when the `ISession` is closed or when they're serialized (if they're passed remotely, for example). So, NHibernate doesn't need to provide functionality for controlling detachment of *subgraphs*. Instead, the application can control the depth of the fetched subgraph (the instances that are currently loaded in memory) using the query language or explicit graph navigation. Then, when the `ISession` is closed, this entire subgraph (all objects associated with a persistence manager) becomes detached.

Let's look at the different states again but, this time, consider the *scope of object identity*.

4.1.4 The scope of object identity

As application developers, we identify an object using .NET object identity (`a==b`). So, if an object changes state, is its .NET identity guaranteed to be the same in the new state? In a layered application, that might not be the case.

In order to explore this topic, it's important to understand the relationship between .NET identity, `object.ReferenceEquals(a,b)`, and database identity, `a.Id==b.Id`. Sometimes both are equivalent; sometimes they aren't. We refer to the conditions under which .NET identity is equivalent to database identity as the *scope of object identity*.

For this scope, there are three common choices:

A primitive persistence layer with no identity scope makes no guarantees that if a row is accessed twice, the same .NET object instance will be returned to the application. This becomes problematic if the application modifies two different instances that both represent the same row in a single transaction (how do you decide which state should be propagated to the database?).

A persistence layer using transaction-scoped identity guarantees that, in the context of a single transaction, there is only one object instance that represents a particular database row. This avoids the previous problem and also allows for some caching to be done at the transaction level.

Process-scoped identity goes one step further and guarantees that there is only one object instance representing the row in the whole process (CLR).

For a typical web or enterprise application, transaction-scoped identity is preferred. Process-scoped identity offers some potential advantages in terms of cache utilization and the programming model for reuse of instances across multiple transactions; however, in a pervasively multithreaded application, the cost of always synchronizing shared access to persistent objects in the global identity map is too high a price to pay. It's simpler, and more scalable, to have each thread work with a distinct set of persistent instances in each transaction scope.

Speaking loosely, we would say that NHibernate implements transaction-scoped identity. Actually, the NHibernate identity scope is the `ISession` instance, so identical objects are guaranteed if the same persistence manager (the `ISession`) is used for several operations. But an `ISession` isn't the same as a (database) transaction—it's a much more flexible element. We'll explore the differences and the consequences of this concept in the next chapter. Let's focus on the persistence lifecycle and identity scope again.

If you request two objects using the same database identifier value in the same `ISession`, the result will be two references to the same in-memory object. The following code example demonstrates this behavior, with several `Load()` operations in two `ISessions`:

```
ISession session1 = sessionFactory.OpenSession();
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

ITransaction tx1 = session1.BeginTransaction();
// Load Category with identifier value "1234"
object a = session1.Load( typeof(Category), 1234 );
object b = session1.Load( typeof(Category), 1234 );
if ( object.ReferenceEquals(a,b) ) {
    System.Console.WriteLine("a and b are identical.");
}
tx1.Commit();
session1.Close();
ISession session2 = sessionFactory.OpenSession();
ITransaction tx2 = session2.BeginTransaction();
// Let's use the generic version of Load()
Category b2 = session2.Load<Category>( 1234 );
if ( ! object.ReferenceEquals(a,b2) ) {
    System.Console.WriteLine("a and b2 are not identical.");
}
tx2.Commit();
session2.Close();

```

Object references **a** and **b** not only have the same database identity, they also have the same .NET identity since they were loaded in the same **ISession**. Once outside this boundary, however, NHibernate doesn't guarantee .NET identity, so **a** and **b2** aren't identical and the message is printed on the console. Of course, a test for database identity—**a.Id==b2.Id**—would still return true.

To further complicate our discussion of identity scopes, we need to consider how the persistence layer handles a reference to an object outside its identity scope. For example, for a persistence layer with transaction-scoped identity such as NHibernate, is a reference to a detached object (that is, an instance persisted or loaded in a previous, completed session) tolerated?

4.1.5 Outside the Identity scope

If an object reference leaves the scope of guaranteed identity, we call it a *reference to a detached object*. Why is this concept useful?

In Windows applications, you usually don't maintain a database transaction across a user interaction. Users take a long time to think about modifications, so for scalability reasons, you must keep database transactions short and release database resources as soon as possible. In this environment, it's useful to be able to reuse a reference to a detached instance. For example, you might want to send an object retrieved in one unit of work to the presentation tier and later reuse it in a second unit of work some time later, after it's been modified by the user. For ASP.NET applications this won't apply, because you should not keep business objects in memory after the page has been rendered – instead you reload them on each request thus not requiring reattachment.

For when you do need to reattach objects, you won't usually wish to reattach the entire object graph in the second unit of work. For performance (and other) reasons, it's important that reassociation of detached instances be selective. NHibernate supports *selective reassociation of detached instances*. This means the application can efficiently reattach a *subgraph* of a graph of detached objects with the current (“second”) NHibernate **ISession**. Once a detached object has been reattached to a new NHibernate persistence manager, it may be considered a persistent instance again, and its state will be synchronized with the database at the end of the transaction. This is due to NHibernate's automatic dirty checking of persistent instances.

Reattachment might result in the creation of new rows in the database when a reference is created from a detached instance to a new transient instance. For example, a new **Bid** might have been added to a detached **Item** while it was on the presentation tier. NHibernate can detect that the **Bid** is new and must be inserted in the database. For this to work, NHibernate must be able to distinguish between a “new” transient instance and

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

an “old” detached instance. Transient instances (such as the `Bid`) might need to be saved; detached instances (such as the `Item`) might need to be reattached (and later updated in the database). There are several ways to distinguish between transient and detached instances, but the nicest approach is to look at the value of the identifier property. NHibernate can examine the identifier of a transient or detached object on reattachment and treat the object (and the associated graph of objects) appropriately. We discuss this important issue further in section 5.3.4, “Distinguishing between transient and detached instances.”

If you want to take advantage of NHibernate’s support for reassociation of detached instances in your own applications, you need to be aware of NHibernate’s identity scope when designing your application—that is, the `ISession` scope that guarantees identical instances. As soon as you leave that scope and have detached instances, another interesting concept comes into play.

We need to discuss the relationship between .NET *equality* and database identity. For a recap on equality, see chapter 3, section 3.5.1 - “Identity versus equality”. Equality is an identity concept that we, the class developers, can control. Sometimes we have to use it for classes that have detached instances. .NET equality is defined by the implementation of the `Equals()` and `GetHashCode()` methods in the persistent classes of the domain model.

4.1.6 Implementing `Equals()` and `GetHashCode()`

The `Equals()` method is called by application code or, more importantly, by the .NET collections. An `ISet` collection (in the library `Iesi.Collections`), for example, calls `Equals()` on each object you put in the `ISet`, to determine (and prevent) duplicate elements.

First let’s consider the default implementation of `Equals()`, defined by `System.Object`, which uses a comparison by .NET identity. NHibernate guarantees that there is a unique instance for each row of the database inside an `ISession`. Therefore, the default identity `Equals()` is appropriate if you never mix instances—that is, if you never put detached instances from different sessions into the same `ISet`. (Actually, the issue we’re exploring is also visible if detached instances are from the same session but have been serialized and deserialized in different scopes.) As soon as you have instances from multiple sessions, however, it becomes possible to have an `ISet` containing two `Items` that each represent the same row of the database table but don’t have the same .NET identity. This would almost always be semantically wrong. Nevertheless, it’s possible to build a complex application with identity (default) equals as long as you exercise discipline when dealing with detached objects from different sessions (and keep an eye on serialization and deserialization). One nice thing about this approach is that you don’t have to write extra code to implement your own notion of equality.

However, if this concept of equality isn’t what you want, you have to override `Equals()` in your persistent classes. Keep in mind that when you override `Equals()`, you always need to also override `GetHashCode()` so the two methods are *consistent* (if two objects are equal, they must have the same hash code). Let’s look at some of the ways you can override `Equals()` and `GetHashCode()` in persistent classes.

Using database identifier equality

A clever approach is to implement `Equals()` to compare just the database identifier property (usually a surrogate primary key) value:

```
public class User {
    ...
    public override bool Equals(object other) {
        if (object.ReferenceEquals(this, other)) return true;
        if (id==null) return false;
        if ( !(other is User) ) return false;
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        User that = (User) other;
        return this.Id == that.Id;
    }
    public override int GetHashCode() {
        return Id==null ?
            base.GetHashCode(this) :
            Id.GetHashCode();
    }
}

```

Notice how this `Equals()` method falls back to .NET identity for transient instances (if `id==null`) that don't have a database identifier value assigned yet. This is reasonable, since they can't have the same persistent identity as another instance.

Unfortunately, this solution has one huge problem: NHibernate doesn't assign identifier values until an entity is saved. So, if the object is added to an `ISet` before being saved, its hash code changes while it's contained by the `ISet`, contrary to the contract defined by this collection. In particular, this problem makes cascade save (discussed later in this chapter) useless for sets. We strongly discourage this solution (database identifier equality).

Comparing by value

A better way is to include all persistent properties of the persistent class, apart from any database identifier property, in the `Equals()` comparison. This is how most people perceive the meaning of `Equals()`; we call it *by value* equality.

When we say "all properties," we don't mean to include collections. Collection state is associated with a different table, so it seems wrong to include it. More important, you don't want to force the entire object graph to be retrieved just to perform `Equals()`. In the case of `User`, this means you shouldn't include the `items` collection (the items sold by this user) in the comparison. So, this is the implementation you could use:

```

public class User {
    ...
    public override bool Equals(object other) {
        if (object.ReferenceEquals(this, other)) return true;
        if ( !(other is User) ) return false;
        User that = (User) other;
        if ( ! this.Username == that.Username )
            return false;
        if ( ! this.Password == that.Password )
            return false;
        return true;
    }
    public override int GetHashCode() {
        int result = 14;
        result = 29 * result + Username.GetHashCode();
        result = 29 * result + Password.GetHashCode();
        return result;
    }
}

```

However, there are again two problems with this approach:

- Instances from different sessions are no longer equal if one is modified (for example, if the user changes his password).
- Instances with different database identity (instances that represent different rows of the database table) could be considered equal, unless there is some combination of properties that are guaranteed to be unique (the database columns have a unique constraint). In the case of `User`, there is a unique property: `Username`.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

To get to the solution we recommend, you need to understand the notion of a *business key*.

Using business key equality

A *business key* is a property, or some combination of properties, that is unique for each instance with the same database identity. Essentially, it's the natural key you'd use if you weren't using a surrogate key. Unlike a natural primary key, it isn't an absolute requirement that the business key never change—as long as it changes rarely, that's enough.

We argue that every entity should have a business key, even if it includes all properties of the class (this would be appropriate for some immutable classes). The business key is what the user thinks of as uniquely identifying a particular record, whereas the surrogate key is what the application and database use.

Business key equality means that the `Equals()` method compares only the properties that form the business key. This is a perfect solution that avoids all the problems described earlier. The only downside is that it requires extra thought to identify the correct business key in the first place. But this effort is required anyway; it's important to identify any unique keys if you want your database to help ensure data integrity via constraint checking.

For the `User` class, `username` is a great candidate business key. It's never null, it's unique, and it changes rarely (if ever):

```
public class User {
    ...
    public override bool Equals(object other) {
        if (object.ReferenceEquals(this, other)) return true;
        if ( !(other is User) ) return false;
        User that = (User) other;
        return this.Username == that.Username );
    }
    public override int GetHashCode() {
        return Username.GetHashCode();
    }
}
```

For some other classes, the business key might be more complex, consisting of a combination of properties. For example, candidate business keys for the `Bid` class are the item ID together with the bid amount, or the item ID together with the date and time of the bid. A good business key for the `BillingDetails` abstract class is the `number` together with the type (subclass) of billing details. Notice that it's almost never correct to override `Equals()` on a subclass and include another property in the comparison. It's tricky to satisfy the requirements that equality be both symmetric and transitive in this case; and, more important, the business key wouldn't correspond to any well-defined candidate natural key in the database (subclass properties may be mapped to a different table).

You might have noticed that the `Equals()` and `GetHashCode()` methods always access the properties of the other object via the getter methods. This is important, since the object instance passed as `other` might be a proxy object, not the actual instance that holds the persistent state. This is one point where NHibernate isn't completely transparent, but it's a good practice to use properties instead of direct instance variable access anyway.

Finally, take care when modifying the value of the business key properties; don't change the value while the domain object is in a set.

So far we've talked about how the broad subject of how the persistence manager behaves when working with instances that are transient, persistent or detached. We've also discussed issues of scope, and the importance of equality and identity. It's now time to take a closer look at the persistence manager and explore the NHibernate `ISession` API in greater detail. We'll come back to detached objects with more details in the next chapter.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

4.2 The persistence manager

Any transparent persistence tool like NHibernate will include some form of *persistence manager* API, which usually provides services for

- Basic CRUD operations
- Query execution
- Control of transactions
- Management of the transaction-level cache

The persistence manager can be exposed by several different interfaces (in the case of NHibernate, `ISession`, `IQuery`, `ICriteria`, and `ITransaction`). Under the covers, the implementations of these interfaces are coupled tightly.

The central interface between the application and NHibernate is `Session`; it's your starting point for all the operations just listed. For most of the rest of this book, we'll refer to the *persistence manager* and the *session* interchangeably; this is consistent with usage in the NHibernate community.

So, how do you start using the session? At the beginning of a unit of work, a thread obtains an instance of `ISession` from the application's `ISessionFactory`. The application might have multiple `ISessionFactory`s if it accesses multiple datasources. But you should never create a new `ISessionFactory` just to service a particular request—creation of an `ISessionFactory` is extremely expensive. On the other hand, `ISession` creation is extremely *inexpensive*; the `ISession` doesn't even obtain an ADO.NET `IDbConnection` until a connection is required.

After opening a new session, you use it to load and save objects.

4.2.1 Making an object persistent

The first thing you want to do with an `ISession` is make a new transient object persistent. To do so, you use the `Save()` method:

```
User user = new User();
user.Name.Firstname = "John";
user.Name.Lastname = "Doe";
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    session.Save(user);
    session.Transaction.Commit();
}
```

First, we instantiate a new transient object `user` as usual. Of course, we might also instantiate it after opening an `ISession`; they aren't related yet. We open a new `ISession` using the `ISessionFactory` referred to by `sessionFactory`, and then we start a new database transaction.

A call to `Save()` makes the transient instance of `User` persistent. It's now associated with the current `ISession`. However, no SQL `INSERT` has yet been executed. The NHibernate `ISession` never executes any SQL statement until absolutely necessary.

The changes made to persistent objects have to be synchronized with the database at some point. This happens when we `Commit()` the NHibernate `ITransaction`. In this case, NHibernate obtains an ADO.NET connection (and transaction) and issues a single SQL `INSERT` statement. Finally, the `ISession` is closed and the ADO.NET connection is released.

Note that it's better (but not required) to fully initialize the `User` instance before associating it with the `ISession`. The SQL `INSERT` statement contains the values that were held by the object *at the point when*

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

`Save()` was called. You can, of course, modify the object after calling `Save()`, and your changes will be propagated to the database as an SQL `UPDATE`.

Everything between `session.BeginTransaction()` and `Transaction.Commit()` occurs in one database transaction. We haven't discussed transactions in detail yet; we'll leave that topic for the next chapter. But keep in mind that all database operations in a transaction scope are *atomic* - they completely succeed or completely fail. If one of the `UPDATE` or `INSERT` statements made on `Transaction.Commit()` fails, all changes made to persistent objects in this transaction will be rolled back at the database level. However, NHibernate does *not* roll back in-memory changes to persistent objects, their state remains exactly as you left it. This is reasonable since a failure of a database transaction is normally non-recoverable and you have to discard the failed `ISession` immediately.

4.2.2 Updating the persistent state of a detached instance

Modifying the `user` after the session is closed will have no effect on its persistent representation in the database. When the session is closed, `user` becomes a *detached* instance. However, it may be reassociated with a new `Session` some time later by calling `Update()` or `Lock()`.

Let's first look at the `Update()` method. Using `Update()` will force an update to the persistent state of the object in the database; an SQL `UPDATE` is scheduled and will be later committed. Here's an example of detached object handling:

```
user.Password = "secret";
using( ISession sessionTwo = sessionFactory.OpenSession() )
    using( sessionTwo.BeginTransaction() ) {
        sessionTwo.Update(user);
        user.Username = "jonny";
        sessionTwo.Transaction.Commit();
    }
```

It doesn't matter if the object is modified before or after it's passed to `Update()`. The important thing is that the call to `Update()` is used to reassociate the detached instance to the new `ISession`, and the current transaction. NHibernate will treat the object as dirty and therefore schedule the SQL `UPDATE` regardless of whether the object has been updated or not. This makes `Update()` a "safe" way of reassociating objects with a `Session` because you know that changes will be propagated to the database. Actually, there is one exception to this, which is when you have enabled `select-before-update` in the persistent class mapping. If you do this, NHibernate will *determine* if the object is dirty rather than assuming it. It does this by executing a `SELECT` statement and comparing the object's current state to the current database state.

Now let's look at the `Lock()` method. A call to `Lock()` associates the object with the `ISession` *without* forcing NHibernate to treat the object as dirty. Consider this example:

```
using( ISession sessionTwo = sessionFactory.OpenSession() ) {
using( sessionTwo.BeginTransaction() ) {
    sessionTwo.Lock(user, LockMode.None);
    user.Password = "secret";
    user.LoginName = "jonny";
    sessionTwo.Transaction.Commit();
}
}
```

When using `Lock()`, it *does* matter whether changes are made before or after the object is associated with the session. Changes made *before* the call to `Lock()` aren't propagated to the database because NHibernate hasn't witnessed those changes; you only use `Lock()` if you're sure that the detached instance hasn't been modified beforehand.

In the code above, we specify `LockMode.None`, which tells NHibernate not to perform a version check or obtain any database-level locks when reassociating the object with the `ISession`. If we specified `LockMode.Read` or `LockMode.Upgrade`, NHibernate would execute a `SELECT` statement in order to

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

perform a version check (and to set an upgrade lock). We take a detailed look at NHibernate lock modes in the next chapter. Having discussed how objects are treated when we reassociate them with a `Session`, let's now look at what happens when we retrieve objects.

4.2.3 Retrieving a persistent object

The `ISession` is also used to query the database and retrieve existing persistent objects. NHibernate is especially powerful in this area, as you'll see later in this chapter and in chapter 7. However, special methods are provided on the `ISession` API for the simplest kind of query: retrieval by identifier. One of these methods is `Get()`, demonstrated here:

```
int userID = 1234;
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    User user = (User) session.Get(typeof(User), userID);
    session.Transaction.Commit();
}
```

The retrieved object `user` may now be passed to the presentation layer for use outside the transaction as a detached instance (after the session has been closed). If no row with the given identifier value exists in the database, the `Get()` returns `null`.

Since NHibernate 1.2, we can use .NET 2.0 generics:

```
User user = session.Get<User>(userID);
```

4.2.4 Updating a persistent object

Any persistent object returned by `Get()` or any other kind of query is already associated with the current `ISession` and transaction context. It can be modified, and its state will be synchronized with the database. This mechanism is called *automatic dirty checking*, which means NHibernate will track and save the changes you make to an object inside a session:

```
int userID = 1234;
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    User user = (User) session.Get(typeof(User), userID);
    user.Password = "secret";
    session.Transaction.Commit();
}
```

First we retrieve the object from the database with the given identifier. We modify the object, and these modifications are propagated to the database when `Transaction.Commit()` is called. Of course, as soon as we close the `ISession`, the instance is considered detached. Batch updates are also possible since NHibernate has been tweaked to use ADO.NET 2.0 batching internal feature. Enabling this feature will make NHibernate perform bulk updates; these updates will therefore become much faster. All you have to do is define the batch size as a NHibernate property:

```
<property name="hibernate.adonet.batch_size">16</property>
```

By default, the batch size is 0 which means that this feature is disabled.

This feature currently works only on .NET 2.0 when using a SQL Server database. And because it uses .NET reflection, it may not work in some restricted environments.

Finally, when using this feature, ADO.NET 2.0 doesn't return the number of rows affected by each statement in the batch which means that NHibernate may not perform optimistic concurrency checking correctly. For example, if a statement affects two rows and another no row (instead of affecting one each), NHibernate will only know that two rows have been affected, and conclude that everything went right.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

4.2.5 Making an object transient

In many use cases, we need our persistent (or detached) objects to become transient again, meaning they will no longer have a corresponding data in the database. As discussed at the beginning of this chapter, persistent objects are those that are in the session and have corresponding data in the database. Making them transient will remove their persistent state from the database. This is easily achieved using the `Delete()` method:

```
int userID = 1234;
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    User user = session.Get<User>(userID);
    session.Delete(user);
    session.Transaction.Commit();
}
```

The SQL `DELETE` will be executed only when the `ISession` is synchronized with the database at the end of the transaction.

After the `ISession` is closed, the `user` object is considered an ordinary transient instance. The transient instance will be destroyed by the garbage collector if it's no longer referenced by any other object, therefore both the in-memory instance and the persistent database row will have been removed.

Similarly, detached objects may also be made transient (Detached objects being those that have corresponding state in the database but which are not in the `ISession`). You don't have to reattach a detached instance to the session with `Update()` or `Lock()`. Instead you can directly delete a detached instance as follows:

```
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    session.Delete(user);
    session.Transaction.Commit();
}
```

In this case, the call to `Delete()` does two things: It associates the object with the `ISession` and then schedules the object for deletion, executed on `Transaction.Commit()`.

You now know the persistence lifecycle and the basic operations of the persistence manager. Together with the persistent class mappings we discussed in chapter 3, you can create your own small NHibernate application. (If you like, you can jump to chapter 10 and read about a handy NHibernate helper class for `ISessionFactory` and `ISession` management.) Keep in mind that we didn't show you any exception-handling code so far, but you should be able to figure out the `try/catch` blocks yourself (like in chapter 2). Map some simple entity classes and components, and then store and load objects in a stand-alone console application (just write a `Main` method). However, as soon as you try to store associated entity objects—that is, when you deal with a more complex object graph—you'll see that calling `Save()` or `Delete()` on each object of the graph isn't an efficient way to write applications.

You'd like to make as few calls to the `ISession` as possible. *Transitive persistence* provides a more natural way to force object state changes and to control the persistence lifecycle.

4.3 Using transitive persistence in NHibernate

Real, nontrivial applications deal not with single objects but rather with graphs of objects. When the application manipulates a graph of persistent objects, the result may be an object graph consisting of persistent, detached, and transient instances. *Transitive persistence* is a technique that allows you to propagate persistence to transient and detached subgraphs automatically.

For example, if we add a newly instantiated `Category` to the already persistent hierarchy of categories, it should automatically become persistent without a call to `session.Save()`. We gave a slightly different example in chapter 4 when we mapped a parent/child relationship between `Bid` and `Item`. In that case, not

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

only were bids automatically made persistent when they were added to an item, but they were also automatically deleted when the owning item was deleted.

There is more than one model for transitive persistence. The best known is *persistence by reachability*, which we'll discuss first. Although some basic principles are the same, NHibernate uses its own, more powerful model, as you'll see later.

4.3.1 Persistence by reachability

An object persistence layer is said to implement persistence by reachability if any instance becomes persistent when the application creates an object reference to the instance from another instance that is already persistent. This behavior is illustrated by the object diagram (note that this isn't a class diagram) in figure 5.2.

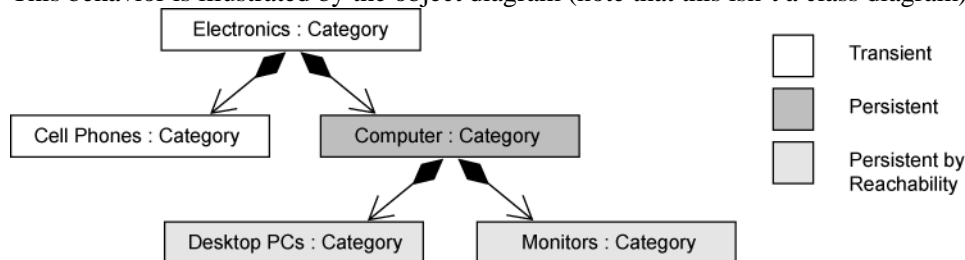


Figure 5.2 Persistence by reachability with a root persistent object

In this example, “Computer” is a persistent object. The objects “Desktop PCs” and “Monitors” are also persistent; they’re reachable from the “Computer” `Category` instance. “Electronics” and “Cell Phones” are transient. Note that we assume navigation is only possible to child categories, and not to the parent—for example, we can call `computer.ChildCategories`. Persistence by reachability is a recursive algorithm: All objects reachable from a persistent instance become persistent either when the original instance is made persistent or just before in-memory state is synchronized with the data store.

Persistence by reachability guarantees referential integrity; any object graph can be completely re-created by loading the persistent root object. An application may walk the object graph from association to association without worrying about the persistent state of the instances. (SQL databases have a different approach to referential integrity, relying on foreign key and other constraints to detect a misbehaving application.)

In the purest form of persistence by reachability, the database has some top-level, or *root*, object from which all persistent objects are reachable. Ideally, an instance should become transient and be deleted from the database if it isn't reachable via references from the root persistent object.

Neither NHibernate nor other ORM solutions implement this form; there is no analog of the root persistent object in an SQL database and no persistent garbage collector that can detect unreferenced instances. Object-oriented data stores might implement a garbage-collection algorithm similar to the one implemented for in-memory objects by the CLR, but this option isn't available in the ORM world; scanning all tables for unreferenced rows won't perform acceptably.

So, persistence by reachability is at best a halfway solution. It helps you make transient objects persistent and propagate their state to the database without many calls to the persistence manager. But (at least, in the context of SQL databases and ORM) it isn't a full solution to the problem of making persistent objects transient and removing their state from the database. This turns out to be a much more difficult problem. You can't simply remove all reachable instances when you remove an object; other persistent instances may hold references to them (remember that entities can be shared). You can't even safely remove instances that aren't referenced by any persistent object in memory; the instances in memory are only a small subset of all objects represented in the database. Let's look at NHibernate's more flexible transitive persistence model.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

4.3.2 Cascading persistence with NHibernate

NHibernate's transitive persistence model uses the same basic concept as persistence by reachability—that is, object associations are examined to determine transitive state. However, NHibernate allows you to specify a *cascade style* for each association mapping, which offers more flexibility and fine-grained control for all state transitions. NHibernate reads the declared style and cascades operations to associated objects automatically.

By default, NHibernate does *not* navigate an association when searching for transient or detached objects, so saving, deleting, or reattaching a `Category` won't affect the child category objects. This is the opposite of the persistence-by-reachability default behavior. If, for a particular association, you wish to enable transitive persistence, you must override this default in the mapping metadata.

You can map entity associations in metadata with the following attributes:

- `cascade="none"`, the default, tells NHibernate to ignore the association.
- `cascade="save-update"` tells NHibernate to navigate the association when the transaction is committed and when an object is passed to `Save()` or `Update()` and save newly instantiated transient instances and persist changes to detached instances.
- `cascade="delete"` tells NHibernate to navigate the association and delete persistent instances when an object is passed to `Delete()`.
- `cascade="all"` means to cascade both save-update and delete, as well as calls to `Evict` and `Lock`.
- `cascade="all-delete-orphan"` means the same as `cascade="all"` but, in addition, NHibernate deletes any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).
- `cascade="delete-orphan"` NHibernate will delete any persistent entity instance that has been removed (dereferenced) from the association (for example, from a collection).

This *association-level cascade style* model is both richer and less safe than persistence by reachability. NHibernate doesn't make the same strong guarantees of referential integrity that persistence by reachability provides. Instead, NHibernate partially delegates referential integrity concerns to the foreign key constraints of the underlying relational database. Of course, there is a good reason for this design decision: It allows NHibernate applications to use *detached* objects efficiently, because you can control reattachment of a detached object graph at the association level.

Let's elaborate on the cascading concept with some example association mappings. We recommend that you read the next section in one turn, because each example builds on the previous one. Our first example is straightforward; it lets you save newly added categories efficiently.

4.3.3 Managing auction categories

System administrators can create new categories, rename categories, and move subcategories around in the category hierarchy. This structure can be seen in figure 4.3.

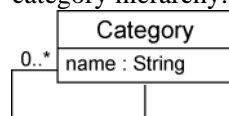


Figure 4.3 Category class with association to itself

Now, we map this class and the association:

```
<class name="Category" table="CATEGORY">
  ...
  <property name="Name" column="CATEGORY_NAME" />
  <many-to-one
    name="ParentCategory"
    class="Category"
    column="PARENT_CATEGORY_ID"
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        cascade="none" />
    <set
        name="ChildCategories"
        table="CATEGORY"
        cascade="save-update"
        inverse="true">
        <key column="PARENT_CATEGORY_ID" />
        <one-to-many class="Category" />
    </set>
    ...
</class>

```

This is a recursive, bidirectional, one-to-many association, as briefly discussed in chapter 3. The one-valued end is mapped with the `<many-to-one>` element and the `Set` typed property with the `<set>`. Both refer to the same foreign key column: `PARENT_CATEGORY_ID`.

Suppose we create a new `Category` as a child category of “Computer” (see figure 4.4).

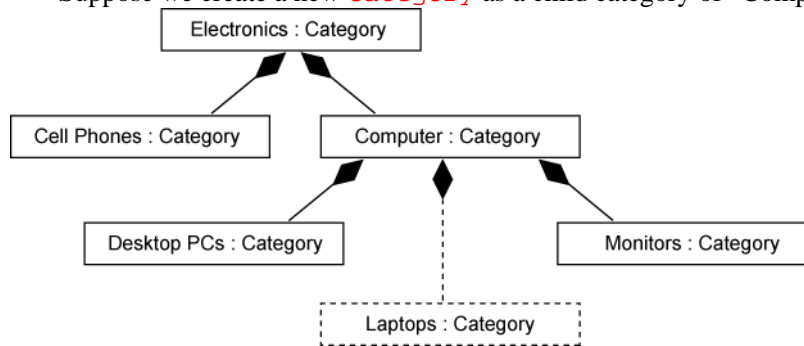


Figure 4.4 Adding a new `Category` to the object graph

We have several ways to create this new “Laptops” object and save it in the database. We could go back to the database and retrieve the “Computer” category to which our new “Laptops” category will belong, add the new category, and commit the transaction:

```

using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    Category computer = session.Get<Category>(computerId);
    Category laptops = new Category("Laptops");
    computer.ChildCategories.Add(laptops);
    laptops.ParentCategory = computer;
    session.Transaction.Commit();
}

```

The `computer` instance is persistent (attached to a session), and the `ChildCategories` association has cascade save enabled. Hence, this code results in the new `laptops` category becoming persistent when `Transaction.Commit()` is called, because NHibernate cascades the dirty-checking operation to the children of `computer`. NHibernate executes an `INSERT` statement.

Let’s do the same thing again, but this time create the link between “Computer” and “Laptops” outside of any transaction (in a real application, it’s useful to manipulate an object graph in a presentation tier—for example, before passing the graph back to the persistence layer to make the changes persistent):

```

Category computer = ... // Loaded in a previous session
Category laptops = new Category("Laptops");
computer.ChildCategories.Add(laptops);
laptops.ParentCategory = computer;

```

The detached `computer` object and any other detached objects it refers to are now associated with the new transient `laptops` object (and vice versa). We make this change to the object graph persistent by saving the new object in a second NHibernate session:

```

using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    // Persist one new category and the link to its parent category
    session.Save(laptops);
    session.Transaction.Commit();
}

```

NHibernate will inspect the database identifier property of the parent category of `laptops` and correctly create the relationship to the “Computer” category in the database. NHibernate inserts the identifier value of the parent into the foreign key field of the new “Laptops” row in `CATEGORY`.

Since `cascade="none"` is defined for the `ParentCategory` association, NHibernate ignores changes to any of the other categories in the hierarchy (“Computer”, “Electronics”). It doesn’t cascade the call to `Save()` to entities referred to by this association. If we had enabled `cascade="save-update"` on the `<many-to-one>` mapping of `ParentCategory`, NHibernate would have had to navigate the whole graph of objects in memory, synchronizing all instances with the database. This process would perform badly, because a lot of useless data access would be required. In this case, we neither needed nor wanted transitive persistence for the `ParentCategory` association.

Why do we have cascading operations? We could have saved the `laptop` object, as shown in the previous example, without any cascade mapping being used. Well, consider the following case:

```

Category computer = ... // Loaded in a previous Session
Category laptops = new Category("Laptops");
Category laptopAccessories = new Category("Laptop Accessories");
Category laptopTabletPCs = new Category("Tablet PCs")
laptops.AddChildCategory(laptopAccessories);
laptops.AddChildCategory(laptopTabletPCs);
computer.AddChildCategory(laptops);

```

(Notice that we use the convenience method `AddChildCategory()` to set both ends of the association link in one call, as described in chapter 3.)

It would be undesirable to have to save each of the three new categories individually. Fortunately, because we mapped the `ChildCategories` association with `cascade="save-update"`, we don’t need to. The same code we used before to save the single “Laptops” category will save all three new categories in a new session:

```

using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    // Persist all three new Category instances
    session.Save(laptops);
    session.Transaction.Commit();
}

```

You’re probably wondering why the cascade style is called `cascade="save-update"` rather than `cascade="save"`. Having just made all three categories persistent previously, suppose we made the following changes to the category hierarchy in a subsequent request (outside of a session and transaction):

```

laptops.Name = "Laptop Computers";
laptopAccessories.Name = "Accessories & Parts";
laptopTabletPCs.Name = "Tablet Computers";
Category laptopBags = new Category("Laptop Bags");
laptops.AddChildCategory(laptopBags);

```

We have added a new category as a child of the “Laptops” category and modified all three existing categories.

The following code updates three old `Category` instances and inserts the new one:

```

using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {

    session.Update(laptops);
    session.Transaction.Commit();
}

```

Specifying `cascade="save-update"` on the `ChildCategories` association accurately reflects the fact that NHibernate determines what is needed to persist the objects to the database. In this case, it will reattach/update the three detached categories (`laptops`, `laptopAccessories`, and `laptopTabletPCs`) and save the new child category (`laptopBags`).

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Notice that the last code example differs from the previous two session examples only in a single method call. The last example uses `Update()` instead of `Save()` because `laptops` was already persistent.

We can rewrite all the examples to use the `SaveOrUpdate()` method. Then the three code snippets are identical:

```
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    // Let NHibernate decide what's new and what's detached
    session.SaveOrUpdate(laptops);
    session.Transaction.Commit();
}
```

The `SaveOrUpdate()` method tells NHibernate to propagate the state of an instance to the database by creating a new database row if the instance is a new transient instance or updating the existing row if the instance is a detached instance. In other words, it does exactly the same thing with the `laptops` category as `cascade="save-update"` did with the child categories of `laptops`.

One final question: How did NHibernate know which children were detached and which were new transient instances?

4.3.4 Distinguishing between transient and detached instances

Since NHibernate doesn't keep a reference to a detached instance, you have to let NHibernate know how to distinguish between a detached instance like `laptops` (if it was created in a previous session) and a new transient instance like `laptopBags`.

A range of options is available. NHibernate will assume that an instance is an unsaved transient instance if:

The identifier property (if it exists) is `null`.

- The version property (if it exists) is `null`.
- You supply an `unsaved-value` in the mapping document for the class, and the value of the `identifier` property matches.
- You supply an `unsaved-value` in the mapping document for the `version` property, and the value of the `version` property matches.

You supply a NHibernate `IInterceptor` and return `true` from `IInterceptor.IsUnsaved()` after checking the instance in your code.

In our domain model, we have used the primitive type `long` as our identifier property type everywhere. As it is not nullable, we have to use the following identifier mapping in all our classes:

```
<class name="Category" table="CATEGORY">
  <id name="Id" unsaved-value="0">
    <generator class="native"/>
  </id>
  ....
</class>
```

The `unsaved-value` attribute tells NHibernate to treat instances of `Category` with an identifier value of `0` as newly instantiated transient instances. The default value for the attribute `unsaved-value` is `null` if the type is nullable; else, it is the default value of the type (that is `0` for numerical type); so, since we've chosen `long` as our identifier property type, we can omit the `unsaved-value` attribute in our auction application classes (we use the same identifier type everywhere). Technically, NHibernate try to guess the `unsaved-value` by instantiating an empty object and retrieving default property values from it.

Unsaved assigned identifiers

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This approach works nicely for synthetic identifiers, but it breaks down in the case of keys assigned by the application, including composite keys in legacy systems. We discuss this issue in chapter 9, section 9.2, “Legacy schemas and composite keys.” Avoid application-assigned (and composite) keys in new applications if possible (this is important for non-versioned entities).

You now have the knowledge to optimize your NHibernate application and reduce the number of calls to the persistence manager if you want to save and delete objects. Check the `unsaved-value` attributes of all your classes and experiment with detached objects to get a feeling for the NHibernate transitive persistence model.

Having focused on how we persist objects with NHibernate, we can now switch perspectives and focus on how we go about retrieving (or loading) them.

4.4 Retrieving objects

Retrieving persistent objects from the database is one of the most interesting (and complex) parts of working with NHibernate. NHibernate provides the following ways to get objects out of the database:

- Navigating the object graph, starting from an already loaded object, by accessing the associated objects through property accessor methods such as `aUser.Address.City`. NHibernate will automatically load (or preload) nodes of the graph while you navigate the graph if the `ISession` is open.
- Retrieving by identifier, which is the most convenient and performant method when the unique identifier value of an object is known.
- Using the Hibernate Query Language (HQL), which is a full object-oriented query language.
- Using the NHibernate `ICriteria` API, which provides a type-safe and object-oriented way to perform queries without the need for string manipulation. This facility includes queries based on an example object.
- Using native SQL queries, where NHibernate takes care of mapping the ADO.NET result sets to graphs of persistent objects.

In your NHibernate applications, you’ll use a combination of these techniques. Each retrieval method may use a different fetching strategy—that is, a strategy that defines what part of the persistent object graph should be retrieved. The goal is to find the best retrieval method and fetching strategy for every use case in your application while at the same time minimizing the number of SQL queries for best performance.

We won’t discuss each retrieval method in much detail in this section; instead we’ll focus on the basic fetching strategies and how to tune NHibernate mapping files for best default fetching performance for all methods. Before we look at the fetching strategies, we’ll give an overview of the retrieval methods. Note that we will also mention the NHibernate caching system, but fully explore it in the next chapter.

Let’s start with the simplest case, retrieval of an object by giving its identifier value (navigating the object graph should be self-explanatory). You saw a simple retrieval by identifier earlier in this chapter, but there is more to know about it.

4.4.1 Retrieving objects by identifier

The following NHibernate code snippet retrieves a `User` object from the database:

```
User user = session.Get<User>(userID);  
And without .NET 2.0 generics:  
User user = (User) session.Get(typeof(User), userID);
```

The `Get()` method is special because the identifier uniquely identifies a single instance of a class. Hence it’s common for applications to use the identifier as a convenient handle to a persistent object. Retrieval by identifier can use the cache when retrieving an object, avoiding a database hit if the object is already cached.

NHibernate also provides a `Load()` method:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```
User user = session.Load<User>(userID);
```

The difference between these two methods is trivial:

If `Load()` can't find the object in the cache or database, an exception is thrown. The `Load()` method never returns `null`. The `Get()` method returns `null` if the object can't be found.

The `Load()` method may return a proxy instead of a real persistent instance (when lazy loading is enabled). A proxy is a placeholder that triggers the loading of the real object when it's accessed for the first time; we discuss proxies later in this section. It is important to understand that `Load()` will return a proxy even if there is no row with the specified identifier; and an exception will be thrown if (and only if) NHibernate try to load it. On the other hand, `Get()` never returns a proxy as it must return null if the entity doesn't exist.

Choosing between `Get()` and `Load()` is easy: If you're certain the persistent object exists, and nonexistence would be considered exceptional, `Load()` is a good option. If you aren't certain there is a persistent instance with the given identifier, use `Get()` and test the return value to see if it's `null`.

What if this object is already in the session's cache as an un-initialized proxy? In this case, `Load()` will return the proxy as-is but `Get()` will initialize it before returning it.

Using `Load()` has a further implication: The application may retrieve a valid *reference* (a proxy) to a persistent instance without hitting the database to retrieve its persistent state. So `Load()` might not throw an exception when it doesn't find the persistent object in the cache or database; the exception may be thrown later, when the proxy is accessed.

There is an interesting application of this behavior. Let say lazy loading is enabled on the class `Category` and analyze the following code:

```
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    Category parent = session.Load<Category>(anId);
    Console.WriteLine( parent.Id );
    Category child = new Category("test");
    child.ParentCategory = parent;
    session.Save(child);
    session.Transaction.Commit();
}
```

We first loaded a category. NHibernate does not hit the database to do this, it returns a proxy. Accessing the identifier of this proxy doesn't cause its initialization (as long as the identifier is mapped with the access strategy "`property`" or "`nosetter`"). Then, we link a new category to the proxy and we save it. An INSERT statement is executed to save the row with the foreign key value of the proxy's identifier. No SELECT statement is executed at all!

Now, let's discover arbitrary queries which are far more flexible than retrieving objects by identifier.

4.4.2 Introducing the Hibernate Query Language

The Hibernate Query Language is an object-oriented dialect of the familiar relational query language SQL. HQL bears close resemblances to ODMG OQL and EJB-QL (from Java); but unlike OQL, it's adapted for use with SQL databases, and it's much more powerful and elegant than EJB-QL. However, JPA QL is actually a subset of HQL. HQL is easy to learn with basic knowledge of SQL.

HQL isn't a data-manipulation language like SQL. It's used only for object retrieval, not for updating, inserting, or deleting data. Object state synchronization is the job of the persistence manager, not the developer.

Most of the time, you'll only need to retrieve objects of a particular class and restrict by the properties of that class. For example, the following query retrieves a user by first name:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
IQuery q = session.createQuery("from User u where u.Firstname = :fname");
q.SetString("fname", "Max");
IList<User> result = q.List<User>();
```

After preparing query `q`, we bind the identifier value to a named parameter, `fname`. The result is returned as a generic `IList` of `User` objects.

Note that, instead of obtaining this list, we can provide one using `q.List(myEmptyList)` and NHibernate will fill it. This is very useful when you want to use a collection with additional functionalities (like advanced data-binding).

HQL is powerful, and even though you may not use the advanced features all the time, you'll need them for some difficult problems. For example, HQL supports the following:

- The ability to apply restrictions to properties of associated objects related by reference or held in collections (to navigate the object graph using query language).
- The ability to retrieve only properties of an entity or entities, without the overhead of loading the entity itself in a transactional scope. This is sometimes called a report query; it's more correctly called projection.
- The ability to order the results of the query.
- The ability to paginate the results.
- Aggregation with `group by`, `having`, and aggregate functions like `sum`, `min`, and `max`.
- Outer joins when retrieving multiple objects per row.
- The ability to call user-defined SQL functions.
- Subqueries (nested queries).

We discuss all these features in chapter 7, together with the optional native SQL query mechanism. We now look at another approach to issuing queries with NHibernate – Query by Criteria.

4.4.3 Query by criteria

The NHibernate *query by criteria* (QBC) API lets you build a query by manipulating criteria objects at runtime. This approach lets you specify constraints dynamically without direct string manipulations, but it doesn't lose much of the flexibility or power of HQL. On the other hand, queries expressed as criteria are often less readable than queries expressed in HQL.

Retrieving a user by first name is easy using a `Criteria` object:

```
ICriteria criteria = session.createCriteria(typeof(User));
criteria.add(Expression.like("Firstname", "Max"));
IList result = criteria.list();
```

An `ICriteria` is a tree of `ICriterion` instances. The `Expression` class provides static factory methods that return `ICriterion` instances. Once the desired criteria tree is built, it's executed against the database.

Many developers prefer QBC, considering it a more object-oriented approach. They also like the fact that the query syntax may be parsed and validated at compile time, whereas HQL expressions aren't parsed until runtime.

The nice thing about the NHibernate `ICriteria` API is the `ICriterion` framework. This framework allows extension by the user, which is difficult in the case of a query language like HQL.

4.4.4 Query by example

As part of the QBC facility, NHibernate supports *query by example* (QBE). The idea behind QBE is that the application supplies an instance of the queried class with certain property values set (to nondefault values). The query returns all persistent instances with matching property values. QBE isn't a particularly powerful

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

approach, but it can be convenient for some applications. The following code snippet demonstrates a NHibernate QBE:

```
User exampleUser = new User();
exampleUser.Firstname = "Max";
ICriteria criteria = session.CreateCriteria(typeof(User));
criteria.add( Example.Create(exampleUser) );
IList result = criteria.List();
```

A typical use case for QBE is a search screen that allows users to specify a range of property values to be matched by the returned result set. This kind of functionality can be difficult to express cleanly in a query language; string manipulations would be required to specify a dynamic set of constraints.

Both the QBC API and the example query mechanism are discussed in more detail in chapter 7.

You now know the basic retrieval options in NHibernate. We focus on the strategies for fetching object graphs in the rest of this section. A fetching strategy defines what part of the object graph (or, what subgraph) is retrieved with a query or load operation.

4.4.5 Fetching strategies

In traditional relational data access, you'd fetch all the data required for a particular computation with a single SQL query, taking advantage of inner and outer joins to retrieve related entities. Some primitive ORM implementations fetch data piecemeal, with many requests for small chunks of data in response to the application's navigating a graph of persistent objects. This approach doesn't make efficient use of the relational database's join capabilities. In fact, this data access strategy scales poorly by nature. One of the most difficult problems in ORM—probably *the* most difficult—is providing for efficient access to relational data, given an application that prefers to treat the data as a graph of objects.

For the kinds of applications we've often worked with (multi-user, distributed, web, and enterprise applications), object retrieval using many round trips to/from the database is unacceptable. Hence we argue that tools should emphasize the R in ORM to a much greater extent than has been traditional.

The problem of fetching object graphs efficiently (with minimal access to the database) has often been addressed by providing association-level fetching strategies specified in metadata of the association mapping. The trouble with this approach is that each piece of code that uses an entity requires a *different* set of associated objects. But this isn't enough. We argue that what is needed is support for fine-grained *runtime* association fetching strategies. NHibernate supports both, it lets you specify a default fetching strategy in the mapping file and then override it at runtime in code.

NHibernate allows you to choose among four fetching strategies for any association, in association metadata and at runtime:

- Immediate fetching—The associated object is fetched immediately, using a sequential database read (or cache lookup).
- Lazy fetching—The associated object or collection is fetched “lazily,” when it's first accessed. This results in a new request to the database (unless the associated object is cached).
- Eager fetching—The associated object or collection is fetched together with the owning object, using an SQL outer join, and no further database request is required.
- Batch fetching—This approach may be used to improve the performance of lazy fetching by retrieving a batch of objects or collections when a lazy association is accessed. (Batch fetching may also be used to improve the performance of immediate fetching.)

Let's look more closely at each fetching strategy.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Immediate fetching

Immediate association fetching occurs when you retrieve an entity from the database and then immediately retrieve another associated entity or entities in a further request to the database or cache. Immediate fetching isn't usually an efficient fetching strategy unless you expect the associated entities to almost always be cached already.

Lazy fetching

When a client requests an entity and its associated graph of objects from the database, it isn't usually necessary to retrieve the whole graph of every (indirectly) associated object. You wouldn't want to load the whole database into memory at once; for example, loading a single `Category` shouldn't trigger the loading of all `Items` in that category.

Lazy fetching lets you decide how much of the object graph is loaded in the first database hit and which associations should be loaded only when they're first accessed. Lazy fetching is a foundational concept in object persistence and the first step to attaining acceptable performance.

Since NHibernate 1.2, all associations are configured for lazy fetching by default; and you can easily change this behavior by setting `default-lazy="false"` in `<hibernate-mapping>` of your mapping files. But we recommend that you keep this strategy and override it at runtime by queries that force eager fetching to occur.

Eager (outer join) fetching

Lazy association fetching can help reduce database load and is often a good default strategy. However, it's a bit like a blind guess as far as performance optimization goes.

Eager fetching lets you explicitly specify which associated objects should be loaded together with the referencing object. NHibernate can then return the associated objects in a single database request, utilizing an SQL `OUTER JOIN`. Performance optimization in NHibernate often involves judicious use of eager fetching for particular transactions. Hence, even though default eager fetching may be declared in the mapping file, it's more common to specify the use of this strategy at runtime for a particular HQL or criteria query.

Batch fetching

Batch fetching isn't strictly an association fetching strategy; it's a technique that may help improve the performance of lazy (or immediate) fetching. Usually, when you load an object or collection, your SQL `WHERE` clause specifies the identifier of the object or object that owns the collection. If batch fetching is enabled, NHibernate looks to see what other proxied instances or uninitialized collections are referenced in the current session and tries to load them at the same time by specifying multiple identifier values in the `WHERE` clause.

We aren't great fans of this approach; eager fetching is almost always faster. Batch fetching is useful for inexperienced users who wish to achieve acceptable performance in NHibernate without having to think too hard about the SQL that will be executed.

We'll now declare the fetching strategy for some associations in our mapping metadata.

4.4.6 Selecting a fetching strategy in mappings

NHibernate lets you select default association fetching strategies by specifying attributes in the mapping metadata. You can override the default strategy using features of NHibernate's query methods, as you'll see in chapter 7. A minor caveat: You don't have to understand every option presented in this section immediately;

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

we recommend that you get an overview first and use this section as a reference when you're optimizing the default fetching strategies in your application.

A wrinkle in NHibernate's mapping format means that collection mappings function slightly differently than single-point associations; so, we'll cover the two cases separately. Let's first consider both ends of the bidirectional association between `Bid` and `Item`.

Single point associations

For a `<many-to-one>` or `<one-to-one>` association, lazy fetching is possible only if the associated class mapping enables proxying. For the `Item` class, we enable proxying by specifying `lazy="true"` (since NHibernate 1.2, this is the default value):

```
<class name="Item" lazy="true">
```

Now, remember the association from `Bid` to `Item`:

```
<many-to-one name="item" class="Item">
```

When we retrieve a `Bid` from the database, the association property may hold an instance of a NHibernate *generated subclass* of `Item` that delegates all method invocations to a different instance of `Item` that is fetched lazily from the database (this is the more elaborate definition of a NHibernate proxy).

In order to delegate method (and property) invocations, these members need to be `virtual`. NHibernate 1.2 uses a validator which verifies that your proxied entities have a default constructor which is not private, that they are not `sealed`, that all public methods and properties are virtual and that there is no public field. It is possible to turn this validator off; but you should carefully think about why you do that. Here is the element to add to your configuration file to turn it off:

```
<property name="hibernate.use_proxy_validator">false</property>
```

Or you can do it programmatically, before building the session factory, using:

```
cfg.Properties[NHibernate.Cfg.Environment.UseProxyValidator]="false".
```

NHibernate uses two different instances so that even polymorphic associations can be proxied—when the proxied object is fetched, it may be an instance of a mapped subclass of `Item` (if there were any subclasses of `Item`, that is). We can even choose any interface implemented by the `Item` class as the type of the proxy. To do so, we declare it using the `proxy` attribute, instead of specifying `lazy="true"`:

```
<class name="Item" proxy="ItemInterface">
```

As soon as we declare the `proxy` or `lazy` attribute on `Item`, any single-point association to `Item` is proxied and fetched lazily, unless that association overrides the fetching strategy by declaring the `outer-join` attribute.

There are three possible values for `outer-join`:

- `outer-join="auto"`—The default. When the attribute isn't specified; NHibernate fetches the associated object lazily if the associated class has proxying enabled, or eagerly using an outer join if proxying is disabled (default).
- `outer-join="true"`—NHibernate always fetches the association eagerly using an outer join, even if proxying is enabled. This allows you to choose different fetching strategies for different associations to the same proxied class. It is equivalent to `fetch="join"`.
- `outer-join="false"`—NHibernate never fetches the association using an outer join, even if proxying is disabled. This is useful if you expect the associated object to exist in the second-level cache (see chapter 5). If it isn't available in the second-level cache, the object is fetched immediately using an extra SQL `SELECT`. This option is equivalent to `fetch="select"`.

So, if we wanted to re-enable eager fetching for the association, now that proxying is enabled, we would specify

```
<many-to-one name="item" class="Item" outer-join="true">
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

For a **one-to-one** association (discussed in more detail in chapter 6), lazy fetching is conceptually possible only when the associated object always exists. We indicate this by specifying `constrained="true"`. For example, if an item can have only one bid, the mapping for the `Bid` is

```
<one-to-one name="item" class="Item" constrained="true">
```

The `constrained` attribute has a slightly similar interpretation to the `not-null` attribute of a `<many-to-one>` mapping. It tells NHibernate that the associated object is *required* and thus cannot be `null`.

To enable batch fetching, we specify the `batch-size` in the mapping for `Item`:

```
<class name="Item" lazy="true" batch-size="9">
```

The batch size limits the number of items that may be retrieved in a single batch. Choose a reasonably small number here.

You'll meet the same attributes (`outer-join`, `batch-size`, and `lazy`) when we consider collections, but the interpretation is slightly different.

Collections

In the case of collections, fetching strategies apply not just to entity associations, but also to collections of values (for example, a collection of strings could be fetched by outer join).

Just like classes, collections have their own proxies, which we usually call *collection wrappers*. Unlike classes, the collection wrapper is always there, even if lazy fetching is disabled (NHibernate needs the wrapper to detect collection modifications).

Collection mappings may declare a `lazy` attribute, an `outer-join` attribute, neither, or both (specifying both isn't meaningful). The meaningful options are as follows:

- Neither attribute specified—This option is equivalent to `outer-join="false" lazy="false"`. The collection is fetched from the second-level cache or by an immediate extra SQL `SELECT`. This option is most useful when the second-level cache is enabled for this collection.
- `outer-join="true"`—NHibernate fetches the association eagerly using an outer join. At the time of this writing, NHibernate is able to fetch only one collection per SQL `SELECT`, so it isn't possible to declare multiple collections belonging to the same persistent class with `outer-join="true"`.
- `lazy="true"`—NHibernate fetches the collection lazily, when it's first accessed. Since NHibernate 1.2, this is the default option and we recommend that you keep this option as a default for all your collection mappings.

We don't recommend eager fetching for collections, so we'll map the item's collection of bids with `lazy="true"`. This option is almost always used for collection mappings (although it is the default since NHibernate 1.2, we will continue to write it to insist on it):

```
<set name="Bids" lazy="true">
  <key column="ITEM_ID"/>
  <one-to-many class="Bid"/>
</set>
```

We can even enable batch fetching for the collection. In this case, the batch size doesn't refer to the number of bids in the batch; it refers to the number of collections of bids:

```
<set name="Bids" lazy="true" batch-size="9">
  <key column="ITEM_ID"/>
  <one-to-many class="Bid"/>
</set>
```

This mapping tells NHibernate to load up to nine collections of bids in one batch, depending on how many uninitialized collections of bids are currently present in the items associated with the session. In other words, if there are five `Item` instances with persistent state in an `ISession`, and all have an uninitialized `Bids` collection, NHibernate will automatically load all five collections in a single SQL query if one is accessed. If there are 11 items, only 9 collections will be fetched. Batch fetching can significantly reduce the number of

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

queries required for hierarchies of objects (for example, when loading the tree of parent and child `Category` objects).

Let's talk about a special case: many-to-many associations (we discuss this mapping in more detail in chapter 6). You usually use a link table (some developers also call it relationship table or association table) that holds only the key values of the two associated tables and therefore allows a many-to-many multiplicity. This additional table has to be considered if you decide to use eager fetching. Look at the following straightforward many-to-many example, which maps the association from `Category` to `Item`:

```
<set name="Items" outer-join="true" table="CATEGORY_ITEM">
  <key column="CATEGORY_ID"/>
  <many-to-many column="ITEM_ID" class="Item"/>
</set>
```

In this case, the eager fetching strategy refers only to the association table `CATEGORY_ITEM`. If we load a `Category` with this fetching strategy, NHibernate will automatically fetch all link entries from `CATEGORY_ITEM` in a single outer join SQL query, but not the item instances from `ITEM`!

The entities contained in the many-to-many association can of course also be fetched eagerly with the same SQL query. The `<many-to-many>` element allows this behavior to be customized:

```
<set name="Items" outer-join="true" table="CATEGORY_ITEM">
  <key column="CATEGORY_ID"/>
  <many-to-many column="ITEM_ID" outer-join="true" class="Item"/>
</set>
```

NHibernate will now fetch all `Items` in a `Category` with a single outer join query when the `Category` is loaded. However, keep in mind that we usually recommend lazy loading as the default fetching strategy and that NHibernate is limited to one eagerly fetched collection per mapped persistent class.

Setting the fetch depth

We'll now discuss a global fetching strategy setting: the *maximum fetch depth*. This setting controls the number of outer-joined tables NHibernate will use in a single SQL query. Consider the complete association chain from `Category` to `Item`, and from `Item` to `Bid`. The first is a many-to-many association and the second is a one-to-many; hence both associations are mapped with collection elements. If we declare `outer-join="true"` for both associations (don't forget the special `<many-to-many>` declaration) and load a single `Category`, how many queries will NHibernate execute? Will only the `Items` be eagerly fetched, or also all the `Bids` of each `Item`?

You probably expect a single query, with an outer join operation including the `CATEGORY`, `CATEGORY_ITEM`, `ITEM`, and `BID` tables. However, this isn't the case by default.

NHibernate's outer join fetch behavior is controlled with the global configuration option `hibernate.max_fetch_depth`. If you set this to `1` (also the default), NHibernate will fetch only the `Category` and the link entries from the `CATEGORY_ITEM` association table. If you set it to `2`, NHibernate executes an outer join that also includes the `Items` in the same SQL query. Setting this option to `3` will not, as you might have expected, also include the bids of each item in the same SQL query. The limitation to one outer joined collection applies here, preventing slow Cartesian products.

Recommended values for the fetch depth depend on the join performance and the size of the database tables; test your applications with low values (less than `4`) first, and decrease or increase the number while tuning your application. The global maximum fetch depth also applies to single-ended association (`<many-to-one>`, `<one-to-one>`) mapped with an eager fetching strategy or using the auto default.

Keep in mind that eager fetching strategies declared in the mapping metadata are effective only if you use retrieval by identifier, use the criteria query API, or navigate through the object graph manually. Any HQL

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

query may specify its own fetching strategy at runtime, thus ignoring the mapping defaults. You can also override the defaults (that is, not ignore them) with criteria queries. This is an important difference, and we cover it in more detail in chapter 7, section 7.3.2, “Fetching associations.”

However, you may sometimes simply like to initialize a proxy or a collection wrapper manually with a simple API call.

Initializing lazy associations

A proxy or collection wrapper is automatically initialized when any of its methods are invoked (except the identifier property getter, which may return the identifier value without fetching the underlying persistent object). However, it’s only possible to initialize a proxy or collection wrapper if it’s currently associated with an open `ISession`. If you close the session and try to access an uninitialized proxy or collection, NHibernate throws a `LazyInitializationException`.

Because of this behavior, it’s sometimes useful to explicitly initialize an object before closing the session. This approach isn’t as flexible as retrieving the complete required object subgraph with an HQL query, using arbitrary fetching strategies at runtime.

We use the static method `NHibernateUtil.Initialize()` for manual initialization:

```
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    Category cat = session.Get<Category>(id);
    NHibernateUtil.Initialize( cat.Items );
    session.Transaction.Commit();
}
foreach( Item item in cat.Items)
...

```

`NHibernateUtil.Initialize()` may be passed a collection wrapper, as in this example, or a proxy. You may also, in similar rare cases, check the current state of a property by calling `NHibernateUtil.IsInitialized()`. (Note that `Initialize()` doesn’t cascade to any associated objects.)

Another solution for this problem is to keep the session open until the application thread finishes, so you can navigate the object graph whenever you like and have NHibernate automatically initialize all lazy references. This is a problem of application design and transaction demarcation; we discuss it again in chapter 8, section 8.1. However, your first choice should be to fetch the complete required graph in the first place, using HQL or criteria queries, with a sensible and optimized default fetching strategy in the mapping metadata for all other cases. NHibernate allows you to look at the underlying SQL that it sends to the database, so it is possible to tune object retrieval if performance problems are observed suspected. This is discussed in the next section.

4.4.7 Tuning object retrieval

In most cases your NHibernate applications will perform well when it comes to fetching data from the database. However, occasionally you may notice that some areas of your application aren’t performing as well as they should. There can be many reasons for this, so we need to understand how to analyze and tune our NHibernate applications so that they work efficiently with the database. Let’s look at the steps involved when you’re tuning the object retrieval operations in your application:

Enable the NHibernate SQL log, as described in chapter 2, section 2.4.2. You should also be prepared to read, understand, and evaluate SQL queries and their performance characteristics for your specific relational model: Will a single join operation be faster than two selects? Are all the indexes used properly, and what is

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

the cache hit ratio inside the database? Get your DBA to help you with the performance evaluation; only he will have the knowledge to decide which SQL execution plan is the best.

Step through your application use case by use case and note how many and what SQL statements NHibernate executes. A use case can be a single screen in your web application or a sequence of user dialogs. This step also involves collecting the object-retrieval methods you use in each use case: walking the graph, retrieval by identifier, HQL, and criteria queries. Your goal is to bring down the number (and complexity) of SQL queries for each use case by tuning the default fetching strategies in metadata.

You may encounter two common issues:

- If the SQL statements use join operations that are too complex and slow, set `outer-join` to false for `<many-to-one>` associations (this is enabled by default). Also try to tune with the global `hibernate.max_fetch_depth` configuration option, but keep in mind that this is best left at a value between 1 and 4.
- If too many SQL statements are executed, use `lazy="true"` for all collection mappings; by default, NHibernate will execute an immediate additional fetch for the collection elements (which, if they're entities, can cascade further into the graph). In rare cases, if you're sure, enable `outer-join="true"` and disable lazy loading for particular collections. Keep in mind that only one collection property per persistent class may be fetched eagerly. Use batch fetching with values between 3 and 15 to further optimize collection fetching if the given unit of work involves several "of the same" collections or if you're accessing a tree of parent and child objects.

After you set a new fetching strategy, rerun the use case and check the generated SQL again. Note the SQL statements, and go to the next use case.

After you optimize all use cases, check every one again and see if any optimizations had side effects for others. With some experience, you'll be able to avoid any negative effects and get it right the first time.

This optimization technique isn't only practical for the default fetching strategies; you can also use it to tune HQL and criteria queries, which can ignore and override the default fetching for specific use cases and units of work. We discuss runtime fetching in chapter 8.

In this section, we've started to think about performance issues, especially issues related to association fetching. Of course, the quickest way to fetch a graph of objects is to fetch it from the cache in memory, as shown in the next chapter.

4.5 Summary

The dynamic aspects of the object/relational mismatch are just as important as the better known and better understood structural mismatch problems. In this chapter, we were primarily concerned with the lifecycle of objects with respect to the persistence mechanism. Now you understand the three object states defined by NHibernate: persistent, detached, and transient. Objects transition between states when you invoke methods of the `ISession` interface or create and remove references from a graph of already persistent instances. This latter behavior is governed by the configurable cascade styles, NHibernate's model for transitive persistence. This model lets you declare the cascading of operations (such as saving or deletion) on an association basis, which is more powerful and flexible than the traditional *persistence by reachability* model. Your goal is to find the best cascading style for each association and therefore minimize the number of persistence manager calls you have to make when storing objects.

Retrieving objects from the database is equally important: You can walk the graph of domain objects by accessing properties and let NHibernate transparently fetch objects. You can also load objects by identifier,

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

write arbitrary queries in the HQL, or create an object-oriented representation of your query using the query by criteria API. In addition, you can use native SQL queries in special cases.

Most of these object-retrieval methods use the default fetching strategies we defined in mapping metadata (HQL ignores them; criteria queries can override them). The correct fetching strategy minimizes the number of SQL statements that have to be executed by lazily, eagerly, or batch-fetching objects. You optimize your NHibernate application by analyzing the SQL executed in each use case and tuning the default and runtime fetching strategies.

Next we explore the closely related topics of *transactions* and *caching*.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Transactions, concurrency, and caching

This chapter covers

- Database transactions and locking
- Long-running conversations
- The NHibernate first- and second-level caches
- The caching system in practice with CaveatEmptor

Now that you understand the basics of object/relational mapping with NHibernate, let's take a closer look at one of the core issues in database application design: *transaction management*. In this chapter, we examine how you use NHibernate to manage transactions, how concurrency is handled, and how caching is related to both aspects. Let's look at our example application.

Some application functionality requires that several different things be done together. For example, when an auction finishes, our CaveatEmptor application has to perform four different tasks:

7. Mark the winning (highest amount) bid.
8. Charge the seller the cost of the auction.
9. Charge the successful bidder the price of the winning bid.
10. Notify the seller and the successful bidder.

What happens if we can't bill the auction costs because of a failure in the external credit card system? Our business requirements might state that either all listed actions must succeed or none must succeed. If so, we call these steps collectively a *transaction* or *unit of work*. If only one step fails, the whole unit of work must fail. We say that the transaction is *atomic*: Several operations are grouped together as a single indivisible unit.

Furthermore, transactions allow multiple users to work concurrently with the same data without compromising the integrity and correctness of the data; a particular transaction shouldn't be visible to and shouldn't influence other concurrently running transactions. Several different strategies are used to implement this behavior, which is called *isolation*. We'll explore them in this chapter.

Transactions are also said to exhibit *consistency* and *durability*. Consistency means that any transaction works with a consistent set of data and leaves the data in a consistent state when the transaction completes. Durability guarantees that once a transaction completes, all changes made during that transaction become persistent and aren't lost even if the system subsequently fails. Atomicity, consistency, isolation, and durability are together known as the ACID criteria.

We begin this chapter with a discussion of system-level *database transactions*, where the database guarantees ACID behavior. We'll look at the ADO.NET and Enterprise Services transactions APIs and see how NHibernate, working as a client of these APIs, is used to control database transactions.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

In an online application, database transactions must have extremely short lifespans. A database transaction should span a single batch of database operations, interleaved with business logic. It should certainly not span interaction with the user. We'll augment your understanding of transactions with the notion of a long-running *user transaction* called *conversation*, where database operations occur in several batches, alternating with user interaction. There are several ways to implement conversations in NHibernate applications, all of which are discussed in this chapter.

Finally, the subject of caching is much more closely related to transactions than it might appear at first sight. For example, caching allows us to keep data close to where it's needed, but at the risk of getting stale over time. Therefore caching strategies need to be balanced to also allow for consistent and durable transactions. In the second half of this chapter, armed with an understanding of transactions, we explore NHibernate's sophisticated cache architecture. You'll learn which data is a good candidate for caching and how to handle concurrency of the cache. We'll then enable caching in the CaveatEmptor application.

Let's begin with the basics and see how transactions work at the lowest level, the database.

5.1 Understanding database transactions

Databases implement the notion of a unit of work as a *database transaction* (sometimes called a *system transaction*).

A database transaction groups data-access operations. A transaction is guaranteed to end in one of two ways: it's either *committed* or *rolled back*. Hence, database transactions are always truly *atomic*. In figure 5.1, you can see this graphically.

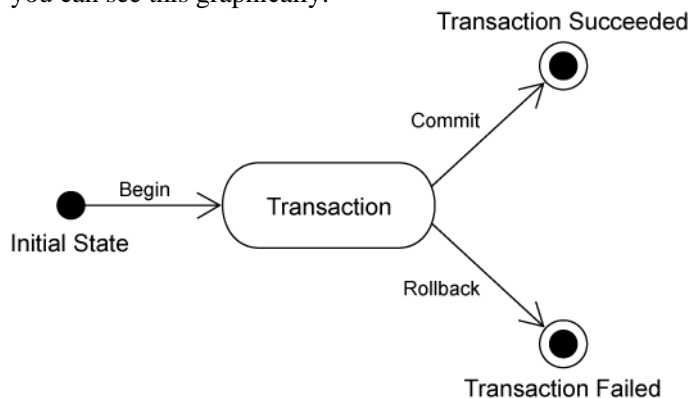


Figure 5.1 System states during a transaction

If several database operations should be executed inside a transaction, you must mark the boundaries of the unit of work. You must start the transaction and, at some point, commit the changes. If an error occurs (either while executing operations or when committing the changes), you have to roll back the transaction to leave the data in a consistent state. This is known as *transaction demarcation*, and (depending on the API you use) it involves more or less manual intervention.

You may already have experience with two transaction-handling programming interfaces: the ADO.NET API and the COM+ automatic transaction processing service.

5.1.1 ADO.NET and Enterprise Services/COM+ transactions

Without Enterprise Services, the ADO.NET **Error! Bookmark not defined.** API is used to mark transaction boundaries. You begin a transaction by calling `BeginTransaction()` on an ADO.NET connection and end it by calling `Commit()`. You may, at any time, force an immediate rollback by calling `Rollback()`. (Easy, huh?)

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

In a system that stores data in multiple databases, a particular unit of work may involve access to more than one data store. In this case, you can't achieve atomicity using ADO.NET alone. You require a transaction manager with support for distributed transactions (two-phase commit). You communicate with the transaction manager using the COM+ automatic transaction processing service.

With Enterprise Services, the automatic transaction processing service is used not only for distributed transactions, but also for declarative transaction-processing features. Declarative transaction-processing allows you to avoid explicit transaction demarcation calls in your application source code; rather, transaction demarcation is controlled by transaction attributes. The declarative transaction attribute specifies how an object participates in a transaction, and is configured programmatically.

We aren't interested in the details of direct ADO.NET or Enterprise Services transaction demarcation. You'll be using these APIs mostly indirectly. Chapter 10, section 10.3 explains how to make NHibernate and Enterprise Services transactions work together.

NHibernate communicates with the database via an `ADO.NETError! Bookmark not defined. IDbConnection` and it provides its own abstraction layer, hiding the underlying transaction API. Using Enterprise Services doesn't require any change in the configuration of NHibernate.

Transaction management is exposed to the application developer via the NHibernate `ITransaction` interface. You aren't forced to use this API—NHibernate lets you control ADO.NET transactions directly. We won't discuss this option as it is discouraged, so instead we'll now focus on the `ITransaction` API and its usage.

5.1.2 The NHibernate `ITransaction` API

The `ITransaction` interface provides methods for declaring the boundaries of a database transaction. See listing 5.1 for an example of the basic usage of `ITransaction`.

Listing 5.1 Using the NHibernate `ITransaction` API

```
using( ISession session = sessions.OpenSession() )
    using( session.BeginTransaction() ) {
        ConcludeAuction();
        session.Transaction.Commit();
    }
```

The call to `session.BeginTransaction()` marks the beginning of a database transaction. This starts an ADO.NET transaction on the ADO.NET connection. With COM+, you don't need to create this transaction; the connection will be automatically enlisted in the current distributed transaction or you will have to do it manually if you have turned automatic transaction enlistment off.

The call to `Transaction.Commit()` synchronizes the `ISession` state with the database. NHibernate then commits the underlying transaction if and only if `BeginTransaction()` started a new transaction (with COM+, you have to vote in favor of completing the distributed transaction).

If `ConcludeAuction()` threw an exception, the `using()` statement will dispose the transaction (here, it means doing a rollback).

Do I need a transaction for read-only operations?

Due to the new connection release mode of NHibernate 1.2, a database connection is opened and closed for each transaction. As long as you are executing a single query, you can let NHibernate manage the transaction.

It's critically important to make sure that the session is closed at the end in order to ensure that the ADO.NET connection is released and returned to the connection pool. (This step is the responsibility of the application.)

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Here is another version showing in details where exceptions can be thrown and how to deal with them: (this version is more complex than the one presented in chapter 2)

```
ISession session = sessions.OpenSession();
ITransaction tx = null;
try {
    tx = session.BeginTransaction();
    ConcludeAuction();

    tx.Commit();
} catch (Exception e) {
    if (tx != null) {
        try {
            tx.Rollback();
        } catch (HibernateException he) {
            //log he
        }
    }
    throw;
} finally {
    try {
        session.Close();
    } catch (HibernateException he) {
        throw;
    }
}
```

As you can see, even rolling back an `ITransaction` and closing the `ISession` can throw an exception. You don't want to use this example as a template in your own application, since you'd rather hide the exception handling with generic infrastructure code. You can, for example, wrap the thrown exception in your own `InfrastructureException`. We discuss this question of application design in more detail in chapter 8, section 8.1, "Inside the layers of a NHibernate application".

NOTE

There is one important aspect you must be aware of: the `ISession` has to be immediately closed and discarded (not reused) when an exception occurs. NHibernate can't retry failed transactions. This is no problem in practice, because database exceptions are usually fatal (constraint violations, for example) and there is no well-defined state to continue after a failed transaction. An application in production shouldn't throw any database exceptions either.

You must also be aware that after committing a transaction, the NHibernate session replaces it by a new transaction. This means that you should keep a reference to the transaction you are committing if you think that you will need it afterward. This is necessary if you need to call `transaction.WasCommitted()`. `session.Transaction.WasCommitted` will always return `false`.

We've noted that the call to `Commit()` synchronizes the `ISession` state with the database. This is called *flushing*, a process you automatically trigger when you use the NHibernate `ITransaction` API.

5.1.3 Flushing the Session

The NHibernate `ISession` implements *transparent write behind*. This means that changes to the domain model made in the scope of an `ISession` aren't immediately propagated to the database. Instead, NHibernate can coalesce many changes into a minimal number of database requests, helping minimize the impact of network latency.

For example, if a single property of an object is changed twice in the same `ITransaction`, NHibernate only needs to execute one SQL `UPDATE`.

NHibernate flushes occur only at the following times:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

- When an `ITransaction` is committed
- Sometimes before a query is executed
- When the application calls `ISession.Flush()` explicitly

Flushing the `ISession` state to the database at the end of a database transaction is required in order to make the changes durable and is the common case. NHibernate doesn't flush before every query. However, if there are changes held in memory that would affect the results of the query, NHibernate will, by default, synchronize first.

You can control this behavior by explicitly setting the NHibernate `FlushMode` via to the property `session.FlushMode`. The flush modes are as follows:

- `FlushMode.Auto`—The default. Enables the behavior just described.
- `FlushMode.Commit`—Specifies that the session won't be flushed before query execution (it will be flushed only at the end of the database transaction). Be aware that this setting may expose you to stale data: modifications you made to objects only in memory may conflict with the results of the query.
- `FlushMode.Never`—Lets you specify that only explicit calls to `Flush()` result in synchronization of session state with the database.

We don't recommend that you change this setting from the default. It's provided to allow performance optimization in rare cases. Likewise, most applications rarely need to call `Flush()` explicitly. This functionality is useful when you're working with triggers, mixing NHibernate with direct ADO.NET, or working with buggy ADO.NET drivers. You should be aware of the option but not necessarily look out for use cases.

We've discussed how NHibernate handles both transactions and the flushing of changes to the database. Another important responsibility of NHibernate is managing actual connections to the database. We discuss this next.

5.1.4 Understanding connection release modes

As a valuable resource, the database connection should be held opened for the shortest amount of time possible. NHibernate is smart enough to open it only when really needed (opening the session doesn't automatically open the connection). Since NHibernate 1.2, it is also possible to define when the database connection should be closed.

There are currently two options available. They are defined by the enumeration `NHibernate.ConnectionReleaseMode`:

- `OnClose`—This was the only mode available in NHibernate 1.0. In this case, the session releases the connection when it is closed.
- `AfterTransaction`—This is the default mode in NHibernate 1.2. The connection is released as soon as the transaction completes.

Note that you can use the `Disconnect()` method of the `ISession` interface to force the release of the connection (without closing the session) and the `Reconnect()` method to tells the session to obtain a new connection when needed.

Obviously, these modes are only activated for connections opened by NHibernate. If you open a connection and send it to NHibernate, you are also responsible of closing this connection.

In order to specify a mode, you must use the configuration parameter `hibernate.connection.release_mode`. Its default (and recommended) value is `auto`. It selects the best mode, which is currently `AfterTransaction`. The two other values are `on_close` and `after_transaction`.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

As NHibernate 1.2.0 has some problems dealing with APIs like `System.Transactions`, you have to use the `OnClose` mode if you discover that the session opens multiple connections in a single transaction.

Now that you understand the basic usage of database transactions with the NHibernate `ITransaction` interface, let's turn our attention more closely to the subject of concurrent data access.

It seems as though you shouldn't have to care about transaction isolation—the term implies that something either is or is not isolated. This is misleading. *Complete* isolation of concurrent transactions is extremely expensive in terms of application scalability, so databases provide several degrees of isolation. For most applications, incomplete transaction isolation is acceptable. It's important to understand the degree of isolation you should choose for an application that uses NHibernate and how NHibernate integrates with the transaction capabilities of the database.

5.1.5 Understanding Isolation Levels

Databases (and other transactional systems) attempt to ensure *transaction isolation*, meaning that, from the point of view of each concurrent transaction, it appears that no other transactions are in progress.

Traditionally, this has been implemented using locking. A transaction may place a lock on a particular item of data, temporarily preventing access to that item by other transactions. Some modern databases such as Oracle and PostgreSQL implement transaction isolation using *multi-version concurrency control*, which is generally considered more scalable. We'll discuss isolation assuming a locking model (most of our observations are also applicable to multi-version concurrency).

This discussion is about database transactions and the isolation level provided by the database. NHibernate doesn't add additional semantics; it uses whatever is available with a given database. If you consider the many years of experience that database vendors have had with implementing concurrency control, you'll clearly see the advantage of this approach. Your part, as a NHibernate application developer, is to understand the capabilities of your database and how to change the database isolation behavior if needed in your particular scenario (and by your data integrity requirements).

Isolation Issues

First, let's look at several phenomena that break full transaction isolation. The ANSI SQL standard defines the standard transaction isolation levels in terms of which of these phenomena are permissible:

- *Lost update*—Two transactions both update a row and then the second transaction aborts, causing both changes to be lost. This occurs in systems that don't implement any locking. The concurrent transactions aren't isolated.
- *Dirty read*—One transaction reads changes made by another transaction that hasn't yet been committed. This is very dangerous, because those changes might later be rolled back.
- *Unrepeatable read*—A transaction reads a row twice and reads different state each time. For example, another transaction may have written to the row, and committed, between the two reads.
- *Second lost updates problem*—A special case of an unrepeatable read. Imagine that two concurrent transactions both read a row, one writes to it and commits, and then the second writes to it and commits. The changes made by the first writer are lost. This problem is also known as *Last Write Win*.
- *Phantom read*—A transaction executes a query twice, and the second result set includes rows that weren't visible in the first result set. (It needs not necessarily be *exactly* the same query.) This situation is caused by another transaction inserting new rows between the execution of the two queries.

Now that you understand all the bad things that could occur, we can define the various *transaction isolation levels* and see what problems they prevent.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Isolation levels

The standard isolation levels are defined by the ANSI SQL standard and you'll use these levels to declare your desired transaction isolation later:

- *Read uncommitted*—Permits dirty reads but not lost updates. One transaction may not write to a row if another uncommitted transaction has already written to it. Any transaction may read any row, however. This isolation level may be implemented using exclusive write locks.
- *Read committed*—Permits unrepeatable reads but not dirty reads. This may be achieved using momentary shared read locks and exclusive write locks. Reading transactions don't block other transactions from accessing a row. However, an uncommitted writing transaction blocks all other transactions from accessing the row.
- *Repeatable read*—Permits neither unrepeatable reads nor dirty reads. Phantom reads may occur. This may be achieved using shared read locks and exclusive write locks. Reading transactions block writing transactions (but not other reading transactions), and writing transactions block all other transactions.
- *Serializable*—Provides the strictest transaction isolation. It emulates serial transaction execution, as if transactions had been executed one after another, serially, rather than concurrently. Serializability may not be implemented using only row-level locks; there must be another mechanism that prevents a newly inserted row from becoming visible to a transaction that has already executed a query that would return the row.

It's nice to know how all these technical terms are defined, but how does that help you choose an isolation level for your application?

5.1.6 Choosing an isolation level

Developers (ourselves included) are often unsure about what transaction isolation level to use in a production application. Too great a degree of isolation will harm performance of a highly concurrent application. Insufficient isolation may cause subtle bugs in our application that can't be reproduced and that we'll never find out about until the system is working under heavy load in the deployed environment.

Note that we refer to *caching* and *optimistic locking* (using versioning) in the following explanation, two concepts explained later in this chapter. You might want to skip this section and come back when it's time to make the decision for an isolation level in your application. Picking the right isolation level is, after all, highly dependent on your particular scenario. The following discussion contains recommendations; nothing is carved in stone.

NHibernate tries hard to be as transparent as possible regarding the transactional semantics of the database. Nevertheless, caching and optimistic locking affect these semantics. So, what is a sensible database isolation level to choose in a NHibernate application?

First, you eliminate the *read uncommitted* isolation level. It's extremely dangerous to use one transaction's uncommitted changes in a different transaction. The rollback or failure of one transaction would affect other concurrent transactions. Rollback of the first transaction could bring other transactions down with it, or perhaps even cause them to leave the database in an inconsistent state. It's possible that changes made by a transaction that ends up being rolled back could be committed anyway, since they could be read and then propagated by another transaction that *is* successful!

Second, most applications don't need *serializable* isolation (phantom reads aren't usually a problem), and this isolation level tends to scale poorly. Few existing applications use serializable isolation in production; rather, they use pessimistic locks (see section 6.1.8, "Using pessimistic locking"), which effectively forces a serialized execution of operations in certain situations.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This leaves you a choice between *read committed* and *repeatable read*. Let's first consider repeatable read. This isolation level eliminates the possibility that one transaction could overwrite changes made by another concurrent transaction (the second lost updates problem) if all data access is performed in a single atomic database transaction. This is an important issue, but using repeatable read isn't the only way to resolve it.

Let's assume you're using versioned data, something that NHibernate can do for you automatically. The combination of the (mandatory) NHibernate first-level session cache and versioning already gives you most of the features of repeatable read isolation. In particular, versioning prevents the second lost update problem, and the first-level session cache ensures that the state of the persistent instances loaded by one transaction is isolated from changes made by other transactions. So, read committed isolation for all database transactions would be acceptable if you use versioned data.

Repeatable read provides a bit more reproducibility for query result sets (only for the duration of the database transaction), but since phantom reads are still possible, there isn't much value in that. (It's also not common for web applications to query the same table twice in a single database transaction.)

You also have to consider the (optional) second-level NHibernate cache. It can provide the same transaction isolation as the underlying database transaction, but it might even weaken isolation. If you're heavily using a cache concurrency strategy for the second-level cache that doesn't preserve repeatable read semantics (for example, the read-write and especially the nonstrict-read-write strategies, both discussed later in this chapter), the choice for a default isolation level is easy: You can't achieve repeatable read anyway, so there's no point slowing down the database. On the other hand, you might not be using second-level caching for critical classes, or you might be using a fully transactional cache that provides repeatable read isolation. Should you use repeatable read in this case? You can if you like, but it's probably not worth the performance cost.

Setting the transaction isolation level allows you to choose a good default locking strategy for all your database transactions. How do you set the isolation level?

5.1.7 Setting an Isolation level

Every ADO.NET connection to a database uses the database's default isolation level, usually read committed or repeatable read. This default can be changed in the database configuration. You may also set the transaction isolation for ADO.NET connections using a NHibernate configuration option:

```
<add
  key="hibernate.connection.isolation"
  value="ReadCommitted"
/>
```

NHibernate will then set this isolation level on every ADO.NET connection obtained from a connection pool before starting a transaction. Some of the sensible values for this option are as follows (you can also find them in `System.Data.IsolationLevel`):

- ReadUncommitted—Read uncommitted isolation
- ReadCommitted—Read committed isolation
- RepeatableRead—Repeatable read isolation
- Serializable—Serializable isolation

Note that NHibernate never changes the isolation level of connections obtained from a datasource provided by COM+. You may change the default isolation using the `Isolation` property of `System.EnterpriseServices.TransactionAttribute`.

So far, we've introduced the issues that surround transaction isolation, the isolation levels available, and how to select the correct one for your application. As you can see, setting the isolation level is a global option

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

that affects all connections and transactions. From time to time, it's useful to specify a more restrictive lock for a particular transaction. NHibernate allows you to explicitly specify the use of a *pessimistic* lock.

5.1.8 Using pessimistic locking

Locking is a mechanism that prevents concurrent access to a particular item of data. When one transaction holds a lock on an item, no concurrent transaction can read and/or modify this item. A lock might be just a momentary lock, held while the item is being read, or it might be held until the completion of the transaction. A *pessimistic lock* is a lock that is acquired when an item of data is read and that is held until transaction completion.

In read-committed mode (our preferred transaction isolation level), the database never acquires pessimistic locks unless explicitly requested by the application. Usually, pessimistic locks aren't the most scalable approach to concurrency. However, in certain special circumstances, they may be used to prevent database-level deadlocks, which result in transaction failure. Some databases (Oracle, MySQL and PostgreSQL, for example, but not SQL Server) provide the SQL `SELECT...FOR UPDATE` syntax to allow the use of explicit pessimistic locks. You can check the NHibernate *Dialects* to find out if your database supports this feature. If your database isn't supported, NHibernate will always execute a normal `SELECT` without the `FOR UPDATE` clause.

The NHibernate `LockMode` class lets you request a pessimistic lock on a particular item. In addition, you can use the `LockMode` to force NHibernate to bypass the cache layer or to execute a simple version check. You'll see the benefit of these operations when we discuss versioning and caching.

Let's see how to use `LockMode`. You might have a transaction that looks like this

```
ITransaction tx = session.BeginTransaction();
Category cat = session.Get<Category>(catId);
cat.Name = "New Name";
tx.Commit();
```

It is possible to make this transaction use a pessimistic lock as follows:

```
ITransaction tx = session.BeginTransaction();
Category cat = session.Get<Category>(catId, LockMode.Upgrade);
cat.Name = "New Name";
tx.Commit();
```

With `LockMode.Upgrade`, NHibernate will load the `Category` using a `SELECT...FOR UPDATE`, thus locking the retrieved rows in the database until they're released when the transaction ends.

NHibernate defines several lock modes:

- `LockMode.None`—Don't go to the database unless the object isn't in either cache.
- `LockMode.Read`—Bypass both levels of the cache, and perform a version check to verify that the object in memory is the same version that currently exists in the database.
- `LockMode.Upgrade`—Bypass both levels of the cache, do a version check (if applicable), and obtain a database-level pessimistic upgrade lock, if that is supported.
- `LockMode.UpgradeNowait`—The same as `UPGRADE`, but use a `SELECT...FOR UPDATE NOWAIT`, if that is supported. This disables waiting for concurrent lock releases, thus throwing a locking exception immediately if the lock can't be obtained.
- `LockMode.Write`—The lock is obtained automatically when NHibernate writes to a row in the current transaction (this is an internal mode; you can't specify it explicitly).

By default, `Load()` and `Get()` use `LockMode.None`. `LockMode.Read` is most useful with `ISession.Lock()` and a detached object. For example:

```
Item item = ... ;
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
Bid bid = new Bid();
item.AddBid(bid);
...
ITransaction tx = session.BeginTransaction();
session.Lock(item, LockMode.Read);
tx.Commit();
```

This code performs a version check on the detached `Item` instance to verify that the database row wasn't updated by another transaction since it was retrieved. If it was updated, a `StaleObjectStateException` is thrown.

Behind the scene, NHibernate executes a `SELECT` to make sure that there is a database row with the identifier (and version, if present) of the detached object. It doesn't check all the columns. This is not a problem when the version is present because it is always updated when persisting the object using NHibernate. If, for some reason, you bypass NHibernate and use ADO.NET, don't forget to update the version.

By specifying an explicit `LockMode` other than `LockMode.None`, you force NHibernate to bypass both levels of the cache and go all the way to the database. We think that most of the time caching is more useful than pessimistic locking, so we don't use an explicit `LockMode` unless we really need it. Our advice is that if you have a professional DBA on your project and let the DBA decide which transactions require pessimistic locking once the application is up and running. This decision should depend on subtle details of the interactions between different transactions and can't be guessed up front.

Let's consider another aspect of concurrent data access. We think that most .NET developers are familiar with the notion of a database transaction and that is what they usually mean by *transaction*. In this book, we consider this to be a *fine-grained* transaction, but we also consider a more coarse-grained notion. Our coarse-grained transactions will correspond to what *the user of the application* considers a single unit of work. Why should this be any different than the fine-grained database transaction?

The database *isolates* the effects of concurrent database transactions. It should appear to the application that each transaction is the only transaction currently accessing the database (even when it isn't). Isolation is expensive. The database must allocate significant resources to each transaction for the duration of the transaction. In particular, as we've discussed, many databases lock rows that have been read or updated by a transaction, preventing access by any other transaction, until the first transaction completes. In highly concurrent systems with hundreds or thousands of updates per second, these locks can prevent scalability if they're held for longer than absolutely necessary. For this reason, you shouldn't hold the database transaction (or even the ADO.NET connection) open while waiting for user input. If a user takes a few minutes to enter data into a form whilst the database is locking resources, then other transactions may be blocked for that entire duration! All this, of course, also applies to a NHibernate `ITransaction`, since it's merely an adaptor to the underlying database transaction mechanism.

If you want to handle long user "think time" while still taking advantage of the ACID attributes of transactions, simple database transactions aren't sufficient. You need a new concept, long-running *user transactions* also known as a *conversation*.

5.2 Working with conversations

Business processes, which might be considered a single unit of work *from the point of view of the user*, necessarily span multiple user client requests. This is especially true when a user makes a decision to update data on the basis of the current state of that data.

In an extreme example, suppose you collect data entered by the user on multiple screens, perhaps using wizard-style step-by-step navigation. You must read and write related items of data in several requests (hence several database transactions) until the user clicks Finish on the last screen. Throughout this process, the data

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

must remain consistent and the user must be informed of any change to the data made by any concurrent transaction. We call this coarse-grained transaction concept a *conversation*, a broader notion of the unit of work.

We'll now restate this definition more precisely. Most .NET applications include several examples of the following type of functionality:

Data is retrieved and displayed on the screen, requiring the first database transaction as data is read.

11. The user has an opportunity to view and then modify the data in their own time. (Of course, no database transaction need here).
12. The modifications are made persistent, which requires a second database transaction as data is written.

In more complicated applications, there may be several such interactions with the user before a particular business process is complete. This leads to the notion of a conversation (sometimes called a *long transaction*, *user transaction*, *application transaction* or *business transaction*). We prefer the terms conversation or user transaction, since these are less vague and emphasize the transaction aspect from the point of view of the user.

During these long user-based transactions, you can't rely on the database to enforce isolation or atomicity of concurrent conversations. Therefore, isolation becomes something your application needs to deal with explicitly — and may even require getting the user's input.

Now, let's discuss conversation with an example.

In our CaveatEmptor application, both the user who posted a comment and any system administrator can open an Edit Comment screen to delete or edit the text of a comment. Suppose two different administrators open the edit screen to view the same comment at the same time. Both edit the comment text and submit their changes. How can we handle this? There are three strategies:

- *Last commit wins*— Both updates are saved to the database, but the last one overwrites the first. No error message is shown to anyone, and the first update is silently lost forever.
- *First commit wins*—The first update is saved, but when second user attempts to save their changes they receive an error message saying “your updates were lost because someone else updated the record whilst you were editing it”. The user must start their edits again and hope they have more luck next time they hit save! This option is often called *optimistic locking* – the application optimistically assumes there won't be problems, but checks and reports if there are.
- *Merge conflicting updates*—The first modification is persisted, and when the second user saves, they are given the option of merging the records. This is also falls under the category of optimistic locking.

The first option, last commit wins, is problematic; the second user overwrites the changes of the first user without seeing the changes made by the first user or even knowing that they existed. In our example, this probably wouldn't matter, but it would be unacceptable in many scenarios. The second and third options are acceptable for most scenarios. In practice there is no single best solution, you must investigate your own business requirements and select one of these three options.

When using NHibernate, the first option happens by default, and therefore requires no work on your part. You should assess which parts of your application, if any, can get away with this easy, but potentially dangerous approach.

If you decide you need the other optimistic locking options, then you will need to add appropriate code to your application. NHibernate can help you implement this using *managed versioning* for *optimistic locking*, also known as *Optimistic Offline Lock*.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

5.2.1 Using managed versioning

Managed versioning relies on either a version number that is incremented or a timestamp that is updated to the current time, every time an object is modified. Note that it has no relation with SQL Server's `TIMESTAMP` column and that database-driven concurrency features are not supported.

For NHibernate managed versioning, we must add a new property to our `Comment` class and map it as a version number using the `<version>` tag. First, let's look at the changes to the `Comment` class with the mapping attributes:

```
[Class(Table="COMMENTS")]
public class Comment {
    ...
    private int version;
    ...
    [Version(Column="VERSION")]
    public int Version {
        get { return version; }
        set { version = value; }
    }
}
```

You can also use a public scope for the setter and getter methods. When using XML, the `<version>` property mapping must come immediately after the identifier property mapping in the mapping file for the `Comment` class:

```
<class name="Comment" table="COMMENTS">
    <id ... >
    </id>
    <version name="Version" column="VERSION" />
    ...
</class>
```

The version number is just a counter value—it doesn't have any useful semantic value. Some people prefer to use a timestamp instead:

```
[Class(Table="COMMENTS")]
public class Comment {
    ...
    private DateTime lastUpdatedDatetime;
    ...
    [Timestamp(Column="LAST_UPDATED")]
    public DateTime LastUpdatedDatetime {
        get { return lastUpdatedDatetime; }
        set { lastUpdatedDatetime = value; }
    }
}
```

In theory, a timestamp is slightly less safe, since two concurrent transactions might both load and update the same item all in the same millisecond; in practice, this is unlikely to occur. However, we recommend that new projects use a numeric version and not a timestamp.

You don't need to set the value of the version or timestamp property yourself; NHibernate will initialize the value when you first save a `Comment`, and increment or reset it whenever the object is modified.

FAQ

Is the version of the parent updated if a child is modified? For example, if a single bid in the collection `bids` of an `Item` is modified, is the version number of the `Item` also increased by one or not? The answer to that and similar questions is simple: NHibernate will increment the version number whenever an object is dirty. This includes all dirty properties, whether they're single-valued or collections. Think about the relationship between `Item` and `Bid`: If a `Bid` is modified, the version of the related `Item` isn't incremented. If we add or remove a `Bid` from the collection of bids, the version of the `Item` will be updated. (Of course, we would make `Bid` an immutable class, since it doesn't make sense to modify bids.)

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Whenever NHibernate updates a comment, it uses the version column in the SQL **WHERE** clause:

```
update COMMENTS set COMMENT_TEXT='New comment text', VERSION=3
where COMMENT_ID=123 and VERSION=2
```

If another transaction had updated the same item since it was read by the current transaction, the **VERSION** column would not contain the value 2, and the row would not be updated. NHibernate would check the row count returned by the ADO.NET driver—which in this case would be the number of rows updated, zero—and throw a **StaleObjectStateException**.

Using this exception, we might show the user of the second transaction an error message (“You have been working with stale data because another user modified it!”) and let the *first commit win*. Alternatively, we could catch the exception, close the current session, and show the second user a new screen, allowing the user to manually *merge changes* between the two versions (using a new session).

It is possible to disable the increment of the version when a specific property is dirty. To do so, you must add **optimistic-lock="false"** to this property’s mapping. This feature is available for properties, components and collections (the owning entity’s version is not incremented).

It is also possible to implement optimistic locking without a version by writing:

```
<class ... optimistic-lock="all">
```

In this case, NHibernate will compare the states of all fields to find if any of them as changed. This feature will only work for persistent objects; it can’t work for detached objects because NHibernate do not have their state when they were loaded.

You may also write **<class ... optimistic-lock="dirty">** if you want NHibernate to allow concurrent modifications as long as they aren’t done on the same columns. This will allow, for example, an administrator to change the name of a customer while another one changes its location.

It is possible to avoid the execution of unnecessary updates (that will trigger *on update* events even when no changes have been made to detached instances) by mapping your entities using **<class ... select-before-update="true">**. NHibernate will select these entities and issue update commands only for dirty instances. However, be aware of the performance implications of this feature.

As you can see, NHibernate makes it easy to use managed versioning to implement optimistic locking. Can you use optimistic locking and pessimistic locking together, or do you have to make a decision for one? And why is it called *optimistic*?

An optimistic approach always assumes that everything will be OK and that conflicting data modifications are rare. Instead of being pessimistic and blocking concurrent data access immediately (and forcing execution to be serialized), optimistic concurrency control will only block at the end of a unit of work and raise an error.

Both strategies have their place and uses, of course. Multi-user applications usually default to optimistic concurrency control and use pessimistic locks when appropriate. Note that the duration of a pessimistic lock in NHibernate is a single database transaction! This means you can’t use an exclusive lock to block concurrent access longer than a single database transaction. We consider this a good thing, because the only solution would be an extremely expensive lock held in memory (or a so called *lock table* in the database) for the duration of, for example, a conversation. This is almost always a performance bottleneck; every data access involves additional lock checks to a synchronized lock manager. You may, if absolutely required in your particular application, implement a simple long pessimistic lock yourself, using NHibernate to manage the lock table. Patterns for this can be found on the NHibernate website; however, we definitely don’t recommend this approach. You have to carefully examine the performance implications of this exceptional case.

Let’s get back to conversations. You now know the basics of managed versioning and optimistic locking. In previous chapters (and earlier in this chapter), we have talked about the NHibernate **ISession** as not being the same as a transaction. In fact, an **ISession** has a flexible scope, and you can use it in different ways with

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

database and conversations. This means that the *granularity* of an `ISession` is flexible; it can be any unit of work you want it to be.

5.2.2 Granularity of a Session

To understand how you can use the NHibernate `ISession`, let's consider its relationship with transactions. Previously, we have discussed two related concepts:

- The scope of object identity (see section 4.1.4)
- The granularity of database and conversations

The NHibernate `ISession` instance defines the scope of object identity. The NHibernate `ITransaction` instance matches the scope of a database transaction.

What is the relationship between an `ISession` and a conversation? Let's start this discussion with the most common usage of the `ISession`.

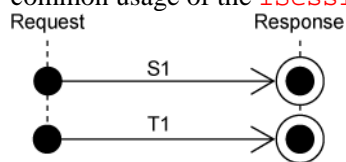


Figure 5.2 Using one to one `ISession` and `ITransaction` per request/response cycle

Usually, we open a new `ISession` for each client request (for example, a web browser request) and begin a new `ITransaction`. After executing the business logic, we commit the database transaction and close the `ISession`, before sending the response to the client (see figure 5.2).

The session (S1) and the database transaction (T1) therefore have the same granularity. If you're not working with the concept of conversation, this simple approach is all you need in your application. We also like to call this approach *session-per-request*.

If you need a long-running conversation, you might, thanks to detached objects (and NHibernate's support for optimistic locking as discussed in the previous section), implement it using the same approach (see figure 5.3).

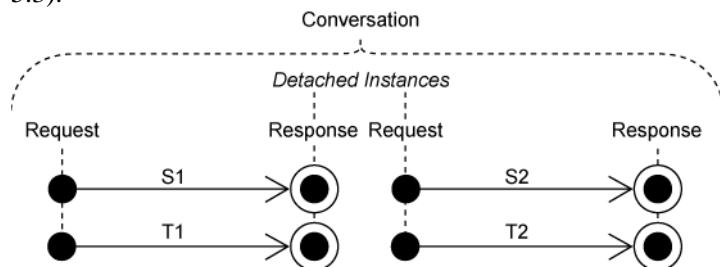


Figure 5.3 Implementing conversations with multiple `ISessions`, one for each request/response cycle

Suppose your conversation spans two client request/response cycles—for example, two HTTP requests in a web application. You could load the interesting objects in a first `ISession` and later reattach them to a new `ISession` after they've been modified by the user. NHibernate will automatically perform a version check. The time between (S1, T1) and (S2, T2) can be "long," as long as your user needs to make his changes. This approach is also known as *session-per-request-with-detached-objects*.

Alternatively, you might prefer to use a single `ISession` that spans multiple requests to implement your conversation. In this case, you don't need to worry about reattaching detached objects, since the objects remain persistent within the context of the one long-running `ISession` (see figure 5.4). Of course, NHibernate is still responsible for performing optimistic locking.

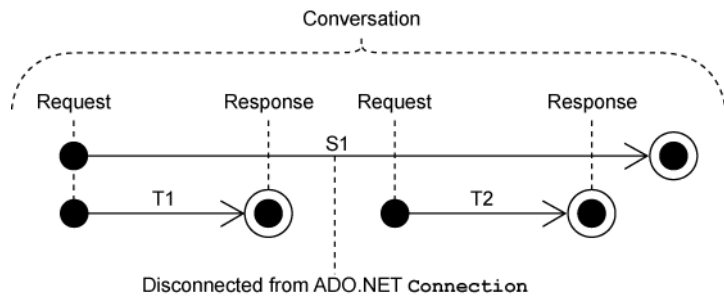


Figure 5.4 Implementing conversations with a long `ISession` using disconnection

A conversation keeps a reference to the session, though the session can be serialized, if really required. The underlying ADO.NET connection has to be closed, of course, and a new connection must be obtained on a subsequent request. This approach is known as *session-per-conversation* or *long session*.

Usually, your first choice should be to keep the NHibernate `ISession` open no longer than a single database transaction (session-per-request). Once the initial database transaction is complete, the longer the session remains open, the greater the chance that it holds stale data in its cache of persistent objects (the session is the mandatory first-level cache). Certainly, you should never reuse a single session for longer than it takes to complete a single conversation.

The question of conversations and the scope of the `ISession` is a matter of application design. We discuss implementation strategies with examples in chapter 10, section 10.2, "Implementing conversations."

Finally, there is an important issue you might be concerned about. If you work with a legacy database schema, you probably can't add version or timestamp columns for NHibernate's optimistic locking.

5.2.3 Other ways to implement optimistic locking

If you don't have version or timestamp columns, NHibernate can still perform optimistic locking, but only for objects that are retrieved and modified in the same `ISession`. If you need optimistic locking for detached objects, you *must* use a version number or timestamp.

This alternative implementation of optimistic locking checks the current database state against the unmodified values of persistent properties at the time the object was retrieved (or the last time the session was flushed). You can enable this functionality by setting the `optimistic-lock` attribute on the class mapping:

```
<class name="Comment" table="COMMENT" optimistic-lock="all">
  <id .....

```

Now, NHibernate will include all properties in the `WHERE` clause:

```
update COMMENTS set COMMENT_TEXT='New text'
where COMMENT_ID=123
and COMMENT_TEXT='Old Text'
and RATING=5
and ITEM_ID=3
and FROM_USER_ID=45
```

Alternatively, NHibernate will include only the modified properties (only `COMMENT_TEXT`, in this example) if you set `optimistic-lock="dirty"`. (Note that this setting also requires you to set the class mapping to `dynamic-update="true"`.)

We don't recommend this approach; it's slower, more complex, and less reliable than version numbers and doesn't work if your conversation spans multiple sessions (which is the case if you're using detached objects).

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note that, when using `optimistic-lock="dirty"`, two concurrent conversations can update the same row as long as they change different columns; and the result can be disastrous for the end-user. So, if you have to use this feature, do it cautiously.

We'll now again switch perspective and consider a new aspect of NHibernate. We already mentioned the close relationship between transactions and caching in the introduction of this chapter. The fundamentals of transactions and locking, and also the session granularity concepts, are of central importance when we consider caching data in the application tier.

5.3 Caching theory and practice

A major justification for our claim that applications using an object/relational persistence layer are expected to outperform applications built using direct ADO.NET is the potential for caching. Although we'll argue passionately that most applications should be designed so that it's possible to achieve acceptable performance *without* the use of a cache, there is no doubt that for some kinds of applications—especially read-mostly applications or applications that keep significant metadata in the database—caching can have an enormous impact on performance.

We start our exploration of caching with some background information. This includes an explanation of the different caching and identity scopes and the impact of caching on transaction isolation. This information and these rules can be applied to caching in general; they aren't only valid for NHibernate applications. This discussion gives you the background to understand why the NHibernate caching system is like it is. We'll then introduce the NHibernate caching system and show you how to enable, tune, and manage the first- and second-level NHibernate cache. We recommend that you carefully study the fundamentals laid out in this section before you start using the cache. Without the basics, you might quickly run into hard-to-debug concurrency problems and risk the integrity of your data.

A cache keeps a representation of current database state close to the application, either in memory or on disk of the server machine. Therefore, the cache is essentially merely a local copy of the data. It sits between your application and the database. The great benefit is that your application can save time by not going to the database every time it needs data. This can be very advantageous when it come to reducing the strain on a busy database server, and ensuring that data is served to the application quickly. The cache may be used to avoid a database hit whenever

- The application performs a lookup by identifier (primary key)
- The persistence layer resolves an association lazily

It's also possible to cache the results of entire queries, so if the same query is issued repeatedly, the entire results are immediately available. As you'll see in chapter 8, this feature is less often used, but the performance gain of caching query results can be impressive in some situations.

Before we look at how NHibernate's cache works, let's walk through the different caching options and see how they're related to identity and concurrency.

5.3.1 Caching strategies and scopes

Caching is such a fundamental concept in object/relational persistence that you can't understand the performance, scalability, or transactional semantics of an ORM implementation without first knowing what caching strategies it uses. There are three main types of cache:

- *Transaction scope*—Attached to the current unit of work, which may be an actual database transaction or a conversation. It's valid and used as long as the unit of work runs. Every unit of work has its own cache.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

- *Process scope*—Shared among many (possibly concurrent) units of work or transactions. This means that data in the process scope cache is accessed by concurrently running transactions, obviously with implications on transaction isolation. A process scope cache might store the persistent instances themselves in the cache, or it might store just their persistent state in some disassembled format.
- *Cluster scope*—Shared among multiple processes on the same machine or among multiple machines in a cluster. It requires some kind of *remote process communication* to maintain consistency. Caching information has to be replicated to all nodes in the cluster. For many (not all) applications, cluster scope caching is of dubious value, since reading and updating the cache might be only marginally faster than going straight to the database. There are many parameters to take into account, so a number of tests and tunings may be required to decide.

Persistence layers might provide multiple levels of caching. For example, a *cache miss* (a cache lookup for an item that isn't contained in the cache) at the transaction scope might be followed by a lookup at the process scope. If that fails, going to the database for the data might be the last resort.

The type of cache used by a persistence layer affects the scope of object identity (the relationship between .NET object identity and database identity).

Caching and object identity

Consider a transaction scope cache. It makes sense if this cache is also used as the identity scope of persistent objects. So, if during a transaction the application attempts to retrieve the same object twice, the transaction scope cache ensures both lookups return the same .NET instance. A transaction scope cache is a good fit for persistence mechanisms that provide transaction-scoped object identity.

In the case of the process scope cache, objects retrieved might be returned *by value*. Instead of storing and returning instances, the cache contains tuples of data. Each unit of work will first retrieve a copy of the state from the cache (a tuple) and then use that to construct its own persistent instance in memory. Unlike the transaction scope cache discussed previously, the scope of the cache and the scope of object identity are no longer the same.

A cluster scope cache always requires remote communication as it is likely to operate over several machines. In the case of POCO-oriented persistence solutions like NHibernate, objects are always passed remotely by value. Therefore, the cluster scope cache handles identity in the same way as the process scope cache; they each store copies of data, and pass that data to the application so they can create their own instances from it. In NHibernate terms, they are both second-level caches, the main difference being that a cluster scope cache can be distributed across several computers if needed.

Let's discuss which scenarios benefit from second-level caching; when to turn on the process (or cluster) scope second-level cache. Note that the first-level transaction scope cache is always on, and is mandatory. So, the decisions to be made are whether to use the second-level cache or not, and to select what type to use, and what data it should be used for.

Caching and transaction isolation

A process or cluster scope cache makes data retrieved from the database in one unit of work visible to another unit of work. Essentially, the cache is allowing cached data to be shared amongst different units of work, multiple threads or even computers. This may have some very nasty side-effects upon transaction isolation. We now discuss some critical considerations when choosing to use a process or cluster scoped cache.

Firstly, if more than one application is updating the database then process scope caching should not be used, or used only for data which changes rarely and may be safely refreshed by a cache expiry. This type of data occurs frequently in content management-type applications, but rarely in financial applications.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

If you are designing your application to scale out over several machines, you will want to build your application to support clustered operation. A process scope cache doesn't maintain consistency between the different caches on different machines in the cluster. To achieve this you should use a cluster scope (distributed) cache instead of the process scope cache.

Many .NET applications share access to their databases with other legacy applications. In this case, you shouldn't use any kind of cache beyond the mandatory transaction scope cache. There is no way for a cache system to know when the legacy application updated the shared data. Actually, it's *possible* to implement application-level functionality to trigger an invalidation of the process (or cluster) scope cache when changes are made to the database, but we don't know of any standard or best way to achieve this. Certainly, it will never be a built-in feature of NHibernate. If you implement such a solution, you'll most likely be on your own, because it's extremely specific to the environment and products used.

After considering non-exclusive data access, you should establish what isolation level is required for the application data. Not every cache implementation respects all transaction isolation levels, and it's critical to find out what is required. Let's look at data that benefits most from a process (or cluster) scoped cache.

A full ORM solution will let you configure second-level caching separately for each class. Good candidate classes for caching are classes that represent

- Data that changes rarely
- Non-critical data (for example, content-management data)
- Data that is local to the application and not shared

Bad candidates for second-level caching are

- Data that is updated often
- Financial data
- Data that is shared with a legacy application

However, these aren't the only rules we usually apply. Many applications have a number of classes with the following properties:

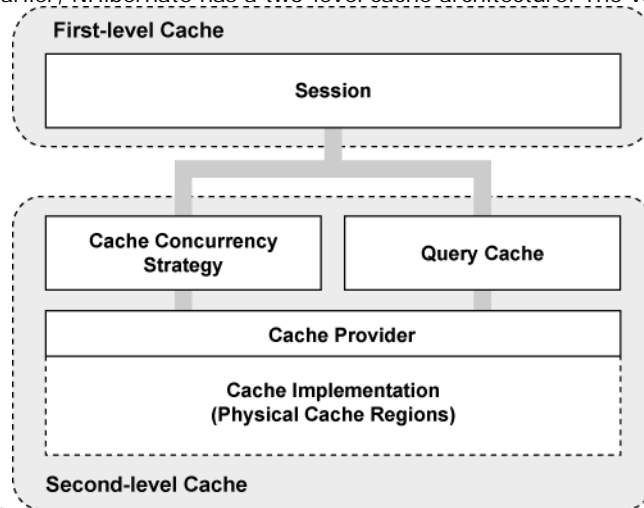
- A small number of instances
- Each instance referenced by many instances of another class or classes
- Instances rarely (or never) updated

This kind of data is sometimes called *reference data*. Reference data is an excellent candidate for caching with a process or cluster scope, and any application that uses reference data heavily will benefit greatly if that data is cached. You allow the data to be refreshed when the cache timeout period expires.

We've shaped a picture of a dual layer caching system in the previous sections, with a transaction scope first-level and an optional second-level process or cluster scope cache. This is close to the NHibernate caching system.

5.3.2 The NHibernate cache architecture

As we said earlier, NHibernate has a two-level cache architecture. The various elements of this system can be seen



in figure 5.5.

Figure 5.5 NHibernate's two-level cache architecture

The first-level cache is the `ISession` itself. A session lifespan corresponds to either a database transaction or a conversation (as explained earlier in this chapter). We consider the cache associated with the `ISession` to be a transaction scope cache. The first-level cache is mandatory and can't be turned off; it also guarantees object identity inside a transaction.

The second-level cache in NHibernate is pluggable and might be scoped to the process or cluster. This is a cache of state (returned by value), not of persistent instances. A cache concurrency strategy defines the transaction isolation details for a particular item of data, whereas the cache provider represents the physical, actual cache implementation. Use of the second-level cache is optional and can be configured on a per-class and per-association basis.

NHibernate also implements a cache for query result sets that integrates closely with the second-level cache. This is an optional feature. We discuss the query cache in chapter 8, since its usage is closely tied to the actual query being executed.

Let's start with using the first-level cache, also called the session cache.

Using the first-level cache

The session cache ensures that when the application requests the same persistent object twice in a particular session, it gets back the same (identical) .NET instance. This sometimes helps avoid unnecessary database traffic. More importantly, it ensures the following:

- The persistence layer isn't vulnerable to stack overflows in the case of circular references in a graph of objects.
- There can never be conflicting representations of the same database row at the end of a database transaction. There is at most a single object representing any database row. All changes made to that object may be safely written to the database (flushed).
- Changes made in a particular unit of work are always immediately visible to all other code executed inside that unit of work.

You don't have to do anything special to enable the session cache. It's always on and, for the reasons shown, can't be turned off.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Whenever you pass an object to `Save()`, `Update()`, or `SaveOrUpdate()`, and whenever you retrieve an object using `Load()`, `Find()`, `List()`, `Iterate()`, or `Filter()`, that object is added to the session cache. When `Flush()` is subsequently called, the state of that object will be synchronized with the database.

If you don't want this synchronization to occur, or if you're processing a huge number of objects and need to manage memory efficiently, you can use the `Evict()` method of the `ISession` to remove the object and its collections from the first-level cache. There are several scenarios where this can be useful.

Managing the first-level cache

Consider this frequently asked question: "I get an `OutOfMemoryException` when I try to load 100,000 objects and manipulate all of them. How can I do mass updates with NHibernate?"

It's our view that ORM isn't suitable for mass update (or mass delete) operations. If you have a use case like this, a different strategy is almost always better: call a stored procedure in the database or use direct SQL `UPDATE` and `DELETE` statements for that particular use case. Don't transfer all the data to main memory for a simple operation if it can be performed more efficiently by the database. If your application is *mostly* mass operation use cases, ORM isn't the right tool for the job!

If you insist on using NHibernate even for mass operations, you can immediately `Evict()` each object after it has been processed (while iterating through a query result), and thus prevent memory exhaustion.

To completely evict all objects from the session cache, call `Session.Clear()`. We aren't trying to convince you that evicting objects from the first-level cache is a bad thing in general, but that good use cases are rare. Sometimes, using projection and a report query, as discussed in chapter 8, section 8.4.5, "Improving performance with report queries," might be a better solution.

Note that eviction, like save or delete operations, can be automatically applied to associated objects. NHibernate will evict associated instances from the `ISession` if the mapping attribute `cascade` is set to `all` or `all-delete-orphan` for a particular association.

When a first-level cache miss occurs, NHibernate tries again with the second-level cache if it's enabled for a particular class or association.

The NHibernate second-level cache

The NHibernate second-level cache has process or cluster scope; all sessions share the same second-level cache. The second-level cache actually has the scope of an `ISessionFactory`.

Persistent instances are stored in the second-level cache in a *disassembled* form. Think of disassembly as a process a bit like serialization (the algorithm is much, much faster than .NET serialization, however).

The internal implementation of this process/cluster scope cache isn't of much interest; more important is the correct usage of the *cache policies*—that is, caching strategies and physical cache providers.

Different kinds of data require different cache policies: the ratio of reads to writes varies, the size of the database tables varies, and some tables are shared with other external applications. So the second-level cache is configurable at the granularity of an individual class or collection role. This lets you, for example, enable the second-level cache for reference data classes and disable it for classes that represent financial records. The cache policy involves setting the following:

- Whether the second-level cache is enabled
- The NHibernate concurrency strategy
- The cache expiration policies (such as expiration or priority)

Not all classes benefit from caching, so it's extremely important to be able to disable the second-level cache. To repeat, the cache is usually useful only for read-mostly classes. If you have data that is updated more often than it's read, don't enable the second-level cache, even if all other conditions for caching are true!

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Furthermore, the second-level cache can be dangerous in systems that share the database with other writing applications. As we explained in earlier sections, you must exercise careful judgment here.

The NHibernate second-level cache is set up in two steps. First, you have to decide which *concurrency strategy* to use. After that, you configure cache expiration and cache attributes using the *cache provider*.

Built-in concurrency strategies

A concurrency strategy is a mediator; it's responsible for storing items of data in the cache and retrieving them from the cache. This is an important role, because it also defines the transaction isolation semantics for that particular item. You'll have to decide, for each persistent class, which cache concurrency strategy to use, if you want to enable the second-level cache.

There are three built-in concurrency strategies, representing decreasing levels of strictness in terms of transaction isolation:

- *read-write*—Maintains *read committed* isolation, using a timestamping mechanism. It's available only in non-clustered environments. Use this strategy for read-mostly data where it's critical to prevent stale data in concurrent transactions, in the rare case of an update.
- *nonstrict-read-write*—Makes no guarantee of consistency between the cache and the database. If there is a possibility of concurrent access to the same entity, you should configure a sufficiently short expiry timeout. Otherwise, you may read stale data in the cache. Use this strategy if data rarely changes (many hours, days or even a week) and a small likelihood of stale data isn't of critical concern. NHibernate invalidates the cached element if a modified object is flushed, but this is an asynchronous operation, without any cache locking or guarantee that the retrieved data is the latest version.
- *read-only*—A concurrency strategy suitable for data which never changes. Use it for reference data only.

Note that with decreasing strictness comes increasing performance. You have to carefully evaluate the performance of a clustered cache with full transaction isolation before using it in production. In many cases, you might be better off disabling the second-level cache for a particular class if stale data isn't an option. First benchmark your application with the second-level cache disabled. Then enable it for good candidate classes, one at a time, while continuously testing the performance of your system and evaluating concurrency strategies.

It's possible to define your own concurrency strategy by implementing `NHibernate.Cache.ICacheConcurrencyStrategy`, but this is a relatively difficult task and only appropriate for extremely rare cases of optimization.

Your next step after considering the concurrency strategies you'll use for your cache candidate classes is to pick a *cache provider*. The provider is a plug-in, the physical implementation of a cache system.

Choosing a cache provider

For now, NHibernate forces you to choose a single cache provider for the whole application. The following providers are released with NHibernate:

- *Hashtable* is not intended for production use. It only caches in memory and can be set using its provider: `NHibernate.Cache.HashtableCacheProvider` available inside `NHibernate.dll`
- *SysCache* relies on `System.Web.Caching.Cache` for the underlying implementation so you can refer to the documentation of the ASP.NET caching feature to understand how it works. Its NHibernate provider is the class `NHibernate.Caches.SysCache.SysCacheProvider` in the library `NHibernate.Caches.SysCache.dll`. This provider should only be used with ASP.NET Web Applications.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

- *Prevalence* makes it possible to use the underlying Bamboo.Prevalence implementation as cache provider. Its NHibernate provider is `NHibernate.Caches.Prevalence.PrevalenceCacheProvider` in the library `NHibernate.Caches.Prevalence.dll`. You can also visit Bamboo.Prevalence's website: <http://bbooprevalence.sourceforge.net/>

There are also some distributed cache providers that you will learn about in the next section. And it's easy to write an adaptor for other products by implementing `NHibernate.Cache.ICacheProvider`.

Every cache provider supports NHibernate Query Cache and is compatible with every concurrency strategy (*read-only*, *nonstrict-read-write*, and *read-write*).

Setting up caching therefore involves two steps:

Look at the mapping files for your persistent classes and decide which cache concurrency strategy you'd like to use for each class and each association.

13. Enable your preferred cache provider in the NHibernate configuration and customize the provider-specific settings.

Let's add caching to our CaveatEmptor `Category` and `Item` classes.

5.3.3 Caching in practice

Remember that you don't have to explicitly enable the first-level cache. So, let's declare caching policies and set up cache providers for the second-level cache in our CaveatEmptor application.

The `Category` has a small number of instances and is updated rarely, and instances are shared among many users, so it's a great candidate for use of the second-level cache. We start by adding the mapping element required to tell NHibernate to cache `Category` instances:

```
[Class(Table="CATEGORY")]
public class Category {
    [Cache(-1, Usage=CacheUsage.ReadWrite)]
    [Id .... ]
    public long Id { ... }
}
```

Note that, like the attribute `[Discriminator]`, you can put `[Cache]` on any field/property; just be careful when mixing it with other attributes (here, we use the position -1 as it must come before).

Here is the corresponding XML mapping:

```
<class
  name="Category"
  table="CATEGORY">
  <cache usage="read-write"/>
  <id .... >
</class>
```

The `usage="read-write"` attribute tells NHibernate to use a read-write concurrency strategy for the `Category` cache. NHibernate will now try the second-level cache whenever we navigate to a `Category` or when we load a `Category` by identifier.

We have chosen `read-write` instead of `nonstrict-read-write`, since `Category` is a highly concurrent class, shared among many concurrent transactions, and it's clear that a read-committed isolation level is good enough. However, `nonstrict-read-write` would probably be an acceptable alternative, since a small probability of inconsistency between the cache and database is acceptable (the category hierarchy has little financial significance).

This mapping was enough to tell NHibernate to cache all simple `Category` property values but not the state of associated entities or collections. Collections require their own `<cache>` element. For the `Items` collection, we'll use a `read-write` concurrency strategy:

```
<class
  name="Category"
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        table="CATEGORY">
        <cache usage="read-write"/>
        <id ....
        <set name="Items" lazy="true">
            <cache usage="read-write"/>
            <key ....
        </set>
    </class>

```

This cache will be used when enumerating the collection `category.Items`, for example. Note that deleting an item that exists on a collection in the second level cache will cause an exception; so, make sure that you remove the item from the collection in the cache before deleting it.

Now, a collection cache holds only the identifiers of the associated item instances. So, if we require the instances themselves to be cached, we must enable caching of the `Item` class. A read-write strategy is especially appropriate here. Our users don't want to make decisions (placing a `Bid`) based on possibly stale data. Let's go a step further and consider the collection of `Bids`. A particular `Bid` in the `Bids` collection is immutable, but we have to map the collection using `read-write`, since new bids may be made at any time (and it's critical that we be immediately aware of new bids):

```

<class
    name="Item"
    table="ITEM">
    <cache usage="read-write"/>
    <id ....
    <set name="Bids" lazy="true">
        <cache usage="read-write"/>
        <key ....
    </set>
</class>

```

To the immutable `Bid` class, we apply a read-only strategy:

```

<class
    name="Bid"
    table="BID">
    <cache usage="read-only"/>
    <id ....
</class>

```

Cached `Bid` data is valid indefinitely, because bids are never updated. No cache invalidation is required. (Instances may be evicted by the cache provider—for example, if the maximum number of objects in the cache is reached.)

`User` is an example of a class that could be cached with the `nonstrict-read-write` strategy, but we aren't certain that it makes sense to cache users at all.

Let's set the cache provider, expiration policies, and physical properties of our cache. We use *cache regions* to configure class and collection caching individually.

Understanding cache regions

NHibernate keeps different classes/collections in different cache *regions*. A region is a named cache: a handle by which you can reference classes and collections in the cache provider configuration and set the expiration policies applicable to that region.

The name of the region is the class name, in the case of a class cache; or the class name together with the property name, in the case of a collection cache. `Category` instances are cached in a region named `NHibernate.Auction.Category`, and the `items` collection is cached in a region named `NHibernate.Auction.Category.Items`.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

You can use the NHibernate configuration property `hibernate.cache.region_prefix` to specify a root region name for a particular `ISessionFactory`. For example, if the prefix was set to `Node1`, `Category` would be cached in a region named `Node1.NHibernate.Auction.Category`. This setting is useful if your application includes multiple `ISessionFactory` instances.

Now that you know about cache regions, let's configure the expiry policies for the `Category` cache. First we'll choose a cache provider.

Setting up a local cache provider

We need to set the property that selects a cache provider:

```
<add
    key="hibernate.cache.provider_class"
    value="NHibernate.Caches.SysCache.SysCacheProvider, NHibernate.Caches.SysCache"
/>
```

Here, we've chosen `SysCache` as our second-level cache.

Now, we need to specify the expiry policies for the cache regions. `SysCache` provides two parameters: an *expiration* value which is the number of seconds to wait before expiring each item (the default value is 300 seconds) and a *priority* value which is a numeric cost of expiring each item, where 1 is a low cost, 5 is the highest, and 3 is normal. Note that only values 1 through 5 are valid and they refer to the `System.Web.Caching.CacheItemPriority` enumeration.

`SysCache` has a configuration file section handler to allow configuring different expirations and priorities for different regions. Here's how we might configure the `Category` class:

```
<?xml version="1.0" ?>
<configuration>
  <configSections>
    <section
      name="syscache"
      type="NHibernate.Caches.SysCache.SysCacheSectionHandler, NHibernate.Caches.SysCache"
    />
  </configSections>
  <syscache>
    <cache region="Category" expiration="36000" priority="5" />
  </syscache>
</configuration>
```

There are a small number of categories, and they're all shared among many concurrent transactions. We therefore define a high expiration value (10 hours) and give it a high priority so that they stay in the cache as long as possible.

`Bids`, on the other hand, are small and immutable, but there are many of them; so we must configure `SysCache` to carefully manage the cache memory consumption. We use both a low expiration value and a low priority:

```
<cache region="Bid" expiration="300" priority="1" />
```

The result is that cached bids are removed from the cache after 5 minutes or if the cache is full (as they have the lowest priority).

Optimal cache eviction policies are, as you can see, specific to the particular data and particular application. You must consider many external factors, including available memory on the application server machine, expected load on the database machine, network latency, existence of legacy applications, and so on. Some of these factors can't possibly be known at development time, so you'll often need to iteratively test the performance impact of different settings in the production environment or a simulation of it.

This is especially true in a more complex scenario, with a replicated cache deployed to a cluster of server machines.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Using a distributed cache

SysCache and Prevalence are excellent cache providers if your application is deployed on a single machine. However, enterprise applications supporting thousands of concurrent users might require more computing power, and scaling your application might be critical to the success of your project. NHibernate applications are naturally scalable—that is, NHibernate behaves the same whether it's deployed to a single machine or too many machines. The only feature of NHibernate that must be configured specifically for clustered operation is the second-level cache. With a few changes to our cache configuration, we're able to use a clustered caching system.

It isn't necessarily wrong to use a purely local (non-cluster-aware) cache provider in a cluster. Some data—especially immutable data, or data that can be refreshed by cache timeout—doesn't require clustered invalidation and may safely be cached locally, even in a clustered environment. We might be able to have each node in the cluster use a local instance of SysCache, and carefully choose sufficiently short `expiration` values.

However, if you require strict cache consistency in a clustered environment, you must use a more sophisticated cache provider. Some distributed cache providers are available for NHibernate. You can consider the followings:

- *MemCache* is released with NHibernate. It uses memcached, a distributed cache system available under Linux, so you can use it with Mono (or use *VMWare Memcached appliance* under Windows). For more details, visit: <http://www.danga.com/memcached/>. Its NHibernate provider is the class `NHibernate.Caches.MemCache.MemCacheProvider` in the library `NHibernate.Caches.MemCache.dll`.
- *NCache* is a commercial distributed cache provider. Its website is <http://www.alachisoft.com/ncache/>.

We will not dig into the details of these distributed cache providers. Distributed caching is a complex topic; we recommend you to read some articles on this topic and to test these providers.

Note that some distributed cache providers work only with some cache concurrency strategies. A nice trick can help us avoid checking our mapping files one by one: Instead of placing a `[Cache]` attribute in our entities or placing `<cache>` elements in our mapping files, we can centralize cache configuration in

`hibernate.cfg.xml`:

```
<hibernate-configuration>
  <session-factory>
    <class-cache
      class="NHibernate.Auction.Model.Bid, NHibernate.Auction"
      usage="read-only"/>
    <collection-cache
      collection="NHibernate.Auction.Model.Item.Bids"
      usage="read-write"/>
  </session-factory>
</hibernate-configuration>
```

We enabled read-only caching for `Bid` and read-write caching for the `Bids` collection in this example. However, there is one important caveat: at the time of this writing, NHibernate will run into a conflict if we also have `<cache>` elements in the mapping file for `Item`. We therefore can't use the global configuration to override the mapping file settings. We recommend that you use the centralized cache configuration from the start, especially if you aren't sure how your application might be deployed. It's also easier to tune cache settings with a centralized configuration.

There is an optional setting to consider. For cluster cache providers, it might be better to set the NHibernate configuration option `hibernate.cache.use_minimal_puts` to `true`. When this setting is enabled, NHibernate will only add an item to the cache after checking to ensure that the item isn't already

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

cached. This strategy performs better if cache writes (puts) are much more expensive than cache reads (gets). This is the case for a replicated cache in a cluster, but not for a local cache (the default is `false`, optimized for a local cache). Whether you're using a cluster or a local cache, you sometimes need to control it programmatically for testing or tuning purposes.

Controlling the second-level cache

NHibernate has some useful methods that will help you test and tune your cache. You may wonder how to disable the second-level cache completely. NHibernate will only load the cache provider and start using the second-level cache if you have any cache declarations in your mapping files or XML configuration file. If you comment them out, the cache is disabled. This is another good reason to prefer centralized cache configuration in `hibernate.cfg.xml`.

Just as the `ISession` provides methods for controlling the first-level cache programmatically, so does the `SessionFactory` for the second-level cache.

You can call `Evict()` to remove an element from the cache, by specifying the class and the object identifier value:

```
SessionFactory.Evict( typeof(Category), 123 );
```

You can also evict all elements of a certain class or only evict a particular collection role:

```
SessionFactory.Evict( typeof(Category) );
```

You'll rarely need these control mechanisms and you should use them with care as they don't respect any transaction isolation semantics of the usage strategy.

5.4 Summary

This chapter was dedicated to transactions (fine-grained and coars-grained), concurrency and data caching.

You learned that for a single unit of work, either all operations should be completely successful or the whole unit of work should fail (and changes made to persistent state should be rolled back). This led us to the notion of a transaction and the ACID attributes. A transaction is atomic, leaves data in a consistent state, and is isolated from concurrently running transactions, and you have the guarantee that data changed by a transaction is durable.

You use two transaction concepts in NHibernate applications: short database transactions and long-running conversations. Usually, you use read committed isolation for database transactions, together with optimistic concurrency control (version and timestamp checking) for long conversations. NHibernate greatly simplifies the implementation of conversations because it manages version numbers and timestamps for you.

Finally, we discussed the fundamentals of caching, and you learned how to use caching effectively in NHibernate applications.

NHibernate provides a dual-layer caching system with a first-level object cache (the `ISession`) and a pluggable second-level data cache. The first-level cache is always active—it's used to resolve circular references in your object graph and to optimize performance in a single unit of work. The second-level cache on the other hand is optional and works best for read-mostly candidate classes. You can configure a non-volatile second-level cache for reference (read-only) data or even a second-level cache with full transaction isolation for critical data. However, you have to carefully examine whether the performance gain is worth the effort. The second-level cache can be customized fine-grained, for each persistent class and even for each collection and class association. Used correctly and thoroughly tested, caching in NHibernate gives you a level of performance that is almost unachievable in a hand-coded data access layer.

Now that we've covered most of the fundamental aspects key to NHibernate applications, we can go to delve into some of the more advanced capabilities of NHibernate. The next chapter will start by discussing

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

some of the advanced NHibernate mapping concepts that will enable you to handle the most demanding persistence requirements.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

6

Advanced mapping concepts

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

In chapter 3, we introduced the most important ORM features provided by NHibernate including basic class and property mappings, inheritance mappings, component mappings, and one-to-many association mappings. We now extend on these topics by turning to the more exotic collection and association mappings that allow you to handle the trickier use cases. It's worth noting that these more exotic mappings should only be used with very careful consideration, in fact it's possible to implement *any* domain model using simpler component mappings (one-to-many associations and one-to-one associations). Throughout this chapter we'll advise you of when you may or may not want to use the advanced features as they're discussed.

Some of these features will require you to have a more in-depth understanding of NHibernate's type system, particularly of the distinction between entity and value types. So, that is where we'll begin.

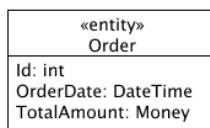
6.1 Understanding the NHibernate type system

We first distinguished between entity and value types back in chapter 3, section 3.6.1 - "Entity and value types". In order to give you a better understanding of the NHibernate type system, we'll elaborate on this a little more.

Entities are the coarse-grained classes in a system. You usually define the features of a system in terms of the entities involved: "the user places a bid for an item" is a typical feature definition that mentions three entities - user, bid and item. In contrast, value types are the much more fine grained classes in a system, such as strings, numbers, dates and monetary amounts. These fine grained classes can be used in many places and serve many purposes; the value type string can store email address, usernames and many other things. Strings are simple value types, but it is possible (but less common) to create value types that are more complex. For example, a value type could contain several fields, like an Address.

So how do we differentiate between value types and entities? From a more formal standpoint, we can say an *entity* is any class whose instances have their own persistent identity, and a *value type* is a class whose instances do not. The entity instances may therefore be in any of the three persistent lifecycle states: transient, detached, or persistent. However, we don't consider these lifecycle states to apply to the simpler value type instances. Furthermore, because entities have their own lifecycle, the `Save()` and `Delete()` methods of the NHibernate `ISession` interface will apply to them, but never to value type instances. To illustrate, let's consider Figure 6.1.1.

Figure 6.1.1 – An order entity with TotalAmount value type



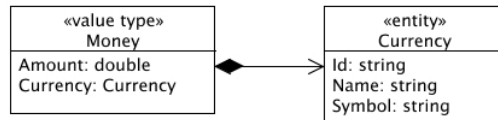
TotalAmount is an instance of *value type* Money. Because value types are completely bound to that their owning entities, TotalAmount is only saved when the Order is saved.

Associations and Value Types

As we said, not all value types are simple. It's possible for value types to also define associations. For example, our Money value type could have a property called Currency that is an association to a Currency entity as shown in figure 6.1.2

Figure 6.1.2 – The Money value type with association to a Currency entity.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>



If your value types have associations, they must *always* point *to* entities. The reason is that, if those associations could point *from* entities *to* value types, a value type could potentially belong to several entities, which isn't desirable. This is one of the great things about value types; if you update a value type instance, you know that it only affects the entity that *owns* it. For example, changing the `TotalAmount` of one `Order` simply cannot accidentally affect others. So far we've talked about value types and entities from an object oriented perspective. To build a more complete picture, we shall now take a look at how the relational model sees value types and entities, and how NHibernate bridges the gap.

Bridging from objects to database

You may be aware that a database architect would see the world of value types and entities slightly differently to this object oriented view of things. In the database, *tables* represent the entities, and columns represent the values. Even join tables and lookup tables are entities. So, if all tables represent entities in the database, does that mean we have to map all tables to entities in our .NET domain model? What about those value types we wanted in our model? NHibernate provides constructs for dealing with this. For example, a many-to-many association mapping hides the intermediate association table from the application, so we don't end up with an unwanted entity in our domain model. Similarly, a collection of value typed strings behaves like a value type from the point of view of the .NET domain model even though it's mapped to its own table in the database.

These features have their uses and can often simplify your C# code. However, over time we have become suspicious of them; these "hidden" entities often end up needing exposure in our applications as business requirements evolve. The many-to-many association table, for example, often has additional columns added as the application matures, so the relationship itself becomes an entity. You might not go far wrong if you make every database-level entity be exposed to the application as an entity class. For example, we'd be inclined to model the many-to-many association as two one-to-many associations to an intervening entity class. We'll leave the final decision to you, and return to the topic of many-to-many entity associations later in this chapter.

Mapping types

So far we've discussed the differences between value types and entities, as seen from the object oriented and relational database perspectives. We know that mapping entities is quite straight forward – entity classes are simply always mapped to database tables using `<class>`, `<subclass>`, and `<joined-subclass>` mapping elements.

Value types need something more, which is where *mapping types* enter the picture. Consider this mapping of the `CaveatEmptor` `User` and email address:

```

<property
  name="Email"
  column="EMAIL"
  type="String"/>
  
```

In ORM, you have to worry about both .NET types *and* SQL data types. In the example above imagine that the `Email` field is a .NET `string`, and `EMAIL` column is an SQL `varchar`. We want to tell NHibernate know how to carry out this conversion, which is where NHibernate mapping types come in. In this case, we've specified the mapping type `"String"`, which we know is appropriate for this particular conversion.

The `String` mapping type isn't the only one built into NHibernate; NHibernate comes with various mapping types that define default persistence strategies for primitive .NET types and certain classes, such as like `DateTime`.

6.1.1 Built-in mapping types

NHibernate's built-in mapping types usually reflect the name of the .NET type they map. Sometimes you'll have a choice of mapping types available to map a particular .NET type to the database. However, the built-in mapping types aren't designed to perform arbitrary conversions, such as mapping a `VARCHAR` field value to a .NET `Int32` property value. If you want this kind of functionality, you will have to define your own *custom value types*. We will get to that topic a little later in this chapter.

We'll now discuss the basic types; date and time, objects, large objects, and various other built-in mapping types and show you what .NET and `System.Data.DbType` data types they handle. `DbTypes` are used to infer the data provider types (hence SQL data types).

.NET primitive mapping types

The basic mapping types in table 7.1 map .NET primitive types to appropriate `DbTypes`.

Table 7.1 Primitive types

Mapping type	.NET type	System.Data.DbType
Int16	System.Int16	DbType.Int16
Int32	System.Int32	DbType.Int32
Int64	System.Int64	DbType.Int64
Single	System.Single	DbType.Single
Double	System.Double	DbType.Double
Decimal	System.Decimal	DbType.Decimal
Byte	System.Byte	DbType.Byte
Char	System.Char	DbType.StringFixedLength - 1 character
AnsiChar	System.Char	DbType.AnsiStringFixedLength - 1 character
Boolean	System.Boolean	DbType.Boolean
Guid	System.Guid	DbType.Guid
PersistentEnum	System.Enum (an enumeration)	The DbType for the underlying value
TrueFalse	System.Boolean	DbType.AnsiStringFixedLength - either 'T' or 'F'
YesNo	System.Boolean	DbType.AnsiStringFixedLength - either 'Y' or 'N'

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

You've probably noticed that your database doesn't support some of the DbTypes listed in table 6.2. However, ADO.NET provides a partial abstraction of vendor-specific SQL data types, allowing NHibernate to work with ANSI-standard types when executing data manipulation language (DML). For database-specific DDL generation, NHibernate translates from the ANSI-standard type to an appropriate vendor-specific type, using the built-in support for specific SQL dialects. (You usually don't have to worry about SQL data types if you're using NHibernate for data access and data schema definition.)

NHibernate supports a number of mapping types coming from Hibernate for compatibility (useful for those coming over from Hibernate or using Hibernate tools to generate hbm.xml files). Table 6.2 lists the additional names of NHibernate mapping types.

Table 6.2 Additional names of NHibernate mapping types

Mapping type	Additional name
Binary	binary
Boolean	boolean
Byte	byte
Character	character
CultureInfo	locale
DateTime	datetime
Decimal	big_decimal
Double	double
Guid	guid
Int16	short
Int32	int
Int32	integer
Int64	long
Single	float
String	string
TrueFalse	true_false
Type	class
YesNo	yes_no

From this table, you can see that writing `type="integer"` or `type="int"` is identical to `type="Int32"`. Note that this table contains many mapping types that will be discussed in the following sections.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Date/time mapping types

Table 6.3 lists NHibernate types associated with dates, times, and timestamps. In your domain model, you may choose to represent date and time data using either `System.DateTime` or `System.TimeSpan`. As they have different purposes, the choice should be easy.

Table 6.3 Date and time typesNHIAMEAP07.doc

Mapping type	.NET type	System.Data.DbType
DateTime	System.DateTime	DbType.DateTime - ignores the milliseconds
Ticks	System.DateTime	DbType.Int64
TimeSpan	System.TimeSpan	DbType.Int64
Timestamp	System.DateTime	DbType.DateTime - as specific as database supports

Object mapping types

All .NET types in tables 6.1 and 6.3 are value types (i.e. derived from `System.ValueType`). This means that they can't be null; unless you use the .NET 2.0 `Nullable<T>` structure or the Nullables add-in, as discussed in the next section. Table 6.4 lists NHibernate types for handling .NET types derived from `System.Object` (which can store null values).

Table 6.4 Nullable object typesNHIAMEAP07.doc

Mapping type	.NET type	System.Data.DbType
String	System.String	DbType.String
AnsiString	System.String	DbType.AnsiString

This table is completed by tables 6.5 and 6.6 which also contain nullable mapping types.

Large object mapping types

Table 6.5 lists NHibernate types for handling binary data and large objects. Note that none of these types may be used as the type of an identifier property.

Table 6.5 Binary and large object typesNHIAMEAP07.doc

Mapping type	.NET type	System.Data.DbType
Binary	System.Byte[]	DbType.Binary
BinaryBlob	System.Byte[]	DbType.Binary
StringClob	System.String	DbType.String

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Serializable	Any System.Object marked with SerializableAttribute	DbType.Binary
--------------	---	---------------

BinaryBlob and StringClob are mainly supported by SQL Server. They can have a very large size and are fully loaded in memory. This can be a performance killer if used to store very large objects. So use this feature carefully. Note that you must set the NHibernate property "prepare_sql" to "true" to enable this feature.

You can find up-to-date design patterns and tips for large object usage on the NHibernate website.

Various CLR mapping types

Table 6.6 lists NHibernate types for various other types of the CLR that may be represented as DbType.Strings in the database.

Table 6.6 Other CLR-related typesNHIAMEAP07.doc

Mapping type	.NET type	System.Data.DbType
CultureInfo	System.Globalization.CultureInfo	DbType.String - 5 chars for culture
Type	System.Type	DbType.String holding Assembly Qualified Name

Certainly, <property> isn't the only NHibernate mapping element that has a type attribute.

6.1.2 Using mapping types

All of the basic mapping types may appear almost anywhere in the NHibernate mapping document, on normal property, identifier property, and other mapping elements.

The <id>, <property>, <version>, <discriminator>, <index>, and <element> elements all define an attribute named type. (There are certain limitations on which mapping basic types may function as an identifier or discriminator type, however.)

You can see how useful the built-in mapping types are in this mapping for the BillingDetails class:

```

<class name="BillingDetails"
      table="BILLING_DETAILS"
      lazy="false"
      discriminator-value="0">

  <id name="Id" type="Int32" column="BILLING_DETAILS_ID">
    <generator class="native"/>
  </id>

  <discriminator type="Char" column="TYPE"/>

  <property name="Number" type="String"/>
  ...
</class>

```

The `BillingDetails` class is mapped as an entity. Its discriminator, id, and Number properties are value typed, and we use the built-in NHibernate mapping types to specify the conversion strategy.

It's often not necessary to explicitly specify a built-in mapping type in the XML mapping document. For instance, if you have a property of .NET type `System.String`, NHibernate will discover this using reflection and select `String` by default. We can easily simplify the previous mapping example:

```

<class name="BillingDetails"
      table="BILLING_DETAILS"
      lazy="false"
      discriminator-value="0">

  <id name="Id" column="BILLING_DETAILS_ID">
    <generator class="native"/>
  </id>

  <discriminator type="Char" column="TYPE"/>

  <property name="Number"/>
  ....
</class>

```

For each of the built-in mapping types, a constant is defined by the class `NHibernate.NHibernateUtil`. For example, `NHibernate.String` represents the `String` mapping type. These constants are useful for query parameter binding, as discussed in more detail in chapter 8:

```

session.CreateQuery("from Item i where i.Description like :desc")
  .SetParameter("desc", desc, NHibernate.String)
  .List();

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

These constants are also useful for programmatic manipulation of the NHibernate mapping meta-model, as discussed in chapter 3.

Of course, NHibernate isn't limited to the built-in mapping types; you can create your own custom mapping types for handling certain scenarios. We'll take a look this next, and explain how the mapping type system is a central to NHibernate's flexibility.

Creating custom mapping types

Object-oriented languages like C# make it easy to define new types by writing new classes. Indeed, this is a fundamental part of the definition of object orientation. If you were limited to the predefined built-in NHibernate mapping types when declaring properties of persistent classes, you'd lose much of C#'s expressiveness. Furthermore, your domain model implementation would be tightly coupled to the physical data model, since new type conversions would be impossible.

In order to avoid that, NHibernate provides a very powerful feature called *custom mapping types*.

NHibernate provides two user-friendly interfaces that applications may use when defining new mapping types, the first being `NHibernate.UserTypes.IUserType`. `IUserType` is suitable for most simple cases and even for some more complex problems. Let's use it in a simple scenario.

Our `Bid` class defines an `Amount` property and our `Item` class defines an `InitialPrice` property, both monetary values. So far, we've only used a simple `System.Double` to represent the value, mapped with `Double` to a single `DbType.Double` column.

Suppose we wanted to support multiple currencies in our auction application and that we had to refactor the existing domain model for this change. One way to implement this change would be to add new properties to `Bid` and `Item`: `AmountCurrency` and `InitialPriceCurrency`. We would then map these new properties to additional `VARCHAR` columns with the built-in `String` mapping type. Imagine if we had currency stored in 100 places, this would be lots of changes. We hope you *never* use this approach!

Creating an implementation of IUserType

Instead, we should create a `MonetaryAmount` class that encapsulates both currency and amount. This is a class of the domain model and doesn't have any dependency on NHibernate interfaces:

```
[Serializable]
public class MonetaryAmount
{
    private readonly double value;
    private readonly string currency;

    public MonetaryAmount(double value, string currency)
    {
        this.value = value;
        this.currency = currency;
    }

    public double Value { get { return value; } }
    public string Currency { get { return currency; } }

    public override bool Equals(object obj) { ... }
    public override int GetHashCode() { ... }
}
```

We've also made life simpler by making `MonetaryAmount` an immutable class, meaning it can't be changed after it's instantiated. We would have to implement `Equals()` and `GetHashCode()` to complete the class - but there is nothing special to consider here aside that they must be consistent, and `GetHashCode()` should return mostly unique numbers.

We will use this new `MonetaryAmount` to replace the `Double`, as defined on the `InitialPrice` property for `Item`. We would benefit by using this new class in other places, such as the `Bid.Amount`.

The next challenge is in mapping our new `MonetaryAmount` properties to the database. Suppose we're working with a legacy database that contains all monetary amounts in USD. Our new class means our *application* code is no longer restricted to a single currency, but it will take time to get the changes done by the database team. Until this happens, we'd like to store just the `Amount` property of `MonetaryAmount` to the database. Because we can't store the currency yet, we'll convert all `Amounts` to USD before we save them, and from USD when we load them.

The first step to handling this is to tell NHibernate how to handle our `MonetaryAmount` type. To do this, we create a `MonetaryAmountUserType` class that implements the NHibernate interface `IUserType`. Our custom mapping type is shown in listing 6.1.

Listing 6.1 Custom mapping type for monetary amounts in USD

```
using System;
using System.Data;
using NHibernate.UserTypes;

public class MonetaryAmountUserType : IUserType {

    private static readonly NHibernate.SqlTypes.SqlType[] SQL_TYPES =
        { NHibernateUtil.Double.SqlType };

    public NHibernate.SqlTypes.SqlType[] SqlTypes {
        get { return SQL_TYPES; }
    }
    public Type ReturnedType { get { return typeof(MonetaryAmount); } }
    public new bool Equals( object x, object y ) {
        if ( object.ReferenceEquals(x,y) ) return true;
        if ( x == null || y == null ) return false;
        return x.Equals(y);
    }
    public object DeepCopy(object value) { return value; }
    public bool IsMutable { get { return false; } }
    public object NullSafeGet(IDataReader dr, string[] names, object owner){
        object obj = NHibernateUtil.Double.NullSafeGet(dr, names[0]);
        if ( obj==null ) return null;
        double valueInUSD = (double) obj;
        return new MonetaryAmount( valueInUSD, "USD");
    }
    public void NullSafeSet(IDbCommand cmd, object obj, int index) {
        if (obj == null) {
            ((IDataParameter)cmd.Parameters[index]).Value = DBNull.Value;
        } else {
            MonetaryAmount anyCurrency = (MonetaryAmount)obj;
            MonetaryAmount amountInUSD =
                MonetaryAmount.Convert( anyCurrency, "USD" );

            ((IDataParameter)cmd.Parameters[index]).Value = amountInUSD.Value;
        }
    }

    public static MonetaryAmount Convert( MonetaryAmount m,
                                         string targetCurrency)
    {
        ...
    }
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

The `SqlTypes` property tells NHibernate what SQL column types to use for DDL schema generation, as seen in #1. The types are subclasses of `NHibernate.SqlTypes.SqlType`. Notice that this property returns an array of types. An implementation of `IUserType` may map a single property to *multiple* columns, but our legacy data model only has a single `Double`.

In #2, we can see that `ReturnedType` tells NHibernate what .NET type is mapped by this `IUserType`.

The `IUserType` is responsible for dirty-checking property values (#3). The `Equals()` method compares the current property value to a previous snapshot and determines whether the property is dirty and must be saved to the database.

The `IUserType` is also partially responsible for creating the snapshot in the first place, as shown in #4. Since `MonetaryAmount` is an immutable class, the `DeepCopy()` method returns its argument. In the case of a mutable type, it would need to return a copy of the argument to be used as the snapshot value. This method is also called when an instance of the type is written to or read from the second-level cache.

NHibernate can make some minor performance optimizations for immutable types. The `IsMutable` (#5) property tells NHibernate that this type is immutable.

The `NullSafeGet()` method shown near #6 retrieves the property value from the ADO.NET `IDataReader`. You can also access the `owner` of the component if you need it for the conversion. All database values are in USD, so you have to convert the `MonetaryAmount` returned by this method before you show it to the user.

In #7, the `NullSafeSet()` method writes the property value to the ADO.NET `IDbCommand`. This method takes whatever currency is set and converts it to a simple `Double` USD value before saving.

Note that, for brevity, we haven't provided a `Convert` function as shown in #8. If we were to implement it, it would have code that converts between various currencies.

Mapping the `InitialPrice` property of `Item` can be done as follows:

```
<property name="InitialPrice"
          column="INITIAL_PRICE"
          type="NHibernate.Auction.CustomTypes.MonetaryAmountUserType,
          NHibernate.Auction"/>
```

This is the simplest kind of transformation that an implementation of `IUserType` could perform. It takes a Value Type class and maps it to a single database column. *Much* more sophisticated things are possible; a custom mapping type could perform validation, it could read and write data to and from an Active Directory, or it could even retrieve persistent objects from a different NHibernate `ISession` for a different database. You're limited mainly by your imagination and performance concerns!

In a perfect world, we'd prefer to represent both the amount *and* currency of our monetary amounts in the database, so we're not limited to storing just USD. We could still use an `IUserType` for this, but it's limited; if an `IUserType` is mapped with more than one property, we can't use them in our HQL or Criteria queries. The NHibernate query engine wouldn't know anything about the individual properties of `MonetaryAmount`. You still access the properties in your C# code (`MonetaryAmount` is just a regular class of the domain model, after all), but not in NHibernate queries.

To allow for a custom value type with multiple properties that can be accessed in queries, we should use the `ICompositeUserType` interface. This interface exposes the properties of our `MonetaryAmount` to NHibernate.

Creating an implementation of *ICompositeUserType*

To demonstrate the flexibility of custom mapping types, we won't have to change our `MonetaryAmount` domain model class at all—we change only the custom mapping type, as shown in listing 6.2.

Listing 6.2 Custom mapping type for monetary amounts in new database schemas

```
using System;
using System.Data;
using NHibernate.UserTypes;

public class MonetaryAmountCompositeUserType : ICompositeUserType {

    public Type ReturnedClass { get { return typeof(MonetaryAmount); } }
    public new bool Equals( object x, object y ) {
        if ( object.ReferenceEquals(x,y) ) return true;
        if ( x == null || y == null ) return false;
        return x.Equals(y);
    }
    public object DeepCopy(object value) { return value; }
    public bool IsMutable { get { return false; } }

    public object NullSafeGet(IDataReader dr, string[] names,
        NHibernate.Engine.ISessionImplementor session, object owner) {
        object obj0 = NHibernateUtil.Double.NullSafeGet(dr, names[0]);
        object obj1 = NHibernateUtil.String.NullSafeGet(dr, names[1]);
        if ( obj0==null || obj1==null ) return null;
        double value = (double) obj0;
        string currency = (string) obj1;
        return new MonetaryAmount(value, currency);
    }

    public void NullSafeSet(IDbCommand cmd, object obj, int index,
        NHibernate.Engine.ISessionImplementor session) {
        if (obj == null) {
            ((IDataParameter)cmd.Parameters[index]).Value = DBNull.Value;
            ((IDataParameter)cmd.Parameters[index+1]).Value = DBNull.Value;
        } else {
            MonetaryAmount amount = (MonetaryAmount)obj;
            ((IDataParameter)cmd.Parameters[index]).Value = amount.Value;
            ((IDataParameter)cmd.Parameters[index+1]).Value = amount.Currency;
        }
    }

    public string[] PropertyNames {
        get { return new string[] { "Value", "Currency" }; }
    }
    public NHibernate.Type.IType[] PropertyTypes {
        get { return new NHibernate.Type.IType[] {
            NHibernateUtil.Double, NHibernateUtil.String }; }
    }
    public object GetPropertyValue(object component, int property) {
        MonetaryAmount amount = (MonetaryAmount) component;
        if (property == 0)
            return amount.Value;
        else
            return amount.Currency;
    }
    public void SetPropertyValue(object comp, int property, object value) {
        throw new Exception("Immutable!");
    }
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

    public object Assemble(object cached,
        NHibernate.Engine.ISessionImplementor session, object owner) {
        return cached;
    }
    public object Disassemble(object value,
        NHibernate.Engine.ISessionImplementor session) {
        return value;
    }
}

```

#1 shows how an implementation of `ICompositeUserType` has its own properties, defined by `PropertyNames`.

Similarly, the properties each have their own type, as defined by `PropertyTypes` (#2).

The `GetPropertyValues()` method, shown in #3, returns the value of an individual property of the `MonetaryAmount`.

Since `MonetaryAmount` is immutable, we can't set property values individually (see #4) This isn't a problem because this method is optional anyway.

In #5, the `Assemble()` method is called when an instance of the type is read from the second-level cache.

The `Disassemble()` method is called when an instance of the type is written to the second-level cache, as shown in #6.

The order of properties must be the same in the `PropertyNames`, `PropertyTypes`, and `GetPropertyValues()` methods. The `InitialPrice` property now maps to two columns, so we declare both in the mapping file. The first column stores the value; the second stores the currency of the `MonetaryAmount`. Note that the *order* of columns must match the order of properties in your type implementation:

```

<property name="InitialPrice"
    type="NHibernate.Auction.CustomTypes.MonetaryAmountCompositeUserType,
    NHibernate.Auction">
    <column name="INITIAL_PRICE"/>
    <column name="INITIAL_PRICE_CURRENCY"/>
</property>

```

In a query, we can now refer to the `Amount` and `Currency` properties of the custom type, even though they don't appear anywhere in the mapping document as individual properties:

```

from Item i
where i.InitialPrice.Value > 100.0
    and i.InitialPrice.Currency = 'XAF'

```

In this example we've expanded the buffer between the .NET object model and the SQL database schema with our custom composite type. Both representations can now handle changes more robustly.

If implementing custom types seems complex, relax; you rarely need to use a custom mapping type. An alternative way to represent the `MonetaryAmount` class is to use a component mapping, as in section 4.4.2, "Using components." The decision to use a custom mapping type is often a matter of taste.

There are few more interfaces that can be used to implement custom types; they are introduced in the next section.

Other interfaces to create custom mapping types

You may find that the interfaces `IUserType` and `ICompositeUserType` do not allow you to easily add more features to your custom types. In this case, you will need to use some of the other interfaces which are in the `NHibernate.UserTypes` namespace:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The `IParameterizedType` interface allows you to supply parameters to your custom type in the mapping file. This interface has a unique method: `SetParameterValues(IDictionary parameters)` that will be called at the initialization of your type. Here is an example of mapping providing a parameter:

```
<property name="Price">
  <type name="NHibernate.Auction.CustomTypes.MonetaryAmountUserType">
    <param name="DefaultCurrency">Euro</param>
  </type>
</property>
```

This mapping tells the custom type to use Euro as currency if it isn't specified.

The `IEnhancedUserType` interface makes it possible to implement a custom type that can be marshalled to and from its string representation. This functionality allows this type to be used as identifier or discriminator type. To create a type that can be used as version, you must implement the `IUserVersionType` interface.

The `INullableUserType` interface allows you to interpret non-null values in a property as null in the database. When using `dynamic-insert` or `dynamic-update`, fields identified as null will not be inserted or updated. This information may also be used when generating the *where clause* of the SQL command when optimistic locking is enabled.

The last interface is different from the previous because it is meant to implement user defined collection types: `IUserCollectionType`. For more details, take a look at the implementation `NHibernate.Test.UserCollection.MyListType` in the source code of `NHibernate`.

Now, let's look at an extremely important application of custom mapping types. Nullable types are found in almost all enterprise applications.

Using Nullable types

With .NET 1.1, primitive types can not be null; but this is no longer the case in .NET 2.0. Let's say that we want to add a `DismissDate` to the class `User`. As long as a user is active, its `DismissDate` should be null. But the `System.DateTime` struct can not be null. And we don't want to use some "magic" value to represent the null state. With .NET 2.0 (and 3.5 of course), you can simply write:

```
public class User
{
  ...
  private DateTime? dismissDate;
  public DateTime? DismissDate
  {
    get { return dismissDate; }
    set { dismissDate = value; }
  }
  ...
}
```

We omit other properties and methods because we focus on the nullable property. And no change is required in the mapping.

If you work with .NET 1.1, the `Nullables` add-in (in the `NHibernateContrib` package for versions prior to `NHibernate 1.2.0`) contains a number of custom mapping types which allow primitive types to be null. For our previous case, we can use the `Nullables.NullableDateTime` class:

```
using Nullables;
[Class]
public class User {
  ...
  private NullableDateTime dismissDate;
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    [Property]
    public NullableDateTime DismissDate
    {
        get { return dismissDate; }
        set { dismissDate = value; }
    }
    ...
}

```

The mapping is quite straightforward:

```

<class name="Example.Person, Example">
    ...
    <property name="DateOfBirth"
        type="Nullables.NHibernate.NullableDateTimeType,
            Nullables.NHibernate" />
</class>

```

It is important to note that, in the mapping, the type of `DismissDate` must be `Nullables.NHibernate.NullableDateTimeType` (from the file `Nullables.NHibernate.dll`). This type is a wrapper used to translate `Nullables` types from/to database types. But if when using the `NHibernate.Mapping.Attributes` library, this operation is automatic, that's why we just had to put the attribute `[Property]`.

The `NullableDateTime` type behaves exactly like `System.DateTime`; there are even implicit operators to easily interact with it. The `Nullables` library contains nullable types for most .NET primitive types supported by `NHibernate`. You can find more details in `NHibernate` documentation.

Using enumerated types

An enumeration (enum) is a special form of value type, which inherits from `System.Enum` and supplies alternate names for the values of an underlying primitive type.

For example, the `Comment` class defines a `Rating`. If you recall, in our `CaveatEmptor` application, users are able to give other comments about other users. Instead of using a simple `int` property for the rating, we create an enumeration:

```

public enum Rating {
    Excellent,
    Ok,
    Low
}

```

We then use this type for the `Rating` property of our `Comment` class. In the database, ratings would be represented as the type of the underlying value. In this case (and by default), it is `Int32`. And that's all we have to do. We may specify `type="Rating"` in our mapping, but it is optional; `NHibernate` can use reflection to find this.

One problem you might run into is using enumerations in `NHibernate` queries. Consider the following query in HQL that retrieves all comments rated "Low":

```

IQuery q =
    session.CreateQuery("from Comment c where c.Rating = Rating.Low");

```

This query doesn't work, because `NHibernate` doesn't know what to do with `Rating.Low` and will try to use it as a literal. We have to use a `bind` parameter and set the rating value for the comparison dynamically (which is what we need for other reasons most of the time):

```

IQuery q =
    session.CreateQuery("from Comment c where c.Rating = :rating");
q.SetParameter("rating",
    Rating.Low,
    NHibernateUtil.Enum(typeof(Rating));

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The last line in this example uses the static helper method `NHibernateUtil.Enum()` to define the `NHibernate Type`, a simple way to tell `NHibernate` about our enumeration mapping and how to deal with the `Rating.Low` value.

We've now discussed all kinds of `NHibernate` mapping types: built-in mapping types, user-defined custom types, and even components (chapter 4). They're all considered value types, because they map objects of value type (not entities) to the database. With a good understanding of what value types are, and how they are mapped, we can now move on to the more complex issue of *collections* of value typed instances.

6.2 Mapping collections of value types

In chapter 4 we introduced using collections to represent entity relationships. We explained how, for example, an `Item` could have a collection of `Bids` in our `CaveatEmptor` application. Collections are not just limited to entity types, and so this section will focus on how you create mappings where collections store instances of a value type. We start this section by showing you how to use basic collections to contain simple value types, such as a list of `string` or `DateTimes`. We then move on to how you work with ordered and sorted collections. Finally, we discuss how you can map collections of *components*, along with the possible pitfalls and how they can be dealt with. You will see many hands-on code samples along the way (please note that all the mappings in this section work with both `.NET 1.1` collections and `.NET 2.0` generics).

6.2.1 Storing value types in Sets, bags, lists, and maps

Suppose that our sellers can attach images to `Items`. An image is accessible only via the containing item; it doesn't need to support associations to any other entity in our system. In this case, it's reasonable to model the image as a value type. `Item` would have a collection of images that `NHibernate` would consider to be part of the `Item`, and therefore without their own persistence lifecycle. In this particular example scenario, let's assume that images are stored as files on the file-system rather than BLOBs in the database, and we'll simply store filenames in the database to record what images each `Item` has. We'll now walk through various ways this can be implemented using `NHibernate`, starting with the simplest implementation – the `set`.

Using a set

The simplest implementation is an `ISet` of `string` filenames. As a reminder, `ISet` is a container that only disallows duplicate objects, and is available in the `Iesi.Collections` library. To store the images against the `Item` using an `ISet`, we add a collection property to the `Item` class as follows:

```
using Iesi.Collections.Generic;

private ISet<string> images = new HashSet<string>();

[Set(Lazy=true, Table="ITEM_IMAGE")]
[Key(1, Column="ITEM_ID")]
[Element(2, TypeType=typeof(string), Column="FILENAME", NotNull=true)]

public ISet<string> Images {
    get { return this.images; }
    set { this.images = value; }
}
```

Here is the corresponding XML mapping:

```
<set name="Images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <element type="String" column="FILENAME" not-null="true"/>
</set>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

In the database, image file names are stored in a table called `ITEM_IMAGE` and linked to their owner through `ITEM_ID`. From the database perspective, we have two entities. However, NHibernate is used to hide this fact so we can present `Images` as merely a *part* of `Item`. The `<key>` element declares the foreign key, `ITEM_ID` of the parent entity. The `<element>` tag declares this collection as a collection of value type instances: in this case, of strings.

As you may recall, a set can't contain duplicate elements, so the primary key of the `ITEM_IMAGE` table consists of both columns in the `<set>` declaration: `ITEM_ID` and `FILENAME`. See figure 6.1 for a table schema example.

ITEM		ITEM_IMAGE	
ITEM_ID	NAME	ITEM_ID	FILENAME
1	Foo	1	fooimage1.jpg
2	Bar	1	fooimage2.jpg
3	Baz	2	barimage1.jpg

Figure 6.1 Table structure and example data for a collection of strings

It doesn't seem likely that we would allow the user to attach the same image more than once, but suppose we did. What kind of mapping would be appropriate then?

Using a bag

An unordered collection that permits duplicate elements is called a *bag*. Curiously, the .NET framework doesn't define an `IBag` interface. NHibernate lets you use an `IList` in .NET to simulate bag behavior; this is consistent with common usage in the .NET community. To use a bag, change the type of `Images` in `Item` from `ISet` to `IList`, probably using `ArrayList` as an implementation.

Changing the table definition from the previous section to permit duplicate `FILENAMES` requires a different primary key. We use an `<idbag>` mapping to attach a surrogate key column to the collection table, much like the synthetic identifiers we use for entity classes:

```
[IdBag(Lazy=true, Table="ITEM_IMAGE")]
    [CollectionId(1, TypeType=typeof(int), Column="ITEM_IMAGE_ID")]
        [Generator(2, Class="sequence")]

    [Key(3, Column="ITEM_ID")]
    [Element(4, TypeType=typeof(string), Column="FILENAME", NotNull=true)]

public IList Images { ... }
```

The XML mapping looks like this:

```
<idbag name="Images" lazy="true" table="ITEM_IMAGE">
  <collection-id type="Int32" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </collection-id>

  <key column="ITEM_ID"/>
  <element type="String" column="FILENAME" not-null="true"/>
</idbag>
```

In this case, the primary key is the generated `ITEM_IMAGE_ID`. You can see a graphical view of the database tables in figure 6.2.

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_IMAGE_ID	ITEM_ID	FILENAME
1	Foo	1	1	fooimage1.jpg
2	Bar	2	1	fooimage1.jpg
3	Baz	3	2	barimage1.jpg

Figure 6.2 Table structure using a bag with a surrogate primary key

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

If you're wondering why we used `<idbag>` rather than `<bag>`, then bear in mind we'll be discussing bags very shortly. Before that, we'll discuss another common approach to storing our images - in an ordered list.

Using a list

A `<list>` mapping requires the addition of an *index column* to the database table. The index column defines the position of the element in the collection. Thus, NHibernate can preserve the ordering of the collection elements when retrieving the collection from the database if we map the collection as a `<list>`:

```
<list name="Images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <index column="POSITION"/>
  <element type="String" column="FILENAME" not-null="true"/>
</list>
```

The primary key consists of the `ITEM_ID` and `POSITION` columns. Notice that duplicate elements (`FILENAME`) are allowed, which is consistent with the semantics of a list. (We don't have to change the `Item` class; the types we used earlier for the bag are the same.)

Note that, even though the `IList` contract doesn't specify that a list is an ordered collection; NHibernate's implementation preserves the ordering when persisting the collection.

If the collection is `[fooimage1.jpg, fooimage1.jpg, fooimage2.jpg]`, the `POSITION` column contains the values 0, 1, and 2, as shown in figure 6.3.

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	POSITION	FILENAME
1	Foo	1	0	fooimage1.jpg
2	Bar	1	1	fooimage1.jpg
3	Baz	1	2	fooimage2.jpg

Figure 6.3 Tables for a list with positional elements

Alternatively, we could use a `.NET` array instead of a list. NHibernate supports this usage; indeed, the details of an array mapping are virtually identical to those of a list. However, we very strongly recommend against the use of arrays, since arrays can't be lazily initialized (there is no way to proxy an array at the CLR level). Here is the mapping:

```
<primitive-array name="Images" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <index column="POSITION"/>
  <element type="String" column="FILENAME" not-null="true"/>
</primitive-array>
```

Now, suppose that our images have user-entered names in addition to the filenames. One way to model this in `.NET` would be to use a `Map`, with names as keys and filenames as values.

Using a map

Mapping a `<map>` (pardon us) is similar to mapping a list:

```
<map name="Images" lazy="true" table="ITEM_IMAGE">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="String" column="FILENAME" not-null="true"/>
</map>
```

The primary key consists of the `ITEM_ID` and `IMAGE_NAME` columns. The `IMAGE_NAME` column stores the keys of the map. Again, duplicate elements are allowed; see figure 6.4 for a graphical view of the tables.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

ITEM		ITEM_IMAGE		
ITEM_ID	NAME	ITEM_ID	IMAGE_NAME	FILENAME
1	Foo	1	Foo Image 1	fooimage1.jpg
2	Bar	1	Foo Image One	fooimage1.jpg
3	Baz	1	Foo Image 2	fooimage2.jpg

Figure 6.4 Tables for a map, using strings as indexes and elements

This Map is unordered. What if we want to always sort our map by the name of the image?

Sorted and ordered collections

In a startling abuse of the English language, the words *sorted* and *ordered* mean different things when it comes to NHibernate persistent collections. A *sorted collection* is sorted in memory using a .NET IComparer. An *ordered collection* is ordered at the database level using an SQL query with an order by clause.

Let's make our map of images a sorted map. This is a simple change to the mapping document:

```
<map name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  sort="natural">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="String" column="FILENAME" not-null="true"/>
</map>
```

By specifying `sort="natural"`, we tell NHibernate to use a `SortedMap`, sorting the image names according to the `CompareTo()` method of `System.String`. If you want some other sorted order—for example, reverse alphabetical order—you can specify the name of a class that implements `System.Collections.IComparer` in the `sort` attribute. For example:

```
<map name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  sort="NHibernate.Auction.ReverseStringComparer, NHibernate.Auction">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="string"/>
  <element type="String" column="FILENAME" not-null="true"/>
</map>
```

NHibernate sorted map uses `System.Collections.SortedList` in its implementation. A sorted set, which behaves like `Iesi.Collections.SortedSet`, is mapped in a similar way:

```
<set name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  sort="natural">
  <key column="ITEM_ID"/>
  <element type="String" column="FILENAME" not-null="true"/>
</set>
```

Bags can't be sorted, and there is no `SortedBag`, unfortunately. Nor may lists be sorted, because the order of list elements is defined by the list index.

Alternatively, you might choose to use an ordered map, using the sorting capabilities of the database instead of in-memory sorting:

```
<map name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  order-by="IMAGE_NAME asc">
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

    <key column="ITEM_ID"/>
    <index column="IMAGE_NAME" type="String"/>
    <element type="String" column="FILENAME" not-null="true"/>
  </map>

```

The expression in the `order-by` attribute is a fragment of an SQL order by clause. In this case, we order by the `IMAGE_NAME` column, in ascending order. You can even write SQL function calls in the `order-by` attribute:

```

<map name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  order-by="lower(FILENAME) asc">
  <key column="ITEM_ID"/>
  <index column="IMAGE_NAME" type="String"/>
  <element type="String" column="FILENAME" not-null="true"/>
</map>

```

Notice that you can order by any column of the collection table. Both sets and bags accept the `order-by` attribute; but again, lists don't. This example uses a bag:

```

<idbag name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  order-by="ITEM_IMAGE_ID desc">
  <collection-id type="Int32" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </collection-id>
  <key column="ITEM_ID"/>
  <element type="String" column="FILENAME" not-null="true"/>
</idbag>

```

Under the covers, NHibernate uses an `Iesi.Collections.ListSet` and a `System.Collections.Specialized.ListDictionary` to implement ordered sets and maps, so this functionality should be carefully used as it doesn't perform well with large numbers of elements.

In a real system, it's likely that we'd need to keep more than just the image name and filename; we'd probably need to create an `Image` class to store this extra information. Of course, we could map `Image` as an entity class; but since we've already concluded that this isn't absolutely necessary, let's see how much further we can get without an `Image` entity, which would require an association mapping and more complex lifecycle handling.

In chapter 3, you saw that NHibernate lets you map user-defined classes as components, which are considered to be value types. This is still true even when component instances are collection elements. Let's now look at how we can map collections of components.

7.2.2 Collections of components

Our `Image` class defines the properties `Name`, `Filename`, `SizeX`, and `SizeY`. It has a single association, with its parent `Item` class, as shown in figure 6.5.

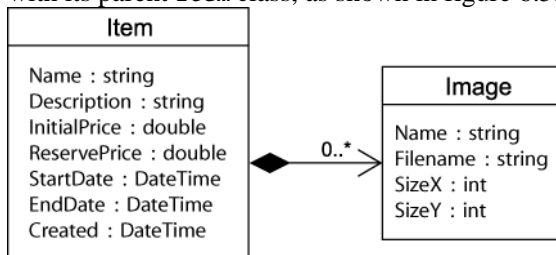


Figure 6.5 Collection of `Image` components in `Item`

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

As you can see from the aggregation association style depicted by a black diamond, `Image` is a component of `Item`, and `Item` is the entity that is responsible for the lifecycle of `Image`. References to images aren't shared, so our first choice is a NHibernate component mapping. The multiplicity of the association further declares this association as many-valued—that is, zero or more `Images` for the same `Item`.

Writing the component class

First, we implement the `Image` class. This is just a POCO, with nothing special to consider. As you know from chapter 4, component classes don't have an identifier property. However, we must implement `Equals()` and `GetHashCode()` to compare the `Name`, `Filename`, `SizeX`, and `SizeY` properties. This allows NHibernate's dirty checking to function correctly. Strictly speaking, implementing `Equals()` and `GetHashCode()` isn't required for all component classes. However, we recommend it for any component class because the implementation is fairly easy and "better safe than sorry" is a good motto.

The `Item` class hasn't actually changed, but the objects in the collection are now `Images` instead of `Strings`. Let's map this to the database.

Mapping the collection

Collections of components are mapped similarly to other collections of value type instances. The only difference is the use of `<composite-element>` in place of the familiar `<element>` tag. An ordered set of images could be mapped like this:

```
<set name="Images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
  <key column="ITEM_ID"/>
  <composite-element class="Namespaces.Image, Assembly">
    <property name="Name" column="IMAGE_NAME" not-null="true"/>
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX" not-null="true"/>
    <property name="SizeY" column="SIZEY" not-null="true"/>
  </composite-element>
</set>
```

This is a `set` of value type instances, so NHibernate must be able to tell instances apart despite the fact they have no separate primary key column. To do this, all columns of the composite are used together to determine if an item is unique: `ITEM_ID`, `IMAGE_NAME`, `FILENAME`, `SIZEX`, and `SIZEY`. Since these columns will all appear in a composite primary key, they cannot be null. This is clearly a disadvantage of this particular mapping. Composite elements in a `set` are sometimes useful, but using a `list`, `bag`, `map` or `idbag` will allow you to get around the not-null restriction.

Bidirectional navigation

So far, the association from `Item` to `Image` is unidirectional. If we decided to make it bidirectional, we would give our `Image` class a property named `Item` that is a reference back to the owning `Item`. In the mapping file, we'd need to add a `<parent>` tag to the mapping:

```
<set name="Images"
    lazy="true"
    table="ITEM_IMAGE"
    order-by="IMAGE_NAME asc">
  <key column="ITEM_ID"/>
  <composite-element class="Image">
    <parent name="Item"/>
    <property name="Name" column="IMAGE_NAME" not-null="true"/>
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX" not-null="true"/>
  </composite-element>
</set>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        <property name="SizeY" column="SIZEY" not-null="true"/>
    </composite-element>
</set>

```

This leads to a potential problem; you'll be able to load `Image` instances by querying for them, but the reference to their parent property will be `null`. The best thing to do is always load the parent in order to access its component parts, or use a full parent/child entity association, as described in chapter 4.

We still have the issue of having to declare all properties as `not-null`, and it would be nice if we could avoid this. Let's now look at how we can use a better primary key for the `IMAGE` table.

Avoiding not-null columns

If a set of `Images` isn't the only solution, other more flexible collection styles are possible. For example, an `<idbag>` offers a surrogate collection key:

```

<idbag name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  order-by="IMAGE_NAME asc">
  <collection-id type="Int32" column="ITEM_IMAGE_ID">
    <generator class="sequence"/>
  </collection-id>
  <key column="ITEM_ID"/>
  <composite-element class="Namespaces.Image, Assembly">
    <property name="Name" column="IMAGE_NAME"/>
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX"/>
    <property name="SizeY" column="SIZEY"/>
  </composite-element>
</idbag>

```

This time, the primary key is the `ITEM_IMAGE_ID` column. NHibernate now doesn't require that we must implement `Equals()` and `GetHashCode()`, nor do we need to declare the properties with `not-null="true"`. They may be nullable in the case of an `idbag`, as shown in figure 6.6.

ITEM_IMAGE

ITEM_IMAGE_ID	ITEM_ID	IMAGE_NAME	FILENAME
1	1	Foo Image 1	fooimage1.jpg
2	1	Foo Image 1	fooimage1.jpg
3	2	Bar Image 1	barimage1.jpg

Figure 6.6 Collection of `Image` components using a bag with a surrogate key

We should point out that there isn't a great deal of difference between this bag mapping and a standard parent/child entity relationship. The tables are identical, and even the C# code is extremely similar; the choice is mainly a matter of taste. Of course, a parent/child relationship supports shared references to the child entity and true bidirectional navigation.

We could even remove the `Name` property from the `Image` class and again use the image name as the key of a map:

```

<map name="Images"
  lazy="true"
  table="ITEM_IMAGE"
  order-by="IMAGE_NAME asc">
  <key column="ITEM_ID"/>
  <index type="String" column="IMAGE_NAME"/>
  <composite-element class="Image">
    <property name="Filename" column="FILENAME" not-null="true"/>
    <property name="SizeX" column="SIZEX"/>

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        <property name="SizeY" column="SIZEY"/>
    </composite-element>
</map>

```

As before, the primary key is composed of `ITEM_ID` and `IMAGE_NAME`.

A composite element class like `Image` isn't limited to simple properties of basic type like `filename`. It may contain components, using the `<nested-composite-element>` declaration, and even `<many-to-one>` associations to entities. It may not own collections, however. A composite element with a many-to-one association is useful, and we'll come back to this kind of mapping later in this chapter.

We're finally finished with value types, and hopefully you have an in-depth view of what is possible and where you can be able to make use of them. The next thing we'll look at is advanced entity association mapping techniques. The simple parent/child association we mapped in chapter 3 is just one of many possible association mapping styles. Like the previous mappings we discussed, most of these mappings are considered "exotic" and will only need to be used in special cases. However, having an awareness of the available techniques will certainly help you solve the thornier mapping challenges you encounter in the wild.

6.3 Mapping entity associations

When we use the word *associations*, we're always referring to relationships between entities. In chapter 4, we demonstrated a unidirectional many-to-one association, made it bidirectional, and finally turned it into a parent/child relationship (one-to-many and many-to-one).

One-to-many associations are the most important and popular type you'll find. In fact, we go so far as to discourage the use of more exotic association styles when a simple bidirectional many-to-one/one-to-many will do the job. In particular, a many-to-many association may always be represented as two many-to-one associations to an intervening class. This model is usually much more extensible, and we'll rarely use a many-to-many mapping in our applications.

Armed with this disclaimer, let's investigate NHibernate's rich association mappings starting with one-to-one associations.

6.3.1 One-to-one associations

We argued in chapter 4 that the relationships between `User` and `Address` were best represented using `<component>` mappings. If you recall, the user has both a `BillingAddress` and a `HomeAddress` in our sample model. Component mappings are usually the simplest way to represent one-to-one relationships, since the lifecycle of one class is almost always dependent on the lifecycle of the other class, and the association is a composition.

But what if we want a dedicated table for `Address`, and to map both `User` and `Address` as entities? In this case, the classes have a true one-to-one association. Because an `Address` is an entity, we'd start by creating the following mapping:

```

<class name="Address" table="ADDRESS" lazy="false">
    <id name="Id" column="ADDRESS_ID">
        <generator class="native"/>
    </id>
    <property name="Street"/>
    <property name="City"/>
    <property name="Zipcode"/>
</class>

```

Note that `Address` now requires an identifier property; it's no longer a component class. There are two different ways to represent a one-to-one association to this `Address` in NHibernate. The first approach adds a foreign key column to the `USER` table.

Using a foreign key association

The easiest way to represent the association from `User` to its `BillingAddress` is to use a `<many-to-one>` mapping with a unique constraint on the foreign key. This may surprise you, since *many* doesn't seem to be a good description of either end of a one-to-one association! However, from NHibernate's point of view, there isn't much difference between the two kinds of foreign key associations. So, we add a foreign key column named `BILLING_ADDRESS_ID` to the `USER` table and map it as follows:

```
<many-to-one name="BillingAddress"
  class="Address"
  column="BILLING_ADDRESS_ID"
  cascade="save-update"/>
```

Note that we've chosen `save-update` as the cascade style. This means the `Address` will become persistent when we create an association from a persistent `User`. The `cascade="all"` cascade would also make sense for this association, since deletion of the `User` should result in deletion of the `Address`.

Our database schema still allows duplicate values in the `BILLING_ADDRESS_ID` column of the `USER` table, so two users could have a reference to the same address. To make this association truly one-to-one, we add `unique="true"` to the `<many-to-one>` element, constraining the relational model so that there can be only one address per user:

```
<many-to-one name="BillingAddress"
  class="Address"
  column="BILLING_ADDRESS_ID"
  cascade="all"
  unique="true"/>
```

This change adds a unique constraint to the `BILLING_ADDRESS_ID` column in the DDL generated by NHibernate—resulting in the table structure illustrated by figure 6.7

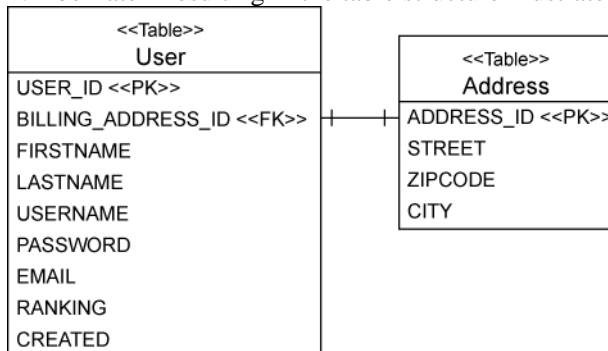


Figure 6.7 A one-to-one association with an extra foreign key column

But what if we want this association to be navigable from `Address` to `User` in .NET? To achieve this, we add a property named `User` that points to the `Address` class, and map it like so in our `Address` mapping:

```
<one-to-one name="User"
  class="User"
  property-ref="BillingAddress"/>
```

This tells NHibernate that the `User` association in `Address` is the reverse direction of the `BillingAddress` association in `User`.

In code, we create the association between the two objects as follows:

```
Address address = new Address();
address.Street = "73 Nowhere Street";
address.City = "Pretoria";
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

address.Zipcode = "1923";
using( session.BeginTransaction() ) {
    User user = (User) session.Get(typeof(User), userId);
    address.User = user;
    user.BillingAddress = address;
    session.Transaction.Commit();
}

```

To finish the mapping, we also have to map the `HomeAddress` property of `User`. This is easy enough: we add another `<many-to-one>` element to the `User` metadata, mapping a new foreign key column, `HOME_ADDRESS_ID`:

```

<many-to-one name="HomeAddress"
    class="Address"
    column="HOME_ADDRESS_ID"
    cascade="save-update"
    unique="true"/>

```

The `USER` table now defines two foreign keys referencing the primary key of the `ADDRESS` table: `HOME_ADDRESS_ID` and `BILLING_ADDRESS_ID`.

Unfortunately, we can't make both the `BillingAddress` and `HomeAddress` associations bidirectional, since we don't know if a particular address is a billing address or a home address. More specifically, we'd have to somehow dynamically decide which property name—`BillingAddress` or `HomeAddress`—to use for the `property-ref` attribute in the mapping of the user property. We *could* try making `Address` an abstract class with subclasses `HomeAddress` and `BillingAddress` and mapping the associations to the subclasses. This approach would work, but it's complex and probably not sensible in this case.

Our advice is to avoid defining more than one one-to-one association between any two classes. If you must, leave the associations unidirectional. If you don't have more than one—if there really is exactly one instance of `Address` per `User`—there is an alternative approach to the one we've just shown. Instead of defining a foreign key column in the `USER` table, you can use a *primary key association*.

Using a primary key association

Two tables related by a primary key association share the same primary key values. The primary key of one table is also a foreign key of the other. The main difficulty with this approach is ensuring that associated instances are assigned the same primary key value when the objects are saved. Before we try to solve this problem, let's see how we would map the primary key association.

For a primary key association, *both* ends of the association are mapped using the `<one-to-one>` declaration. This also means that we can no longer map both the billing and home address, only one property. Each row in the `USER` table has a corresponding row in the `ADDRESS` table. Two addresses would require an additional table, and this mapping style therefore wouldn't be adequate. Let's call this single address property `Address` and map it with the `User`:

```

<one-to-one name="Address"
    class="Address"
    cascade="save-update"/>

```

Next, here's the `User` of `Address`:

```

<one-to-one name="User"
    class="User"
    constrained="true"/>

```

The most interesting thing here is the use of `constrained="true"`. It tells NHibernate that there is a foreign key constraint on the primary key of `ADDRESS` that refers to the primary key of `USER`.

Now we must ensure that newly saved instances of `Address` are assigned the same identifier value as their `User`. We use a special NHibernate identifier-generation strategy called `foreign`:

```

<class name="Address" table="ADDRESS" lazy="false">
    <id name="Id" column="ADDRESS_ID">

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    <generator class="foreign">
      <param name="property">User</param>
    </generator>
  </id>
  ...
  <one-to-one name="User"
    class="User"
    constrained="true"/>
</class>

```

The `<param>` named `property` of the `foreign` generator allows us to name a one-to-one association of the `Address` class—in this case, the `user` association. The `foreign` generator inspects the associated object (the `User`) and uses its identifier as the identifier of the new `Address`. Look at the table structure in figure 6.8.

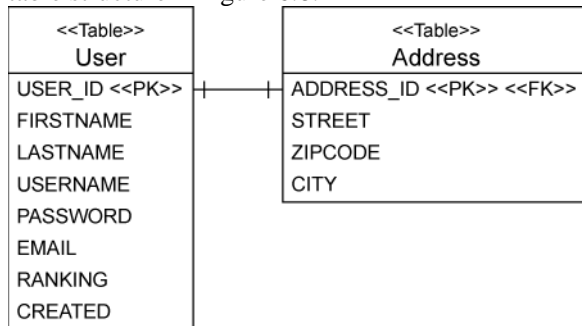


Figure 6.8 The tables for a one-to-one association with shared primary key values

The code to create the object association is unchanged for a primary key association; it's the same code we used earlier for the many-to-one mapping style.

There is now just one remaining entity association multiplicity we haven't discussed: many-to-many.

6.3.2 Many-to-many associations

The association between `Category` and `Item` is a many-to-many association, as you can see in figure 6.9.

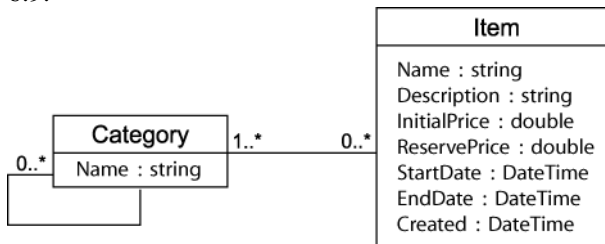


Figure 6.9 A many-to-many valued association between `Category` and `Item`

As we've explained earlier in this section, we avoid the use of many-to-many associations as there is almost always other information that must be attached to the links between associated instances, and the best way to represent this information is via an intermediate *association class*. Nevertheless, it's the purpose of this section to implement a real many-to-many entity association. Let's start with a unidirectional example.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

A unidirectional many-to-many association

If you only require unidirectional navigation, the mapping is straightforward. Unidirectional many-to-many associations are no more difficult than the collections of value type instances we covered previously. For example, if the `Category` has a set of `Items`, we can use this mapping:

```
<set name="Items"
    table="CATEGORY_ITEM"
    lazy="true"
    cascade="save-update">
  <key column="CATEGORY_ID"/>
  <many-to-many class="Item" column="ITEM_ID"/>
</set>
```

Just like a collection of value type instances, a many-to-many association has its own table, the *link table* or *association table*. In this case, the link table has two columns: the foreign keys of the `CATEGORY` and `ITEM` tables. The primary key is composed of both columns. The full table structure is shown in figure 6.10.

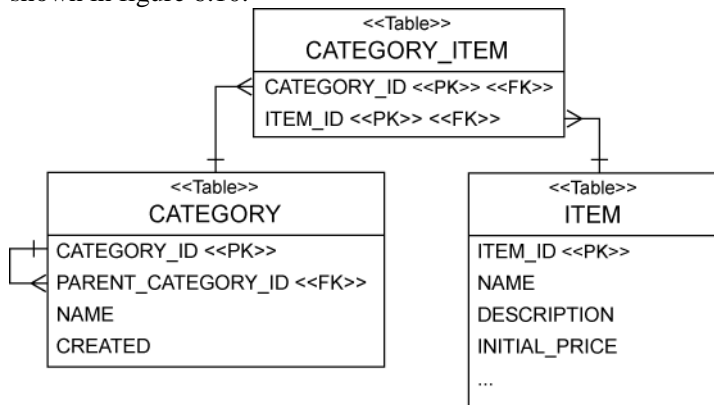


Figure 6.10 Many-to-many entity association mapped to an association table

We can also use a bag with a separate primary key column:

```
<idbag name="Items"
    table="CATEGORY_ITEM"
    lazy="true"
    cascade="save-update">
  <collection-id type="Int32" column="CATEGORY_ITEM_ID">
    <generator class="sequence"/>
  </collection-id>
  <key column="CATEGORY_ID"/>
  <many-to-many class="Item" column="ITEM_ID"/>
</idbag>
```

As usual with an `<idbag>` mapping, the primary key is a surrogate key column, `CATEGORY_ITEM_ID`, and duplicate links are therefore allowed (the same `Item` can be added to a particular `Category` twice).

Other variations we can even use are the indexed map or list collections. The following example uses a list:

```
<list name="Items"
    table="CATEGORY_ITEM"
    lazy="true"
    cascade="save-update">
  <key column="CATEGORY_ID"/>
  <index column="DISPLAY_POSITION"/>
  <many-to-many class="Item" column="ITEM_ID"/>
</list>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The primary key consists of the `CATEGORY_ID` and `DISPLAY_POSITION` columns. This mapping guarantees that every `Item` knows its position in the `Category`.

Creating an object association in .NET code is easy:

```
using( session.BeginTransaction() ) {
    Category cat = (Category) session.Get(typeof(Category), categoryId);
    Item item = (Item) session.Get(typeof(Item), itemId);

    cat.Items.Add(item);
    session.Transaction.Commit();
}
```

Bidirectional many-to-many associations are slightly more difficult.

A bidirectional many-to-many association

When we mapped a bidirectional one-to-many association in chapter 3, section 3.6, “Introducing associations”, we explained why one end of the association must be mapped with `inverse="true"`. Feel free to review that section, as it is relevant for bidirectional many-to-many associations too. In particular, each row of the link table is represented by *two* collection elements, one element at each end of the association. For example, we might create an `Item` class with a collection of `Category` instances, and a `Category` class with a collection of `Item` instances. When it comes to creating relationships in .NET code, it might look something like this:

```
cat.Items.Add(item);
item.Categories.Add(cat);
```

Regardless of multiplicity, a bidirectional association requires that you set both ends of the association.

When you map a bidirectional many-to-many association, you must declare one end of the association using `inverse="true"` to define which side’s state is used to update the link table. You can choose for yourself which end that should be.

Recall this mapping for the `Items` collection from the previous section:

```
<class name="Category" table="CATEGORY">
    ...
    <set name="Items"
        table="CATEGORY_ITEM"
        lazy="true"
        cascade="save-update">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </set>
</class>
```

We can reuse this mapping for the `Category` end of the bidirectional association. We map the `Item` end as follows:

```
<class name="Item" table="ITEM">
    ...
    <set name="Categories"
        table="CATEGORY_ITEM"
        lazy="true"
        inverse="true"
        cascade="save-update">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </set>
</class>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note the use of `inverse="true"`. Once again, this setting tells NHibernate to ignore changes made to the `categories` collection and use the other end of the association - the `items` collection - as the representation that should be synchronized with the database.

We've chosen `cascade="save-update"` for both ends of the collection, which suited our needs well. Note that `cascade="all"`, `cascade="delete"`, and `cascade="all-delete-orphans"` aren't meaningful for many-to-many associations, since an instance with potentially many parents shouldn't be deleted when just one parent is deleted.

Another thing to consider is what kinds of collections may be used for bidirectional many-to-many associations? Do you need to use the same type of collection at each end? It's reasonable to use, for example, a list at the end not marked `inverse="true"` (or explicitly set `false`) and a bag at the end that is marked `inverse="true"`.

You can use any of the mappings we've shown for unidirectional many-to-many associations for the noninverse end of the bidirectional association. `<set>`, `<idbag>`, `<list>`, and `<map>` are all possible, and the mappings are identical to those shown previously.

For the inverse end, `<set>` is acceptable, as is the following bag mapping:

```
<class name="Item" table="ITEM">
  ...
  <bag name="Categories"
    table="CATEGORY_ITEM"
    lazy="true"
    inverse="true" cascade="save-update">
    <key column="ITEM_ID"/>
    <many-to-many class="Category" column="CATEGORY_ID"/>
  </bag>
</class>
```

This is the first time we've shown the `<bag>` declaration: It's similar to an `<idbag>` mapping, but it doesn't involve a surrogate key column. It lets you use an `IList` (with bag semantics) in a persistent class instead of an `ISet`. Thus it's preferred if the noninverse side of a many-to-many association mapping is using a map, list, or bag (which all permit duplicates). Remember that a bag doesn't preserve the order of elements.

No other mappings should be used for the inverse end of a many-to-many association. Indexed collections such as lists and maps can't be used, since NHibernate won't initialize or maintain the index column if `inverse="true"`. This is also true and important to remember for all other association mappings involving collections: an indexed collection, or even arrays, can't be set to `inverse="true"`.

We already frowned at the use of a many-to-many association and suggested the use of composite element mappings as an alternative. Let's see how this works.

Using a collection of components for a many-to-many association

Suppose we need to record some information each time we add an `Item` to a `Category`. For example, we might need to store the date and the name of the user who added the item to this category. We need a C# class to represent this information:

```
public class CategorizedItem {
    private string username;
    private DateTime dateAdded;
    private Item item;
    private Category category;
    ....
}
```

You'll see that we omitted the properties and `Equals()` and `GetHashCode()` methods, but they would be necessary for this component class.

We map the Items collection on Category as shown below. If you prefer using mapping attributes in your code, you should be able to easily deduce the mapping using attributes; just be careful when ordering them:

```
<set name="Items" lazy="true" table="CATEGORY_ITEMS">
  <key column="CATEGORY_ID"/>
  <composite-element class="CategorizedItem">
    <parent name="Category"/>
    <many-to-one name="Item"
      class="Item"
      column="ITEM_ID"
      not-null="true"/>
    <property name="Username" column="USERNAME" not-null="true"/>
    <property name="DateAdded" column="DATE_ADDED" not-null="true"/>
  </composite-element>
</set>
```

We use the `<many-to-one>` element to declare the association to Item, and we use the `<property>` mappings to declare the extra association-related information. The link table now has four columns: CATEGORY_ID, ITEM_ID, USERNAME, and DATE_ADDED. The columns of the CategorizedItem properties should never be null: otherwise we can't identify a single link entry, because they're all part of the table's primary key. You can see the table structure in figure 6.11.

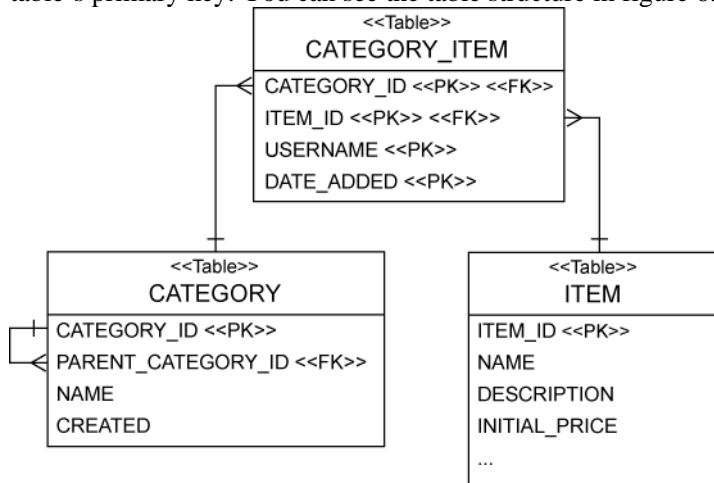


Figure 6.11 Many-to-many entity association table using a component

In fact, rather than mapping just the Username, we might like to keep an actual reference to the User object. In this case, we have the following *ternary association* mapping:

```
<set name="Items" lazy="true" table="CATEGORY_ITEMS">
  <key column="CATEGORY_ID"/>
  <composite-element class="CategorizedItem">
    <parent name="Category"/>
    <many-to-one name="Item"
      class="Item"
      column="ITEM_ID"
      not-null="true"/>
    <many-to-one name="User"
      class="User"
      column="USER_ID"
      not-null="true"/>
    <property name="DateAdded" column="DATE_ADDED" not-null="true"/>
  </composite-element>
</set>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This is a fairly exotic beast! If you find yourself with a mapping like this, you should ask whether it might be better to map `CategorizedItem` as an entity class and use two one-to-many associations. Furthermore, there is no way to make this mapping bidirectional: a component, such as `CategorizedItem` can't, by definition, have shared references. You can't navigate from `Item` to `CategorizedItem`.

We talked about some limitations of many-to-many mappings in the previous section. One of them, the restriction to nonindexed collections for the inverse end of an association, also applies to one-to-many associations, if they're bidirectional. Let's take a closer look at one-to-many and many-to-one again, to refresh your memory and elaborate on what we discussed in chapter 4.

One-to-many associations

You already know most of what you need to know about one-to-many associations from chapter 3. We mapped a typical parent/child relationship between two entity persistent classes, `Item` and `Bid`. This was a bidirectional association, using a `<one-to-many>` and a `<many-to-one>` mapping. The "many" end of this association was implemented in C# with an `ISet`; we had a collection of `Bids` in the `Item` class. Let's reconsider this mapping and walk through some special cases.

Using a bag with set semantics

For example, if you absolutely need an `ICollection` of children in your parent C# class, it's possible to use a `<bag>` mapping in place of a set. In our example, first we have to replace the type of the `Bids` collection in the `Item` persistent class with an `ICollection`. The mapping for the association between `Item` and `Bid` is then left essentially unchanged:

```
<class
  name="Bid"
  table="BID">
  ...
  <many-to-one
    name="Item"
    column="ITEM_ID"
    class="Item"
    not-null="true"/>
</class>

<class
  name="Item"
  table="ITEM">
  ...
  <bag
    name="Bids"
    inverse="true"
    cascade="all-delete-orphan">
    <key column="ITEM_ID"/>
    <one-to-many class="Bid"/>
  </bag>
</class>
```

We renamed the `<set>` element to `<bag>`, making no other changes. Note, however, that this change isn't useful: the underlying table structure doesn't support duplicates, so the `<bag>` mapping results in an association with set semantics. Some tastes prefer the use of `ICollection`s even for associations with set semantics, but ours doesn't, so we recommend using `<set>` mappings for typical parent/child relationships.

The obvious, but wrong solution would be to use a real `<list>` mapping for the `Bids` with an additional column holding the position of the elements. Remember the NHibernate limitation we introduced earlier in this chapter: you can't use indexed collections on an *inverse* side of an

association. The `inverse="true"` side of the association isn't considered when NHibernate saves the object state, so NHibernate will ignore the index of the elements and not update the position column.

However, if your parent/child relationship will only be unidirectional where navigation is only possible from parent to child, you could even use an indexed collection type because the "many" end would no longer be inverse. Good uses for unidirectional one-to-many associations are uncommon in practice, and we don't have one in our auction application. You may remember that we started with the `Item` and `Bid` mapping in chapter 4, making it first unidirectional, but we quickly introduced the other side of the mapping.

Let's find a different example to implement a unidirectional one-to-many association with an indexed collection.

Unidirectional mapping

For the purposes of this discussion, we now suppose that the association between `Category` and `Item` is to be remodeled as a one-to-many association; an `Item` now belongs to at most one category and doesn't own a reference to its current category. In C# code, we model this as a collection named `Items` in the `Category` class; we don't have to change anything if we don't use an indexed collection. If `Items` is implemented as an `ISet`, we use the following mapping:

```
<set name="Items" lazy="true">
  <key column="CATEGORY_ID"/>
  <one-to-many class="Item"/>
</set>
```

Remember that one-to-many association mappings don't need to declare a table name. NHibernate already knows that the column names in the collection mapping (in this case, only `CATEGORY_ID`) belong to the `ITEM` table. The table structure is shown in figure 6.12.

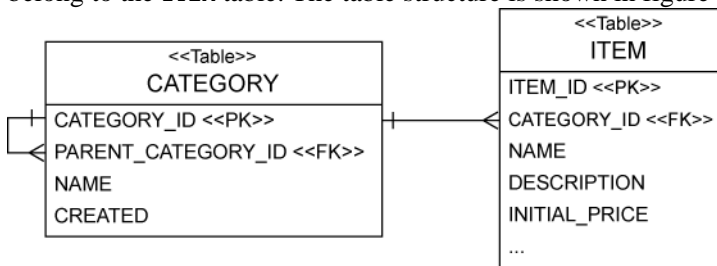


Figure 6.12 A standard one-to-many association using a foreign key column

The other side of the association, the `Item` class, has no mapping reference to `Category`. We can now also use an indexed collection in the `Category`. For example, after we change the `Items` property to `List`:

```
<list name="Items" lazy="true">
  <key>
    <column name="CATEGORY_ID" not-null="false"/>
  </key>
  <index column="DISPLAY_POSITION"/>
  <one-to-many class="Item"/>
</list>
```

Note the new `DISPLAY_POSITION` column in the `ITEM` table, which holds the position of the `Item` elements in the collection.

There is an important issue to consider, which, in our experience, puzzles many NHibernate users at first. In a unidirectional one-to-many association, the foreign key column `CATEGORY_ID` in the `ITEM` must be nullable. An `Item` could be saved without knowing anything about a `Category`—it's a stand-

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

alone entity! This is a consistent model and mapping, and you might have to think about it twice if you deal with a not-null foreign key and a parent/child relationship. Using a bidirectional association (and a Set) is the correct solution.

Now that you know about all the association mapping techniques for normal entities, you may want to consider inheritance; how do all these associations work between various levels of an inheritance hierarchy? Of course, what we really want is *polymorphic* behavior, so let's see how NHibernate deals with polymorphic entity associations.

6.4 Mapping polymorphic associations

Polymorphism is a defining feature of object-oriented languages like C#. Therefore, support for polymorphic associations and queries is a fundamental requirement of an ORM solution like NHibernate. Surprisingly, we've managed to get this far without needing to talk much about polymorphism. Even more surprisingly, there isn't much to say on the topic—polymorphism is so easy to use in NHibernate that we don't need to spend a lot of effort explaining this feature.

To give you a good overview of how polymorphic associations are used, we'll first consider a many-to-one association to a class that might have subclasses. In this case, NHibernate guarantees that you can create links to any subclass instance just as you would to instances of the base class. Following that, we'll also guide you through setting up polymorphic collections, and then go on to explain the particular issues with the "table per concrete class" mapping.

6.4.1 Polymorphic many-to-one associations

A *polymorphic association* is an association that may refer to instances of a subclass, where the parent class was explicitly specified in the mapping metadata. For this example, imagine that we don't have many `BillingDetails` per `User`, but only one, as shown in figure 6.13.

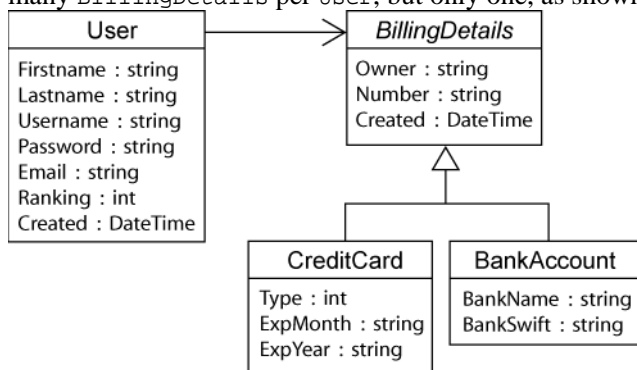


Figure 6.13 The user has only one billing information object.

The user needs a unidirectional association to some `BillingDetails`, which can be `CreditCard` details or `BankAccount` details. We map this association to the abstract class `BillingDetails` as follows:

```

<many-to-one name="BillingDetails"
  class="BillingDetails"
  column="BILLING_DETAILS_ID"
  cascade="save-update"/>
  
```

But since `BillingDetails` is abstract, the association must refer to an instance of one of its subclasses—`CreditCard` or `BankAccount`—at runtime.

All the association mappings we've introduced so far in this chapter support polymorphism. You don't have to do anything special to use polymorphic associations in NHibernate - just specify the

name of any mapped persistent class in your association mapping. Then, if that class declares any <subclass> or <joined-subclass> elements, the association is naturally polymorphic.

The following code demonstrates the creation of an association to an instance of the `CreditCard` subclass:

```
CreditCard cc = new CreditCard();
cc.Number = ccNumber;
cc.Type = ccType;
cc.ExpiryDate = ccExpiryDate;

using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {

    User user = (User) session.Get(typeof(User), uid);
    user.BillingDetails = cc;
    session.Transaction.Commit();
}
```

Now, when we navigate the association in a second transaction, NHibernate automatically retrieves the `CreditCard` instance:

```
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {

    User user = (User) session.Get(typeof(User), uid);
    user.BillingDetails.Pay(paymentAmount);
    session.Transaction.Commit();
}
```

Note that, when `user.BillingDetails.Pay(paymentAmount)`, the call is against the appropriate subclass.

There is one thing to watch out for: if `BillingDetails` was mapped with `lazy="true"`, NHibernate would proxy the `BillingDetails` association. In this case, we wouldn't be able to perform a typecast to the concrete class `CreditCard` at runtime, and even the `is` operator would behave strangely:

```
User user = (User) session.Get(typeof(User), uid);
BillingDetails bd = user.BillingDetails;
Assert.IsFalse( bd is CreditCard );
CreditCard cc = (CreditCard) bd;
```

In this code, the typecast on the last line fails because `bd` is a proxy instance and when creating it, NHibernate doesn't know yet that it is a `CreditCard`; all it knows is that it is a `BillingDetails`. When a method is invoked on the proxy, the call is delegated to an instance of `CreditCard` that is fetched lazily. To perform a proxy-safe typecast, use `Session.Load()`:

```
User user = (User) session.Get(typeof(User), uid);
BillingDetails bd = user.BillingDetails;
CreditCard cc =
    (CreditCard) session.Load( typeof(CreditCard), bd.Id );
expiryDate = cc.ExpiryDate;
```

After the call to `load`, `bd` and `cc` refer to two different proxy instances, which both delegate to the same underlying `CreditCard` instance. Also, because proxy instances were created, no database hit has been incurred yet.

Note that you can avoid these issues by avoiding lazy fetching, as in the following code, using a query technique discussed in the next chapter:

```
User user = (User) session.CreateCriteria(typeof(User))
    .Add(Expression.Expression.Eq("id", uid) )
    .SetFetchMode("BillingDetails", FetchMode.Eager)
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        .UniqueResult();
        CreditCard cc = (CreditCard) user.BillingDetails;
        expiryDate = cc.ExpiryDate;

```

So, `BillingDetails` will have been fetched eagerly in this case thus avoiding the lazy load. Truly object-oriented code shouldn't use `is` or numerous typecasts. If you find yourself running into problems with proxies, you should question your design, asking whether there is a more polymorphic approach.

One-to-one associations are handled the same way. What about many-valued associations?

6.4.2 Polymorphic collections

Let's refactor the previous example to its original form as introduced in our `CaveatEmptor` application. If `User` owns many `BillingDetails`, we use a bidirectional one-to-many. In `BillingDetails`, we have the following:

```

<many-to-one name="User"
  class="User"
  column="USER_ID"/>

```

In the `Users` mapping, we have this:

```

<set name="BillingDetails"
  lazy="true"
  cascade="save-update"
  inverse="true">
  <key column="USER_ID"/>
  <one-to-many class="BillingDetails"/>
</set>

```

Adding a `CreditCard` is easy:

```

CreditCard cc = new CreditCard();
cc.Number = ccNumber;
cc.Type = ccType;
cc.ExpiryDate = ccExpiryDate;

using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {

    User user = (User) session.Get(typeof(User), uid);
    user.AddBillingDetails(cc);
    session.Transaction.Commit();
}

```

As usual, our `user.AddBillingDetails(cc)` function will ensure the association is set at both ends by calling `BillingDetails.Add(cc)` and `cc.User=this`.

We can iterate over the collection and handle instances of `CreditCard` and `BankAccount`:

```

using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {

    User user = (User) session.Get(typeof(User), uid);
    foreach(BillingDetails bd in user.BillingDetails){
        bd.Pay(ccPaymentAmount);
        session.Transaction.Commit();
    }
}

```

Note that, `bd.Pay(...)` will be a call the appropriate `BillingDetails` subclass instance. In the examples so far, we've assumed that `BillingDetails` is a class mapped explicitly in the NHibernate mapping document, and that the inheritance mapping strategy is table-per-hierarchy or table-per-subclass. We haven't yet considered the case of a table-per-concrete-class mapping strategy, where `BillingDetails` wouldn't be mentioned explicitly in the mapping file, but only in the C# definition of the subclasses.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

6.4.3 Polymorphic associations and table-per-concrete-class

In section 3.7.1, “Table per concrete class”, we defined the *table-per-concrete-class* mapping strategy and observed that this mapping strategy makes it difficult to represent a polymorphic association, because you can’t map a foreign key relationship to the table of the abstract base class. There is no table for the base class with this strategy; you only have tables for concrete classes.

Suppose that we want to represent a polymorphic many-to-one association from `User` to `BillingDetails`, where the `BillingDetails` class hierarchy is mapped using this table-per-concrete-class strategy. There is a `CREDIT_CARD` table and a `BANK_ACCOUNT` table, but no `BILLING_DETAILS` table. We need two pieces of information in the `USER` table to uniquely identify the associated `CreditCard` or `BankAccount`:

- n The name of the table in which the associated instance resides
- n The identifier of the associated instance

The `USER` table requires the addition of a `BILLING_DETAILS_TYPE` column, in addition to the `BILLING_DETAILS_ID`. We use a `NHibernate` `<any>` element to map this association:

```
<any name="BillingDetails"
    meta-type="String"
    id-type="Int32"
    cascade="save-update">
  <meta-value value="CREDIT_CARD" class="CreditCard"/>
  <meta-value value="BANK_ACCOUNT" class="BankAccount"/>
  <column name="BILLING_DETAILS_TYPE"/>
  <column name="BILLING_DETAILS_ID"/>
</any>
```

The `meta-type` attribute specifies the `NHibernate` type of the `BILLING_DETAILS_TYPE` column; the `id-type` attribute specifies the type of the `BILLING_DETAILS_ID` column (`CreditCard` and `BankAccount` must have the same identifier type). Note that the order of the `<column>` elements is important: first the type, then the identifier.

The `<meta-value>` elements tell `NHibernate` how to interpret the value of the `BILLING_DETAILS_TYPE` column. We can use any value we like as a type discriminator. For example, we can encode the information in two characters:

```
<any name="BillingDetails"
    meta-type="String"
    id-type="Int32"
    cascade="save-update">
  <meta-value value="CC" class="CreditCard"/>
  <meta-value value="CA" class="BankAccount"/>
  <column name="BILLING_DETAILS_TYPE"/>
  <column name="BILLING_DETAILS_ID"/>
</any>
```

Actually, the `<meta-value>` elements are optional; if you omit them, `NHibernate` will use the fully qualified names of the classes. An example of this table structure is shown in figure 6.14.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

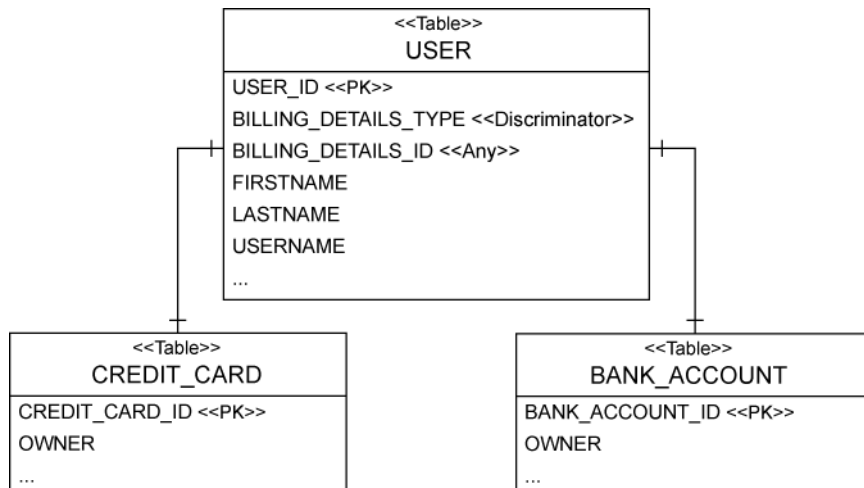


Figure 6.14 Using a discriminator column with an *any* association

Here is the first major problem with this kind of association: we can't add a foreign key constraint to the `BILLING_DETAILS_ID` column, since some values refer to the `BANK_ACCOUNT` table and others to the `CREDIT_CARD` table. Thus, we need to come up with some other way to ensure integrity. A database trigger might be one way of achieving this.

Furthermore, it's difficult to write SQL table joins for this association. In particular, the NHibernate query facilities don't support this kind of association mapping, nor may this association be fetched using an outer join. We discourage the use of `<any>` associations for all but the most special cases.

As you can see, polymorphism is messier in the case of a table-per-concrete-class inheritance mapping strategy. We don't usually use this mapping strategy when polymorphic associations are required. As long as you stick to the other inheritance-mapping strategies, polymorphism is straightforward, and you don't usually need to think about it.

6.5 Summary

This chapter covered the finer points of ORM and techniques sometimes required to solve the structural mismatch problem. We can now fully map all the entities and associations in the `CaveatEmptor` domain model.

We also discussed the NHibernate type system, which distinguishes *entities* from *value types*. An entity instance has its own lifecycle and persistent identity; an instance of a value type is completely dependant on an owning entity.

NHibernate defines a rich variety of built-in value mapping types that bridge the gap between .NET types and SQL types. When these predefined types are insufficient, you can easily extend them using custom types or component mappings and even implement arbitrary conversions from .NET to SQL data types.

Collection-valued properties are considered to be of value type. A collection doesn't have its own persistent identity and belongs to a single owning entity. You've seen how to map collections, including collections of value-typed instances and many-valued entity associations.

NHibernate supports one-to-one, one-to-many, and many-to-many associations between entities. In practice, we recommend against the overuse of many-to-many associations. Associations in NHibernate are naturally polymorphic. We also talked about bidirectional behavior of such relationships.

Having covered mapping techniques in great detail, the next chapter will now move away from the topic of mapping, and turn to the subject of object retrieval. This builds on the concepts introduced in chapter 4, only giving more detailed information allowing you to build more *efficient* and flexible

queries. We will also cover some of the more advanced topics that surround object retrieval, such as report queries, fetching associations and caching.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>



Retrieving objects efficiently

Queries are the most interesting part of writing good data access code. A complex query may require a long time to get right, and its impact on the performance of an application can be tremendous. As with regular SQL, writing NHibernate queries becomes much easier with experience.

If you've been using handwritten SQL for a number of years, you might be concerned that ORM will take away some of the expressiveness and flexibility that you're used to. This is seldom the case; NHibernate's powerful query facilities allow you to do almost anything you would in SQL, and in some cases more. For the rare cases where you can't make NHibernate's own query facilities do exactly what you want, NHibernate allows you to retrieve objects using the native SQL dialect of your database.

In chapter 4, section 4.4, we mentioned that there are three ways to express queries in NHibernate. First is the HQL :

```
session.CreateQuery("from Category c where c.Name like 'Laptop%'");
```

Next is the ICriteria API:

```
session.CreateCriteria(typeof(Category))
    .Add( Expression.Like("Name", "Laptop%") );
```

Finally, there is direct SQL, which automatically maps result sets to objects:

```
session.CreateSQLQuery(
    "select {c.*} from CATEGORY {c} where NAME like 'Laptop%'",
    "c",
    typeof(Category));
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

This chapter covers in-depth techniques using all three methods. You may also use this chapter as a reference, hence some sections are written in a less verbose style but show many small code examples for different use cases.

We'll start our exploration by showing you how queries are actually *executed* with NHibernate. Or put another way, rather than focusing on the queries themselves, we'll focus on the various techniques for actually creating and running queries using NHibernate. Following that, we'll move on to discuss the particulars of how queries are composed.

7.1 Executing queries

The `IQuery` and `ICriteria` interfaces both define several methods for controlling execution of a query. To execute a query in your application, you need to obtain an instance of one of these query interfaces using the `ISession`. Let's take a quick look at how you might do that.

7.1.1 The query interfaces

To create a new `IQuery` instance, call either `CreateQuery()` or `CreateSQLQuery()`. `IQuery` can be used to prepare a HQL query as follows:

```
IQuery hqlQuery = session.CreateQuery("from User");
```

This query is now setup to fetch all user objects in the databases. We can also achieve the same thing using the `CreateSQLQuery()` method using the native SQL dialect of the underlying database:

```
IQuery sqlQuery = session.CreateSQLQuery(
    "select {u.*} from USERS {u}", "u",
    typeof(User));
```

You will learn more about running SQL queries in section 8.5.4, "Native SQL queries". Finally, here's how we'd use the strongly typed `ICriteria` interface to do the same thing in a different way:

```
ICriteria crit = session.CreateCriteria(typeof(User));
```

This last example uses `CreateCriteria()` to get a list of objects back. You'll notice that the *root entity* type we want the query to return is specified as `User`. We'll study criteria queries in detail later on, and now continue our discussion of creating and running queries by looking at another useful concept - pagination.

Paging the result

Pagination is a commonly used technique, and you've probably seen it in action several times. For example, an eCommerce web site may display lists of products over a number of pages, each showing only 10 or 20 products at a time. Typically, users navigate to next or previous pages by clicking appropriate links in the page. When writing data access code for this scenario, the developers needed to work out how to show the correct page of records at any given time – and that's what pagination is all about.

In NHibernate, both the `IQuery` and `ICriteria` interfaces make pagination simple, as demonstrated below:

```
IQuery query =
    session.CreateQuery("from User u order by u.Name asc");
query.SetFirstResult(0);
query.SetMaxResults(10);
```

The call to `SetMaxResults(10)` limits the query result set to the first 10 objects selected by the database. What if we wanted to get some results for the next page?

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

ICriteria crit = session.CreateCriteria(typeof(User));
crit.AddOrder( Order.Asc("Name") );
crit.SetFirstResult(10);
crit.SetMaxResults(10);
IList<User> results = crit.List<User>();

```

Starting from the 10th object, we retrieve the next 10 objects. Note that there is no standard way to express pagination in SQL, and each database vendor often provides a different syntax and approach. Fortunately, NHibernate knows the tricks for each vendor, so paging is very easily done regardless of your particular database.

IQuery and ICriteria also expose a *fluent-interface* that allows *method-chaining*. To demonstrate, we've rewritten the two previous examples to take advantage of this:

```

IList<User> results =
    session.CreateQuery("from User u order by u.Name asc")
        .SetFirstResult(0)
        .SetMaxResults(10)
        .List<User>();

IList<User> results =
    session.CreateCriteria(typeof(User))
        .AddOrder( Order.Asc("Name") )
        .SetFirstResult(40)
        .SetMaxResults(20)
        .List<User>();

```

Chaining method calls in this way is considered to be less verbose and easier to write, and it's possible to do with many of NHibernate's APIs.

Now that we've created queries and set up pagination, we'll look at how you go about getting the results of a query.

Listing and iterating results

The List() method executes the query and returns the results as a list:

```

IList<User> result = session.CreateQuery("from User").List<User>();

```

When writing queries, sometimes we only want a single instance to be returned. For example, if we wanted to find the highest bid, we might get that instance by reading it from the result list by index: result[0]. Alternatively, we could use SetMaxResults(1), and execute the query with the UniqueResult() method:

```

Bid maxBid =
    (Bid) session.CreateQuery("from Bid b order by b.Amount desc")
        .SetMaxResults(1)
        .UniqueResult();

Bid bid = (Bid) session.CreateCriteria(typeof(Bid))
        .Add( Expression.Eq("Id", id) )
        .UniqueResult();

```

You need to be sure that your query only returns one object, otherwise an exception will be thrown.

The IQuery and ISession interfaces also provide an Enumerable() method, which returns the same result as List() or Find(), but which uses a different strategy for retrieving the results. When you use Enumerable() to execute a query, NHibernate retrieves only the primary key (identifier) values in the first SQL select; it tries to find the rest of the state of the objects in the cache, before querying again for the rest of the property values. This technique can be used to optimize loading in specific cases, as discussed in section 7.7, "Optimizing object retrieval."

Why not use ISession.Find() instead of IQuery.List()?

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The `ISession` API provides shortcut methods for simple queries. Instead of creating an `IQuery` instance, you can also call `ISession.Find("from User")`. The result is the same as from `IQuery.List()`. The same is true for `Enumerable()`. However, that query shortcut methods on the `ISession` API will be removed in the future to reduce the bloat of session methods. We recommend always using the `IQuery` API.

Finally, another important factor of constructing queries is that of binding parameters. The `IQuery` interface lets you achieve this in a flexible manner, as we'll now discuss.

7.1.2 Binding parameters

Allowing developers to bind values to queries is an important feature for any data access library because it permits them to construct queries that are both maintainable and secure. We'll demonstrate the types of parameter binding available in NHibernate below, but let's first look at the potential problems of *not* binding parameters.

The problem of sql injection attacks

Consider the following code:

```
string queryString =
    "from Item i where i.Description like '" + searchString + "'";
IList result = session.CreateQuery(queryString).List();
```

This code is plain and simple BAD! You might know why? Simply put, the code would potentially leave your application open to *Sql Injection* attacks. This is where a malicious user attempts to trick your application into running their own SQL against the database, so they can cause damage or circumnavigate application security. If that user typed the `searchString` below:

```
foo' and CallSomeStoredProcedure() and 'bar' = 'bar
```

The `queryString` sent to the database would be

```
from Item i where i.Description like 'foo' and CallSomeStoredProcedure() and 'bar' =
'bar'
```

As you can see, the original `queryString` would no longer be a simple search for a string, but would also execute a stored procedure in the database!

One of the main problems here is that our application isn't checking the values passed in from the user interface. Because of this, the quote characters aren't escaped, and therefore the user can inject their own SQL. Users might even accidentally crash your application just by putting a single quote in the search string. The golden rule is "never pass unchecked values from user input to the database"! Fortunately, this problem is easily avoided by using parameters. With parameters, our query might look like this:

```
string queryString =
    "from Items I where i.Description like :searchString"
```

When we use parameters, queries and parameters are sent to the database separately, so the database can ensure they are dealt with securely and efficiently.

Another reason to use parameters is that it helps NHibernate to be more efficient; this is because NHibernate keeps track of the queries you execute. When parameters are used it only needs to keep

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

one copy of the query in memory, even if it's run thousands of times with different parameters each time.

So, now we understand the importance of parameters, the next question is how do we use them in our NHibernate queries? There are two approaches to *parameter binding*: named parameters and positional parameters. We discuss these in turn below.

Using named parameters

Using named parameters, we can rewrite the query as

```
string queryString =
    "from Item item where item.Description like :searchString";
```

The colon followed by a parameter name indicates a named parameter. Then, we can use the `IQuery` interface to bind a value to the `searchString` parameter:

```
IList result = session.CreateQuery(queryString)
    .SetString("searchString", searchString)
    .List();
```

Because `searchString` is a user-supplied string variable, we use the `SetString()` method of the `IQuery` interface to bind it to the named parameter (`searchString`). This code is cleaner, much safer, and performs better, because a single compiled SQL statement can be reused if only bind parameters change.

Often, you'll need multiple parameters:

```
string queryString = @"from Item item
    where item.Description like :searchString
    and item.Date > :minDate";

IList result = session.CreateQuery(queryString)
    .SetString("searchString", searchString)
    .SetDate("minDate", minDate)
    .List();
```

Using positional parameters

If you prefer, you can use positional parameters:

```
string queryString = @"from Item item
    where item.Description like ?
    and item.Date > ?";

IList result = session.createQuery(queryString)
    .SetString(0, searchString)
    .SetDate(1, minDate)
    .List();
```

Not only is this code less self-documenting than the alternative with named parameters, it's also much more vulnerable to easy breakage if we change the query string slightly:

```
string queryString = @"from Item item
    where item.Date > ?
    and item.Description like ?";
```

Every change of the position of the bind parameters requires a change to the parameter-binding code. This leads to fragile and maintenance-intensive code. We recommend that you avoid positional parameters.

Last, a named parameter may appear multiple times in the query string:

```
string userSearch =
    @"from User u where u.Username like :searchString
    or u.Email like :searchString";

IList result = session.CreateQuery(userSearch)
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
.SetString("searchString", searchString)
.List();
```

Binding arbitrary arguments

We've used `SetString()` and `SetDate()` to bind arguments to query parameters. The `IQuery` interface provides similar convenience methods for binding arguments of most of the NHibernate built-in types: everything from `SetInt32()` to `SetTimestamp()` and `SetEnum()`.

A particularly useful method is `SetEntity()`, which lets you bind a persistent entity:

```
session.CreateQuery("from Item item where item.Seller = :seller")
    .SetEntity("seller", seller)
    .List();
```

However, there is also a generic method that allows you to bind an argument of any NHibernate type:

```
string queryString = @"from Item item
    where item.Seller=:seller and
    item.Description like :desc";

session.CreateQuery(queryString)
    .SetParameter( "seller", seller,
        NHibernateUtil.Entity(typeof(User)) )
    .SetParameter( "desc", description, NHibernateUtil.String )
    .List();
```

This even works for custom user-defined types like `MonetaryAmount`:

```
IQuery q =
    session.CreateQuery("from Bid bid where bid.Amount > :amount");
q.SetParameter( "amount",
    givenAmount,
    NHibernateUtil.Custom(typeof(MonetaryAmountUserType)) );
IList<Bid> result = q.List<Bid>();
```

For some parameter types, it's possible to guess the NHibernate type from the class of the parameter value. In this case, you don't need to specify the NHibernate type explicitly:

```
string queryString = @"from Item item
    where item.Seller = :seller and
    item.Description like :desc";

session.CreateQuery(queryString)
    .SetParameter("seller", seller)
    .SetParameter("desc", description)
    .List();
```

As you can see, it even works with entities, such as `seller`. This approach works nicely for string, int, and bool parameters, for example, but not so well for `DateTime`, where the NHibernate type might be `Timestamp`, or `DateTime`. In that case, you have to use the appropriate binding method or explicitly use `NHibernateUtil.DateTime` (or any other NHibernate type) as the third argument to `SetParameter()`.

If we had a POCO with `Seller` and `Description` properties, we could use the `SetProperties()` method to bind the query parameters. For example, we could pass query parameters in an instance of the `Item` class:

```
Item item = new Item();
item.Seller = seller;
item.Description = description;

string queryString = @"from Item item
    where item.Seller=:seller and
    item.Description like :desc";
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
session.CreateQuery(queryString).SetProperties(item).List();
```

`SetProperties()` matches the names of POCO properties to named parameters in the query string, using `SetParameter()` to guess the NHibernate type and bind the value. In practice, this turns out to be less useful than it sounds, since some common NHibernate types aren't guessable (`DateTime`, in particular).

The parameter-binding methods of `IQuery` are null-safe, making this code legal:

```
session.CreateQuery("from User as u where u.Email = :email")
    .SetString("email", null)
    .List();
```

However, the result of this code is almost certainly not what we intended. The resulting SQL will contain a comparison like `username = null`, which always evaluates to null in SQL ternary logic. Instead, we must use the `is null` operator:

```
session.CreateQuery("from User as u where u.Email is null").List();
```

So far, the HQL code examples we've shown all use embedded HQL query string literals. This isn't unreasonable for simple queries, but once we start considering complex queries that must be split over multiple lines, it starts to get a bit unwieldy.

7.1.3 Using named queries

We don't like to see HQL string literals scattered all over the C# code unless they're necessary. NHibernate lets you store query strings outside of your code, a technique that is called *named queries*. This allows you to store all queries related to a particular persistent class along with the other metadata of that class in an XML mapping file. The name of the query is used to call it from the application.

The `GetNamedQuery()` method obtains an `IQuery` instance for a named query:

```
session.GetNamedQuery("FindItemsByDescription")
    .SetString("description", description)
    .List();
```

In this example, we execute the named query `FindItemsByDescription` after binding a string argument to a named parameter. The named query is defined in mapping metadata, e.g. in `Item.hbm.xml`, using the `<query>` element:

```
<query name="FindItemsByDescription"><![CDATA[
    from Item item where item.Description like :description
]]></query>
```

Named queries don't have to be HQL strings; they might even be native SQL queries—and your C# code doesn't need to know the difference:

```

<sql-query name="FindItemsByDescription"><![CDATA[
    select {i.*} from ITEM {i} where DESCRIPTION like :description
]]>
<return alias="i" class="Item"/>
</sql-query>

```

This is useful if you think you might want to optimize your queries later by fine-tuning the SQL. It's also a good solution if you have to port a legacy application to NHibernate, where SQL code was isolated from the hand-coded ADO.NET routines. With named queries, you can easily port the queries one by one to mapping files.

7.1.4 Using query substitutions

It is often necessary, or at least useful, to use a different word to name an object in a query. For example, with a boolean property like `User.IsAdmin`, you will write:

```
from User u where u.IsAdmin = 1
```

But, by adding this property to your configuration file:

```

<property name="hibernate.query.substitutions">
    true 1, false 0
</property>

```

You can write:

```
from User u where u.IsAdmin = true
```

Note that, this feature can also be used to rename SQL functions.

We've now wrapped up our discussion on creating and running queries, so now it's time to actually focus on the queries themselves. The next section covers HQL, starting with simple queries and moving on to some far more advanced topics.

7.2 Basic queries for objects

Let's start with simple queries to become familiar with the HQL syntax and semantics. Although we show the criteria alternative for most HQL queries, keep in mind that HQL is the preferred approach for complex queries. Usually, the criteria can be derived if you know the HQL equivalent; it's much more difficult the other way around.

NOTE

Testing NHibernate queries— You can use the open source tool NHibernate Query Analyzer to execute NHibernate queries ad hoc. It lets you select NHibernate mapping documents (or write them), setup NHibernate configuration, and then view the result of HQL queries you type interactively. More details are provided in the section 7.6.4.

7.2.1 The simplest query

The simplest query retrieves all instances of a particular persistent class. In HQL, it looks like this:

```
from Bid
```

Using the `ICriteria` interface, it looks like this:

```
ICriteria c = session.CreateCriteria(typeof(Bid));
```

Both would generate the following SQL behind the scenes:

```
select B.BID_ID, B.AMOUNT, B.ITEM_ID, B.CREATED from BID B
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Even for this simple case, you can see that HQL is less verbose than SQL.

7.2.2 Using aliases

When you query a class using HQL you often need to assign an *alias* to the queried class, which you use as reference in other parts of the query:

```
from Bid as bid
```

The `as` keyword is always optional. The following is equivalent:

```
from Bid bid
```

Think of this as being a bit like the temporary variable declaration in the following C# code:

```
for ( int i = 0; i < allQueriedBids.Count; i++ ) {  
    Bid bid = (Bid) allQueriedBids[i];  
    ...  
}
```

We assign the alias `bid` to queried instances of the `Bid` class, allowing us to refer to their property values later in the code (or query). To remind yourself of the similarity, we recommend that you use the same naming convention for aliases that you use for temporary variables (camelCase, usually). However, we use shorter aliases in some of the examples in this book (for example, `i` instead of `item`) to keep the printed code readable.

NOTE

We never write HQL keywords in uppercase; we never write SQL keywords in uppercase either. It looks ugly and antiquated—most modern terminals can display both uppercase and lowercase characters. However, HQL isn't case-sensitive for keywords, so you can write `FROM Bid AS bid` if you like shouting.

By contrast, a criteria query defines an implicit alias. The root entity in a criteria query is always assigned the alias `this`. We discuss this topic in more detail later, when we're joining associations with criteria queries. You don't have to think much about aliases when using the `ICriteria` API.

7.2.3 Polymorphic queries

We described HQL as an object-oriented query language, so it should support *polymorphic queries*—that is, queries for instances of a class and all instances of its subclasses, respectively. You already know enough HQL that we can demonstrate this. Consider the following query:

```
from BillingDetails
```

This query returns objects of the type `BillingDetails`, which is an abstract class. So, in this case, the concrete objects are of the subtypes of `BillingDetails`: `CreditCard` and `BankAccount`. If we only want instances of a particular subclass, we may use

```
from CreditCard
```

The class named in the `from` clause doesn't need to be a mapped persistent class; any class will do. The following query returns all persistent objects in the entire database:

```
from System.Object
```

Of course, this also works for interfaces—this query returns all serializable persistent objects (actually, only those implementing the interface `ISerializable`):

```
from System.ISerializable
```

Criteria queries also support polymorphism:

```
session.CreateCriteria(typeof(BillingDetails)).List();
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This query returns instances of `BillingDetails` and its subclasses. Likewise, the following criteria query returns all persistent objects:

```
session.CreateCriteria(typeof(System.Object)).List();
```

Polymorphism applies not only to classes named explicitly in the `from` clause, but also to polymorphic associations, as you'll see later.

Now that we've discussed the `from` clause, let's move on to the other parts of HQL.

7.2.4 Restriction

We don't usually want to retrieve all instances of a class when we run a query. Instead, we would want to express some constraints on the property values of our objects, so only a subset of objects is retrieved. This is called *restriction*, and in both HQL and SQL, restriction is achieved using the `where` clause. A `where` clause can be simple or complex, but let's start with a simple HQL example:

```
from User u where u.Email = 'foo@hibernate.org'
```

Notice that the constraint is expressed in terms of a property, `Email`, of the `User` class, and that we use an object-oriented notion: Just as in C#, `u.Email` may not be abbreviated to plain `Email`.

For a criteria query, we must construct an `ICriterion` object to express the constraint. The `Expression` class provides factory methods for built-in `ICriterion` types. Let's create the same query using criteria and immediately execute it:

```
ICriterion emailEq = Expression.Eq("Email", "foo@hibernate.org");
ICriteria crit = session.CreateCriteria(typeof(User));
crit.add(emailEq);
User user = (User) crit.UniqueResult();
```

We create an `ICriterion` instance holding the simple `Expression` for an equality comparison and add it to the `ICriteria`. The `UniqueResult()` method executes the query and returns exactly one object as a result.

Usually, we would write this a bit less verbosely, using method chaining:

```
User user = (User) session.CreateCriteria(typeof(User))
    .Add( Expression.Eq("Email", "foo@hibernate.org") )
    .UniqueResult();
```

The SQL generated by these queries is

```
select U.USER_ID, U.FIRSTNAME, U.LASTNAME, U.USERNAME, U.EMAIL
from USER U
where U.EMAIL = 'foo@hibernate.org'
```

It is common to have a restriction which should always be used; most of the time, it is used to ignore deprecated data. You may, for example, have an `Active` property and write:

```
select User u where u.Email = 'foo@hibernate.org' and u.Active = 1
```

But this is dangerous because you may forget this restriction; a better solution, in this case, is to change your mapping:

```
<class name="User" where="ACTIVE=1">
```

Now, you can simply write:

```
from User u where u.Email = 'foo@hibernate.org'
```

This query will generate the SQL query:

```
select U.USER_ID, U.FIRSTNAME, U.LASTNAME, U.USERNAME, U.EMAIL
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
from USER U
where U.EMAIL = 'foo@hibernate.org' and U.ACTIVE = 1
```

Note that the `ACTIVE` column doesn't have to be mapped. And, since NHibernate 1.2.0, this attribute is also used when calling `ISession.Load()` and `ISession.Get()`. This feature is also available for collections:

```
<bag name="Users" where="ACTIVE=1">
```

Here, the collection `Users` will only contain users whose `ACTIVE` value is 1. This approach may be useful, but we recommend considering filters for most scenarios (read section 7.5.2, "Collection filters").

You can, of course, use various other comparison operators for restriction.

7.2.5 Comparison operators

A restriction is expressed using ternary logic. The `where` clause is a logical expression that evaluates to true, false, or null for each tuple of objects. You construct logical expressions by comparing properties of objects to other properties or literal values using HQL's built-in comparison operators.

FAQ

What is ternary logic? A row is included in an SQL result set if and only if the `where` clause evaluates to true. In C#, `notNullObject==null` evaluates to false and `null==null` evaluates to true. In SQL, `NOT_NULL_COLUMN=null` and `null=null` both evaluate to null, not true. Thus, SQL needs a special operator, `IS NULL`, to test whether a value is null. This ternary logic is a way of handling expressions that may be applied to null column values. It is a (debatable) SQL extension to the familiar binary logic of the relational model and of typical programming languages such as C#.

HQL supports the same basic operators as SQL: `=`, `<>`, `<`, `>`, `>=`, `<=`, `between`, `not between`, `in`, and `not in`. For example:

```
from Bid bid where bid.Amount between 1 and 10
from Bid bid where bid.Amount > 100
from User u where u.Email in ( 'foo@hibernate.org', 'bar@hibernate.org' )
```

In the case of criteria queries, all the same operators are available via the `Expression` class:

```
session.CreateCriteria(typeof(Bid))
    .Add( Expression.Between("Amount", 1, 10) )
    .List();

session.CreateCriteria(typeof(Bid))
    .Add( Expression.Gt("Amount", 100) )
    .List();

string[] emails = { "foo@NHibernate.org", "bar@NHibernate.org" };
session.CreateCriteria(typeof(User))
    .Add( Expression.In("Email", emails) )
    .List();
```

Because the underlying database implements ternary logic, testing for null values requires some care. Remember that `null = null` doesn't evaluate to true in the database, but to null. All comparisons that use the null operator in fact evaluate to null. Both HQL and the `ICriteria` API provide an SQL-style `is null` operator:

```
from User u where u.Email is null
```

This query returns all users with no email address. The same semantic is available in the `ICriteria` API:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

session.CreateCriteria(typeof(User))
    .Add( Expression.IsNull("Email") )
    .List();

```

We also need to be able to find users who *do* have an email address:

```

from User u where u.Email is not null

session.CreateCriteria(typeof(User))
    .Add( Expression.IsNotNull("Email") )
    .List();

```

Finally, the HQL `where` clause supports arithmetic expressions (but the `ICriteria` API doesn't):

```

from Bid bid where ( bid.Amount / 0.71 ) - 100.0 > 0.0

```

For string-based searches, you need to be able to perform case-insensitive matching and matches on fragments of strings in restriction expressions.

7.2.6 String matching

The `like` operator allows wildcard searches, where the wildcard symbols are `%` and `_`, just as in SQL:

```

from User u where u.Firstname like "S%"

```

This expression restricts the result to users with a first name starting with a capital `S`. You can also negate the `like` operator, for example using a substring match expression:

```

from User u where u.Firstname not like "%Foo S%"

```

For criteria queries, wildcard searches may use either the same wildcard symbols or specify a `MatchMode`. NHibernate provides the `MatchMode` as part of the `ICriteria` query API; we use it for writing string match expressions without string manipulation. These two queries are equivalent:

```

session.CreateCriteria(typeof(User))
    .Add( Expression.Like("Firstname", "S%") )
    .List();

session.CreateCriteria(typeof(User))
    .Add( Expression.Like("Firstname", "S", MatchMode.Start) )
    .List();

```

The allowed `MatchModes` are `Start`, `End`, `Anywhere`, and `Exact`.

An extremely powerful feature of HQL is the ability to call arbitrary SQL functions in the `where` clause. If your database supports user-defined functions (most do), you can put this functionality to all sorts of uses, good or evil. For the moment, let's consider the usefulness of the standard ANSI SQL functions `upper()` and `lower()`. They can be used for case-insensitive searching:

```

from User u where lower(u.Email) = 'foo@hibernate.org'

```

The `ICriteria` API doesn't currently support SQL function calls. It does, however, provide a special facility for case-insensitive searching:

```

session.CreateCriteria(typeof(User))
    .Add( Expression.Eq("Email", "foo@hibernate.org").IgnoreCase() )
    .List();

```

Unfortunately, HQL doesn't provide a standard string-concatenation operator; instead, it supports whatever syntax your database provides. Here is an example for SQL Server:

```

from User user
where ( user.Firstname + ' ' + user.Lastname ) like 'S% K%'

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

We'll return to some more exotic features of the HQL where clause later in this chapter. We only used single expressions for restrictions in this section; let's combine several with logical operators.

7.2.7 Logical operators

Logical operators (and parentheses for grouping) are used to combine expressions:

```
from User user
  where user.Firstname like "S%" and user.Lastname like "K%"

from User user
  where ( user.Firstname like "S%" and user.Lastname like "K%" )
  or user.Email in ( 'foo@hibernate.org', 'bar@hibernate.org' )
```

If you add multiple `ICriterion` instances to the one `ICriteria` instance, they're applied conjunctively (that is, using `and`):

```
session.CreateCriteria(typeof(User))
    .Add( Expression.Like("Firstname", "S%") )
    .Add( Expression.Like("Lastname", "K%") )
```

If you need disjunction (`or`), you have two options. The first is to use `Expression.Or()` together with `Expression.And()`:

```
ICriteria crit = session.CreateCriteria(typeof(User))
    .Add(
        Expression.Or(
            Expression.And(
                Expression.Like("Firstname", "S%"),
                Expression.Like("Lastname", "K%")
            ),
            Expression.In("Email", emails)
        )
    );
```

The second option is to use `Expression.Disjunction()` together with `Expression.Conjunction()`:

```
ICriteria crit = session.CreateCriteria(typeof(User))
    .Add( Expression.Disjunction()
        .Add( Expression.Conjunction()
            .Add( Expression.Like("Firstname", "S%") )
            .Add( Expression.Like("Lastname", "K%") )
        )
        .Add( Expression.In("Email", emails) )
    );
```

We think both options are ugly, even after spending five minutes trying to format them for maximum readability. So, unless you're constructing a query on the fly, the HQL string is much easier to understand. Complex criteria queries are useful only when they're created programmatically; for example, in the case of a complex search screen with several optional search criteria, we might have a `CriteriaBuilder` that translates user restrictions to `ICriteria` instances.

7.2.8 Ordering query results

All query languages provide a mechanism for ordering query results. HQL provides an *order by* clause, similar to SQL.

This query returns all users, ordered by username:

```
from User u order by u.Username
```

You specify ascending and descending order using `asc` or `desc`:

```
from User u order by u.Username desc
```

Finally, you can order by multiple properties:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
from User u order by u.Lastname asc, u.Firstname asc
```

The `ICriteria` API provides a similar facility:

```
IList results = session.CreateCriteria(typeof(User))
    .AddOrder( Order.Asc("Lastname") )
    .AddOrder( Order.Asc("Firstname") )
    .List();
```

Thus far, we've only discussed the basic concepts of HQL and criteria queries. You've learned how to write a simple `from` clause and use aliases for classes. We've combined various restriction expressions with logical operators. However, we've focused on single persistent classes—that is, we've only referenced a single class in the `from` clause. An important query technique we haven't discussed yet is the *joining of associations* at runtime.

7.3 Joining associations

When querying databases, sometimes we want to combine data in two or more relations. This is achieved using a *join*. For example, we might join the data in the `ITEM` and `BID` tables, as shown in figure 7.1. Note that not all columns and possible rows are shown; hence the dotted lines.

ITEM			BID		
ITEM_ID	NAME	INITIAL_PRICE	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00	1	1	10.00
2	Bar	50.00	2	1	20.00
3	Baz	1.00	3	2	55.50

Figure 7.1 The `ITEM` and `BID` tables are obvious candidates for a join operation.

What most people think of when they hear the word *join* in the context of SQL databases is an *inner join*. An inner join is one of several types of joins, and it's the easiest to understand. Consider the SQL statement and result in figure 7.2. This SQL statement is an *ANSI-style join*.

```
from ITEM I inner join BID B on I.ITEM_ID = B.ITEM_ID
```

ITEM_ID	NAME	INITIAL_PRICE	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00	1	1	10.00
1	Foo	2.00	2	1	20.00
2	Bar	50.00	3	2	55.50

Figure 7.2 The result table of an ANSI-style inner join of two tables

If we join tables `ITEM` and `BID` with an inner join, using their common attributes (the `ITEM_ID` column), we get all items and their bids in a new result table. Note that the result of this operation contains only items that have bids. If we want all items, and null values instead of bid data when there is no corresponding bid, we use a (*left*) *outer join*, as shown in figure 7.3.

```
from ITEM I left outer join BID B on I.ITEM_ID = B.ITEM_ID
```

ITEM_ID	NAME	INITIAL_PRICE	BID_ID	ITEM_ID	AMOUNT
1	Foo	2.00	1	1	10.00
1	Foo	2.00	2	1	20.00
2	Bar	50.00	3	2	55.50
3	Baz	1.00	null	null	null

Figure 7.3 The result of an ANSI-style left outer join of two tables

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

You can think of a table join as working as follows. First, you get a Cartesian product of the two tables by taking all possible combinations of `ITEM` rows with `BID` rows. Second, you filter these joined rows using a *join condition*. Note that the database has much more sophisticated algorithms to evaluate a join; it usually doesn't build a memory-consuming product and then filter all rows. The join condition is just a boolean expression that evaluates to true if the joined row is to be included in the result. In the case of the left outer join, each row in the (left) `ITEM` table that never satisfies the join condition is also included in the result, with null values returned for all columns of `BID`. (A *right* outer join would retrieve all bids and null if a bid has no item—certainly not a sensible query in our situation.)

In SQL, the join condition is usually specified explicitly; it isn't possible to simply use the name of a foreign key constraint to specify how two tables are to be joined. Instead, we have to specify the join condition in the `on` clause for an ANSI-style join or in the `where` clause for a so-called *theta-style join*, where `I.ITEM_ID = B.ITEM_ID`.

7.3.1 NHibernate join options

In NHibernate queries, you don't usually specify a join condition explicitly. Rather, you specify the name of a mapped class association so that NHibernate can work out the join for you. For example, the `Item` class has an association named `bids` with the `Bid` class. If we name this association in our query, NHibernate has enough information in the mapping document to then deduce the join expression. This helps make queries less verbose and more readable.

HQL provides four ways of expressing inner and outer joins:

- n An *ordinary* join in the `from` clause
- n A *fetch* join in the `from` clause
- n A *theta-style* join in the `where` clause
- n An *implicit* association join

We'll discuss all of these options in this chapter, but because the “ordinary” and “fetch” `from` clause joins have the clearest syntax, we'll discuss these first.

When working with NHibernate, there are usually several reasons why you might want to use a join, and it's important to note that NHibernate actually lets you differentiate between the *purposes* for joining. Let's put this in the context of a short example. Suppose we're querying `Items`, there are three possible reasons why we might be interested in joining the `Bids`.

Firstly, we might want to retrieve `Items` returned on the basis of some criterion that should be applied to their `Bids`. For example, you might want all `Items` that have a bid of more than \$100; hence this requires an *inner join*.

Alternatively, we may be running a query where we're mainly interested in only the `Items` without any special criterion for `Bids`. We may or may not want to access the `Bids` for an item, but we want the option for NHibernate to lazily load them when we first access the collection.

Another reason for joining to `Bids` is that we may want to execute an outer join to load all `Items` along with their `bids` in the same select, something we called *eager fetching* earlier. Remember that we prefer to map all associations lazy by default, so an eager, outer-join fetch query can be used to override this default fetching strategy at runtime. We'll discuss this scenario first.

7.3.2 Fetching associations

In HQL, you can specify that an association should be eagerly fetched by an outer join using the `fetch` keyword in the `from` clause:

```
from Item item
left join fetch item.Bids
where item.Description like '%part%'
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This query returns all items with a description that contains the string `part`, and all their bids, in a single select. When executed, it returns a list of `Item` instances, with their `bids` collections fully initialized. We call this a *from clause fetch join*. The purpose of a fetch join is performance optimization: We use this syntax only because we want eager initialization of the `bids` collections in a single SQL select.

We can do the same thing using the `ICriteria` API:

```
session.CreateCriteria(typeof(Item))
    .SetFetchMode("Bids", FetchMode.Eager)
    .Add( Expression.Like("Description", "part", MatchMode.Anywhere) )
    .List();
```

Both of these queries result in the following SQL:

```
select I.DESCRPTION, I.CREATED, I.SUCCESSFUL_BID, B.BID_ID,
B.AMOUNT, B.ITEM_ID, B.CREATED
from ITEM I
left outer join BID B on I.ITEM_ID = B.ITEM_ID
where I.DESCRPTION like '%part%'
```

We can also prefetch many-to-one or one-to-one associations using the same syntax:

```
from Bid bid
left join fetch bid.Item
left join fetch bid.Bidder
    where bid.Amount > 100

session.CreateCriteria(typeof(Bid))
    .SetFetchMode("Item", FetchMode.Eager)
    .SetFetchMode("Bidder", FetchMode.Eager)
    .Add( Expression.Gt("Amount", 100 ) )
    .List();
```

These queries execute the following SQL:

```
select I.DESCRPTION, I.CREATED, I.SUCCESSFUL_BID,
B.BID_ID, B.AMOUNT, B.ITEM_ID, B.CREATED,
U.USERNAME, U.PASSWORD, U.FIRSTNAME, U.LASTNAME
from BID B
left outer join ITEM I on I.ITEM_ID = B.ITEM_ID
left outer join USER U on U.USER_ID = B.BIDDER_ID
where B.AMOUNT > 100
```

Note that the `left` keyword is optional in HQL, so we could rewrite the previous examples using `join fetch`. Although this looks straightforward to use, there are a couple of things to consider and remember:

- n *HQL always ignores the mapping document eager fetch (outer join) setting.* If you've mapped some associations to be fetched by outer join, by setting `outer-join="true"` or `fetch="join"` on the association mapping, any HQL query will ignore this preference. With HQL, if you want eager fetching you need to ask for it in the query string. HQL is designed to be as flexible as possible: You can completely (re)define the fetching strategy that should be used at runtime. In comparison, the criteria will take full notice of your mappings! If you specify `outer-join="true"` in the mapping file, the criteria query will fetch that association by outer join—just like `ISession.Get()` or `ISession.Load()` for retrieval by identifier. For a criteria query, you can explicitly disable outer join fetching by calling `SetFetchMode("Bids", FetchMode.Lazy)`.
- n *NHibernate currently limits you to fetching just one collection eagerly.* This is a reasonable restriction, since fetching more than one collection in a single query would be a Cartesian product result. This restriction might be relaxed in a future version of NHibernate, but we

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

encourage you to think about the size of the result set if more than one collection is fetched in an outer join. The amount of data that would have to be transported between database and application can easily grow into the megabyte range, and most of it would be thrown away immediately (NHibernate *flattens* the tabular result set to build the object graph). You may fetch as many one-to-one or many-to-one associations as you like.

- n *If you fetch a collection, NHibernate doesn't return a distinct result list.* For example, an individual `Item` might appear several times in the result `IList`, if you outer-join fetch the bids. You'll probably need to make the results distinct yourself using, for example: `distinctResults = new HashSet(resultList);`. An `ISet` doesn't allow duplicate elements.

This is how NHibernate implements what we call *runtime association fetching strategies*, a powerful feature that is essential for achieving high performance in ORM. Let's continue with the other join operations.

7.3.3 Using aliases with joins

We've already discussed the role of the `where` clause in expressing restriction. Often, you'll need to apply restriction criteria to multiple associated classes (joined tables). If we want to do this using an HQL `from` clause join, we need to assign an alias to the joined class:

```
from Item item
join item.Bids bid
  where item.Description like '%part%'
  and bid.Amount > 100
```

This query assigns the alias `item` to the class `Item` and the alias `bid` to the joined `Item`'s bids. We then use both aliases to express our restriction criteria in the `where` clause.

The resulting SQL is as follows:

```
select I.DESCRPTION, I.CREATED, I.SUCCESSFUL_BID,
B.BID_ID, B.AMOUNT, B.ITEM_ID, B.CREATED
from ITEM I
inner join BID B on I.ITEM_ID = B.ITEM_ID
where I.DESCRPTION like '%part%'
and B.AMOUNT > 100
```

The query returns all combinations of associated `Bids` and `Items`. But, unlike a `fetch join`, the `Bids` collection of the `Item` isn't initialized by the query! So what do we mean by a *combination* here? We mean an ordered pair: `(bid, item)`. In the query result, NHibernate represents an ordered pair as an array. Let's discuss a full code example with the result of such a query:

```
IQuery q = session.CreateQuery("from Item item join item.Bids bid");
foreach( object[] pair in q.List() ) {
    Item item = (Item) pair[0];
    Bid bid = (Bid) pair[1];
}
```

Instead of an `IList` of `Items`, this query returns an `IList` of `object[]` arrays. At index 0 is the `Item`, and at index 1 is the `Bid`. A particular `Item` may appear multiple times, once for each associated `Bid`.

This is all different from the case of a query with an eager `fetch join`. The query with the `fetch join` returned an `IList` of `Items`, with initialized `Bids` collections.

If we don't want the `Bids` in the query result, we can specify a `select` clause in HQL. This clause is optional (it isn't in SQL), so we only have to use it when we aren't satisfied with the result returned by default. We use the alias in a `select` clause to retrieve only the selected objects:

```
select item
from Item item
join item.Bids bid
  where item.Description like '%part%'
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
and bid.Amount > 100
```

Now the generated SQL looks like this:

```
select I.DESCRPTION, I.CREATED, I.SUCCESSFUL_BID,
from ITEM I
inner join BID B on I.ITEM_ID = B.ITEM_ID
where I.DESCRPTION like '%part%'
and B.AMOUNT > 100
```

The query result contains just Items, and because it's an *inner join*, only Items that have Bids:

```
IQuery q = session.CreateQuery("select i from Item i join i.Bids b");
foreach( Item item in q.List<Item>() {
    ...
}
```

As you can see, using aliases in HQL is the same for both direct classes and joined associations. We assign aliases in the from clause and use them in the where and in the optional select clause. The select clause in HQL is much more powerful; we discuss it in detail later in this chapter.

There are two ways to express a join in the ICriteria API; hence there are two ways to use aliases for restriction. The first is the CreateCriteria() method of the Criteria interface. It means that you can nest calls to CreateCriteria():

```
ICriteria itemCriteria = session.CreateCriteria(typeof(Item));
itemCriteria.Add( Expression.Like("Description",
                                "part",
                                MatchMode.Anywhere) );
ICriteria bidCriteria = itemCriteria.CreateCriteria("Bids");
bidCriteria.Add( Expression.Gt( "Amount", 100 ) );

IList results = itemCriteria.List();
```

We'd usually write the query as follows, using method chaining:

```
IList results =
    session.CreateCriteria(typeof(Item))
        .Add( Expression.Like("Description", "part", MatchMode.Anywhere) )
        .CreateCriteria("Bids")
        .Add( Expression.Gt("Amount", 100) )
        .List();
```

The creation of an ICriteria instance for the Bids of the Item results in an inner join between the tables of the two classes. Note that we may call List() on either ICriteria instance without changing the query results.

The second way to express this query using the ICriteria API is to assign an alias to the joined entity:

```
IList results =
    session.CreateCriteria(typeof(Item))
        .CreateAlias("Bids", "bid")
        .Add( Expression.Like("Description", "%part%") )
        .Add( Expression.Gt("bid.Amount", 100) )
        .List();
```

This approach doesn't use a second instance of ICriteria. So, properties of the joined entity must be qualified by the alias assigned in CreateAlias(). Properties of the *root entity* (Item) may be referred to without the qualifying alias or by using the alias "this". Thus the following is equivalent:

```
IList results =
    session.CreateCriteria(typeof(Item))
        .CreateAlias("Bids", "bid")
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        .Add( Expression.Like("this.Description", "%part%") )
        .Add( Expression.Gt("bid.Amount", 100)
        .List();

```

By default, a criteria query returns only the root entity—in this case, the `Items`—in the query result. Let's summarize with a full example:

```

IList<Item> results =
    session.CreateCriteria(typeof(Item))
        .CreateAlias("Bids", "bid")
        .Add( Expression.Like("this.Description", "%part%") )
        .Add( Expression.Gt("bid.Amount", 100) )
        .List<Item>();

foreach( Item item in results ) {
    // Do something
}

```

Keep in mind that the `Bids` collection of each `Item` isn't initialized. A limitation of criteria queries is that you can't combine a `CreateAlias` with an eager fetch mode; for example, `SetFetchMode("Bids", FetchMode.Eager)` isn't valid.

Sometimes you'd like a less verbose way to express a join. In NHibernate, you can use an *implicit association join*.

7.3.4 Using implicit joins

So far, we've used simple qualified property names like `bid.Amount` and `item.Description` in our HQL queries. HQL supports multipart property path expressions for two purposes:

- n Querying components
- n Expressing implicit association joins

The first use is straightforward:

```

from User u where u.Address.City = 'Bangkok'

```

We express the parts of the mapped component `Address` with dot notation. This usage is also supported by the `ICriteria` API:

```

session.CreateCriteria(typeof(User))
    .Add( Expression.Eq("Address.City", "Bangkok") );

```

The second usage, implicit association joining, is available only in HQL. For example:

```

from Bid bid where bid.Item.Description like '%part%'

```

This result in an implicit join on the many-to-one associations from `Bid` to `Item`. Implicit joins are always directed along many-to-one or one-to-one associations, never through a collection-valued association (you can't write `item.Bids.Amount`).

Multiple joins are possible in a single property path expression. If the association from `Item` to `Category` would be many-to-one (instead of the current many-to-many), we could write

```

from Bid bid where bid.Item.Category.Name like 'Laptop%'

```

We frown on the use of this syntactic sugar for more complex queries. Joins are important, and especially when optimizing queries, you need to be able to see at a glance how many of them there are. Consider the following query (again, using a many-to-one from `Item` to `Category`):

```

from Bid bid
    where bid.Item.Category.Name like 'Laptop%'
    and bid.Item.SuccessfulBid.Amount > 100

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

How many joins are required to express this in SQL? Even if you get the answer right, we bet it takes you more than a few seconds. The answer is three; the generated SQL looks something like this:

```
select ...
from BID B
inner join ITEM I on B.ITEM_ID = I.ITEM_ID
inner join CATEGORY C on I.CATEGORY_ID = C.CATEGORY_ID
inner join BID SB on I.SUCCESSFUL_BID_ID = SB.BID_ID
where C.NAME like 'Laptop%'
and SB.AMOUNT > 100
```

It's more obvious if we express the same query like this:

```
from Bid bid
join bid.Item item
  where item.Category.Name like 'Laptop%'
  and item.SuccessfulBid.Amount > 100
```

We can even be more verbose:

```
from Bid as bid
join bid.Item as item
join item.Category as cat
join item.SuccessfulBid as winningBid
  where cat.Name like 'Laptop%'
  and winningBid.Amount > 100
```

Let's continue with join conditions using arbitrary attributes, expressed in *theta-style*.

7.3.5 *Theta-style joins*

A cartesian product allows you to retrieve all possible combinations of instances of two or more classes. This query returns all ordered pairs of `Users` and `Category` objects:

```
from User, Category
```

Obviously, this generally isn't useful. There is one case where it's commonly used: *theta-style joins*.

In traditional SQL, a theta-style join is a Cartesian product, together with a join condition in the `where` clause, which is applied on the product to restrict the result.

In HQL, the theta-style syntax is useful when your join condition isn't a foreign key relationship mapped to a class association. For example, suppose we store the `User`'s name in log records instead of mapping an association from `LogRecord` to `User`. The classes don't "know" anything about each other, because they aren't associated. We can then find all the `Users` and their `LogRecords` with the following theta-style join:

```
from User user, LogRecord log where user.Username = log.Username
```

The join condition here is the `username`, presented as an attribute in both classes. If both entities have the same `username`, they're joined (with an inner join) in the result. The query result consists of ordered pairs:

```
IList results = session.CreateQuery(
    @"from User user, LogRecord log
     where user.Username = log.Username"
)
.List();

foreach( Object[] pair in results )
  User user = (User) pair[0];
  LogRecord log = (LogRecord) pair[1];
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

We can change the result by adding a `select` clause.

You probably won't need to use theta-style joins often. Note that the `ICriteria` API doesn't provide any means for expressing Cartesian products or theta-style joins. It's also currently not possible in NHibernate to outer-join two tables that don't have a mapped association.

7.3.6 Comparing identifiers

It's extremely common to perform queries that compare primary key or foreign key values to either query parameters or other primary or foreign key values. If you think about this in more object-oriented terms, what you're doing is comparing object references. HQL supports the following:

```
from Item i, User u
  where i.Seller = u and u.Username = 'steve'
```

In this query, `i.Seller` refers to the foreign key to the `USER` table in the `ITEM` table (on the `SELLER_ID` column), and `user` refers to the primary key of the `USER` table (on the `USER_ID` column). This next query uses a theta-style join and is equivalent to the much preferred ANSI style:

```
from Item i join i.Seller u
  where u.Username = 'steve'
```

On the other hand, the following theta-style join *cannot* be re-expressed as a `from` clause join:

```
from Item i, Bid b
  where i.Seller = b.Bidder
```

In this case, `i.Seller` and `b.Bidder` are both foreign keys of the `USER` table. Note that this is an important query in our application; we use it to identify people bidding for their own items.

We might also like to compare a foreign key value to a query parameter—for example, to find all Comments from a User:

```
User givenUser = ...
IQuery q =
  session.CreateQuery("from Comment c where c.FromUser = :user");
q.SetEntity("user", givenUser);
IList results = q.List();
```

Alternatively, sometimes we'd prefer to express these kinds of queries in terms of identifier values rather than object references. You can refer to an identifier value by either the name of the identifier property (if there is one) or the special property name `id`. Every persistent entity class has this special HQL property, even if you don't implement an identifier property on the class (see chapter 3, section 3.5.2, "Database identity with NHibernate").

These queries are exactly equivalent to the previous queries:

```
from Item i, User u
  where i.Seller.id = u.id and u.Username = 'steve'
from Item i, Bid b
  where i.Seller.id = b.Bidder.id
```

However, we can now use the identifier value as a query parameter:

```
long userId = ...
IQuery q =
  session.CreateQuery("from Comment c where c.FromUser.id = :id"); q.SetInt64("id",
  userId);
IList results = q.List();
```

You might have noticed that there is a world of difference between the following queries:

```
from Bid b where b.Item.id = 1
from Bid b where b.Item.Description like '%part%'
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The second query uses an implicit table join; the first has no joins at all.

We've now covered most of the features of NHibernate's query facilities that are commonly needed for retrieving objects for manipulation in business logic. In the next section, we'll change our focus and discuss features of HQL that are used mainly for analysis and reporting functionality.

7.4 Writing report queries

Reporting queries take advantage of the database's ability to perform efficient grouping and aggregation of data.

They're more relational in nature; they don't always return entities. For example, instead of retrieving complete `Item` entities, a report query might only retrieve their names and prices. If this is the only information we need for a report screen, we don't need transactional entities and can save the small overhead of automatic dirty-checking and caching in the `ISession`.

Let's consider the structure of an HQL query again.:

```
[select ...] from ... [where ...]
[group by ... [having ...]] [order by ...]
```

The only mandatory clause of an HQL query is the `from` clause, so all other clauses are optional. So far, we've discussed the `from`, `where`, and `order by` clauses. We also used the `select` clause to declare which entities should be returned in a join query.

In reporting queries, you use the `select` clause for *projection* and the `group by` and `having` clauses for aggregation. Let's look at what we mean by projection.

7.4.1 Projection

The `select` clause performs projection. It lets you specify which objects or properties of objects you want in your query results. For example, as you've already seen, the following query returns ordered pairs of `Items` and `Bids`:

```
from Item item join item.Bids bid where bid.Amount > 100
```

If we only wanted the `Items`, we should use this query instead:

```
select item from Item item join item.Bids bid where bid.Amount > 100
```

Or, if we were just displaying a list page to the user, it might be adequate to just retrieve a few properties of those objects needed for that page:

```
select item.id, item.Description, bid.Amount
from Item item join item.Bids bid
where bid.Amount > 100
```

This query returns an array of objects for each row. Because there are three items in the `select` clause, each `object[]` will have 3 elements. Also, because it's a report query, the objects in the result aren't NHibernate entities and aren't therefore transactional. Let's execute the query with some code:

```
IList results = session.CreateQuery(
    @"select item.id, item.Description, bid.Amount
    from Item item join item.Bids bid
    where bid.Amount > 100"
)
.List();

foreach( Object[] row in results ) {
    long id = (long) row[0];
    string description = (string) row[1];
    double amount = (double) row[2];

    // ... show values in a report screen
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

If you're used to working with domain objects, this example will seem quite ugly and verbose. NHibernate gives us a another approach to this, called *dynamic instantiation*.

Using dynamic instantiation

If you don't find working with arrays of values a little cumbersome, NHibernate let's us use dynamic instantiation and define a class to represent each row of results. We can do this using the HQL `select new` construct:

```
select new ItemRow( item.id, item.Description, bid.Amount )
  from Item item join item.Bids bid
  where bid.Amount > 100
```

The `ItemRow` class is one we'd write just for our report screen, and note that we also have to give it an appropriate constructor. This query returns newly instantiated (but transient) instances of `ItemRow`, as you can see in the next example:

```
IList results = session.CreateQuery(
    @"select new ItemRow( item.id, item.Description, bid.Amount )
      from Item item join item.Bids bid
      where bid.Amount > 100"
)
.List();

foreach( ItemRow row in results ) {
    // Do something
}
```

The custom `ItemRow` class doesn't have to be a persistent class that has it's own mapping file, but in order for NHibernate to "see" it, we need to import it using:

```
<hibernate-mapping>
  <import class="ItemRow" />
</hibernate-mapping>
```

`ItemRow` is therefore only a data-transfer class, useful in report generation.

Getting distinct results

When you use a `select` clause, the elements of the result are no longer guaranteed to be unique. For example, `Items` descriptions aren't unique, so the following query might return the same description more than once:

```
select item.Description from Item item
```

It's difficult to see how it could possibly be meaningful to have two identical rows in a query result, so if you think duplicates are likely, you should use the `distinct` keyword:

```
select distinct item.Description from Item item
```

This eliminates duplicates from the returned list of `Item` descriptions.

Calling SQL functions

You may recall that you can call database-specific SQL functions in the `where` clause. It's also possible, at least for some NHibernate SQL dialects, to call database-specific SQL functions from the `select` clause. For example, the following query retrieves the current date and time from the database server (SQL Server syntax), together with a property of `Item`:

```
select item.StartDate, getdate() from Item item
```

The technique of database functions in the `select` clause is of course not limited to database-dependent functions, but to other, more generic (or standardized) SQL functions as well:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```
select item.StartDate, item.EndDate, upper(item.Name)
from Item item
```

This query returns an `object[]` with the starting and ending date of an item auction, and the name of the item all in uppercase.

Let's now look at calling SQL *aggregate functions*.

7.4.2 Using aggregation

NHibernate recognizes the following aggregate functions: `count()`, `min()`, `max()`, `sum()`, and `avg()`.

This query counts all the Items:

```
select count(*) from Item
```

The result is returned as an Integer:

```
int count =
    (int) session.CreateQuery("select count(*) from Item")
                .UniqueResult();
```

Notice how we use `*`, which has the same semantics as in SQL.

The next variation of the query counts all Items that have a `successfulBid`:

```
select count(item.SuccessfulBid) from Item item
```

This query calculates the total of all the successful Bids:

```
select sum(item.SuccessfulBid.Amount) from Item item
```

The query returns a value of the same type as the summed elements; in this case `double`. Notice the use of an implicit join in the `select` clause: We navigate the association (`SuccessfulBid`) from `Item` to `Bid` by referencing it with a dot.

The next query returns the minimum and maximum bid amounts for a particular Item:

```
select min(bid.Amount), max(bid.Amount)
from Bid bid where bid.Item.id = 1
```

The result is an ordered pair of doubles (two instances of `double` in an `object[]` array).

The special `count(distinct)` function ignores duplicates:

```
select count(distinct item.Description) from Item item
```

When you call an aggregate function in the `select` clause without specifying any grouping in a `group by` clause, you collapse the result down to a single row containing your aggregated value(s). This means (in the absence of a `group by` clause) any `select` clause that contains an aggregate function must contain only aggregate functions.

So, for more advanced statistics and reporting, you'll need to be able to perform *grouping*.

7.4.3 Grouping

Just like in SQL, any property or alias that appears in HQL outside of an aggregate function in the `select` clause must also appear in the `group by` clause.

Consider the next query, which counts the number of users with each particular last name:

```
select u.Lastname, count(u) from User u
group by u.Lastname
```

Now look at the generated SQL:

```
select U.LAST_NAME, count(U.USER_ID)
from USER U
group by U.LAST_NAME
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

In this example, the `u.Lastname` isn't inside an aggregate function; we use it to group the result. We also don't need to specify the property we'd like to count in HQL. The generated SQL will automatically use the primary key if we use an alias that has been set in the `from` clause.

The next query finds the average bid amount for each item:

```
select bid.Item.id, avg(bid.Amount) from Bid bid
group by bid.Item.id
```

This query returns ordered pairs of `Item` identifiers and average bid amount. Notice how we use the `id` special property to refer to the identifier of a persistent class no matter what the identifier's real property name is.

The next query counts the number of bids and calculates the average bid per unsold item:

```
select bid.Item.id, count(bid), avg(bid.Amount)
from Bid bid
where bid.Item.SuccessfulBid is null
group by bid.Item.id
```

This query uses an implicit association join. For an explicit ordinary join in the `from` clause (not a fetch join), we can re-express it as follows:

```
select bidItem.id, count(bid), avg(bid.Amount)
from Bid bid
join bid.Item bidItem
where bidItem.SuccessfulBid is null
group by bidItem.id
```

To initialize the `bids` collection of the `Items`, we can use a fetch join and refer to the associations starting on the other side:

```
select item.id, count(bid), avg(bid.Amount)
from Item item
fetch join item.Bids bid
where item.SuccessfulBid is null
group by item.id
```

Sometimes, you'll want to further restrict the result by selecting only particular values of a group.

7.4.4 Restricting groups with having

The `where` clause is used to perform the relational operation of restriction on rows. The `having` clause performs restriction on groups.

For example, the next query counts users with each last name that begins with *K*:

```
select user.Lastname, count(user)
from User user
group by user.Lastname
having user.Lastname like 'K%'
```

The same rules govern the `select` and `having` clauses: Only grouped properties may appear outside an aggregate function. The next query counts the number of bids per unsold item, returning results for only those items that have more than 10 bids:

```
select item.id, count(bid), avg(bid.Amount)
from Item item
join item.Bids bid
where item.SuccessfulBid is null
group by item.id
having count(bid) > 10
```

Most report queries use a `select` clause to choose a list of projected or aggregated properties. You've seen that when more than one property or alias is listed in the `select` clause, NHibernate returns the query results as tuples: Each row of the query result list is an instance of `object[]`. Tuples are

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

inconvenient and non-typesafe, so NHibernate provides the `select new` constructor as mentioned earlier. You can create new objects dynamically with this technique and also use it in combination with aggregation and grouping.

If we define a class called `ItemBidSummary` with a constructor that takes a long, a string, and an int, we can use the following query:

```
select new ItemBidSummary( bid.Item.id, count(bid), avg(bid.Amount) )
    from Bid bid
    where bid.item.SuccessfulBid is null
    group by bid.Item.id
```

In the result of this query, each element is an instance of `ItemBidSummary`, which is a summary of an `Item`, the number of bids for that item, and the average bid amount. This approach is typesafe, and a data transfer class such as `ItemBidSummary` can easily be extended for special formatted printing of values in reports.

7.4.5 Improving performance with report queries

Report queries can have an impact on the performance of your application. Let's explore this issue in more depth.

The only time we've seen any significant overhead in NHibernate code compared to direct ADO.NET queries—and then only for unrealistically simple test cases—is in the special case of read-only queries against a local database. It's possible for a database to completely cache query results in memory and respond quickly, so benchmarks are generally useless if the dataset is small: Plain SQL and ADO.NET will always be the fastest option.

On the other hand, even with a small result set, NHibernate must still do the work of adding the resulting objects of a query to the `ISession` cache (perhaps also the second-level cache) and manage uniqueness, and so on. Report queries give you a way to avoid the overhead of managing the `ISession` cache. The overhead of a NHibernate report query compared to direct SQL/ADO.NET isn't usually measurable, even in unrealistic extreme cases like loading one million objects from a local database without network latency.

Report queries using projection in HQL let you specify exactly which properties you wish to retrieve. For report queries, you aren't selecting entities, but only properties or aggregated values:

```
select user.Lastname, count(user)
    from User user
    group by user.Lastname
```

This query doesn't return a persistent entity, so NHibernate doesn't add a transactional object to the `ISession` cache. Furthermore, NHibernate won't be tracking changes to these returned objects.

Reporting queries result in faster release of allocated memory, since objects aren't kept in the `ISession` cache until the `ISession` is closed—they may be garbage-collected as soon as they're dereferenced by the application, after executing the report.

These considerations are almost always extremely minor, so don't go out and rewrite all your read-only transactions to use report queries instead of transactional, cached, and monitored objects. Report queries are more verbose and (arguably) less object-oriented. They also make less efficient use of NHibernate's caches, which is much more important once you consider the overhead of remote communication with the database in production systems. We follow the "don't optimize prematurely" wisdom, and urge you to wait until you find an actual case where you have a real performance problem before using this optimization.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

7.4.6 Obtaining DataSets

It may happen that you have to interact with a component using DataSets. Many report engines, like Crystal Reports, have limited support for POCO (but which may be enough). Their common data source is either the database directly or a DataSet. But, as NHibernate do not return DataSets, we have to find a solution.

The most common work-around is to directly use ADO.NET to get the DataSet; this solution may fit in many situations but it doesn't take advantage of NHibernate features and requires careful monitoring of possible changes as they can make NHibernate caches stale.

Another solution is to use NHibernate to query data and fill a DataSet with the result. This operation can be done manually by writing a code similar to the one required for DTOs, but it will become tedious when dealing with numerous entities. In this case, Code Generation can help greatly simplify the process.

Now, let's get back to regular entity queries. There are still many NHibernate features waiting to be discovered.

7.5 Advanced query techniques

You'll use advanced query techniques less frequently with NHibernate, but it will be helpful to know about them. In this section, we talk about programmatically building criteria with "example objects", a topic we briefly introduced earlier.

Filtering collections is also a handy technique: You can use the database instead of filtering objects in memory. Subqueries and queries in native SQL will round out your knowledge of NHibernate query techniques.

7.5.1 Dynamic queries

It's common for queries to be built programmatically by combining several optional query criteria depending on user input. For example, a system administrator may wish to search for users by any combination of first name or last name, and to retrieve the result ordered by username. Using HQL, we could build the query using string manipulations:

```
public IList<User> FindUsers(string firstname,
                             string lastname) {

    StringBuilder queryString = new StringBuilder();
    bool conditionFound = false;

    if (firstname != null) {
        queryString.Append("lower(u.Firstname) like :firstname ");
        conditionFound=true;
    }
    if (lastname != null) {
        if (conditionFound) queryString.Append("and ");
        queryString.Append("lower(u.Lastname) like :lastname ");
        conditionFound=true;
    }

    string fromClause = conditionFound ?
        "from User u where " :
        "from User u ";

    queryString.Insert(0, fromClause).Append("order by u.username");

    IQuery query = GetSession().CreateQuery( queryString.ToString() );

    if (firstname != null)
        query.SetString( "firstname",
                        '%' + firstname.ToLower() + '%' );
    if (lastname != null)
        query.SetString( "lastname",
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        '%' + lastname.ToLower() + '%' );
    return query.List<User>();
}

```

This code is tedious and noisy, so let's try a different approach. The `ICriteria` API looks promising:

```

public IList<User> FindUsers(string firstname,
                             string lastname) {
    ICriteria crit = GetSession().CreateCriteria(typeof(User));
    if (firstname != null) {
        crit.Add( Expression.InsensitiveLike("Firstname",
                                             firstname,
                                             MatchMode.Anywhere) );
    }
    if (lastname != null) {
        crit.Add( Expression.InsensitiveLike("Lastname",
                                             lastname,
                                             MatchMode.Anywhere) );
    }
    crit.AddOrder( Order.Asc("Username") );
    return crit.List<User>();
}

```

This code is much shorter and more readable. Note that the `InsensitiveLike()` operator performs a case-insensitive match. There seems no doubt that this is a better approach. However, for search screens with many optional search criteria, there is an even better way.

First, observe that as we add new search criteria, the parameter list of `FindUsers()` grows. It would be better to capture the searchable properties as an object. Since all the search properties belong to the `User` class, why not use an instance of `User`?

QBE uses this idea; you provide an instance of the queried class with some properties initialized, and the query returns all persistent instances with matching property values. NHibernate implements QBE as part of the `ICriteria` query API:

```

public IList<User> FindUsers(User u) {
    Example exampleUser =
        Example.Create(u).IgnoreCase().EnableLike(MatchMode.Anywhere);
    return GetSession().CreateCriteria(typeof(User))
        .Add(exampleUser)
        .List<User>();
}

```

The call to `Create()` returns a new instance of `Example` for the given instance of `User`. The `IgnoreCase()` method puts the example query into a case-insensitive mode for all string-valued properties. The call to `EnableLike()` specifies that the SQL like operator should be used for all string-valued properties, and specifies a `MatchMode`.

We've significantly simplified the code *again*. The nicest thing about NHibernate `Example` queries is that an `Example` is just an ordinary `ICriterion`. So, you can freely mix and match QBE with QBC.

Let's see how this works by further restricting the search results to users with unsold `Items`. For this purpose, we add an `ICriteria` to the example user, constraining the result using its `Items` collection of `Items`:

```

public IList<User> FindUsers(User u) {
    Example exampleUser =
        Example.Create(u).IgnoreCase().EnableLike(MatchMode.Anywhere);

```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        return GetSession().CreateCriteria(typeof(User))
            .Add( exampleUser )
            .CreateCriteria("Items")
            .Add( Expression.IsNull("SuccessfulBid") )
            .List<User>();
    }

```

Even better, we can combine `User` properties and `Item` properties in the same search:

```

public IList<User> FindUsers(User u, Item i) {
    Example exampleUser =
        Example.Create(u).IgnoreCase().EnableLike(MatchMode.Anywhere);

    Example exampleItem =
        Example.Create(i).IgnoreCase().EnableLike(MatchMode.Anywhere);

    return GetSession().CreateCriteria(typeof(User))
        .Add( exampleUser )
        .CreateCriteria("Items")
        .Add( exampleItem )
        .List<User>();
}

```

At this point, we invite you to step back and consider how much code would be required to implement this search screen using handcoded SQL/ADO.NET. It's rather a lot, if we listed it here it would stretch for *pages*.

7.5.2 Collection filters

You'll commonly want to execute a query against all elements of a particular collection. For instance, we might have an `Item` and wish to retrieve all bids for that particular item, ordered by the amount of the bid. We already know one good way to write this query:

```

IList results =
    session.CreateQuery(@"from Bid b where b.Item = :item
                        order by b.Amount asc")
        .SetEntity("item", item)
        .List();

```

This query works perfectly, since the association between bids and items is bidirectional and each `Bid` knows its `Item`. Imagine that this association was unidirectional: `Item` has a collection of `Bids`, but there is no inverse association from `Bid` to `Item`.

We could try the following query:

```

string query = @"select bid from Item item join item.Bids bid
                where item = :item order by bid.Amount asc";

IList results = session.CreateQuery(query)
                .SetEntity("item", item)
                .List();

```

This query is inefficient—it uses an unnecessary join. A better, more elegant solution is to use a *collection filter*: a special query that can be applied to a persistent collection or array. It's commonly used to further restrict or order a result. We use it on an already loaded `Item` and its collection of bids:

```

IList results = session.CreateFilter( item.Bids,
                                    "order by this.Amount asc" )
                .List();

```

This filter is equivalent to the first query shown earlier and results in identical SQL. Collection filters have an implicit `from` clause and an implicit `where` condition. The alias `this` refers implicitly to elements of the collection of bids.

NHibernate collection filters are *not* executed in memory. The collection of bids may be uninitialized when the filter is called and, if so, will remain uninitialized. Furthermore, filters don't apply to transient collections or query results; they may only be applied to a persistent collection currently referenced by an object associated with the NHibernate session.

The only required clause of an HQL query is `from`. Since a collection filter has an implicit `from` clause, the following is a valid filter:

```
IList results = session.CreateFilter( item.Bids, "" ).List();
```

To the great surprise of everyone (including the designer of this feature), this trivial filter turns out to be useful! You can use it to paginate collection elements:

```
IList results = session.CreateFilter( item.Bids, "" )
    .SetFirstResult(50)
    .SetMaxResults(25)
    .List();
```

Usually, however, we'd use an `order by` with paginated queries.

Even though you don't need a `from` clause in a collection filter, you can include one if you like. A collection filter doesn't even need to return elements of the collection being filtered. The next query returns any `Category` with the same name as a category in the given collection:

```
string filterString =
    "select other from Category other where this.Name = other.Name";

IList results =
    session.CreateFilter( cat.ChildCategories, filterString )
        .List();
```

The following query returns a collection of `Users` who have bid on the item:

```
IList results =
    session.CreateFilter( item.Bids,
        "select this.Bidder" )
        .List();
```

The next query returns all these users' bids (including those for other items):

```
IList results = session.CreateFilter(
    item.Bids,
    "select elements(this.Bidder.Bids)" )
    .List();
```

Note that the query uses the special HQL `elements()` function (explained later) to select all elements of a collection.

The most important reason for the existence of collection filters is to allow the application to retrieve some elements of a collection without initializing the entire collection. In the case of very large collections, this is important to achieve acceptable performance. The following query retrieves all bids made by a user in the past week:

```
IList results =
    session.CreateFilter( user.Bids,
        "where this.Created > :oneWeekAgo" )
        .SetDateTime("oneWeekAgo", oneWeekAgo)
        .List();
```

Again, this query does *not* initialize the `Bids` collection of the `User`.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

7.5.3 Subqueries

Subselects are an important and powerful feature of SQL . A subselect is a select query embedded in another query, usually in the `select`, `from`, or `where` clause.

HQL supports subqueries in the `where` clause. We can't think of many good uses for subqueries in the `from` clause, although `select` clause subqueries might be a nice future extension. (You might remember from chapter 3, section 3.4.2, that a *derived property* mapping is in fact a `select` clause subselect.) Note that some platforms supported by NHibernate don't implement subselects. If you desire portability among many different databases, you shouldn't use this feature.

The result of a subquery might contain either a single row or multiple rows. Typically, subqueries that return single rows perform aggregation. The following subquery returns the total number of items sold by a user; the outer query returns all users who have sold more than 10 items:

```
from User u where 10 < (
    select count(i) from u.Items i where i.SuccessfulBid is not null
)
```

This is a *correlated subquery*—it refers to an alias (`u`) from the outer query. The next subquery is an *uncorrelated subquery*:

```
from Bid bid where bid.Amount + 1 >= (
    select max(b.Amount) from Bid b
)
```

The subquery in this example returns the maximum bid amount in the entire system; the outer query returns all bids whose amount is within one (dollar) of that amount.

Note that in both cases, the subquery is enclosed in parentheses. This is always required.

Uncorrelated subqueries are harmless; there is no reason not to use them when convenient, although they can always be rewritten as two queries (after all, they don't reference each other). You should think more carefully about the performance impact of correlated subqueries. On a mature database, the performance cost of a simple correlated subquery is similar to the cost of a join. However, it isn't necessarily possible to rewrite a correlated subquery using several separate queries.

If a subquery returns multiple rows, it's combined with *quantification*. ANSI SQL (and HQL) defines the following quantifiers:

- n any
- n all
- n some (a synonym for any)
- n in (a synonym for = any)

For example, the following query returns items where all bids are less than 100:

```
from Item item where 100 > all ( select b.Amount from item.Bids b )
```

The next query returns all items with bids greater than 100:

```
from Item item where 100 < any ( select b.Amount from item.Bids b )
```

This query returns items with a bid of exactly 100:

```
from Item item where 100 = some ( select b.Amount from item.Bids b )
```

So does this one:

```
from Item item where 100 in ( select b.Amount from item.Bids b )
```

HQL supports a shortcut syntax for subqueries that operate on elements or indices of a collection. The following query uses the special HQL `elements()` function:

```
IList list = session.CreateQuery(@"from Category c
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        where :item in elements(c.Items)")
        .SetEntity("item", item)
        .List();

```

The query returns all categories to which the item belongs and is equivalent to the following HQL, where the subquery is more explicit:

```

IList results = session.CreateQuery(@"from Category c
        where :item in (from c.Items)")
        .SetEntity("item", item)
        .List();

```

Along with `elements()`, HQL provides `indices()`, `maxelement()`, `minelement()`, `maxindex()`, `minindex()`, and `size()`, each of which is equivalent to a certain correlated subquery against the passed collection. Refer to the NHibernate documentation for more information about these special functions; they're rarely used.

Subqueries are an advanced technique; you should question their frequent use, since queries with subqueries can often be rewritten using only joins and aggregation. However, they're powerful and useful from time to time.

By now, we hope you're convinced that NHibernate's query facilities are flexible, powerful, and easy to use. HQL provides almost all the functionality of ANSI standard SQL. Of course, on rare occasions you *do* need to resort to handcrafted SQL, especially when you wish to take advantage of database features that go beyond the functionality specified by the ANSI standard.

7.6 Native SQL

We can think of some good examples why you might use native SQL queries in NHibernate: HQL provides no mechanism for specifying SQL query hints, it also doesn't support hierarchical queries (such as the Oracle `CONNECT BY` clause) and you may need to quickly port SQL code to your application. We suppose that you'll stumble on other examples.

In these relatively rare cases, you're free to resort to using the ADO.NET API directly. However, doing so means writing the tedious code by hand to transform the result of the query to an object graph. You can avoid all this work by using NHibernate's built-in support for native SQL queries.

NHibernate allows you to execute arbitrary SQL queries to retrieve scalar values or even entities. These queries can be written in your C# code or in your mapping files. In the later case, it is also possible to call stored procedures. You can even override the SQL commands that NHibernate generates for the CRUD operations. All these techniques will be covered in the following pages.

7.6.1 Using the ISQLQuery API

`ISQLQuery` instances are created by calling the method `ISession.CreateSQLQuery()`, passing in a SQL query string. Then, you may use the methods of `ISQLQuery` to provide more details about your query.

A SQL query can return scalar values from individual columns, an complete entity (along with its associations and collections) or multiple entities. It also supports all the features of HQL queries which means you can use parameters and paging, etc.

Scalar and entity queries

The simplest native queries are scalar queries. For example:

```

Ilist results = session.CreateSQLQuery("SELECT * FROM ITEM")
        .AddScalar("ITEM_ID", NHibernateUtil.Int64)
        .AddScalar("NAME", NHibernateUtil.String)
        .AddScalar("CREATED", NHibernateUtil.Date)
        .List();

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

This query won't return Item objects, but instead it will return the specified columns of all items as arrays of objects (object[]). These columns override the * in the SELECT.

If you want to retrieve entities, you can do this:

```
Ilist<Item> results = session.CreateSQLQuery("SELECT * FROM ITEM")
    .AddEntity(typeof(Item))
    .List<Item>();
```

You may also manage associations and collections by *joining* them. Let's see how we can eagerly load the items with their sellers:

```
Ilist<Item> results = session.CreateSQLQuery(
    @"SELECT ITEM_ID, NAME, CREATED, SELLER_ID, ...
        USER_ID, FIRSTNAME, ...
    FROM ITEM i, USER u,
    WHERE i.SELLER_ID = u.USER_ID" )
    .AddEntity("item", typeof(Item))
    .AddJoin("item.Seller")
    .List<Item>();
```

In this case, the query is more complex because we must specify the columns of both tables. We must also specify the alias "item" in AddEntity() in order to join its Seller.

Here is how you may try to join the Bids collection of the items:

```
Ilist<Item> results = session.CreateSQLQuery(
    @"SELECT i.ITEM_ID, NAME, CREATED, ...
        BID_ID, CREATED, b.ITEM_ID, ...
    FROM ITEM i, BID b,
    WHERE i.ITEM_ID = b.ITEM_ID" )
    .AddEntity("item", typeof(Item))
    .AddJoin("item.Bids")
    .List<Item>();
```

This query is similar to the previous one. A good knowledge of SQL was enough to write it and take care of the ambiguous column name ITEM_ID. However, unfortunately this will not work in NHibernate; it will not recognize i.ITEM_ID because it isn't specified like that in the mapping file or attributes.

It is time to introduce a new trick that will address this problem. We demonstrate this in the next section that explains how to retrieve many entity types in a single query.

Multiple entities queries

Querying more than one entity increases the chance of having column name duplications. Thankfully, this problem is simple to solve using placeholders.

Placeholders are necessary because an SQL query result may return the state of multiple entity instances in each row, or even the state of multiple instances of the same entity. What we need is a way to distinguish between the different entities. NHibernate uses its own naming scheme where column aliases are placed in the SQL to correctly map column values to the properties of particular instances. However, when using our own SQL we don't want the user to have to understand all this. Instead, native SQL queries are specified with placeholders for the column aliases, which are much simpler.

The following native SQL query shows what these placeholders—the names enclosed in braces—look like:

```
Ilist<Item> results = session.CreateSQLQuery(
    @"SELECT i.ITEM_ID as {item.id},
        i.NAME as {item.Name},
        i.CREATED as {item.Created}, ...
    FROM ITEM i" )
    .AddEntity("item", typeof(Item))
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```
.List<Item>();
```

Each placeholder specifies an HQL-style property name. And, we must provide the entity class that is referred to by `item` in the placeholders.

Here's a shortcut if you don't want to specify every columns explicitly:

```
Ilist<Item> results = session.CreateSQLQuery(
    @"SELECT {item.*}
    FROM ITEM" )
    .AddEntity("item", typeof(Item))
    .List<Item>();
```

The `{item.*}` placeholder is replaced with a list of the mapped column names and correct column aliases for all properties of the `Item` entity.

Now, let's see how we can return multiple entities:

```
Ilist results = session.CreateSQLQuery(
    @"SELECT {item.*}, {user.*}
    FROM ITEM i INNER JOIN USER u
    ON i.SELLER_ID = u.USER_ID" )
    .AddEntity("item", typeof(Item))
    .AddEntity("user", typeof(User))
    .List();
```

This query will return tuples of entities; as usual, NHibernate represents a tuple as an instance of `object[]`.

As with HQL, it is usually recommended to keep these queries out of your code by writing them in your mappings as discussed next.

7.6.2 Named SQL queries

Named SQL queries are queries defined in NHibernate mapping files. Here is how you can rewrite the previous example:

```
<sql-query name="FindItemsAndSellers">
  <return alias="item" class="Item"/>
  <return alias="user" class="User"/>
  <![CDATA[
    SELECT {item.*}, {user.*}
    FROM ITEM i INNER JOIN USER u
    ON i.SELLER_ID = u.USER_ID
  ]]>
</sql-query>
```

This named query can be executed from code as follows:

```
Ilist results = session.GetNamedQuery("FindItemAndSellers")
    .List();
```

Comparing the named query to the inline version discussed previously, we can see `<return>` element replaces the method `AddEntity()`. `<return-join>` is used for associations and collections, and `<return-scalar>` to return scalar values. You may also load a collection only using `<load-collection>`.

If you frequently return the same information, you can externalize it using `<resultset>`. Here is a complete example:

```
<resultset name="FullItem">
  <return alias="item" class="Item"/>
  <return-join alias="user" property="item.Seller"/>
  <return-join alias="bid" property="item.Bids"/>
  <return-scalar column="diff" type="int"/>
</resultset>
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

</resultset>

<sql-query name="FindItemsWithSellersAndBids" resultset-ref="FullItem">
  <![CDATA[
    SELECT {item.*}, {user.*}, {bid.*},
           i.RESERVE_PRICE-i.INITIAL_PRICE as diff
    FROM ITEM i
    INNER JOIN USER u ON i.SELLER_ID = u.USER_ID
    LEFT OUTER JOIN BID b ON i.ITEM_ID = b.ITEM_ID
  ]]>
</sql-query>

```

When executed, this query will return tuples containing an item with its seller and bids and a computed scalar value (diff).

Note that `<resultset>` elements can also be referred in code using the method `ISQLQuery.SetResultSetMapping()`.

Named SQL queries allow you to avoid the `{}`-syntax and define your own column aliases. Here is a simple example:

```

<sql-query name="FindItems">
  <return alias="item" class="Item">
    <return-property name="Name" column="MY_NAME"/>
    <return-property name="InitialPrice">
      <return-column name="MY_ITEM_PRICE_VALUE"/>
      <return-column name="MY_ITEM_PRICE_CURRENCY"/>
    </return-property>
  </return>
  <![CDATA[
    SELECT i.ITEM_ID as {item.id}, ...
           i.NAME as MY_NAME,
           i.INITIAL_PRICE as MY_ITEM_PRICE_VALUE,
           i.INITIAL_PRICE_CURRENCY as MY_ITEM_PRICE_CURRENCY,
    FROM ITEM i
  ]]>
</sql-query>

```

In this example, we also use the `{}`-syntax for columns we don't want to customize. In chapter 6, section 6.1.2, we defined item's initial price as a composite user type, so this example shows how to load it.

Since the native SQL is tightly coupled to the actual mapped tables and columns, we strongly recommend that you define all native SQL queries in the mapping document instead of embedding them in the C# code.

Using stored procedures

NHibernate named SQL queries can call stored procedures and functions since version 1.2.0.

If we create a stored procedure like (using a SQL Server database):

```

CREATE PROCEDURE FindItems_SP AS
  SELECT ITEM_ID, NAME, INITIAL_PRICE, INITIAL_PRICE_CURRENCY, ...
  FROM ITEM

```

We can call it using the this query:

```

<sql-query name="FindItems">
  <return alias="item" class="Item">
    <return-property name="id" column="ITEM_ID"/>
    <return-property name="Name" column="NAME"/>
    <return-property name="InitialPrice">
      <return-column name="INITIAL_PRICE"/>
      <return-column name="INITIAL_PRICE_CURRENCY"/>
    </return-property>
  ...
</sql-query>

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    </return>
    exec FindItems_SP
</sql-query>

```

Unlike the previous example, we must map all properties here; this is obvious as NHibernate can not inject its own columns aliases.

Note that the stored procedure must return a resultset to be able to work with NHibernate. Another limitation is that associations and collections are not supported; a SQL query calling a stored procedure can only return scalar values and entities.

Finally, stored procedures are database-dependent. Therefore, the mapping for a SQL Server database may not be the same as for an Oracle database.

If, in some special cases, you need even more control over the SQL that is executed, or if you want to call a stored procedure that isn't supported, NHibernate offers you a way to get an ADO.NET connection. The property `session.Connection` returns the currently active ADO.NET `IDbConnection` from the `ISession`. It's not your responsibility to close this connection, just to execute whatever SQL statements you like and then continue using the `ISession` (and finally, close the `ISession`). The same is true for transactions; you must not commit or roll back this connection yourself (unless you completely manage the connection for NHibernate).

7.6.3 Customizing Create, Retrieve, Update, Delete commands

In most cases, the commands generated by NHibernate to save your entities are acceptable. However, it may happen that you need to perform a specific operation and therefore override NHibernate's generated SQL. NHibernate allows you to specify the SQL statements for create, update, and delete operations.

The custom commands are written in the mapping of the concerned class. For example:

```

<class name="Item">
    ...
    <sql-insert>
        INSERT INTO ITEM (NAME, ..., ITEM_ID) VALUES (UPPER(?), ..., ?)
    </sql-insert>
    <sql-update>UPDATE ITEM SET NAME=UPPER(?), ... WHERE ITEM_ID=?</sql-update>
    <sql-delete>exec DeleteItem_SP ?</sql-delete>
</class>

```

In this example, `<sql-insert>` and `<sql-update>` respectively save and update an item with a custom logic (converting names to upper case). And, as you can see in `<sql-delete>`, these custom commands can also call stored procedures. In this last case, the order of the positional parameters must be respected (as you can see here, the identifier is generally the last parameter). The order is defined by NHibernate and you can read it by enabling debug logging and reading the static SQL commands that are generated by NHibernate (remember to do that before writing these custom commands).

Note that the custom `<sql-insert>` will be ignored if you use `identity` to generate identifier values for the class. Your custom commands are required to affect the same number of rows as NHibernate-generated SQL would. You can disable this verification by adding `check="none"` to your commands.

Retrieve commands are defined as named queries. For example, here is a query to load an item with pessimistic lock:

```

<sql-query name="LoadItem">
    <return alias="item" class="Item" lock-mode="upgrade"/>
    SELECT {item.*}
    FROM ITEM

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        WHERE ITEM_ID = ?
        FOR UPDATE
    </sql-query>

```

Then, it must be referenced in the mapping:

```

<class name="Item">
    ...
    <loader query-ref="LoadItem"/>
</class>

```

It is also possible to customize how a collection should be loaded. In this case, the named query will use the `<load-collection>` tag. Here is an example for the `Bids` collection of `Item`:

```

<sql-query name="LoadItemBids">
    <load-collection alias="bid" role="Item.Bids"/>
    SELECT {bid.*}
    FROM BID
    WHERE ITEM_ID = :id
</sql-query>

```

Here is how it is referenced:

```

<bag name="Bids" ...>
    ...
    <loader query-ref="LoadItemBids"/>
</bag>

```

When you're writing queries and testing them in your application, you may encounter one of the common performance issues with ORM. Fortunately, we know how to avoid (or, at least, limit) their impact. This process is called *optimizing object retrieval*. Let's walk through the most common issues.

7.7 *Optimizing object retrieval*

Performance-tuning your application should first include the most obvious settings, such as the best fetching strategies and use of proxies, as shown in chapter 4. Note that we consider enabling the second-level cache to be the last optimization you usually make.

The *fetch joins*, part of the *runtime fetching strategies*, as introduced in this chapter, deserve some extra attention. However, some design issues can't be resolved by tuning, but can only be avoided if possible.

7.7.1 *Solving the n+1 selects problem*

The biggest performance killer in applications that persist objects to SQL databases is the *n+1 selects problem*. When you tune the performance of a NHibernate application, this problem is the first thing you'll usually need to address.

It's normal (and recommended) to map almost all associations for lazy initialization. This means you generally set all collections to `lazy="true"` and even change some of the one-to-one and many-to-one associations to not use outer joins by default. This is the only way to avoid retrieving all objects in the database in every transaction. Unfortunately, this decision exposes you to the n+1 selects problem. It's easy to understand this problem by considering a simple query that retrieves all `Items` for a particular user:

```

IList<Item> results = session.CreateCriteria(typeof(Item))
    .Add( Expression.Eq("item.Seller", user) )
    .List<Item>();

```

This query returns a list of items, where each collection of bids is an uninitialized collection wrapper. Suppose that we now wish to find the maximum bid for each item. The following code would be one way to do this:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

IList maxAmounts = new ArrayList();
foreach( Item item in results ) {
    double maxAmount = 0;
    foreach ( Bid bid in item.Bids ) {
        if( maxAmount < bid.Amount )
            maxAmount = bid.Amount;
    }
    maxAmounts.Add( new MaxAmount( item.Id, maxAmount ) );
}

```

But there is a huge problem with this solution (aside from the fact that this would be much better executed in the database using aggregation functions): Each time we access the collection of bids, NHibernate must fetch this lazy collection from the database for each item. If the initial query returns 20 items, the entire transaction requires 1 initial `select` that retrieves the items plus 20 additional `selects` to load the `bids` collections of each item. This might easily result in unacceptable latency in a system that accesses the database across a network. Usually you don't explicitly create such operations, because you should quickly see doing so is suboptimal. However, the `n+1 selects` problem is often hidden in more complex application logic, and you may not recognize it by looking at a single routine.

The first attempt to solve this problem might be to enable *batch fetching*. We change our mapping for the `bids` collection to look like this:

```
<set name="bids" lazy="true" inverse="true" batch-size="10">
```

With batch fetching enabled, NHibernate prefetches the next 10 items when collection is first accessed. This reduces the problem from `n+1 selects` to `n/10 + 1 selects`. For many applications, this may be sufficient to achieve acceptable latency. On the other hand, it also means that in some other transactions, collections are fetched unnecessarily. It isn't the best we can do in terms of reducing the number of round trips to the database.

A much, much better solution is to take advantage of HQL aggregation and perform the work of calculating the maximum bid on the database. Thus we avoid the problem:

```

string query = @"select new MaxAmount( item.id, max(bid.Amount) )
                from Item item join item.Bids bid"
                where item.Seller = :user group by item.id";

IList maxAmounts = session.CreateQuery(query)
                    .SetEntity("user", user)
                    .List();

```

Unfortunately, this isn't a complete solution to the generic issue. In general, we may need to do more complex processing on the bids than merely calculating the maximum amount. We'd prefer to do this processing in the .NET application.

We can try enabling eager fetching at the level of the mapping document:

```
<set name="Bids" lazy="false" inverse="true" outer-join="true">
```

The `outer-join` attribute is available for collections and other associations. It forces NHibernate to load the association eagerly, using an SQL outer join. You may also use the `fetch` attribute; `fetch="select"` is equivalent to `outer-join="false"`, and `fetch="join"` is equivalent to `outer-join="true"`.

Note that, as previously mentioned, HQL queries ignore the `outer-join` attribute; but we might be using a criteria query.

This mapping avoids the problem as far as this transaction is concerned; we're now able to load all bids in the initial `select`. Unfortunately, any other transaction that retrieves items using `Get()`, `Load()`, or a criteria query will also retrieve all the bids at once. Retrieving unnecessary data imposes extra

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

load on both the database server and the application server and may also reduce the concurrency of the system, creating too many unnecessary read locks at the database level.

Hence we consider eager fetching at the level of the mapping file to be almost always a bad approach. The `outer-join` attribute of collection mappings is arguably a misfeature of NHibernate (fortunately, it's disabled by default). Occasionally it makes sense to enable `outer-join` for a `<many-to-one>` or `<one-to-one>` association (the default is `auto`; see chapter 4, section 4.4.6, "Single point associations"), but we'd never do this in the case of a collection.

Our recommended solution for this problem is to take advantage of NHibernate's support for runtime (code-level) declarations of association fetching strategies. The example can be implemented like this:

```
IList<Item> results = session.CreateCriteria(typeof(Item))
                        .Add( Expression.Eq("item.Seller", user) )
                        .SetFetchMode("Bids", FetchMode.Eager)
                        .List<Item>();

// Make results distinct
ISet<Item> distinctResults = new HashSet<Item>(results);

IList maxAmounts = new ArrayList();
foreach ( Item item in distinctResults ) {
    double maxAmount = 0;
    foreach ( Bid bid in item.Bids ) {
        if( maxAmount < bid.Amount )
            maxAmount = bid.Amount;
    }
    maxAmounts.Add( new MaxAmount( item.Id, maxAmount ) );
}
```

We disabled batch fetching and eager fetching at the mapping level; the collection is lazy by default. Instead, we enable eager fetching for this query alone by calling `SetFetchMode()`. As discussed earlier in this chapter, this is equivalent to a `fetch join` in the `from` clause of an HQL query.

The previous code example has one extra complication: The result list returned by the NHibernate criteria query isn't guaranteed to be distinct. In the case of a query that fetches a collection by outer join, it will contain duplicate items. It's the application's responsibility to make the results distinct if that is required. We implement this by adding the results to a `HashSet` (from the library `Java.Collections`) and then iterating the set.

So, we have established a general solution to the `n+1 selects` problem. Rather than retrieving just the top-level objects in the initial query and then fetching needed associations as the application navigates the object graph, we follow a two-step process:

- 1 Fetch all needed data in the initial query by specifying exactly which associations will be accessed in the following unit of work.
- 2 Navigate the object graph, which will consist entirely of objects that have already been fetched from the database.

This is the only true solution to the mismatch between the object-oriented world, where data is accessed by navigation, and the relational world, where data is accessed by joining.

Another efficient solution, for deep graphs of objects, is to issue one query per level and let NHibernate resolve the references between the objects. For example, we can query categories, asking NHibernate to fetch their items. Then, we can query these items, asking NHibernate to fetch their bids.

Finally, there is one further solution to the `n+1 selects` problem. For some classes or collections with a sufficiently small number of instances, it's possible to keep all instances in the second-level cache, avoiding the need for database access. Obviously, this solution is preferred where and when it's possible (it isn't possible in the case of the `Bids` of an `Item`, because we wouldn't enable caching for this kind of data).

The n+1 selects problem may appear whenever we use the `List()` method of `IQuery` to retrieve the result. As we mentioned earlier, this issue can be hidden in more complex logic; we highly recommend the optimization strategies mentioned in chapter 4, section 4.4.7, “Tuning object retrieval” to find such scenarios. It’s also possible to generate too many selects by using `Find()`, the shortcut for queries on the `ISession` API, or `Load()` and `Get()`.

There is a third query API method we haven’t discussed yet. It’s extremely important to understand when it’s applicable, because it produces n+1 selects!

7.7.2 Using `Enumerable()` queries

The `Enumerable()` method of the `ISession` and `IQuery` interfaces behaves differently than the `Find()` and `List()` methods. It’s provided specifically to let you take full advantage of the second-level cache.

Consider the following code:

```
IQuery categoryByName =
    session.CreateQuery("from Category c where c.Name like :name");
categoryByName.SetString("name", categoryNamePattern);
IList categories = categoryByName.List();
```

This query results in execution of an SQL `select`, with all columns of the `CATEGORY` table included in the `select` clause:

```
select CATEGORY_ID, NAME, PARENT_ID from CATEGORY where NAME like ?
```

If we expect that categories are already cached in the session or second-level cache, then we only need the identifier value (the key to the cache). This will reduce the amount of data we have to fetch from the database. The following SQL would be slightly more efficient:

```
select CATEGORY_ID from CATEGORY where NAME like ?
```

We can use the `Enumerable()` method:

```
IQuery categoryByName =
    session.CreateQuery("from Category c where c.Name like :name");
categoryByName.SetString("name", categoryNamePattern);
IEnumerable<Category> categories = categoryByName.Enumerable<Category>();
```

The initial query only retrieves the category primary key values. We then iterate through the result, and `NHibernate` looks up each `Category` in the current session and in the second-level cache. If a cache miss occurs, `NHibernate` executes an additional `select`, retrieving the category by its primary key from the database.

In most cases, this is a minor optimization. It’s usually much more important to minimize *row* reads than to minimize *column* reads. Still, if your object has large string fields, this technique may be useful to minimize data packets on the network and, therefore, latency.

Let’s talk about another optimization, which also isn’t applicable in every case. So far, we’ve only discussed caching the results of a lookup by identifier (including implicit lookups, such as loading a lazy association) in chapter 5. It’s also possible to cache the results of `NHibernate` queries.

7.7.3 Caching queries

For applications that perform many queries and few inserts, deletes, or updates, caching queries can have an impact on performance. However, if the application performs many writes, the query cache won’t be utilized efficiently. `NHibernate` expires a cached query result set when there is *any* insert, update, or delete of any row of a table that appears in the query.

Just as not all classes or collections should be cached, not all queries should be cached or will benefit from caching. For example, if a search screen has many different search criteria, then it will

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

not happen often enough that the user chooses the same criterion many times. In this case, the cached query results will be underused, and we'd be better off not enabling caching for that query.

Note that the query cache does *not* cache the entities returned in the query result set, just the identifier values. NHibernate will, however, fully cache the value typed data returned by a projection query. For example, the projection query "select u, b.Created from User u, Bid b where b.Bidder = u" will result in caching of the identifiers of the users and the date object when they made their bids. It's the responsibility of the second-level cache (in conjunction with the session cache) to cache the actual state of entities. So, if the cached query you just saw is executed again, NHibernate will have the bid-creation dates in the query cache but perform a lookup in the session and second-level cache (or even execute SQL again) for each user that was in the result. This is similar to the lookup strategy of `Enumerable()`, as explained in the previous section.

The query cache must be enabled using, for example:

```
<add key="hibernate.cache.use_query_cache" value="true" />
```

However, this setting alone isn't enough for NHibernate to cache query results. By default, NHibernate queries always ignore the cache. To enable query caching for a particular query (to allow its results to be added to the cache, and to allow it to draw its results *from* the cache), you use the `IQuery` interface:

```
IQuery categoryByName =
    session.CreateQuery("from Category c where c.Name = :name");
categoryByName.SetString("name", categoryName);
categoryByName.SetCacheable(true);
```

Even this doesn't give you sufficient granularity, however. Different queries may require different query expiration policies. NHibernate allows you to specify a different named cache *region* for each query:

```
IQuery userByName =
    session.CreateQuery("from User u where u.Username= :uname");
userByName.SetString("uname", username);
userByName.SetCacheable(true);
userByName.SetCacheRegion("UserQueries");
```

You can now configure the cache expiration policies using the region name. When query caching is enabled, the cache regions are as follows:

- n The default query cache region, `null`
- n Each named region
- n The *timestamp cache*, `NHibernate.Cache.UpdateTimestampsCache`, which is a special region that holds timestamps of the most recent updates to each table

NHibernate uses the timestamp cache to decide if a cached query result set is stale. NHibernate looks in the timestamp cache for the timestamp of the most recent insert, update, or delete made to the queried table. If it's later than the timestamp of the cached query results, then the cached results are discarded and a new query is issued. For best results, you should configure the timestamp cache so that the update timestamp for a table doesn't expire from the cache while queries against the table are still cached in one of the other regions. The easiest way is to turn off expiry for the timestamp cache.

Some final words about performance optimization: Remember that issues like the `n+1` selects problem can slow your application to unacceptable performance. Try to avoid the problem by using the best fetching strategy. Verify that your object-retrieval technique is the best for your use case before you look into caching anything.

From our point of view, caching at the second level is an important feature, but it isn't the first option when optimizing performance. Errors in the design of queries or an unnecessarily complex part of your object model can't be improved with a "cache it all" approach. If an application only performs

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

at an acceptable level with a *hot cache* (a full cache) after several hours or days of runtime, you should check it for serious design mistakes, unperformant queries, and *n+1 selects* problems.

7.7.4 Using profilers and NHibernate Query Analyzer

In most cases, there are many ways to write a query; and it may be hard to select the optimal. Profiler tools can help testing the performance of these options. So, use them as often as possible.

When working with HQL queries, you may also wonder if the generated SQL is optimal. There is a tool called NHibernate Query Analyzer which allows you to dynamically write and execute queries on your domain model. It displays the generated SQL query in real-time and display the result of your query when you execute it. So it is also very helpful when learning HQL.

For more details, refer to the documentation on its website: <http://www.ayende.com/projects/nhibernate-query-analyzer.aspx>

Note that some HQL queries in this chapter will not work with the released version of CaveatEmptor because we assumed a different kind of mapping was used in their specific cases in order to illustrate specific functionalities.

7.8. Summary

We don't expect that you know everything about HQL and criteria after reading this chapter once. However, the chapter will be useful as a reference in your daily work with NHibernate, and we encourage you to come back and reread sections whenever you need to.

The code examples in this chapter show the three basic NHibernate query techniques: HQL, Query by Criteria (including a Query by Example mechanism), and direct execution of database-specific SQL queries.

We consider HQL the most powerful method. HQL queries are easy to understand, and they use persistent class and property names instead of table and column names. HQL is polymorphic: You can retrieve all objects with a given interface by querying for that interface. With HQL, you have the full power of arbitrary restrictions and projection of results, with logical operators and function calls just as in SQL, but always on the object level using class and property names. You can use named parameters to bind query arguments in a secure and type-safe way. Report-style queries are also supported, and this is an important area where other ORM solutions usually lack features.

Most of this is also true for criteria-based queries; but instead of using a query string, you use a typesafe API to construct the query. If you use refactoring tools like Resharper, you also benefit from your criteria queries being considered in refactorings. So-called *example objects* can be combined with criteria—for example, to retrieve “all items that look like the given example.”

The most important part of object retrieval is the efficient loading of associated objects—that is, how you define the part of the object graph you'd like to load from the database in one operation. NHibernate provides lazy, eager, and batch fetching strategies, in mapping metadata and dynamically at runtime. You can use association joins and result iteration to prevent common problems such as the *n+1 selects problem*. Your goal is to minimize database roundtrips with many small queries, but at the same time, you also try to minimize the amount of data loaded in one query.

The best query and the ideal object-retrieval strategy depend on your use case, but you should be well prepared with the examples in this chapter and NHibernate's excellent runtime fetching strategies.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>



Developing NHibernate Applications

This chapter covers

- n Implementing layered applications
- n Solving issues when setting up .NET applications using NHibernate
- n Achieving design goals
- n Solving debugging and performance problems
- n Integrating services like audit logging

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

At this point, you may be thinking " I know all about NHibernate features, but how do I fit them all together to build a full NHibernate application?" It's time for us answer that question, and to apply all the knowledge we've gained to implement applications as part of a real world development process.

If you recall, we discussed the desired architecture of a NHibernate application in chapter 1, section 1.1.3, "Layered Architecture". This provided a great birds-eye view of things, but of course we need to somehow get from that point to working executable code.

We'll discuss the development process layer by layer, showing the internals of each layer, and how each should handle their designated responsibilities, and how they communicate with each other.

Because this book focuses on NHibernate, the domain model and persistence layers draw most of our attention. However, using NHibernate will require design decisions throughout all application layers, so we'll be sure to give detail in these where we feel it will help.

The complexity of defining and building your layered application depends on the complexity of the problem in hand. For instance, our "Hello World" application in chapter 2 would be relatively trivial for you. However, building an application as complex as the CaveatEmptor example might prove slightly more challenging! By following the advice given in this chapter, you should be able to find your way through any difficult patches. Note that we will present many ideas and patterns, and it is important that you understand their pros and cons so you can make the right decisions in the right places.

The first part of this chapter focuses on the implementation of a NHibernate application. First, we will rediscover the classic architecture of a NHibernate application. We will talk about its layers, their purposes and briefly about their implementations. After that, we will quickly talk about the development of some kind of .NET applications; we will deal with the issues that you may encounter when starting to write the application or when deploying it. Finally, we will discuss about problem solving in the context of NHibernate applications. This part also serves as a map for the two next chapters. It will provide references to upcoming sections for more details.

The second part of this chapter is about services: How to integrate loosely coupled components to a NHibernate application. You will learn how to use the `IInterceptor` interface to efficiently integrate services like audit logging. We will also briefly talk about some other alternatives.

Let's start at the beginning with the architecture of a NHibernate application and its implementation.

8.1 Inside the layers of a NHibernate application

Chapter 1 presented an overview of the layered architecture of a NHibernate application, but you probably still have many questions about its implementation. These final chapters should give you the answers you need. Note that this section will often direct you to the following chapters for more detailed explanations.

Let's review the importance of disciplined application layering that we emphasized in the first chapters of this book. Layering helps you achieve separation of concerns, making code more readable by grouping code that does similar things. On the other hand, layering carries a price: Each extra layer increases the amount of code it takes to implement a simple piece of functionality—and more code makes the functionality itself more difficult to change.

We won't try to form any conclusions about the right number of layers to use (and certainly not about what those layers should be) since the "best" design varies from application to application and a complete discussion of application architecture is well outside the scope of this book. We merely observe that, in our opinion, a layer should only exist if it is required, as it increases the complexity and costs of the development.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

In this section, we go through the layers introduced in chapter 1, explaining their roles and discussing their implementations. That way, you will progressively learn how to develop a NHibernate application.

These layers are:

- n The business layer (with the domain model)
- n The persistence layer
- n The presentation layer

Looking at these layers, we realize that only two of them matter to the end-user: The business layer matters because it embodies the problem the application is supposed to solve. The presentation layer matters to the user because it allows them to issue commands to the application and see the results.

The persistence layer is something the user doesn't really care about directly, but which is obviously essential to most applications. It's useful to remember that persistence is a *service* that is used by the application (like those presented in section 9.4), and which is there to permit loading and saving of an application's data. Keeping this viewpoint in-mind will help you achieve a good separation of concerns.

If you think carefully about the implementation of these layers, you will see that they all exist to augment the domain model in some way. This is why we say that the development process of a NHibernate application is *domain-centric*, and this approach is called Domain-Driven Development (DDD). Because testing is part of the development process, we also talk about testing these layers and see how we can apply Test-Driven Development (TDD).

Before going any further, an introduction to patterns, DDD and TDD is required.

8.1.1 *Patterns and methodologies*

The breadth of this chapter requires us to mention many patterns and practices related to software development. We will give a brief introduction to each as we discuss them, although if you find you're in unfamiliar territory with we'd encourage you to follow any references we give to get a deeper understanding of the topics.

In case you haven't studied patterns before, here's the general idea. Many problems, although different in their formulation, are solved in a similar way. So, it is possible to formulate an abstract solution that can be applied to solve these problems. Therefore, a pattern is a description or template recommending a solution to a recurring problem. Hundreds of patterns have been discovered and documented over the years, and we'll give references to those books and articles we consider to be most important.

One of the most famous and popular patterns is the *Singleton pattern*, have you heard of it? This pattern solves problems where only one instance of a class should be available to a system, and how that instance can be globally accessible. Another pattern we've discussed already is "Domain model" pattern, which pattern describes a way to capture the behavior and data of a system using collaborating classes, rather than record sets or some other device.

There are many benefits in learning patterns: They are well tested, time proven methods that let you leverage the experience of other professionals. They also give us common vocabulary for efficiently communicating aspects of our software with others. We'll now look at a pattern that is gaining increasing popularity in mainstream software development, and which is highly applicable to NHibernate – the Model View Controller pattern.

The MVC pattern

Model / View / Controller (MVC) is a popular pattern commonly used in both web and desktop applications. It prescribes separating application concerns into three categories: Model, View and Controller. This approach is illustrated in figure 8.1.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

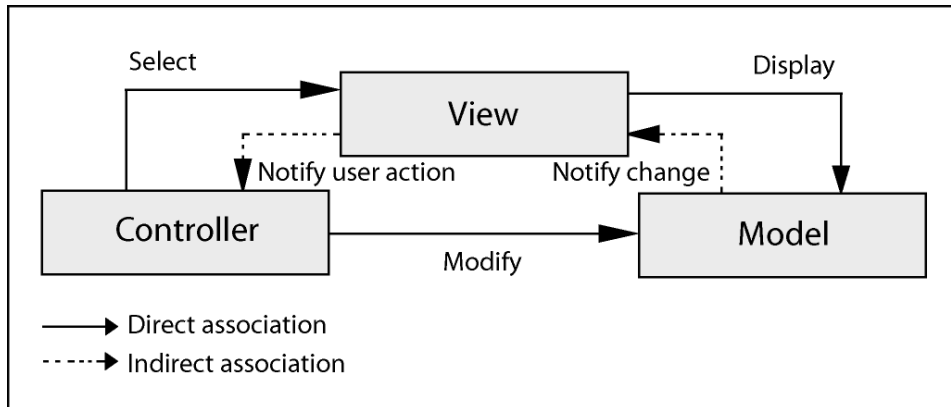


Figure 8.1 Model / View / Controller pattern

Let's look more closely at the key parts of this figure, starting with the View. The View is all about displaying data to the user, and collecting their input (mouse clicks, data entry etc). The View could be a web page, a windows form, or a speech-driven audio menu like those found in telephone systems. A view gets its data from one or more models, so essentially a view's purpose in life is to display models and receive user input. In many cases, these models are the domain objects in your system, which we've talked about many times throughout this book. So where does the controller fit into all this? Well, when the user interacts with the view, the view would notify the controller about those interactions. It might tell the controller about a mouse click, a drop-down-list selection, or some data entry. The controller is the coordinating brain between user and domain model. In many cases, the controller could simply issue updates to the model, setting properties and calling methods.

The direct associations define the dependencies between these components; the Controller knows about the View and the Model, the View only directly knows about the Model, and the Model is totally independent – it doesn't know about any controllers or views. The indirect (dotted line) associations indicate that a component might communicate with another component without directly depending on it. In .NET, events give us this capability, and this gives us a flavor of the *Observer pattern*. For more details of that, [read chapter 10](#), section 10.3.1.

When applying the MVC pattern to a NHibernate application, the Model is represented by the entities; the View is represented by the user interface displaying the Model (in a Web application, it is the web page). And, the Controller is the brain; it uses user inputs to process the Model and ask the View to display the result.

In .NET applications, it is common to spread the Controller in many layers (the presentation and the business layer). It requires some discipline to avoid putting everything in the code-behind of the web page or windows form; however, it is worth the effort because it increases the reusability and testability of your code. The presentation layer must only contain code related to formatting, displaying and retrieving information. If you're aware of the Microsoft MVC project, you may realize that it aims to provide many of these benefits.

Domain-Driven Development

Domain-Driven Development (DDD) is a popular approach to evolving rich domain models.

The goal of DDD is to allow the customers and developers to work together with minimal friction. Developers and customers establish a common language that they use to communicate concepts in business-level conversations, and furthermore this very same language is used to portray concepts in the software code. Therefore, the language of the software and the business are the same, leaving little room for ambiguity.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Having the end-user and the developer working together speeds the roundtrip between the decision making and the implementation.

Practically speaking, the development process is centered on the domain model. It is written first to make sure that the business is correctly understood and represented. The other components and layers are implemented and adapted based on the evolution of the domain model.

All NHibernate applications somehow follow this approach; even the “Hello World” application, in chapter 2, does. Therefore, we encourage you to learn more about this methodology because it will help you building NHibernate applications. To learn more about DDD, refer to the books Domain-Driven Design [Evans 2004] and Applying Domain-Driven Design and Patterns [Nilsson 2006].

Test-Driven Development

Although we tend to focus on the implementing business logic and , testing must also be part of the development process. Test-Driven Development (TDD) goes as far as to make the first step, even before the implementation. Applying TDD means thinking of a solution to a problem, writing the tests of this solution, and then implementing the solution, making the tests pass. TDD is not popular with new comers; however, once you are used to the process, you see the benefits in the short and long run.

You may only be familiar with the classic application testing for quality assurance where you build a feature, with its UI, and then you test this feature using the UI. Although this method is important, it is not enough in complex applications. Can you test the whole UI each time you change something in the domain model? Are you sure that you will test it correctly and not miss a hole somewhere?

Here comes unit testing. As you can guess, it is about testing each single *unit* (generally a class). Note that the tests are automated. This means that you can run them literally just by pressing on a button and drinking your coffee while the computer works for you. And using a domain model makes this approach much simpler (due to the separation of concerns).

The .NET framework has some popular testing libraries, the most popular being the NUnit (<http://nunit.org/>). This chapter will briefly introduce you to the unit testing of the domain model and the business layer using this library.

Like DDD, This methodology is an Agile Software Development practice (http://en.wikipedia.org/wiki/agile_software_development). It is out of the scope of this book to dig in these topics. For more details about NUnit and TDD, read Test-Driven Development in Microsoft .NET [Newkirk et al 2004]. You may also take a look at tools like ReSharper (<http://www.jetbrains.com/resharper/>) which help a lot when applying TDD.

You can learn more about patterns in the books Patterns of Enterprise Application Architecture [Fowler 2003] and Design Patterns [Gamma et al 1995].

With these methodologies and patterns in mind, let’s see how we can develop a layer of an application.

8.1.2 *Development process of a layer*

A clear separation of concerns allows us to develop the layers of an application almost independently. However, as the layered architecture points out, there is still an order in which the layer should be implemented (based on the dependencies). And that’s why we generally start with the domain model; hence, applying DDD.

In the following sections, we will go through the development of each layer from the domain model (which has no dependency) to the presentation layer (which is the top layer depending on all others).

There are two parts in this process: the implementation and the testing. When talking about implementing a layer, we will see what this layer is made of and how its elements can be written. After that, we give you an idea of how these elements can be tested. A good understanding of the implementation is required in order to test it; that’s why the testing comes after. Once you understand this process, you are free to apply TDD (write the test first, then the implementation).

Figure 8.2 illustrates the development process that we are following.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

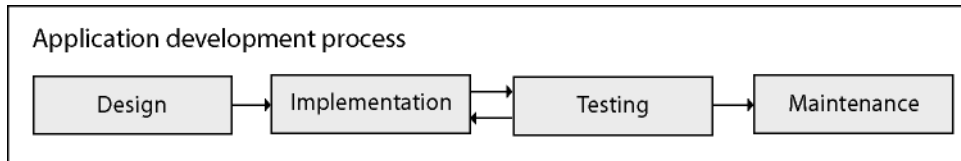


Figure 8.2 Simplistic representation of an application development process

This figure shows that we first design an application; then we implement and test it. These steps are done in a loop (like when applying TDD). Once the application is done, the maintenance starts. Note that, unlike in this figure, the development process of a real-world application generally has a big loop between the design and the implementation/testing. The reason is that the requirements of the application evolve as it is developed.

The design of a NHibernate application has been discussed since chapter 1 and some additions are made in section 9.3.1. We will talk about its implementation (and testing) in this section and in the following chapters. And we talk about the maintenance in section 8.3.2.

Now, we can start talking about the implementation and testing of the heart of the application: The domain model.

8.1.3 The domain model

The domain model is commonly perceived as part of the business layer. However, we will discuss them separately. The domain model is the heart of the business. It tells what the business is doing (the domain), and represents the entities manipulated in the domain by the business (the model).

It is conceptually independent of any other layer, even of the business layer, as it doesn't use any class that isn't itself part of the domain model. You should already be familiar with it as we have been implementing entities since the chapter 2.

The CaveatEmptor application has a more complex domain model as illustrated in chapter 4, figure 4.2. Its implementation requires nine classes for the entities and even more for the other components.

Implementation

In the context of enterprise application development, implementing a domain model may not be as simple as it looks. This section will only enumerate the steps of this process along with some of the problems you may encounter; we will deeply cover it in chapter 10.

First, you may write it from scratch, or you can generate it (from the mapping or from the database). In this case, you may have to deal with a legacy database. Although a legacy database with a weird design can make this step complex, it is generally easy enough; you just have to write a set of classes with constructors and properties.

Then, implementing the behavior and rules of the entities (the business logic) can be messy if you don't do it correctly. You must be careful when you choose where and how you implement them in order to avoid unnecessary constraints.

As if it wasn't enough, the other layers may still have an influence on the domain model: It may not be completely persistence ignorant or it may implement some mechanisms, for example, to ease the data-binding on the presentation layer.

Finally, you may have to communicate with other components/services/applications which can not manipulate your domain model as-is; in .NET applications, it is quite common to have a specific set of classes (*Data Transfer Objects* or DataSet) as the communication medium. In this case, you must convert your entities to enable this communication; and this conversion can be quite troublesome. Would you give up your domain model to avoid this problem?

All these problems require a good amount of thinking and experimentation. Thankfully, chapter 10 provides you with a wide range of options to solve them.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Testing

If your previous applications didn't have any test for your domain model, you should really reconsider your position. Remember, your domain model is the heart of your software and, unlike the presentation layer, breaking it may not be easily visible.

Note that, when unit testing your domain model, the units are the entities.

There are globally two kinds of tests: Data integrity tests and logic tests. Let's take a simple example for each of them and see how we can do it using NUnit. To make it even more fun, we will apply TDD.

Data integrity tests are simple tests to make sure that the content of the entities are valid (according to the business logic) and stay valid, no matter what happens.

For example, if you have a `User` entity in your domain model, its name will be mandatory and, because it is stored in the database, this name must not exceed a number of characters (let's say 64).

We have written three tests to codify this specification in listing 8.1.

Listing 8.1 Unit testing an entity

```
using NUnit.Framework;

[TestFixture]
public class UserFixture {

    [Test]
    public void WorkingName() {                               |1
        string random = new Random().Next().ToString();
        User u = new User();
        u.Name = random;

        Assert.AreEqual(random, u.Name);
    }

    [Test, ExpectedException( typeof(BusinessException) )]
    public void NotNullableName() {                           |2
        User u = new User();
        u.Name = null;
    }

    [Test, ExpectedException( typeof(BusinessException) )]
    public void TooLongName() {                               |3
        User u = new User();
        u.Name = "".PadLeft(65, 'x');
    }
}
```

#1 There is a property Name keeping its value

#2 The name can not be null

#3 The name must not exceed 64 characters

Tests are grouped in public classes called *fixtures*. NUnit uses attributes to identify them, this is why this class is decorated with the attribute `[TestFixture]`. Tests are methods marked using `[Test]`. Note that these methods must be public and must take no parameters.

The first test makes sure that there is a property called `Name` in the class `User` that keeps the value we assign to it. We use a random value to avoid cheats. The actual test is done using `Assert.AreEqual(...)`; The class `Assert` belongs to NUnit and provides a wide range of methods for various testing.

The two other tests use a different approach. They try to set invalid values to the property `Name` and expect it to throw a `BusinessException`. If it doesn't, the test fails. As explained in chapter 10, section 10.4.2, you may allow invalid values and validate changed entities before saving them.

Remember that this class should stay in a separated test library.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Now, we can move on to the implementation of the part of the class `User` that will make these three tests pass.

First, we have to create the class `User` and its property `Name`, then we use a field `_name` to keep the value assigned to this property. At this point, the first test passes.

After that, we add a validation to make the two other tests pass. Here is the final result:

```
public class User {
    private string _name;

    public string Name {
        get { return _name; }
        set {
            if( string.IsNullOrEmpty(value) || value.Length>64 )
                throw new BusinessException("Invalid name");
            _name = value;
        }
    }
}
```

This code is quite easy to understand: in the setter of the property `Name`, we throw an exception if the name that is about to be set is invalid (null, empty or too long); then we keep this value in the field `_name`.

The second kind of tests for an entity is the logic test which tests any behavior in the entity. For example, when changing his password, the user must provide the old one and twice the new one.

Here are some tests for this method: (we assume that a newly created user has a blank password and that there is no encryption)

```
[Test]
public void WorkingPassword() {
    string random = new Random().Next().ToString();
    User u = new User();
    u.ChangePassword("", random, random);

    Assert.AreEqual(random, u.Password);
}

[Test, ExpectedException( typeof(BusinessException) )]
public void BadOldPassword() {
    User u = new User();
    u.ChangePassword("?", "", "");
}

[Test, ExpectedException( typeof(BusinessException) )]
public void DifferentNewPasswords() {
    User u = new User();
    u.ChangePassword("", "x", "y");
}
```

There is little to explain. The first test make sure that we can successfully change the password and the two others make sure that we can't change the password with an invalid old password or a new password different to the confirmation password.

Here is an implementation of the method `ChangePassword()` that makes these tests pass:

```
public void ChangePassword(string oldPwd, string newPwd, string confirmPwd) {
    if( (_password != oldPwd) || (newPwd != confirmPwd) )
        throw new BusinessException("...");
    _password = newPwd;
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note that you should probably separate the two validations to provide meaningful messages in the thrown exception. And, in the real world, you should move these validations to the business layer like it is done in the next section.

What you must remember is that a domain model can (should) be efficiently tested. Its autonomy makes it very easy not only to implement, but also to test.

Let's continue with the other classes forming the business aspect of the application.

8.1.4 The business layer

The business layer acts as a *gateway* that upper layers (like the presentation layer) must use to manipulate the entities.

It is common to see .NET applications (mainly those using `DataSet`) directly accessing the database. You already know that it is wrong and why (if you forgot, go back reading chapter 1, section 1.2, "Implementing persistence in .NET").

The business layer plays several roles: It is a layer on top of the persistence layer. It performs high-level business logic (that can not be performed by the entities themselves). It may also integrate services like security and audit logging.

The Controller (of the MVC pattern) can also be perceived as being part of this layer. It pilots the flow of information between the end-user (through the View) and the Model. However, it is a good practice to keep the controllers as a thin layer on top of the *core* of the business layer.

Implementation

Depending of the coupling between the entities, you can write one business class to manage each entity or for one for many entities. You may also have business classes for some use cases / scenarios.

When implementing the CRUD-like operations, try not to mimic the persistence layer. At this level, save and update are not business words. It is actually easy to find the right words when seeing the problem from a business perspective. For example, when we place a bid on an item, although we are saving this bid in the database (technically speaking), we shouldn't call the method performing this operation `SaveBid()`; in this case, `PlaceBid()` is more expressive. Obviously, `PlaceBid()` (in the business layer) will send the bid for persistence using a method from the persistence layer that can be called `Save()` (or `MakePersistent()` as explained in chapter 11, section 11.1).

Let's change our previous example to illustrate a piece of the business layer. What if we want to allow administrators to change users' passwords (without knowing their current ones)?

In this case, the class `User` can not perform this logic because it doesn't know which user is currently logged; unless you make this information available using, for example, the Singleton pattern (however, this may not be a good idea).

Here is how the method `ChangePassword()` can be implemented: (now, this method belongs to a class in the business layer)

```
public void ChangePassword( User u,
    string oldPwd, string newPwd, string confirmPwd ) {

    if( u != null )
        throw new ArgumentNullException("u");

    if( LoggedUser == null )
        throw new BusinessException("Must be logged");

    if( ( ! LoggedUser.IsAdministrator && u.Password != oldPwd )
        || ( newPwd != confirmPwd ) )
        throw new BusinessException("...");

    u.Password = newPwd;

    UserPersister.Save(u); // Persistence layer
}
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

There are many validations in this method; however, note that we don't validate the new password; this is up to the property `User.Password`. By the way, you can make the setter of this property internal (providing the domain model and the business layer are in the same library) to make sure that the presentation layer can't change the password directly.

Note that, instead of receiving an instance of the class `User`, this method may receive the user's identifier and load it internally. The advantages are that the business layer is sure that nothing else has changed in the user and that it may be easier for the presentation layer to provide an identifier (for example, because it doesn't keep the user instance). However, it is less object-oriented.

About the query methods, unless you are writing a heavily customizable search engine UI, you should provide methods for all kind of queries that the end-user can run. Do not let the presentation layer build arbitrary queries. It makes the presentation layer aware of the persistence layer, it may become a security issue and it makes the application less testable. On the other hand, you should use an extensible approach to avoid having to create too many methods.

You may also include some code for audit logging to keep track of what happened and who did it (invaluable when debugging an application in production). However, you will learn in section 9.4 another way to deal with this kind of services.

We talk about the implementation of the business logic in the chapter 10, section 10.4. We can't really talk about the general implementation because it heavily depends on the application. Just make sure that you cleanly separate it from the other layers. You may, for example, put the business classes in a `Business` namespace and the controllers in a `Controllers` namespace.

We also can't really give much detail about the implementation of controllers because they tend to be very platform-dependent. You will discover, in appendix C, a library which allows cleanly writing controllers.

Testing

As you would guess, testing the business layer is crucial. Practically, it is very similar to testing the domain model. If you understood unit testing as illustrated in listing 9.1, you should be able to easily test the previous method `ChangePassword()`.

However, because this layer uses the persistence layer, it may become troublesome in some places. Some complex business logic may require a specific persistence strategy. These borderline scenarios require a compromise between the separation of concerns and the ease of implementation and testing.

The tests for the business layer should only be related to the business layer itself. You can take a look at the testing the persistence layer in the next section to see what must not be tested here. Even the tests for the domain model's entities should be separated.

8.1.4 The persistence layer

The persistence layer provides CRUD methods for entities. Thanks to NHibernate, it can be implemented as a service (that is, non intrusive) for the domain model.

However, some libraries like ActiveRecord (Cf. appendix C) merge it in the domain model. We generally see that as a bad practice; however, the reason behind this design choice is simplicity: When writing a new application that isn't very complex (in term of layers coupling and integration issues), having a persistence ignorant domain model isn't necessarily a requirement.

Anyway, we recommend that you build the persistence layer separately and that you hide it behind the business layer. But we agree that it depends of your programming style.

Implementation

The general practice, when implementing the persistence layer, is to write one persistence class per entity, commonly called `EntityNameDAO`. DAO stands for Data Access Object. It is a well known

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

pattern to implement persistence layers. In the previous code listing, `UserPersister` could have been called `UserDAO`.

Persistence classes have methods for simple CRUD operations and can allow the business layer to execute custom queries. We explained this approach in chapter 11, section 11.1.2.

The presentation layer (and other upper layers) should not be able to access this layer, unless the application is simple, in which case the business layer is not required (however, beware of "simple" applications evolving and becoming very complex). If this layer is in the same library as the business layer, its classes can be made internal. Else, simply avoid adding a reference in the presentation library to the persistence library.

Also note that it should abstract database semantic (unless for performance reasons). The reason is that the persistence layer is between the business layer and the database, hence, it should hide the database to the business layer.

Testing

As the domain model, there are two ways to test the persistence layer. You can test the correctness of the mapping between the domain model and the database, and you can also test the *persistence logic*.

Testing the mapping means making sure that the NHibernate mapping is correctly written and those entities are correctly loaded and persisted. Practically, it implies saving an entity with a random content, loading it and making sure that the content hasn't changed.

When testing the *persistence logic*, the logic is represented by any code customizing the way NHibernate works. Example: The queries (the where clause, the ordering, etc.). The idea is to make sure you get the data as intended.

However, avoid testing NHibernate itself. There are NHibernate tests for that. Let's take this test as example:

```
session1.Save( entity );  
Assert.IsNotNull( session2.Get<Entity>(id) );
```

This test is useless because if `Save()` succeeded, then the entity was saved. You will have a hard time if you don't trust NHibernate and all the libraries you use to do their job. On the other hand, you may make sure that the proper lazy loading and cascading options are enabled.

By the way, in case you didn't notice, we use two different sessions (`session1` and `session2`) because `session1.Get<Entity>(id)` will not hit the database; it will use its (first-level) cache instead. If creating two sessions in a test is too costly for you, you can either use `session1.Clear()` or provide your own database connection by calling `sessionFactory.OpenSession(yourDbConnection)`. For more details about the caches, read chapter 6, section 6.3.

Note that persistence tests are slower than other tests because of the cost of using a database; although you can speed them up using in-memory capable RDBMSs like SQLite. It is also possible to mock the database; however, the tests may become less meaningful. Mocking is a technique allowing faking a component to avoid its dependencies. For more details, read http://en.wikipedia.org/wiki/Mock_object.

As the persistence layer is hidden behind the business layer, you may actually test the persistence layer through the business layer. Although it clutters the business layer tests with "unrelated" tests, it may be acceptable for simple cases.

8.1.6 The presentation layer

The presentation layer is basically the User Interface (and its code-behind). It serves as a bridge between the end-user and the business layer.

Its primary work is to format and display information (the entities); it also receives commands and information from the end-user to send them to the business layer (or the Controller).

Implementation

Implementing the presentation layer is largely outside the scope of this book. However, using NHibernate may have some consequences on it. Read the next section for more details about deployment issues of .NET Applications using NHibernate.

From the domain model point of view, the biggest issue is displaying and retrieving entities. .NET provides some powerful data-binding mechanisms that may be hard to leverage with a domain model. We show in chapter 10, section 10.5, that there are many alternatives to data-bind entities.

In order to show how a typical command-handling method can look like, we will implement the method that can be called when the end-user click on the button to change a password.

```
private void btnChangePassword_Click(object sender, EventArgs e) {
    try {
        Business.ChangePassword( editedUser,
            editOldPwd.Text, editNewPwd.Text, editConfirmPwd.Text );

        MessageBox.Show("Success!");
        // Or go to the success page
    }
    catch(Exception ex) {
        MessageBox.Show("Failed: " + ex.Message);
        // Or go to the failed page
    }
}
```

This method simply sends the information to the business layer and displays a message after. Note that this method plays the role of the Controller. Strictly speaking, it should just call the Controller which will then do exactly what is in this method. This is an example of situations where the code-behind is used to implement the controller logic which means that the controller is not part of the business layer, but is merged in the presentation layer.

We use the generic name `Business` for the business class because its name can widely vary depending on the way you organize your business layer.

By the way, you should probably log errors; it may save you a lot of work when trying to figure out what happened in an application in production.

Testing

The presentation layer is the hardest to test automatically. The reason is that it is something inherently visual; hence the common way to test it is by running the application and seeing if it works.

On the other hand, if you write your application like described in this chapter, the persistence layer should consist of the design code (HTML for Web applications) and a thin code-behind for formatting and data-binding. This code can be easily tested visually: it isn't complex and it is harder to break.

Note that there are some techniques and libraries to test the presentation layer. However, we will not cover them. For more details, start by reading http://en.wikipedia.org/wiki/List_of_GUI_testing_tools.

When implementing these layers, you may encounter some issues when trying to make your NHibernate application work with some .NET features; let's see how you can solve them.

8.2 Solving issues related to .NET features

Applications using NHibernate tend to use some .NET features that can be troublesome because of some constraints in the way they work or because of security restrictions.

In this section, we provide talk about two specific features: working in a web environment and using .NET remoting.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

8.2.1 Web application

Web applications are much more restricted than Windows applications. They have a specific structure and they must follow many rules. This section aims to give you guidelines on how you should develop an application with a Web interface.

First, let's get started with Web application development, then we will talk about a specific security issue.

Quick Start

You shouldn't have major issues starting an application using Web pages as presentation layer. In order to achieve a good separation of concerns, the other layers must not be in the Web application; practically speaking, it means that the code-behind of pages and `App_Code` section must only contain the presentation layer logic (that is, formatting, displaying, retrieving information and calling the other layers). And the other layers can be in one library (or many, if necessary). Note that this is also quite true for Windows applications.

There are two common issues when implementing the presentation layer of a NHibernate application: Data-binding the entities (and their collections) and managing the session to efficiently persist these entities. We talk about data-binding in chapter 10, section 10.5 and session management in chapter 11.

Code access security

The .NET framework has a powerful security policy. It allows, for example, Web administrators to limit the amount of permissions given to assemblies based on their provenance. This is required because Web applications are generally accessible by a lot of uncontrollable persons; hence they must be carefully configured to avoid security issues.

Web servers are often configured for medium-trust. So if you intend to deploy your application in a public hosting service, you may have some issues getting it to work.

The medium-trust policy enforces some restrictions that affect NHibernate applications: You can't use reflection to access private members of a class and you can't use proxies because they can't be generated due to tightened security.

Therefore, you must map your database to public fields/properties, you must turn off lazy loading by setting `Lazy=false` in the mapping of each class and you must turn off the reflection optimizer. Read chapter 3, sections 3.5.3 and 3.5.4 for more details.

In the case that your NHibernate configuration properties are in `Web.config`, you must add an extra attribute to the declaration of its section:

```
<section name="nhibernate" type="..." requirePermission="false" />
```

Setting the attribute `requirePermission` to `false` allows NHibernate to read this section when loading the configuration.

8.2.2 .NET Remoting

Many NHibernate applications use .NET Remoting to make the business (or persistence) layer accessible across a network.

This is mainly the case for Windows applications. In this scenario, most layers can be executed on the client's computer, except the persistence layer which is executed on the server. That way, the database access can be restricted so that only the server knows how to access it and the client communicates with the server using, for example, a layer of *marshal-by-reference* objects.

The domain model must be serializable. As NHibernate proxies are not serializable, it is also required to return entities with fully initialized associations. In some edge cases, you may consider using data transfer objects. For more details, read chapter 11, section 11.3.1.

Once the application is done, you may ask yourself whether you indeed achieved your initial goals. And after the development is done, the maintenance starts (fixing bugs, improving performance, etc.). Let's see how we can help with these boring and stressful tasks.

8.3 Achieving goals and problem solving

Now that we've looked at the development process of a NHibernate application, let's take a step back and think again about the design of this application. Even with a good implementation and set of tests, a poorly designed application would be useless. This is why you should have some design goals and ways to measure how much they are achieved.

A good understanding of the development process is required to fully take advantage of this section. However, do not make the mistake of delaying this task when actually developing an application. You should realize that the costs to change the design of an application increase rapidly as the application is implemented. And don't hesitate, in the middle of the development to take a break and look back at your initial goals and how far you are from them.

Assessing an application status is not always easy. Developers can also have a hard time understanding what is wrong with their application. Having some skills in problem solving is definitively a plus when dealing with complex frameworks like NHibernate.

Finally, you must remember that a single tool can not do every job. Learn to choose wisely NHibernate when it is the best option and fallback to other alternatives for other situations.

Hopefully, by the end of this section, you will have a better understanding of how to deal with these tasks. Let's start with the first one in the development process: Achieving the design goals.

8.3.1 Design goals applied to a NHibernate application

A NHibernate application is simply a .NET application using NHibernate. However, because of the central role played by NHibernate, there are some implications which must be taken in consideration when designing a NHibernate application.

As explained in the following section, no tool suits every job. NHibernate is certainly a good framework to solve the ORM mismatch; but it is also a complex one. It requires some skills to be correctly used. We advise you to carefully test NHibernate and your competence before starting to use it in a mission-critical environment.

There are six design goals defined by the MSDN and we are going to look at each of them with NHibernate in mind.

Availability and Reliability

This is the ability of an application to be present and ready for use. Basically, your application should aim to be bug-free.

NHibernate has an impact on the way your application is tested. Because it is not intrusive, the business logic in your domain model and business layer can be fully tested outside NHibernate's scope. Note that you must still test the way you use NHibernate, see the performance and scalability section.

Extensive testing is the first recommendation to achieve this goal; testing the internals of your application and its interactions with the outside world. If your application interacts with external services, verify that a failure in one of these services will not cause your application to crash.

Finally, if your application provides services to other systems, verify that they can't crash your application by sending invalid information or using your services in a specific way. The end-user can be considered as part of these systems. The first step here is to validate inputs you receive from these systems.

Manageability and Securability

This is the ability of an application to be administered and to protect its resources and its users. The purpose of manageability is to ease the (re)configuration and the maintenance of the application.

Another benefit of NHibernate is that it encourages the separation of concerns in your application (layered architecture), hence increase its manageability. NHibernate is also configurable using XML files, hence allowing production-time changes.

Security was not a big concern few years ago, but it is gaining more attention. The first advice related to security is to keep your connection string in a safe place (not plain text and not in an assembly); for example, you can keep it encrypted in `Web.config`.

The general advice is to implement your application with security in mind and to encourage the use of the minimal amount of privileges.

Performance and Scalability

Performance is the measure of an application's operation under load. Developers tend to consider this goal as the most important. It is also the most misunderstood goal (for example, it is commonly confused with scalability).

NHibernate is a layer on top of ADO.NET; do some tests to make sure your application performs well, identify bottlenecks and make sure that NHibernate is efficiently used (lazy/eager fetching, caching, etc.). As last resort, NHibernate allows you to fallback to classic ADO.NET (the underlying connection is accessible through the property: `ISession.Connection`); we are not against stored procedures for batch processing. Note that NHibernate 1.2.0 is now faster than hand-coded data access for classic operations on SQL Server because it is using an unexposed batching feature of the .NET framework.

The next section gives some tips to improve the performance of your application. Remember that performance optimization is not something to do at the end of the implementation; think about it from the start (but not too much).

Scalability is the ability of an application to match increasing demand with an increase in resources. There are many techniques to achieve this goal. You can use asynchronous programming (wrapping expensive calls using asynchronous delegates). You should also use the ADO.NET connection pool (and maybe increase its size); in this case, avoid using a connection string per user.

Another best practice is to open a NHibernate session as late as possible and to close it as soon as possible. You may also consider using a distributed cache (read chapter 6, section 6.3, "Caching theory and practice").

Final note

You may certainly aim at fulfilling all these design goals; however, a design choice which has a positive effect on one goal may at the same time have a negative effect on another one.

Remember that it is certainly important to have strict rules to make sure that these goals are kept in mind through the whole development process; but it is also important to well balance the amount of effort put on an optimization with the outcome of this work.

Do not work on a feature more than it is worth and do not optimize blindly, that is without any way to know whether this optimization is really needed and to measure the optimization improvement.

8.3.2 Identifying and solving problems

There is nothing more frustrating than getting an error and not understanding where it comes from. However, a strict debugging process helps easily fixing most errors. Errors in .NET applications are generally thrown exceptions.

End-users can also complain when using a bug-free application. The main complain being that it is too slow. This problem can be very frustrating for the end-users and you can be certain that they will bug you for that.

Bug solving process

Before thinking about fixing a bug, make sure that there is an infrastructure to catch and log it and make sure that the end-user receives a useful message. This is very important in production environment.

The first step to fix a bug revealed through an exception is to read the content of this exception. Not only the message, but also the stack trace, inner exceptions, etc. (everything returned by `exception.ToString()`).

Then, you should try to understand the meaning of this exception (refer to the documentation) and start investigating on its origin. A good technique at this stage is to isolate the problem until its origin (and solution) becomes obvious. Practically speaking, it means removing processes until the culprit one is located. If you have a hard time understanding a NHibernate exception, read in appendix C the section about asking for help.

Once you have fully identified the problem, TDD recommends that you write some tests reproducing this problem; these tests will help avoiding that this problem comes back later unnoticed.

You may take some time rethinking about the design of your application to see if this problem isn't the symptom of a bigger one.

Finally, you can fix the bug, making the tests you wrote pass.

Improving the performance

If you read this section, you may just have receive one of these performance complains we talked about. Here are some common mistakes you may have done and tips to help dramatically improve your application's performance (and make the end-users happier).

By the way, you should consider writing performance tests at an early stage to avoid waiting for the end-user to tell you that your application is too slow. And don't forget to also optimize your database (adding the correct indexes, etc.).

A mistake that some new NHibernate developers commit is that they create the session factory more than required. This is a very expensive process. Most of the time, it is done once at the start of the application. Avoid keeping the session factory at a place that lives shorter than your application (like keeping it in a web page request).

Another common mistake, related to the fact that NHibernate makes it so easy to load entities, is that you may load more information than you need (without even knowing it). For example, associations and collections are fully initialized when lazy loading is not enabled. So even when loading a single entity, you may end fetching a whole object graph. The general advice here is to always enable lazy loading and to carefully write your queries.

Another related problem, arising when enabling lazy loading, is the n+1 select problem. For more details, read chapter 8, section 8.6.1, "Solving the n+1 selects problem". By the way, a nice way to spot this issue early is to measure the number of queries executed per page; you can easily achieve that by writing a tool to watch logs from `NHibernate.SQL` at `DEBUG` level. If it is higher than a certain limit, you have got a problem to solve and do it immediately, before you forget what is going on in this page. You may also measure other performance-killer operations (like the number of remote calls per page) and global performance information (like the time it takes to process each page).

You should also try to load the information you need using the minimal number of queries (however, avoid expensive queries like those involving Cartesian product). Note that it is generally more important to minimize the number of entities loaded (row count) than the number of fields loaded for each entity (column count).

Chapter 8 describes many features that can help you writing optimized queries.

Now, let's talk about a less known issue. This one is related to the way NHibernate works. When you load entities, the NHibernate session keeps a number of information about them (for transparent

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

persistence, dirty checking, etc.). And when committing/flushing, the session uses this information to perform the required operations.

There is a specific situation where this process can be a performance bottleneck: When you load a lot of entities to update only few of them, this process will be slower than it should be. The reason is that the session will check all these entities to find those that must be updated. You should help it avoiding this waste by evicting unchanged entities or using another session to save changed entities.

As a last resort, consider using the 2nd level cache (and the query cache) to hit the database less often and reuse previous results. Read chapter 6, section 6.3, "Caching theory and practice", for more details on the pros and cons of this feature.

Throwing exceptions

It is common in other environments (like C++) to write code like:

```
if( something goes wrong ) return -1;
```

However, .NET guidelines recommend using exceptions. They are much more powerful and harder to miss. It is very important to understand this concept because NHibernate relies on them to provide error reports and your application should do the same.

Although you should already be familiar with this concept, we will briefly review it and explain how to handle NHibernate exceptions.

Your first step should be to create your own exceptions to provide better information (hence being able to better handle them). Use .NET build-in exceptions only when meaningful (for example `ArgumentNullException` when reporting a null argument).

In these chapters, we use `BusinessException` when a business rule is broken. You may add more specific exceptions if you need to.

Another good practice is to not let exceptions from external libraries reach the presentation layer (or any facade like `WebService`) untouched. You should wrap them in your own exceptions.

Here is an example showing the implementation of a simple `Save()` method in a class of the persistence layer (this class could be called `UserDAO`).

```
public void Save( User user ) {
    try {
        session.SaveOrUpdate( user );
    }
    catch( HibernateException ex ) {
        log.Error( "Error while saving a user.", ex );
        throw new PersistenceException( "Error while saving a user.", ex );
    }
}
```

`HibernateException` is the (base) exception thrown by NHibernate. Obviously, the upper layer must close the session if an exception is thrown. Note that you can move this exception handling to the business layer (to provide a business-friendly message to the end-user).

If you come across a (performance) problem when using NHibernate and you find it hard to solve, you should step back and ask yourself if it was really the right tool for this process.

8.3.3 Use the right tool for the right job

As explained in chapter 1, Object/Relational Mapping (ORM) and NHibernate are certainly powerful tools. However, we take great care not to make NHibernate appear to be a silver bullet. It isn't a solution that will make all your database problems go away magically.

Writing database applications is one of the more challenging tasks in software development. NHibernate's job is to reduce the amount of code you have to write for the most common 90 percent of use cases (common CRUD and reporting).

The next 5 percent are more difficult; queries become complex, transaction semantics are unclear at first, and performance bottlenecks are hidden. You can solve these problems with NHibernate elegantly and keep your application portable, but you'll also need some experience to get it right.

NHibernate's learning curve is high at first. In our experience, a developer needs at least two to four weeks to learn the basics. Don't jump on NHibernate one week before your project deadline—it won't save you. Be prepared to invest more time than you would need for another web application framework or simple utility.

Finally, use SQL and ADO.NET for the 5 percent of use cases you can't implement with NHibernate, such as mass data manipulation or complex reporting queries with vendor-specific SQL functions. You should realize that there are many tasks which are inherently not object-oriented. Forcing ORM can easily cripple the performance of your application.

Once again: Use the right tool for the right job.

There are other places where you must carefully think about the right approach. For example, adding services can be done in many ways, each with their pros and cons. Let's study the case of audit logging.

8.4 Integrating services: Case of Audit logging

So far, we have talked about the development of a NHibernate application without taking into account many aspects and features that may be hard to plug in our layered architecture. In this section, we will talk about the integration of these services.

In the context of this book, a *service* is a clearly separated subsystem (set of classes) that is integrated to the main application to add some functionality. Note that talking about independent services (for example using COM) is largely out of the scope of this book.

The most common services are audit logging and security. It is also common to have some business component implemented as services. For example, in a messaging platform, a service may analyze messages to detect misbehaving users or filter strong language.

We call them *services* because they should be loosely coupled with the business logic (although it isn't that important for simple applications). An important property of services is that, due to their loose coupling, they can evolve and be configured independently; it is also easy to disable them without deeply affecting the main application. This is why they are often part of the non-functional requirements.

Audit logging is the process of recording changes done on data (occurring events, in general). An *audit log* is a database table that contains information about changes made to other data, specifically about the *event* that results in the change. For example, we might record information about creation and update events for auction `Items`. The information that's recorded usually includes the user, the date and time of the event, what type of event occurred, and the item that was changed.

NHibernate has special facilities for implementing audit logs (and other similar aspects that require a persistence event mechanism). In this section, we will use the `IInterceptor` interface to implement audit logging. But first, we will briefly talk about the hard way (doing it *manually*), so that you can grasp the level of difficulty of this problem. Then, we discover how `IInterceptor` makes it much easier.

8.4.1 Doing it the hard-way

The *hard-way* (or the *manual-way*) describes an approach that requires a continuous amount of effort through the development process. In the case of audit logging, it means calling the audit logging service each time it is needed.

Can't I use database triggers?

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Audit logs are often handled using database triggers, and we think this is an excellent approach. However, it's sometimes better for the application to take responsibility, especially if a complex processing is used or if portability between different databases is required.

Practically speaking, you can have an implementation similar to the following:

```
public void Save( User user ) {
    try {
        session.SaveOrUpdate( user );
        AuditLog.LogEvent( LogType.Update, user );
    }
    catch { ... }
}
```

The call to the method `LogEvent()` of the class `AuditLog` will generate and save a log about this change. `LogType` is an enumeration; it is better than using a string. Note that you may use the *Observer pattern* to remove the dependency on this service; read chapter 10, section 10.3.2 for more details about this pattern.

The main advantage is that a better message can be generated for each operation (due to the fact that we know exactly what we are doing each time we call this service).

The disadvantages of this approach are that it is verbose (it clutters the code), and more important: It can be forgot or bypassed; this is unacceptable when building a trusty audit logging service.

This approach may work much better for other services; so do not discard it completely.

8.4.2 Doing it the NHibernate-way

Let's see how NHibernate allows us to automate audit logging. The advantages of this approach are the disadvantages of the hard-way, and vice-versa.

You need to perform several steps to implement this approach:

- 1 Mark the persistent classes for which you want to enable logging.
- 2 Define the information that should be logged: user, date, time, type of modification, and so on.
- 3 Tie it all together with a NHibernate `IInterceptor` that automatically creates the audit trail for you.

Creating the marker attribute

We first create a marker attribute, `AuditableAttribute`. We use this attribute to mark all persistent classes that should be automatically audited:

```
[AttributeUsage( AttributeTargets.Class, AllowMultiple=false )]
[Serializable]
public class AuditableAttribute : Attribute {
}
```

This attribute can be applied once on classes; you may add some properties to customize the logging per class (for example, a localized name to use instead of the class name). Enabling audit logging for a particular persistent class is now trivial; we just add it to the class declaration. Here's an example, for `Item`:

```
[Auditable]
public class Item {
    ...
}
```

Note that, using an attribute implies relying on `entity.ToString()` to obtain logging details because there will be no other means for the audit logging service to extract them; unless you use a big `switch` statement to cast the object (feasible if this service is aware of the domain model).

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Instead of an attribute, we may create an `IAuditable` interface. That way, the entities can actively participate to the logging process. Anyway, you can do both and choose the best one for each entity.

Creating and mapping the log record

Now we create a new persistent class, `AuditLogRecord`. This class represents the information we want to log in the audit database table:

```
public class AuditLogRecord {
    public long Id;
    public string Message;
    public long EntityId;
    public Type EntityType;
    public long UserId;
    public DateTime Created;

    internal AuditLogRecord() {}

    public AuditLogRecord(string message,
                           long entityId,
                           Type entityType,
                           long userId) {
        this.Message = message;
        this.EntityId = entityId;
        this.EntityType = entityType;
        this.UserId = userId;
        this.Created = DateTime.Now;
    }
}
```

You shouldn't consider this class part of your domain model. Hence you don't need to be as cautious about exposing public fields. The `AuditLogRecord` is part of your persistence layer and possibly shares the same assembly with other persistence-related classes, such as your custom mapping types.

Next, we map this class to the `AUDIT_LOG` database table:

```
<hibernate-mapping default-access="field">
  <class name="NHibernate.Auction.Persistence.Audit.AuditLogRecord,
           NHibernate.Auction.Persistence"
        table="AUDIT_LOG"
        mutable="false">
    <id name="Id" column="AUDIT_LOG_ID">
      <generator class="native"/>
    </id>
    <property name="Message" column="MESSAGE"/>
    <property name="EntityId" column="ENTITY_ID"/>
    <property name="EntityType" column="ENTITY_CLASS"/>
    <property name="UserId" column="USER_ID"/>
    <property name="Created" column="CREATED"/>
  </class>
</hibernate-mapping>
```

We marked the class `mutable="false"`, since `AuditLogRecords` are immutable, `NHibernate` will now no longer update the record, even if you try to.

The audit logging concern is somewhat orthogonal to the business logic that causes the log-able event. It's possible to mix logic for audit logging with the business logic, but in many applications it's preferable that audit logging be handled in a central piece of code, transparently to the business logic. We wouldn't manually create a new `AuditLogRecord` and save it whenever an `Item` is modified.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

NHibernate offers an extension point, so you can plug in an audit-log routine (or any other similar event listener). This extension is known as a NHibernate `IInterceptor`.

Writing an interceptor

We'd prefer that a `LogEvent()` method be called automatically when we call `Save()`. The best way to do this with NHibernate is to implement the `IInterceptor` interface. Here's an example:

Listing 8.2 IInterceptor implementation for audit logging

```
public class AuditLogInterceptor : NHibernate.Cfg.EmptyInterceptor {  
    private ISession session;  
    private long userId;  
  
    private ISet inserts = new HashSet();           |1  
    private ISet updates = new HashSet();         |1  
  
    public ISession Session {                     |2  
        get { return this.session; }  
        set { this.session = value; }  
    }  
    public long UserId {                          |3  
        get { return this.userId; }  
        set { this.userId = value; }  
    }  
  
    public virtual bool OnSave(object entity,      |4  
        object id,  
        object[] state,  
        string[] propertyNames,  
        IType[] types) {  
  
        if ( entity.GetType().GetCustomAttributes(  
            typeof(AuditableAttribute), false).Length > 0 )  
            inserts.Add(entity);  
  
        return base.OnSave(entity, id, state, propertyNames, types);  
    }  
  
    public virtual bool OnFlushDirty(object entity, |5  
        object id,  
        object[] currentState,  
        object[] previousState,  
        string[] propertyNames,  
        IType[] types) {  
  
        if ( entity.GetType().GetCustomAttributes(  
            typeof(AuditableAttribute), false).Length > 0 )  
            updates.Add(entity);  
  
        return base.OnFlushDirty(entity, id,  
            currentState, previousState, propertyNames, types);  
    }  
  
    public virtual void PostFlush(System.Collections.ICollection c) { |6  
        try {  
            foreach(object entity in inserts) {  
                AuditLog.LogEvent(LogType.Create,  
                    entity,  
                    userId,  
                    session.Connection);  
            }  
            foreach(object entity in updates) {  
                AuditLog.LogEvent(LogType.Update,  
                    entity,  
                    userId,  
                    session.Connection);  
            }  
        }  
    }  
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Using a temporary Session

It should be clear why we require an `ISession` instance inside the `AuditLogInterceptor`. The interceptor has to create and persist `AuditLogRecord` objects, so a first attempt for the `OnSave()` method could have been the following routine:

```
if ( entity.GetType().GetCustomAttributes(
    typeof(AuditableAttribute), false).Length > 0 ) {
    try {
        object entityId = session.GetIdentifier(entity);
        AuditLogRecord logRecord = new AuditLogRecord( ... );
        // ... set the log information

        session.Save(logRecord);
    } catch (HibernateException ex) {
        throw new CallbackException(ex);
    }
}
```

Look how we use `session.GetIdentifier(entity)` to easily get the identifier. This implementation seems straightforward: create a new `AuditLogRecord` instance and save it, using the current session. However, it doesn't work.

It's illegal to invoke the original NHibernate `ISession` from an `IInterceptor` callback. The session is in a fragile state during interceptor calls. A nice trick that avoids this issue is to open a new `ISession` for the sole purpose of saving a single `AuditLogRecord` object. To keep this as fast as possible, you reuse the ADO.NET connection from the original `ISession`.

This *temporary session* handling is encapsulated in the `AuditLog` helper class:

Listing 8.3 Temporary session pattern

```
public class AuditLog {
    public static void LogEvent(
        LogType logType,
        object entity,
        long userId,
        IDbConnection connection) {
        using(ISession tempSession = sessionFactory.OpenSession(connection)) { |1
            AuditLogRecord record =
                new AuditLogRecord(logType.ToString(),
                                   tempSession.GetIdentifier(entity),
                                   entity.GetType(),
                                   userId); |2
            tempSession.Save(record); |2
            tempSession.Flush(); |2
        }
    }
}
```

#1 Create a new session reusing an opened connection
#2 Create and save the log record

Note that this method never commits or starts any database transactions; all it does is executing additional `INSERT` statements on an existing ADO.NET connection and inside the current database transaction. Using a temporary `ISession` for some operations on the same ADO.NET connection and transaction is a nice trick you may also find useful in other scenarios.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

The *NHibernate-way* is powerful, simple, and easier to integrate. However, there are other kinds of operations that can not work using it. In the case of audit logging, the *NHibernate-way* only logs operations per-entity. You can't log an operation affecting many entities or unrelated to persistence.

The bottom line is that you will probably use both approaches.

We encourage you to experiment and try different interceptor patterns. The *NHibernate* website also has examples using nested interceptors or logging a complete history (including updated property and collection information) for an entity.

8.4.3 Other ways of integrating services

The approaches we covered common and quite simple to implement. However, more complex applications may require a more loosely coupled approach. In this section, we discover patterns called *Inversion of Control* and *Dependency Injection*. We also give you a hint on how you can use a logging library to merge your logs with *NHibernate* logs.

Inversion of Control and Dependency Injection

It is outside the scope of this book to cover these patterns; however, if you don't know them, a brief introduction would certainly be helpful.

The motivation of these patterns is to avoid high-coupling between services addressing different concerns.

Let's take an example: In our auction application, ending an auction can require that we update the database, send a notification to the winner, collect the payment and dispatch the item. These steps require that we communicate with different services. A high-coupling with them may cripple the manageability and flexibility of our application. Configuring and changing these services can become very hard.

The solution brought by *Inversion of Control* is to externalize the binding between the application and the services. Practically speaking, it means that you define interfaces (contracts) to communicate with the services and you use a configuration file (generally written in XML) to specify the service to use for each interface. That way, changing the service can be done by simply editing the configuration file.

In the case of audit logging, you can create an interface called `IAuditLog` and specify in the configuration file that the class (service) to use is the `AuditLog` class defined in listing 9.3.

There are many libraries that provide these features, each with its pros and cons. The two of the most popular are *Castle Windsor* (<http://www.castleproject.org/container/>) and *Spring.NET* (<http://www.springframework.net/>).

For more details, read http://en.wikipedia.org/wiki/Dependency_injection.

Integrating NHibernate logging

You may decide to use a logging library instead of (or in addition to) saving logs using *NHibernate*. This kind of logging is generally not for auditing but for maintenance.

It is easy to merge your logs with *NHibernate* logs using *log4net*. This library provides various destinations for the logs; it is even possible to send them through e-mails or to save them in a database. However, beware of the performance costs.

If you can't use it, you still have another option: Wrap *log4net* library (to redirect method calls) to be able to switch from *log4net* to other solutions like the *Enterprise Library* or the *System.Diagnostics* API.

Although logging is a non-functional requirement (that few end-users care about), they are invaluable to debug applications in production. So think twice before deciding that you don't need it.

8.5 Summary

This chapter focused on application development and the integration issues you may encounter when writing NHibernate applications. We first talked the practical implementation of a layered application. We discuss how the domain model and the business layer should be implemented and tested. We then talked about the persistence layer and ended with the presentation layer.

The next objective of this chapter was to help integrating NHibernate applications into production environments. We specifically talked about the medium-trust issue you may encounter when developing Web applications.

After that, we summarize how NHibernate can help achieving the standard design goals of a .NET application. NHibernate has an impact on the way you design an application and a careful usage of its features can greatly improve the quality of your application. We also gave few tips helping identifying and solving bugs and performance issues.

In the last section of the chapter, we talked about integrating services in a NHibernate application. We discuss the pros and cons of the hard-way and the NHibernate-way. We also talk about few more alternatives.

We implemented audit logging for persistent entities with an implementation of the NHibernate `IInterceptor` interface. Our custom interceptor uses a temporary `ISession` trick to track modification events in an audit history table.

You are now ready to dig in the details of implementing the two layers directly related to NHibernate: The domain model layer and the persistence layer. These are the topics covered in the next two chapters. Go on to the next chapter to start with the domain model layer.

9

Writing Real World Domain Models

This chapter covers

- n Domain Model development processes and legacy schemas mapping
- n Understanding persistence ignorance and business logic
- n Implementing GUI data binding
- n Approaches to obtain a DataSet from entities

This chapter focuses on some advanced aspects of the development of a domain model.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Since the first chapters, you have been familiarized with what an entity looks like. However, we have been manually implementing simple entities that contain data and learning how to map this data so that NHibernate can load and save it.

In the first section of this chapter, you will learn many automated ways of implementing the domain model, creating the database, and writing the mapping. In the case that you are working against an existent database, you may have to map some exotic tables. The next section presents various features of NHibernate that help you support these mapping.

In the real world, a domain model contains much more than data. It contains behavior. In order to avoid unnecessary dependencies, this chapter will also teach how to implement entities unaware of the persistence layer. After that, we explain what the business logic of a domain model is and how to implement it.

At this point, you will have a fully functional domain model. It will be time to interact with the other layers. The two last sections explain how to data bind an entity to the presentation layer and how to fill a DataSet with the content of an entity in order to communicate with components requiring DataSets.

Now, let's start at the beginning and discover efficient ways of implementing the domain model.

9.1 Development processes and tools

In the first chapters, we always started by defining the domain model before creating the database and writing the mapping. However, in the real world, it isn't always like that. In this section, we will present the processes that can be used to develop the domain model, the database, and the mapping in any order. You will learn how tools can help you go from to the others.

Once you have one of these representations (the domain model, the database or the mapping), NHibernate provides tools that can be used to (partially) generate the other representations. Generally, you will have to complete and customize the generated code.

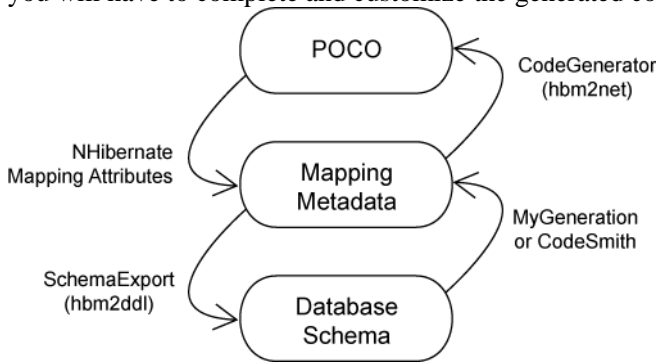


Figure 9.1 Development processes

We will review these various processes in this section.

9.1.1 Generating the mapping and the database from entities

We have been using this approach in this book. It is commonly called *top-down development*. In the absence of an existing data model, this is the most comfortable development style for most .NET developers.

When using this approach, we start with an existing .NET domain model (ideally implemented with POCOs) and complete freedom with respect to the database schema.

Then, we use a library like NHibernate.Mapping.Attributes to generate the mapping (or we can manually write it using a XML editor). Appendix B contains some useful tips when using NHibernate.Mapping.Attributes. You will also discover another attributes library in Appendix C.

Finally, we let NHibernate's hbm2ddl tool generate the database schema using the mapping metadata. This tool is part of NHibernate library. It is not a graphical tool. Its interface is the class `NHibernate.Tool.hbm2ddl.SchemaExport`; hence it is also sometimes called `SchemaExport`.

When planning to use this tool, some special elements and attributes can be used in the mapping files. Most of them are relevant only for a customized schema. NHibernate tries to use sensible defaults if you don't specify your own names and strategies (for example, the column names); however, be warned that a professional DBA might not accept this default schema without manual changes. Nevertheless, the defaults may be satisfactory for a development or prototype environment.

Note that you may also use a naming strategy (as explained in chapter 3, section 3.5.7) to change the way entities names are converted into tables names.

Preparing the mapping metadata

In this example, we've marked up the mapping for the `Item` class with hbm2ddl-specific attributes and elements. These optional definitions integrate seamlessly with the other mapping elements, as you can see in listing 9.1.

Listing 9.1 Additional elements in the `Item` mapping for `SchemaExport`

```
<class name="Item" table="ITEM">
  <id name="Id" type="String">
    <column name="ITEM_ID" sql-type="char(32)"/> |1
    <generator class="uuid.hex"/>
  </id>

  <property name="Name" type="String">
    <column name="NAME"
      not-null="true" |2
      length="255" |2
      index="IDX_ITEMNAME"/> |2
  </property>

  <property name="Description"
    type="String"
    column="DESCRIPTION"
    length="4000"/> |3

  <property name="InitialPrice"
    type="MonetaryAmount">
    <column name="INITIAL_PRICE" check="INITIAL_PRICE > 0"/> |4
    <column name="INITIAL_PRICE_CURRENCY"/>
  </property>

  <set name="Categories" table="CATEGORY_ITEM" cascade="none">
    <key>
      <column="ITEM_ID" sql-type="char(32)"/> |5
    </key>
    <many-to-many class="Category">
      <column="CATEGORY_ID" sql-type="char(32)"/>
    </many-to-many>
  </set>

  ...
</class>
```

hbm2ddl automatically generates a `NVARCHAR` typed column if a property (even the identifier property) is of mapping type `String`. We know the identifier generator `uuid.hex` always generates strings that are 32 characters long; so, we use a `CHAR` SQL type and set its size fixed at 32 characters #1. The

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

nested `<column>` element is required for this declaration because there is no attribute to specify the SQL datatype on the `<id>` element.

The `column`, `not-null`, and `length` attributes are also available on the `<property>` element, but we want to create an additional index in the database, hence we again use a nested `<column>` element #2. This index will speed our searches for items by name. If we reuse the same index name on other property mappings, we can create an index that includes multiple database columns. The value of this attribute is also used to name the index in the database catalog.

For the description field, we chose the lazy approach, using the attributes on the `<property>` element instead of a `<column>` element. The `DESCRIPTION` column will be generated as `VARCHAR(4000)` #3.

The custom user-defined type `MonetaryAmount` requires two database columns to work with. We have to use the `<column>` element. The `check` attribute #4 triggers the creation of a *check constraint*; the value in that column must match the given arbitrary SQL expression. Note that there is also a `check` attribute for the `<class>` element, which is useful for multicolumn check constraints.

A `<column>` element can also be used to declare the foreign key fields in an association mapping. Otherwise, the columns of our association table `CATEGORY_ITEM` would be `NVARCHAR(32)` instead of the more appropriate `CHAR(32)` type #5.

We've grouped all attributes relevant for schema generation in table 19.1; some of them weren't included in the previous `Item` mapping example.

Table 9.1 XML mapping attributes for `hbm2ddl`

Attribute	Value	Description
Column	string	Usable in most mapping elements; declares the name of the SQL column. <code>hbm2ddl</code> (and NHibernate's core) defaults to the name of the .NET property) if the <code>column</code> attribute is omitted and no nested <code><column></code> element is present. This behavior may be changed by implementing a custom <code>INamingStrategy</code> ; see the section 3.5.7, "Naming conventions" in chapter 3.
not-null	true/false	Forces the generation of a NOT NULL column constraint. Available as an attribute on most mapping elements and also on the dedicated <code><column></code> element.
Unique	true/false	Forces the generation of a single-column UNIQUE constraint. Available for various mapping elements.
Length	integer	Can be used to define a "length" of a datatype. For example, <code>length="4000"</code> for a <code>string</code> mapped property generates a <code>NVARCHAR(4000)</code> column. This attribute is also used to define the precision of decimal types.
index	string	Defines the name of a database index that can be shared by multiple elements. An index on a single column is also possible. Only available with the <code><column></code> element.
unique-key	string	Enables unique constraints involving multiple database columns. All elements using this attribute must share the same constraint name to be part of a single constraint definition. This is a <code><column></code> element-only attribute.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

sql-type	string	Overrides hbm2ddl's automatic detection of the SQL datatype; useful for database specific data types. Be aware that this effectively prevents database independence: hbm2ddl will automatically generate a VARCHAR or VARCHAR2 (for Oracle), but it will always use a declared SQL-type instead, if present. This attribute can only be used with the dedicated <column> element.
foreign-key	string	Names a foreign-key constraint, available for <many-to-one>, <one-to-one>, <key>, and <many-to-many> mapping elements. Note that inverse="true" sides of an association mapping won't be considered for foreign key naming, only the non-inverse side. If no names are provided, NHibernate generates unique random names.

After you've reviewed (probably together with a DBA) your mapping files and added schema-related attributes, you can create the schema.

Creating the schema

The hbm2ddl tool is instrumented using an instance of the class SchemaExport. For example:

```
Configuration cfg = new Configuration();
cfg.Configure();
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.Create(false, true);
```

This example creates and initializes a NHibernate configuration. Then, it creates an instance of SchemaExport that uses the mapping and database connection properties of the configuration to generate and execute the SQL commands creating the tables of the database.

Here is the public interface of this class with a brief description of each method:

```
public class SchemaExport
{
    public SchemaExport(Configuration cfg);

    // Specify the database connection properties separately
    public SchemaExport(Configuration cfg, IDictionary connectionProperties);

    // The generated script will be written to this file
    public SchemaExport SetOutputFile(string filename);

    // Set the SQL end of statement delimiter (usually semicolon)
    public SchemaExport SetDelimiter(string delimiter);

    // Run the creation schema script
    public void Create(bool script, bool export);

    // Run the drop schema script
    public void Drop(bool script, bool export);

    // To execute both the drop and create DDL scripts.
    public void Execute(bool script, bool export, bool justDrop, bool format);
    public void Execute(bool script, bool export, bool justDrop, bool format,
        IDbConnection connection, TextWriter exportOutput);
}
```

Table 9.2 explains the meaning of the parameters of the Execute() methods.

Table 9.2 hbm2ddl.SchemaExport.Execute() parameters description

Option	Description
--------	-------------

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Script	Output the generated script to the console.
Export	Execute the generated script against the database.
justDrop	Only drop the tables and clean the database.
Format	Format the generated script nicely instead of using one row for each statement.
Connection	Opened database connection to use when export is true.
exportOutput	Output the generated script to this writer.

This tool is indispensable when applying TDD (explained in chapter 8, section 8.1.1) because it frees you from manually modifying the database whenever the mapping changes. All you have to do is call it before running your tests and you will get a fresh and up-to-date database to work on. Note that it is also available as a NAnt task: `NHibernate.Tasks.Hbm2DdlTask`. For more details, read its API documentation.

However, the fact that this tool completely regenerates the database each time means that you can't use it if you want to migrate data from an old version of the database. Thankfully, the section 9.1.6 provides some approaches to handle this issue.

Creating more database objects

If you need to execute arbitrary SQL statements when generating your database, you can add them to your mapping document. This is especially useful to create triggers and stored procedures used in the mapping.

These statements are written in `<database-object>` elements. If they are in the `<create>` sub-element, they are executed when creating the database. Else, they are in the `<drop>` sub-element, and they are executed when dropping the database.

Here is an example:

```
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2">
  <database-object>
    <create>
      CREATE PROCEDURE FindItems_SP AS
      SELECT ITEM_ID, NAME, INITIAL_PRICE, INITIAL_PRICE_CURRENCY, ...
      FROM ITEM
    </create>
    <drop>
      DROP PROCEDURE FindItems_SP
    </drop>
  </database-object>
  <dialect-scope name="NHibernate.Dialect.MsSql2005Dialect"/>
  <dialect-scope name="NHibernate.Dialect.MsSql2000Dialect"/>
</hibernate-mapping>
```

This example provides the code need to create and drop the stored procedure used at the end of the section 7.6.2 in chapter 7.

As these SQL statements can be dialect-dependent, it is also possible to use `<dialect-scope>` to specify for which dialect they must be executed. In the previous example, the code will only be executed on SQL Server databases.

9.1.2 Generating entities from the mapping

The mapping document provides sufficient information to completely deduce the DDL schema and to generate working POCOs. Furthermore, the mapping document isn't too

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

verbose. So, you may prefer *middle-out development*, where you begin with a handwritten NHibernate mapping document and generate the DDL using `hbm2ddl` and the domain model using code generation. This approach is mainly appealing when migrating from existing Hibernate mapping (used by a Java application).

The tool used to generate entities from the mapping is called `hbm2net`. It is quite similar to `hbm2ddl`; however, it is available as a separated library along with a console application (`NHibernate.Tool.hbm2net.Console`) and a NAnt task (`NHibernate.Tasks.Hbm2NetTask`).

Before using this tool, you must make sure that your mapping provides all the required information like the properties' types. Then, you can execute it using its `CodeGenerator` class:

```
string[] args = new string[] {  
    "--config=hbm2net.config", "--output=DomainModel", "*.hbm.xml" };  
NHibernate.Tool.hbm2net.CodeGenerator.Main(args);
```

Here is the equivalent using its Console Application:

```
NHibernate.Tool.hbm2net.Console.exe  
--config=hbm2net.config --output=DomainModel *.hbm.xml
```

This code will generate a C# class for each mapping document in the current directory and save it in the `DomainModel` directory. The content of the file `hbm2net.config` looks like the following listing:

```
<?xml version="1.0" ?>  
<codegen>  
  
    <meta attribute="implements">  
        NHibernate.InAction.CaveatEmptor.Persistence.Audit.IAuditable  
    </meta>  
  
    <generate  
        renderer="NHibernate.Tool.hbm2net.BasicRenderer"/>  
  
    <generate  
        renderer="NHibernate.Tool.hbm2net.FinderRenderer"  
        suffix="Finder" />  
  
</codegen>
```

The generated C# classes will inherit from the specified `IAuditable` interface. Renderers are used to generate specific parts of the C# class; there is even a `VelocityRenderer`, based on the `NVelocity` library, which allows you to use a template. Refer to their API documentation for more details.

Note that this tool is not as complete as Hibernate's `hbm2java`; refer to the latter's documentation for more details.

Most .NET developers feel more comfortable using the *top-down development* with an attribute library like `NHibernate.Mapping.Attributes` which gives a maximum control, or using the *bottom-up development* when there is an existing data model.

9.1.3 Generating the mapping and the entities from the database

The *bottom-up development* begins with an existing database schema and data model. In this case, the easiest way to proceed is to use a code generation tool like `ActiveWriter MyGeneration` or `CodeSmith` to generate NHibernate mapping documents and skeletal POCO persistent classes (data containers with fields and simple implementation of properties, but no logic). You'll usually have to enhance and modify the generated NHibernate mapping by hand, because not all class association details and .NET-specific meta-information can be automatically generated from an SQL schema.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

This type of code generation is generally template-based: You write a template describing your mapping and POJO with placeholders (for the names, types, etc.). The code generator executes this template for each table in the database. This process is quite intuitive. It is even possible to preserve hand-written regions of code that is not overwritten when regenerating the classes.

MyGeneration is a free .NET code generator; you can learn more about it on its website: <http://www.mygenerationsoftware.com/>. It comes with a simple NHibernate template that you can customize at will.

CodeSmith is a similar tool that is available in free and commercial editions. For more details, go to <http://www.codesmithtools.com/>.

There is also a work-in-progress Visual Studio add-in called ActiveWriter (<http://www.altinoren.com/activewriter/>). It has the benefit of being visually appealing and it provides entities with ActiveRecord attributes (introduced in Appendix C).

Is it a bad thing to write anemic domain models?

As explained in section 10.4, a domain model is made of data and behavior. When using a simplistic code generator, you may be tempted to write your domain model as a data container and move all the behavior in other layers. In this case, your domain model is said to be anemic. For more details, read http://en.wikipedia.org/wiki/Anemic_Domain_Model.

This is not necessarily a bad thing. This approach may work well for simple applications. However, you must realize that it goes against the basic idea of object-oriented design. The behavior may not be correctly represented in other layers, leading to code duplication and other issues. Worse, it may end up in the wrong layers (like the presentation layer).

9.1.4 Meet in the middle

The most difficult scenario combines existing .NET classes and an existing relational schema. It is very hard to map arbitrary domain models to a given schema.

This scenario usually requires at least some refactoring of the .NET classes, database schema, or both. The mapping document must almost certainly be written by hand (although it is possible to use `NHibernate.Mapping.Attributes`). This is an incredibly painful scenario that is, fortunately, exceedingly rare.

When trying to use this scenario, don't hesitate to take full advantage of the numerous extension interfaces of NHibernate. They were introduced in chapter 3, section 3.1.4.

9.1.5 Automatic database schema maintenance

Once you've deployed an application, it becomes difficult to alter the database schema. This can even be the case in development, if your scenario requires test data that has to be redeployed after every schema change. With `hbm2ddl`, your only choice is to drop the existing structure and create it again, possibly followed by a time-consuming test data import.

Hibernate comes bundled with a tool for schema evolution, `SchemaUpdate`, which is used to update an existing SQL database schema; it drops obsolete tables, columns, and constraints. However, at the time of writing, this tool is not available for NHibernate. Therefore, we need another way to solve this problem.

The most basic solution is to manually save each ALTER command used to progressively upgrade the database. However, it might be difficult to keep track of all these changes when maintaining many versions of the database at the same time.

You may find also commercial software that can analyze a database schema before and after a change and then, generate SQL commands to alter and/or migrate the data from the old version of the database to the new version.

There is a quite simple way of managing the evolution of a database. It requires versioning that database and writing a service that keep track of the changes done in each version.

The basic idea is to add a separated table with a column "Version". This is the only table that must not change in the lifecycle of the database. It must also contain a unique row specifying the current version of the database. This value is set when creating the database and updated each time the database evolves.

The changes done on the database must be written as SQL statements. These statements will generally create new tables, alter old ones, migrate data from old tables to new ones and drop these old tables.

Now, all you need is to write a system that keeps track of the changes for each version. Here is a simple example:

```
DatabaseSystem.AddUpdate(1.0, 1.1, new string[] {"SQL statements..."});
```

Each time a new version of the database is defined, the changes are added to this system. When a database must be updated, this system will read its current version, execute all the changes done since this version and set the version to its new value. And the migration is done.

Note that you can easily support downgrading if required.

This approach is similar to the one used by the open source project Migrator: <http://code.macournoyer.com/migrator/>.

So far, we have assumed that the database can be easily mapped to the domain model. However, some old databases may be harder to map.

9.2 Legacy schemas and composite keys

Some data requires special treatment in addition to the general principles we've discussed in the rest of the book. In this section, we'll describe important kinds of data that introduce extra complexity into your NHibernate code.

When your application inherits an existing legacy database schema, you want to make as few changes to the existing schema as possible. Every change you make could break other existing applications that access the database and require expensive migration of existing data. In general, it isn't possible to build a new application and make no changes to the existing data model—a new application usually means additional business requirements that naturally require evolution of the database schema.

We'll therefore consider two types of problems: problems that relate to changing business requirements (which generally can't be solved without schema changes) and problems that relate only to how you wish to represent the same business problem in your new application (which can usually—but not always—be solved without database schema changes). You can usually spot the first kind of problem by looking at the *logical* data model. The second type more often relates to the implementation of the logical data model as a physical database schema.

If you accept this observation, you'll see that the kinds of problems that require schema changes are those that call for addition of new entities, refactoring of existing entities, addition of new attributes to existing entities, and modification of the associations between entities. The problems that can be solved *without* schema changes usually involve inconvenient column definitions for a particular entity.

Let's now concentrate on the second kind of problems. These inconvenient column definitions most commonly fall into two categories:

- n Use of natural (especially composite) keys

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

n Inconvenient column types

We've mentioned that we think natural primary keys are a bad idea. Natural keys often make it difficult to refactor the data model when business requirements change. They may even, in extreme cases, impact performance. Unfortunately, many legacy schemas use (natural) composite keys heavily, and, for the very reason that we discourage the use of composite keys, it may be difficult to change the legacy schema to use surrogate keys. Therefore, NHibernate supports the use of natural keys. If the natural key is a composite key, support is via the `<composite-id>` mapping.

The second category of problems can usually be solved using a custom NHibernate mapping type (implementing the interfaces `IUserType` or `ICompositeUserType`), as described in chapter 7.

Let's look at some examples that illustrate the solutions for both problems. We'll start with natural key mappings.

9.2.1 Mapping a table with a natural key

Our `USER` table has a synthetic primary key, `USER_ID`, and a unique key constraint on `USERNAME`. Here's a portion of our NHibernate mapping:

```
<class name="User" table="USER">
  <id name="Id" column="USER_ID">
    <generator class="native"/>
  </id>

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

  <property name="Username"
    column="USERNAME"
    unique="true"
    not-null="true"/>
  ...
</class>
```

Notice that a synthetic identifier mapping may specify an `unsaved-value`, allowing NHibernate to determine whether an instance is a detached instance or a new transient instance. Hence, the following code snippet may be used to create a new persistent user:

```
User user = new User();
user.Username = "john";
user.Firstname = "John";
user.Lastname = "Doe";
session.SaveOrUpdate(user); // Generates id value by side-effect
System.Console.WriteLine( session.GetIdentifier(user) ); // Prints 1
session.Flush();
```

If you encountered a `USER` table in a legacy schema, `USERNAME` would probably be the primary key. In this case, we would have no synthetic identifier; instead, we'd use the assigned identifier generator strategy to indicate to NHibernate that the identifier is a natural key assigned by the application before the object is saved:

```
<class name="User" table="USER">
  <id name="Username" column="USERNAME">
    <generator class="assigned"/>
  </id>

  <version name="Version"
    column="VERSION"
    unsaved-value="-1"/>
  ...
</class>
```

We no longer can take advantage of the `unsaved-value` attribute in the `<id>` mapping. An assigned identifier can't be used to determine whether an instance is detached or transient—since it's assigned

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

by the application. Instead, we specify an `unsaved-value` mapping for the `<version>` property. Doing so achieves the same effect by essentially the same mechanism. The code to save a new `User` isn't changed:

```
User user = new User();
user.Username = "john"; // Assign a primary key value
user.Firstname = "John";
user.Lastname = "Doe";
session.SaveOrUpdate(user); // Will save, since version is -1
System.Console.WriteLine( session.GetIdentifier(user) ); // Prints "john"
session.Flush();
```

However, we will have to change the declaration of the `version` property in the `User` class to assign the value `-1` (`private int version = -1`).

If a class with a natural key does *not* declare a `version` or `timestamp` property, it's more difficult to get `SaveOrUpdate()` and `cascades` to work correctly. You might use a custom `NHibernate IInterceptor` as discussed later in this chapter. (On the other hand, if you're happy to use explicit `Save()` and explicit `Update()` instead of `SaveOrUpdate()` and `cascades`, `NHibernate` doesn't need to be able to distinguish between transient and detached instances; so, you can safely ignore this advice.)

Composite natural keys extend the same ideas.

9.2.2 Mapping a table with a composite key

As far as `NHibernate` is concerned, a composite key may be handled as an assigned identifier of value type (the `NHibernate` type is a component). Suppose the primary key of our `user` table consisted of a `USERNAME` and an `ORGANIZATION_ID`. We could add a property named `OrganizationId` to the `User` class:

```
[Class(Table="USER")]
public class User {

    [CompositeId]
        [KeyProperty(1, Name="Username", Column="USERNAME")]
        [KeyProperty(2, Name="OrganizationId", Column="ORGANIZATION_ID")]

    public string Username { ... }
    public int OrganizationId { ... }

    [Version(Column="VERSION", UnsavedValue="0")]

    public int Version { ... }

    ...
}
```

Here is the corresponding XML mapping:

```
<class name="User" table="USER">
    <composite-id>
        <key-property name="Username"
            column="USERNAME" />
        <key-property name="OrganizationId"
            column="ORGANIZATION_ID" />
    </composite-id>
    <version name="Version"
        column="VERSION"
        unsaved-value="0" />
    ...
</class>
```

The code to save a new `User` would look like this:

```
User user = new User();
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

// Assign a primary key value
user.Username = "john";
user.OrganizationId = 37;

// Set property values
user.Firstname = "John";
user.Lastname = "Doe";
session.SaveOrUpdate(user); // will save, since version is 0
session.Flush();

```

But what object could we use as the identifier when we called `Load()` or `Get()`? It's possible to use an instance of the `User`; for example:

```

User user = new User();

// Assign a primary key value
user.Username = "john";
user.OrganizationId = 37;

// Load the persistent state into user
session.Load(user, user);

```

In this code snippet, `User` acts as its own identifier class. Note that we now have to implement `Equals()/GetHashCode()` for this class (and make it `Serializable`). We can change that by using a separated class as identifier.

Using a composite identifier class

It's much more elegant to define a separate *composite identifier class* that declares just the key properties. Let's call this class `UserId`:

```

[Serializable] public class UserId {
    private string username;
    private string organizationId;

    public UserId(string username, string organizationId) {
        this.username = username;
        this.organizationId = organizationId;
    }

    // Properties here...

    public override bool Equals(object o) {
        if (o == null) return false;
        if (object.ReferenceEquals(this, o)) return true;
        UserId userId = o as UserId;
        if (userId == null) return false;
        if (organizationId != userId.OrganizationId)
            return false;
        if (username != userId.Username)
            return false;
        return true;
    }

    public override int GetHashCode() {
        return username.GetHashCode() + 27 * organizationId.GetHashCode();
    }
}

```

It's critical that we implement `Equals()` and `GetHashCode()` correctly, since NHibernate uses these methods to do cache lookups. Furthermore, the hash code must be consistent over time. This means that if the column `USERNAME` is case insensitive, it must be normalized (to upper/lower case strings) before using them. Composite key classes are also expected to be `Serializable`.

Now, we'd remove the `UserName` and `OrganizationId` properties from `User` and add a `UserId` property. We'd use the following mapping:

```
<class name="User" table="USER">
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

    <composite-id name="UserId" class="UserId">
      <key-property name="UserName"
        column="USERNAME"/>
      <key-property name="OrganizationId"
        column="ORGANIZATION_ID"/>
    </composite-id>
    <version name="Version"
      column="VERSION"
      unsaved-value="0"/>
    ...
  </class>

```

We could save a new instance using this code:

```

User user = new User();

// Assign a primary key value
user.UserId = new UserId("john", 42);

// Set property values
user.Firstname = "John";
user.Lastname = "Doe";

session.SaveOrUpdate(user); // will save, since version is 0
session.Flush();

```

The following code shows how to load an instance:

```

UserId id = new UserId("john", 42);

User user = (User) session.Load(typeof(User), id);

```

Now, suppose the ORGANIZATION_ID was a foreign key to the ORGANIZATION table, and that we wished to represent this association in our C# model. Our recommended way to do this is to use a <many-to-one> association mapped with insert="false" update="false", as follows:

```

<class name="User" table="USER">
  <composite-id name="UserId" class="UserId">
    <key-property name="UserName"
      column="USERNAME"/>
    <key-property name="OrganizationId"
      column="ORGANIZATION_ID"/>
  </composite-id>
  <version name="Version"
    column="VERSION"
    unsaved-value="0"/>
  <many-to-one name="Organization"
    class="Organization"
    column="ORGANIZATION_ID"
    insert="false" update="false"/>
  ...
</class>

```

This use of insert="false" update="false" tells NHibernate to ignore that property when updating or inserting a User, but we may of course read it with john.Organization.

An alternative approach is to use a <key-many-to-one>:

```

<class name="User" table="USER">
  <composite-id name="UserId" class="UserId">
    <key-property name="UserName"

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        column="USERNAME"/>
    <key-many-to-one name="Organization"
        class="Organization"
        column="ORGANIZATION_ID"/>
</composite-id>
    <version name="Version"
        column="VERSION"
        unsaved-value="0"/>
    ...
</class>

```

However, it's usually inconvenient to have an association in a composite identifier class, so this approach isn't recommended except in special circumstances.

Referencing an entity with a composite key

Since `USER` has a composite primary key, any referencing foreign key is also composite. For example, the association from `Item` to `User` (the seller) is now mapped to a composite foreign key. To our relief, NHibernate can hide this detail from the C# code. We can use the following association mapping for `Item`:

```

<many-to-one name="Seller" class="User">
    <column name="USERNAME"/>
    <column name="ORGANIZATION_ID"/>
</many-to-one>

```

Any collection owned by the `User` class will also have a composite foreign key—for example, the inverse association, `Items`, sold by this user:

```

<set name="Items" lazy="true" inverse="true">
    <key>
        <column name="USERNAME"/>
        <column name="ORGANIZATION_ID"/>
    </key>
    <one-to-many class="Item"/>
</set>

```

Note that the order in which columns are listed is significant and should match the order in which they appear inside the `<composite-id>` element.

Let's turn to our second legacy schema problem, inconvenient columns.

9.2.3 Using a custom type to map legacy columns

The phrase *inconvenient column type* covers a broad range of problems: for example, use of the `CHAR` (instead of `VARCHAR`) column type, use of a `VARCHAR` column to represent numeric data, and use of a special value instead of an SQL `NULL`. It's straightforward to implement an `IUserType` implementation to handle legacy `CHAR` values (by trimming the string returned by the `ADO.NET` data reader), to perform type conversions between numeric and string data types, or to convert special values to a C# `null`. We won't show code for any of these common problems; we'll leave that to you—they're all easy if you study chapter 7, section 7.1.3, "Creating custom mapping types" carefully.

We'll look at a slightly more interesting problem. So far, our `User` class has two properties to represent a user's names: `Firstname` and `Lastname`. As soon as we add an `Initial`, our `User` class will become messy. Thanks to NHibernate's component support, we can easily improve our model with a single `Name` property of a new `Name` C# type (which encapsulates the details).

Also suppose that there is a single `NAME` column in the database. We need to map the concatenation of three different properties of `Name` to one column. The following implementation of `IUserType` demonstrates how this can be accomplished (we make the simplifying assumption that the `Initial` is never null):

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

public class NameUserType : IUserType {
    private static readonly NHibernate.SqlTypes.SqlType[] SQL_TYPES =
        {NHibernate.NHibernateUtil.AnsiString.SqlType};
    public NHibernate.SqlTypes.SqlType[] SqlTypes { get { return SQL_TYPES; } }

    public Type ReturnedType { get { return typeof(Name); } }

    public bool IsMutable {
        get { return true; }
    }

    public object DeepCopy(object value) {
        Name name = (Name) value;
        return new Name(name.Firstname,
            name.Initial,
            name.Lastname);
    }

    new public bool Equals(object x, object y) {
        // use equals() implementation on Name class
        return x==null ? y==null : x.Equals(y);
    }

    public object NullSafeGet(IDataReader dr, string[] names, object owner) {
        string dbName =
            (string) NHibernateUtil.AnsiString.NullSafeGet(dr, names);
        if (dbName==null) return null;
        string[] tokens = dbName.Split();

        Name realName =
            new Name( tokens[0],
                tokens[1],
                tokens[2] );
        return realName;
    }

    public void NullSafeSet(IDbCommand cmd, object obj, int index) {
        Name name = (Name) obj;
        String nameString = (name==null) ?
            null :
            name.Firstname
            + ' ' + name.Initial
            + ' ' + name.Lastname;

        NHibernateUtil.AnsiString.NullSafeSet(cmd, nameString, index);
    }
}

```

Notice that this implementation delegates to one of the NHibernate built-in types for some functionality. This is a common pattern, but it isn't a requirement.

We hope you can now see how many different kinds of problems having to do with inconvenient column definitions can be solved by clever user of NHibernate custom types. Remember that every time NHibernate reads data from an ADO.NET `IDataReader` or writes data to an ADO.NET `IDbCommand`, it goes via an `IType`. In almost every case, that `IType` could be a custom type. (This includes associations—a NHibernate `ManyToOneType`, for example, delegates to the identifier type of the associated class, which might be a custom type.)

One further problem often arises in the context of working with legacy data: integrating database triggers.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

9.2.4 Working with triggers

There are some reasonable motivations for using triggers even in a brand-new database, so legacy data isn't the only context in which problems arise. Triggers and ORM are often a problematic combination. It's difficult to synchronize the effect of a trigger with the in-memory representation of the data.

Suppose the `ITEM` table has a `CREATED` column mapped to a `Created` property of type `DateTime`, which is initialized by an insert trigger. The following mapping is appropriate:

```
<property name="Created"
  type="Timestamp"
  column="CREATED"
  insert="false"
  update="false"/>
```

Notice that we map this property `insert="false"` and `update="false"` to indicate that it isn't to be included in SQL INSERTS or UPDATES.

After saving a new `Item`, NHibernate won't be aware of the value assigned to this column by the trigger, since it occurs after the `INSERT` or the item row. If we need to use the value in our application, we have to tell NHibernate explicitly to reload the object with a new SQL `SELECT`. For example:

```
Item item = new Item();
...
NHibernateHelper.BeginTransaction();
ISession session = NHibernateHelper.Session;

session.Save(item);
session.Flush(); // Force the INSERT to occur
session.Refresh(item); // Reload the object with a SELECT

System.Console.WriteLine( item.Created );

NHibernateHelper.CommitTransaction();
NHibernateHelper.CloseSession();
```

Most problems involving triggers may be solved this way, using an explicit `Flush()` to force immediate execution of the trigger, perhaps followed by a call to `Refresh()` to retrieve the result of the trigger.

You should be aware of one special problem when you're using detached objects with a database with triggers. Since no snapshot is available when a detached object is re-associated with a session using `Update()` or `SaveOrUpdate()`, NHibernate may execute unnecessary SQL `UPDATE` statements to ensure that the database state is completely synchronized with the session state. This may cause an `UPDATE` trigger to fire inconveniently. You can avoid this behavior by enabling `select-before-update` in the mapping for the class that is persisted to the table with the trigger. If the `ITEM` table has an update trigger, we can use the following mapping:

```
<class name="Item"
  table="ITEM"
  select-before-update="true">
  ...
</class>
```

This setting forces NHibernate to retrieve a snapshot of the current database state using an SQL `SELECT`, enabling the subsequent `UPDATE` to be avoided if the state of the in-memory `Item` is the same.

Let's summarize our discussion of legacy data models: NHibernate offers several strategies to deal with (natural) composite keys and inconvenient columns. However, our recommendation is that you carefully examine whether a schema change is possible. In our experience, many developers immediately dismiss database schema changes as too complex and time-consuming, and they look for a NHibernate solution. Sometimes this opinion isn't justified, and we urge you to consider schema evolution as a natural part of your data's lifecycle. If making table changes and exporting/importing

data solves the problem, one day of work might save many days in the long run—when many workarounds and special cases become a burden.

Now that we are done developing and mapping the data-side of the domain model, it is time to dig into its behavior and especially how much it is supposed to know about persistence.

9.3 Understanding persistence ignorance

In the description of the layers of a NHibernate application in chapter 9, section 9.1.3, we highlighted that the domain model shouldn't depend of any other layer or service (though it isn't a strict rule). This is important because it influences its portability. The less dependent it is, the better it can be modified, tested and reused.

This recommendation leads to the notion of *persistence ignorance* (PI). A persistence ignorant entity is an entity which has no knowledge of the way it is persisted (it doesn't even know that it can be persisted).

Practically speaking, it means that the entity doesn't have methods like `Save()` or static (factory) methods like `Load()` and doesn't have any reference to the persistence layer. This is already the case for the entities we have been writing.

Going one step further, we could also say that entities shouldn't have an `Identifier` property and a `Version` property. The reason would be that primary keys and optimistic control have nothing to do with the business domain. However, they are certainly useful (and even required to write efficient applications).

Note that persistence ignorance is not a requirement for most domain models; it is useful when persistence awareness would hinder the portability and flexibility of the domain model. In the Appendix C, we present a library called `ActiveRecord` that sacrifices persistence ignorance in favor of simplicity.

Now, let's see how we can implement an entity that is as free from persistence-related code as possible while still being functional.

9.3.1 Abstracting persistence-related code

A common compromise, in the level of persistence awareness, is to separate persistence-related code from the business code in the implementation of an entity. You may simply perform a visual separation using `#region` in order to improve the readability of the code or you can go as far as to create an abstract base class for entities to deal with that.

Let's take a look at how we can implement the latter solution. We will use `NHibernate.Mapping.Attributes` because it allows the base class to abstract the mapping information along with the code. You will see that the end-result can be acceptable as long as you don't mind inheriting from this base class (if you mind, just copy the content of this class in your entities).

Note that this implementation presents many independent ideas and patterns; feel free to extract some of them for your applications.

We are going to implement an abstract class that entities can inherit from to gain the persistence-related code they need. This class will provide an identifier and a version property along with proper overloading of `System.Object` methods. We will call this class `VersionedEntity` and you can see its implementation in the listing 9.2.

Listing 9.2 Base class abstracting persistence-related code

```
[Serializable]
public abstract class VersionedEntity {
    private Guid id = Guid.NewGuid();
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

private int version = 0;

[Id(Name = "Id", Access = "nosetter.camelcase-underscore")] |1
    [Generator(1, Class = "assigned")] |1
public virtual Guid Id { |1
    get { return id; }
}

[Version(Access = "nosetter.camelcase-underscore")] |2
public virtual int Version { |2
    get { return version; }
}

public override string ToString() { |3
    return GetType().FullName + "#" + Id;
}
public override bool Equals(object obj) { |3
    VersionedEntity entity = obj as VersionedEntity;
    if (entity == null) return false;
    return Id == entity.Id;
}
public override int GetHashCode() { |3
    return Id.GetHashCode();
}
}
}

```

Using an assigned Guid as identifier #1 provides many advantages. For example, It simplifies the implementation of `Equals()` and `GetHashCode()`.

The version #2 is used for optimistic concurrency control, explained in chapter 6, section 5.2.1, "Using managed versioning".

These implementation #3 of `System.Object` methods are simple but effective.

Note that you can replace the initialization of the identifier by:

```

private Guid id
    = (Guid) new NHibernate.Id.GuidCombGenerator().Generate(null, null);

```

This initialization uses the `guid.comb` identifier generator. You can read its advantages in the table 4.1 of chapter 4, section 4.3.3.

However, as you aren't supposed to reference `NHibernate` here, you should create a private static method into `VersionedEntity` using the same algorithm as the method `GuidCombGenerator.GenerateComb()`. Remember that `NHibernate` is licensed under the LGPL; therefore, its source code is publicly available. For more details, read http://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License.

Implementing persistence-abstracted entities

When using this class, your entities will only have to map its business properties. It will inherit what the persistence layer (and `NHibernate`) needs: The identifier, the version and the overloading of `System.Object` methods.

You may ask yourself if the use of `NHibernate.Mapping.Attributes` decreases the persistence ignorance of your entities (as it is about mapping, which is a persistence concern).

This is true to some extent. However, attributes have no behavior and do not add any constraint to your model. They are just like documentation, a useful documentation because the impedance mismatch is not magically solved. You should always remember that your model is persisted in a relation database and that it has some implications. The pros and cons of attributes are enumerated in chapter 3, section 3.4.

Let's take a simple example to illustrate the documentation aspect of these attributes:

```
[Class]
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

public class User : VersionedEntity {
    private string name;

    [Property(Length = 64, Access = "field.camelcase-underscore")]
    string Name { ... }
}

```

Without the information `Length=64`, a careless developer could think that names can be unlimited in length and the end-user will obtain an application that truncates his name for an unknown reason.

By the way, as you can see, using `VersionedEntity` makes this implementation completely free of code unrelated to the business domain while being fully functional.

The fact that the domain model isn't aware of other layers (like the presentation layer) means that it can not directly inform them whenever something happens in it (for example to notifying when a change is done so that the GUI is refreshed). Thankfully, there is a pattern to solve this kind of problem.

9.3.2 Applying the Observer pattern to an entity

The Observer pattern allows a subject to notify observers without a hard-coded link between them. This pattern is often used in a MVC architecture, as explained in chapter 8, section 8.1.1.

This pattern can be used, for example, so that the user interface registers to observe an event that may be raised by the entity whenever its internal state changes.

In order to implement this pattern, you need to add the event to the entity. Then, the observer must register to the event to start listening. Most of the time, the registration is done just after creating or loading the entity.

Let's take an example to illustrate how to implement this pattern. In the previous example, the class `User` has a property `Name`. If we want to inform the presentation layer when this property changes, here is the direct (and bad) way:

```

public class User {
    public string Name {
        get { return name; }
        set {
            if (name==value) return;
            name = value;
            PresentationLayer.User_NameChanged(this);
        }
    }
}

```

Here, we assume that the entity has access to the presentation layer that provides a method to call when it changes. The problem, in this implementation, is that the entity is tied to the presentation layer and that is very bad because, you cannot use it in any other context (for example, when testing).

Here is the solution using the Observer pattern:

```

public delegate void NameChangedEventHandler(
    object sender, EventArgs e );

public class User {
    private string name;

    public string Name {
        get { return name; }
        set {
            if (name==value) return;
            name = value;
            OnNameChanged();
        }
    }
}

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

    }
    public event NameChangedEventHandler NameChanged;
    protected virtual void OnNameChanged() {
        if (NameChanged != null)
            NameChanged(this, EventArgs.Empty);
    }
}

```

We define a delegate for the event called `NameChanged`. Then, in the implementation of the property (here `Name`), we raise the event after changing the value. We use the method `OnNameChanged()` for that because it is a recommended .NET guideline. Read the official documentation of .NET for more details about the implementation and the usage of events.

The next step is to listen to the event:

```

User user = BusinessLayer.LoadUser(userId);
user.NameChanged += User_NameChanged;

```

In this code, we load a user and register the `NameChanged` event just after. The method `User_NameChanged()` will be called whenever this property changes.

Note that the .NET framework provides an interface for this scenario. It is called `INotifyPropertyChanged`. Here is an implementation of the class `User` inheriting from this interface:

```

using System.ComponentModel;

public class User : INotifyPropertyChanged {
    private string name;

    public string Name {
        get { return name; }
        set {
            if (name==value) return;
            name = value;
            OnPropertyChanged("Name");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    protected virtual void OnPropertyChanged(string propertyName) {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

This solution is similar to the previous one. However, it will work for all properties and may be useful in other situations (like data binding).

The Observer pattern can also be used in many other situations. For example, for security purpose:

```

public class SecurityService {
    public void User_IsAdministratorChanging(object sender, EventArgs e) {
        if ( ! loggedUser.IsAdministrator )
            throw new SecurityException("Not allowed");
    }
}

```

Here, the security service listens to the event `User.IsAdministratorChanging`. It is therefore able to cancel a modification by throwing an exception if the logged user is not an administrator.

This section has explained how to avoid cluttering the domain model with unrelated concerns. Now, let's talk about its main concern: The business logic.

9.4 Implementing the Business Logic

In this book, we call *business logic* any code that dictates how the entities should behave. It defines what can be done with the entities and enforces business rules on the data they contain.

Note that we are not strictly speaking about the domain model only, but about the business layer in general.

The business layer can contain many kind of business logic. We will use a little case study to cover them all. Let's say that we want to implement a sub-system of our CaveatEmptor application that allow users to place a bid on an item, and when this is done, all other bidders are notified of this new bid via e-mail.

It is not possible to do that inside the `Bid` or the `Item` entities because sending e-mails is not their responsibility. The business layer should do it. Here is an example of how:

```
public void PlaceBid(int itemId, Bid bid) {
    using (ItemDAO persister = new ItemDAO()) {
        Item item = persister.LoadWithBids(itemId);
        item.PlaceBid(bid);
        foreach (Bid bid in item.Bids)
            Notify(bid.Author, bid, ...);
    }
}
```

Before going further, we need to explain few details: This method takes the identifier of the item and a bid for convenience purpose. In its implementation, it loads the item using a DAO called `ItemDAO`. For more details about the DAO pattern, read the chapter 11, section 11.1. This class can load items and will automatically persist the changes done when it is closed by `using()` because it keeps the `ISession` instance that it uses open.

After loading the item, we place the bid and notify the authors of all the other bids. Note that, because the NHibernate session is still open, the `Bids` collection can be lazily loaded. However, as we know that we will need it, we use a `Load()` method that eagerly loads it, which result in a faster application.

Now, let's dissect this method to see how various kind of business logic are executed when it is called.

9.4.1 Business logic in the business layer

The method `BusinessLayer.PlaceBid()` contains logic that belongs to the business layer. Aside executive code like this, it can also contain business rules like security validation:

```
BusinessLayer.PlaceBid(int itemId, Bid bid) {
    If ( loggedUser.IsBanned )
        throw new SecurityException("Not allowed");
    // ...
}
```

Here, before placing the bid, we make sure that the logged user is not banned.

It is also common to use the `IInterceptor` API when a business rule is hooked to the persistence of entities. Read the section 9.4 of chapter 9 to see how it is done. For complex business rules, you might consider using a rules engine.

The remaining business logic is inside the domain model.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

9.4.2 Business logic in the domain model

The business logic in an entity is a codification of what this entity is supposed to do. Here is a simple implementation of the method `Item.PlaceBid()`:

```
public void PlaceBid(Bid bid) {
    if (bid == null)
        throw new ArgumentNullException("bid");
    if (bid.Amount < CurrentMaxBid.Amount )
        throw new BusinessException("Bid too low.");
    if (this.EndDate < DateTime.Now)
        throw new BusinessException("Auction already ended.");
    Bids.Add(bid);
    CurrentMaxBid = bid;
}
```

This implementation has the purpose of illustrating the different kind of business logic. Chapter 11 contains a detailed discussion about the real implementation of this method.

In this implementation, we start with various checking and then we perform the action. A specific kind of checking is the data integrity checking. It can be easily identified when it prevents the user from assigning a value that conflict with its nature. It is the case when, for example, we don't allow a null string as user's name.

Sometimes, it happens that some logic must be executed at a specific time (for example, before saving the entity). This is the case when the many properties must be filled before hand. In this case, you can add a method that will be called at that time.

For example, if some validation logic must be performed before saving an entity, a method should be added for that in this entity; it could be called `Validate()`. This method can either be called each time that the persistence layer is called, or using the `IInterceptor` API. As for its implementation, validating individual properties can be done by reusing the checking done in these properties (in order to avoid code duplication). For example, with the implementation of this property:

```
public string Name {
    get { return name; }
    set {
        if ( string.IsNullOrEmpty(value) )
            throw new BusinessException("Name required.");

        if (name==value) return;
        name = value;
    }
}
```

We can test that the name is correctly initialized without duplicating the checking. We just have to write:

```
public void Validate() {
    Name = Name;
}
```

This nice trick only works if the checking: `if (string.IsNullOrEmpty(value))` is done before the code: `if (name==value) return;` once we are done with the validation of properties one by one, we can make the validations between the properties like, for example, checking that a minimum value is really smaller than the maximum one.

It is also common to delay the validation of properties until the entity must be saved. This is useful when the properties might keep intermediate values that are not valid yet.

When implementing the business logic of the domain model, you must be careful to avoid unwanted dependencies. They should be move at an upper layer (most of the time, the business layer). For example:

```
public string Password {
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    if ( CryptographicService.IsNotAStrongPassword(value) )
        throw new BusinessException("Not strong enough.");
    password = value;
    MailingService.NotifyPasswordChange(this);
}

```

This is obviously wrong; the domain model must not depend of these cryptographic and mailing services. The business layer must contain this logic.

There are also some *rules* that are not supposed to be implemented in the domain model and not even in the application itself.

9.4.3 Rules that are not business rules

Some *rules* should not be implemented in any layer of the application. An easy way to find them is to see if they really are *business rules*.

These *rules* are generally test the code, to make sure that everything went as expected. For example:

```

public void PlaceBid(int itemId, Bid bid) {
    ...
    item.PlaceBid(bid);
    if ( ! item.Bids.Contains(bid) )
        throw new Exception("PlaceBid() failed.");
    ...
}

```

This is wrong because, if `Item.PlaceBid()` fails, it should already throw an exception, therefore this *rule* should be a test that makes sure that an exception is really thrown when `Item.PlaceBid()` fails.

Let's take another example:

```

public void LoadMinAndMaxBids() {
    ...
    min = BidDAO.LoadMinBid(item);
    max = BidDAO.LoadMaxBid(item);
    if (max < min )
        throw new Exception("Something is wrong with the queries");
    ...
}

```

This is a test for the persistence layer. If it fails, it means that there is a bug in its implementation.

For more details about testing, read the chapter 9, section 9.1.

So far, we have implemented the internal structure of the domain model, taking care of its data and its behavior. Now, it is time to address some issues related to the environment in which this domain model is used.

9.5 Data binding entities

The presentation layer is used by the end-user to display and modify the entities of our NHibernate application. This implies that the data inside our entities are displayed using .NET GUI controls and that the inputs of the end-user are sent back to change these entities.

Although this data transfer can be done manually, .NET provides a way to create a link between an object (called the data source) and a control so that a change to one of them is reverberated to the other. This way is called *data binding*. In the context of NHibernate, we will not call these objects entities but POCOs, to emphasis the fact that they don't have any special infrastructure to ease data binding (which is the case for DataSets, for example).

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

People used to DataSets (and the wizards of Visual Studio .NET) find data binding quite different when starting to write POCOs. Thankfully, most .NET GUI controls support a basic data binding for POCOs and .NET provides some interfaces to improve this support.

In this section, we will discuss a number of alternatives for data binding; these alternatives apply equally to Windows and Web applications. We will first see how we can interact with .NET GUI controls without using data binding. Then, we will data-bind POCOs and enumerate a number of extensions that improve these capabilities. We will also see how NHibernate can help implementing data binding. Finally, we will discover a library that helps a lot when data binding POCOs.

There are three kinds of data in a POCO: A simple property (whose type is a primitive type), a reference to another POCO (as component or many-to-one relationship), and a collection of POCOs (or primitives).

In this discussion, we will ignore the reference to another POCO because it is generally visualized by displaying one of its properties (its name, for example) and a special mechanism (a button, for example) is provided to view or edit this reference.

In order to cover how the two other kind of data (simple properties and collections) can be data bound, we will take the example of writing a form to manage a user and his billing details (as defined for our auction application in chapter 4, section 4.1.2). This form will receive the user and allow us to update him and his list of billing details.

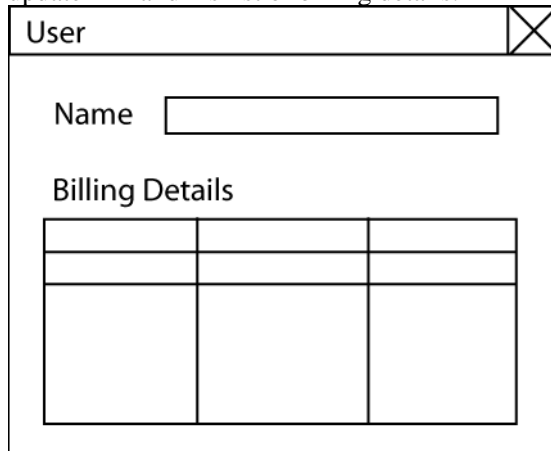


Figure 9.3 Domain model bound to an user interface

The interesting aspect of this example is that `BillingDetails` is an abstract class, so the entities in the collection can either be `BankAccount` or `CreditCard` instances. This complication will give us the opportunity to see the limitations of some approaches.

Note that we don't give a thorough explanation of the .NET APIs we will use; if you need to learn more about them, refer to the official documentation of .NET. You may also read the book *Data Binding with Windows Forms 2.0* [Noyes 2006].

Let's start by ignoring all these APIs and displaying/retrieving data manually.

9.5.1 Implementing manual data binding

The idea behind this approach is very simple: We copy the data of POCOs from/to the GUI. When we need to display something, we take it from the POCO and send it to the GUI:

```
editName.Text = user.Name;
```

When we need to process these POCOs, we retrieve any changes in the GUI and apply them back in the POCOs:

```
user.Name = editName.Text;
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

For the collection, we must add its items one by one in the control.

This approach is simple to understand and implement. It is also good for customization. For example, when displaying an identifier (of the type integer), you may decide to display *New* for a transient entity (instead of 0).

It is also straightforward to support polymorphism:

```
if(billingDetails is BankAccount) {
    ...
    editBankName.Text = (billingDetails as BankAccount).BankName;
}
else {
    ...
    editExpYear.Text = (billingDetails as CreditCard).ExpYear.ToString();
}
```

However, it can be tedious to implement for complex objects and it is less interesting in Windows applications because they can have very chatty forms.

9.5.2 Using data-bound controls

In this case, we rely on the support of data binding public properties and collections on controls. Here is an example for the Name:

```
editName.DataBindings.Add("Text", user, "Name");
```

In this example, the control `editName` (a `TextBox`) is data bound to the property `Name` of the `User` instance. For the collection, the simplest solution is to use the `DataSource` property of the control:

```
dataGridView.DataSource = user.BillingDetails;
```

The `DataGridView` control will use the `BillingDetails` collection as data source. However, this solution is very limited. For example, it doesn't support polymorphism, which means that you can only edit the properties of the class `BillingDetails`. You cannot edit the properties of the subclasses `BankAccount` and `CreditCard`.

Numerous helper classes and extensions can be used to improve this support: `ObjectDataSource`, `BindingSource`; `BindingList`, `IEditableObject`, `INotifyPropertyChanged`, etc. We suggest that you look at these APIs to see which ones suit your needs.

If you really value simplicity in your domain model and still want to do powerful data binding, you can implement wrapping classes (using the *Adapter pattern*) that would represent a *presentation model*:

```
BillingDetailsWrapper detailsWrapper = new BillingDetailsWrapper(details);
editBillingDetails.DataSource = detailsWrapper;
```

In this case, you have two classes with specific purposes that give you more control: The entity keeps focus on its business value and the wrapper provides data binding capabilities on top of the entity. However, they also give you more work as you have two classes to implement instead of one.

Another benefit is that you can add some properties for reporting purpose. A common example is to add a `FullName` property that returns the first name and the last name concatenated.

9.5.3 Data binding using NHibernate

If you think about the way NHibernate works, you will realize that it does some kind of data binding: When you load an entity, it fills it with its data; and when you save it, it retrieves this data.

It is possible to access the part of NHibernate responsible of this work and use it for our benefits. Chapter 3, section 3.5.10 explains how to use this API. It shows how to extract the names of the

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

properties in an entity along with the values contained in an instance. Here is a code based on that explanation:

```
User user = UserDao.Load(userId);

NHibernate.Metadata.IClassMetadata meta =
    sessionFactory.GetClassMetadata( typeof(User) );

string[] metaPropertyNames = meta.PropertyNames;
object[] propertyValues = meta.GetPropertyValues(user);

for (int i=0; i<metaPropertyNames.Length; i++) {

    Label label = new Label();
    label.Text = metaPropertyNames[i];
    Controls.Add(label);

    TextBox edit = new TextBox();
    edit.Text = propertyValues[i].ToString();
    Controls.Add(edit);

}
```

This simplistic implementation retrieves the data of a user and generates labels and textboxes to display them. Note that the position of these controls should be defined.

The interface `IClassMetadata` also has the method: `SetPropertyValues(object entity, object[] values);` it can be used to copy the data from the GUI to the entity. Note that you must keep them in the same order as when you loaded them.

Although this approach seems powerful, it has several drawbacks that are not acceptable in production application: Even with a well-designed algorithm, the resulting layout of the GUI will be far from perfect. You may have some problems with the formatting of the values (for example, dates). There are better controls than `TextBox` for some types of data (for example, `DateTimePicker`). Finally, this approach requires an extra amount of work to support references to other POCOs and collections.

It is certainly possible to solve these issues with some efforts; and this approach can help when prototyping an application. Therefore, you should add it to your toolbox.

9.5.4 Data binding using ObjectViews

ObjectViews is an open source library written specifically to help data bind POCOs to .NET Windows controls.

It is largely outside the scope of this book to cover this library. However, it is worth mentioning that it supports both the data binding of individual POCOs and of collections.

Note that, at the time of this writing, *ObjectViews* is still based on .NET 1.1 and doesn't evolve anymore.

You can download this library (with a helpful example application) on its website: <http://sourceforge.net/projects/objectviews/>.

9.6 Filling a DataSet with entities data

`DataSets` are widely used, mostly by data-centric applications leveraging the nice wizards of tools like Visual Studio .NET to generate code. However, they are quite different to POCOs. If, for some reason, your domain model must somehow communicate with a component using `DataSets`, you will have to find a solution to this problem.

Before starting, remember that you can always execute classic ADO.NET code by opening a database connection yourself or by letting *NHibernate* do it and use the `ISession.Connection` property to retrieve it. In this case, you will have to be careful to not work with stale data or change something without clearing the related *NHibernate* second-level cache. For more details, read chapter 6, section 6.3.

You might also consider rewriting the component using `DataSets` for better consistency.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

If both of these options are not applicable, you will have to convert your entities from/to DataSets. In the following sections, we will only consider going from entities to a DataSet filled with their data. You shouldn't have any major problem reversing this process.

9.6.1 Manual conversion

A DataSet is an in-memory data container that mimicks the structure of a relational database. Filling it means adding rows to its tables. It is relatively easy to figure out the code required to fill a DataSet. Here, we assume that you are working with typed dataset, as they are easier to work with.

Listing 9.3 contains a method that does this work for the Item entity. It is complete because it handles the simple properties, the reference Seller to the User entity and the Bids collection.

Listing 9.3 Filling a DataSet with the content of an entity

```
static private ArrayList adding = new ArrayList();
static public void FillDataSet(TypedDataset dataset, Item item) {
    if ( adding.Contains(item) ) |1
        return; |1
    adding.Add(item); |1

    TypedDataset.ItemRow row;
    if ( dataset.Item.Rows.Contains(item.ItemID) ) |2
        row = dataset.Item.FindByItemID(item.ItemID); |2
    else |2
        row = dataset.Item.NewItemRow(); |2

    row.ItemID = item.ItemID; |3
    row.Name = item.Name; |3
    ... |3

    if ( item.Seller == null ) |4
        row.SetSellerIDNull(); |4
    else { |4
        if ( ! dataset.User.Rows.Contains(item.Seller.UserID) ) |4
            FillDataSet(dataset, item.Seller); |4
        row.SellerID = item.Seller.UserID; |4
    }

    if ( NHibernateUtil.IsInitialized(item.Bids) ) |5
        foreach (Bid bid in item.Bids) |5
            FillDataSet(dataset, bid); |5

    if ( ! dataset.Item.Rows.Contains(item.ItemID) ) |6
        dataset.Item.AddItemRow(row); |6

    adding.Remove(item); |7
}
```

A collection is used to keep the list of entities that are currently being added #1. This is required to avoid infinite recursive calls when there is a circular reference.

If the entity is already in the DataSet, it must be updated, else its row must be created #2.

Filling the simple properties is straightforward #3.

Handling references to other entities is a little bit more complex: We must either set it to null or add it if it is not in the DataSet yet #4.

Handling collections requires that we first make sure that it is already loaded (unless, in your case, you want it to be lazy loaded). Then we just need to add the bids one by one #5.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Now, we can add the row if it is a new one #6.

Finally, we remove the entity from this collection #7 because we are done adding it.

If you are using a code generator like explained in the first section of this chapter, you may be able to generate this code for all your entities. It would be a huge save of time.

Now, let's see how NHibernate can help achieve the same result a little bit faster.

9.6.2 NHibernate-assisted conversion

If you are working with a non-typed `DataSet`, it is possible to use an approach similar to the one explained in section 9.5.3: You can extract the classes' names and properties' names and use them as tables' names and columns' names.

Actually, there is a method in NHibernate that is a close implementation of this idea. It is the method `ToString(object entity)` of the class `NHibernate.Impl.Printer`. Take a look at its implementation before starting your implementation.

Succeeding to implement this approach would mean that it is generic enough to work with any entity because you will only be manipulating metadata.

However, it means that the domain model dictates the schema of the `DataSet`. Note that, this issue can be solved by using the mapping between the domain model and the database (as the `DataSet` schema is generally based on it).

With this support of communicating with a component using `DataSets`, we are done implementing our real-world domain model.

9.7 Summary

Writing real world domain models can be particularly tricky because of the influence of its environment. Hopefully, this chapter has helped you understand this process.

The first step is to implement the domain model, the database and write the mapping between these two. Before this chapter, we always wrote them manually one by one. Now, we know how to generate them. We even have an idea of how we can automate the migration of the database as the domain model evolves.

When writing the mapping, this chapter explained how to handle legacy databases. NHibernate supports the mapping of natural and composite keys. As a last resort, it is possible to implement user types to handle custom situations. It is also possible to work with a database using triggers.

After implementing and mapping the data of the domain model, we moved to its business logic. We explained what persistence ignorance means and how to write a clean domain model free of unwanted dependencies. Then, we listed the different kinds of business logic, explaining how they should be implemented. We also gave some advices of errors to avoid.

When we have completed the domain model, we need to display it. This is where data binding comes to play. As we saw, it can require a good amount of work to correctly be done.

We completed this chapter by looking at how we can obtain a `DataSet` from the content of an entity. We saw that, although it might require a decent amount of time at first, this process can be automated.

This chapter was just an opening to the real world of domain models. You may still need to do some research to find the perfect answer to your needs.

Now, it is time to move to another layer: The persistence layer. So far, we have been writing simple and short persistence operations. It is time to look at what the real world looks like on that side.

10

Advanced Persistence Techniques

This chapter covers

- n Designing the persistence layer
- n Implementing reusable Data Access Objects
- n Implementing conversations
- n Supporting Enterprise Services transactions

You have reached the last chapter of this book. In the previous chapters, you learnt how to implement persistence in a .NET application. You discovered the features of NHibernate and you learned how to design a layered application.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

With this knowledge, you should be able to create the domain model, map it to the database, and implement the business layer and the presentation layer. However, we haven't given enough details about the persistence layer. In chapter 2, the class `Persister` represents the persistence layer. In a real-world application, you will need much more than that.

This chapter starts with the presentation of the *Data Access Object* (DAO) pattern. It is a popular pattern that deals with the organization of the persistence layer. We will go as far as building a persistence layer made of generic objects with a neat organization.

You will also learn the basics of session management. We will return to the interesting topic of conversations (chapter 5) and show practical examples of the various ways conversations can be implemented with NHibernate.

This chapter will end with a discussion on distributed applications. It will explain how to make a NHibernate application participate to a distributed transaction.

10.1 Designing the persistence layer

NHibernate is intended to be used in just about any architectural scenario imaginable (as long as the application is based on .NET). It might run in an ASP.NET application, or in a Windows/Console application. It might even run inside a Web/Windows Service.

These environments are very similar as far as NHibernate is concerned; only few changes are required to port a NHibernate application from one environment to another, as long as the application is correctly layered.

We don't expect your application design to exactly match the scenario we show, and we don't expect you to integrate NHibernate using exactly the code that we use. Rather, we'll demonstrate some common patterns and let you adapt them to your own tastes. For this reason, our examples are plain C#, using no third-party frameworks.

We emphasized the importance of disciplined application layering in chapter 1. Layering helps you achieve separation of concerns, making code more readable by grouping code that does similar things. On the other hand, layering carries a price: Each extra layer increases the amount of code it takes to implement a simple piece of functionality—and more code makes the functionality itself more difficult to change.

We won't try to form any conclusions about the right number of layers to use (and certainly not about what those layers should be) since the "best" design varies from application to application and a complete discussion of application architecture is well outside the scope of this book. We merely observe that, in our opinion, a layer should only exist if it is required, as it increases the complexity and costs of the development. However, we do agree that a dedicated persistence layer is a sensible choice for most applications and that persistence-related code shouldn't be mixed with business logic or presentation.

In this section, we'll show you how to separate NHibernate-related code from your business and presentation layers in a Console application. As we said before, it will be easy to re-use this persistence layer in another (Web or Windows) application.

We need a simple use case from the `CaveatEmptor` application to demonstrate these ideas: When a user places a bid on an item, `CaveatEmptor` must perform the following tasks, all in a single request:

- 1 Check that the amount entered by the user is greater than the maximum existing bid for the item.
- 2 Check that the auction hasn't yet ended.
- 3 Create a new bid for the item.

If either of the checks fails, the user should be informed of the reason for the failure; if both checks are successful, the user should be informed that the new bid has been made. These checks are our *business rules*. If a failure occurs while accessing the database, the user should be informed that the system is currently unavailable (an infrastructure concern).

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Let's see how we can implement this with a simple implementation.

10.1.1 Implementing a simple persistence layer

In the "Hello World" application of chapter 2, the class `Persister` represented alone the persistence layer: It internally initialized the session factory and opened sessions as needed. This simplistic design cannot work for large application. You would have too many methods to perform CRUD operations on all entities.

In this section, we will split the persistence layer into a number of classes, each responsible of a specific concern.

First, we need a way for our application to obtain new `ISession` instances. We'll write a simple *helper* (or utility) class to handle configuration and `ISessionFactory` initialization (see chapter 3) and provide easy access to new `ISessions`. The full code for this class is shown in listing 10.1.

Listing 10.1 A simple NHibernate helper class

```
public class NHibernateHelper {  
    public static readonly ISessionFactory SessionFactory;           |1  
    static NHibernateHelper() {                                     |2  
        try {  
            Configuration cfg = new Configuration();  
            SessionFactory = cfg.Configure().BuildSessionFactory(); |3  
        } catch (Exception ex) {  
            Console.Error.WriteLine(ex);                           |4  
            throw new Exception("NHibernate initialization failed", ex);  
        }  
    }  
    public static ISession OpenSession() {                          |5  
        return SessionFactory.OpenSession();  
    }  
}
```

The `ISessionFactory` is bound to a static (and readonly) variable #1. All our threads can share this one constant, because the `ISessionFactory` implementation is thread-safe. This session factory is created in a *static constructor* #2. This constructor is executed at the first access to this helper class.

The process of building the `ISessionFactory` from a `Configuration` #3 is the same as always. We catch and log the exception #4; Of course, you should use your own logging mechanism rather than `Console.Error`. Our utility class has just one public method, a factory method for new `ISessions` #5. It is a convenient method to shorten the code required for the most common usage of this class: Opening new sessions.

This (very trivial) implementation stores the `ISessionFactory` in a static variable.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Note that this design is completely cluster-safe. The `ISessionFactory` implementation is essentially stateless (it keeps no state relative to running transactions), except for the second-level cache. It's the responsibility of the cache provider to maintain cache consistency across a cluster. So, you can safely have as many actual `ISessionFactory` instances as you like (in practice, you want as few as possible, since the `ISessionFactory` consumes significant resources and is expensive to initialize).

Now that we've solved the problem of where to put the `ISessionFactory` instance (a FAQ), we continue with our use-case implementation.

Performing all the operations inside the same method

In this section, we will write the code that implements the “place bid” use case in one `PlaceBid()` method (see listing 10.2). We're assuming some kind of framework, and we don't show how to read user inputs or how to forward results to the presentation layer. (Note that we don't consider this first implementation to be a good one—we'll make substantial improvements later.)

Listing 10.2 Implementing a simple use case in one method

```
public void PlaceBid(long itemId, long userId, double bidAmount) {
    try {
        using(ISession session = NHibernateHelper.OpenSession()) |1
        using(session.BeginTransaction()) {
            // Load (and lock) requested Item |2
            Item item = session.Load<Item>(itemId, LockMode.Upgrade);
            // Check auction still valid |3
            if ( item.EndDate < DateTime.Now ) {
                throw new BusinessException("Auction already ended.");
            }
            // Check amount of Bid |4
            IQuery q =
                session.CreateQuery(@"select max(b.Amount)
                                   from Bid b where b.Item = :item");
            q.SetEntity("item", item);
            double maxBidAmount = (double) q.UniqueResult();
            if (maxBidAmount > bidAmount) {
                throw new BusinessException("Bid too low.");
            }
            // Add new Bid to Item |5
            User bidder = session.Load<User>(userId);
            Bid newBid = new Bid(bidAmount, item, bidder);
            item.AddBid(newBid);
            session.Transaction.Commit(); |6
            // Operation succeeded
        }
    } catch (HibernateException ex) { |7
        throw new InfrastructureException(
            "Error while accessing the database", ex );
    }
}
```

First, we get a new `ISession` using our utility class #1 and then start a database transaction. These session and transaction will be closed by the `using()` statement. If we don't commit the transaction (or if this commit fails), the transaction will be rolled back.

We load the `Item` from the database #2, using its identifier value, obtaining a pessimistic lock (this prevents two simultaneous bids for the same item).

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

If the end date of the auction is earlier than the current date #3, we throw an exception so that the persistence layer displays an error message. Usually you'll want more sophisticated error handling for this exception, with a qualified error message.

Using an HQL query, we check if there is a higher bid for the current item in the database #4. If there is, we display an error message. If all checks are successful, we place the new bid by adding it to the item #5. We don't have to save it manually; it will be saved using transitive persistence (cascading from the `Item` to `Bid`).

Committing the database transaction flushes the current state of the `ISession` to the database #6.

This `try-catch` block #7 is responsible for exceptions thrown when rolling back the transaction or closing the session; it is wrapped to abstract NHibernate details.

This implementation of the method `PlaceBid()` doesn't assume the existence of any other class. It does the work of the persistence layer (by using NHibernate) and of the business layer.

Based on what you learnt in chapter 9, it is obvious that some check done in this method should be done by the domain model.

Creating a "smart" domain model

Currently, our `PlaceBid()` method contains some business logic. Let's move it to its right place: the `Item` entity.

First, we add the new method `PlaceBid()` to the `Item` class:

```
public Bid PlaceBid(User bidder, double bidAmount, double maxBidAmount) {
    if ( this.EndDate < DateTime.Now )
        throw new BusinessException("Auction already ended.");

    if (maxBidAmount > bidAmount) {
        throw new BusinessException("Bid too low.");
    }

    // Create new Bid
    Bid newBid = new Bid(bidAmount, this, bidder);

    // Place bid for this Item
    this.AddBid(newBid);
    return newBid;
}
```

This code enforces business rules that constrain the state of our business objects but don't execute data-access code. The motivation is to encapsulate business logic in classes of the domain model without any dependency on persistent data access.

You might have discovered that this method of `Item` requires the current highest bid amount. It is obvious that the domain model shouldn't use the persistence layer to query this value and it would be inefficient to iterate its collection of bids looking for it. So we prefer to ask the upper layer to provide this value.

Now, we simplify our `PlaceBid()` method to the following:

```
public void PlaceBid(long itemId, long userId, double bidAmount) {
    try {
        using (ISession session = NHibernateHelper.OpenSession())
            using (session.BeginTransaction()) {

                // Load (and lock) requested Item
                Item item = session.Load<Item>(itemId, LockMode.Upgrade);

                // Retrieve the highest bid amount
                IQuery q =
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

        session.CreateQuery(@"select max(b.Amount)
                           from Bid b where b.Item = :item");
q.SetEntity("item", item);
double maxBidAmount = (double) q.UniqueResult();

// Retrieve the bidder and place the bid
User bidder = session.Load<User>(userId);
item.PlaceBid(bidder, bidAmount, maxBidAmount);

session.Transaction.Commit();

// Operation succeeded
    }
} catch (HibernateException ex) {
    throw new InfrastructureException(
        "Error while accessing the database", ex );
}
}
}

```

The business logic for placing a bid is now encapsulated in the domain model. The remaining code belongs to the persistence layer and the business layer.

It is time to clearly separate these layers. As introduced in chapter 2, we can create a `Persister` class. This class would contain a method to load items and users and another to load the highest bid amount. It would also contain methods to load and save all the other entities of our domain model. The result would be a very big class with many redundancies.

There are many patterns to address this problem. Let's discover one of the most populars.

Introducing the Data Access Object pattern

Mixing data access code (responsibility of the persistence layer) with control logic (part of the business layer) violates our emphasis on separation of concerns. For all but the simplest applications, it makes sense to hide NHibernate API calls behind a facade with higher-level business semantics. There is more than one way to design this facade—some small applications might use a single class for all persistence operations; some might a class for each operation—but we prefer the DAO pattern.

A DAO defines an interface to persistence operations (CRUD and finder methods) relating to a particular persistent entity. It advises you to group code that relates to persistence of that entity. Another common name for this pattern is Gateway (though they have a slightly different meaning).

Let's create an `ItemDAO` class, which will eventually implement all persistence code related to items. For now, it contains only the `FindById()` method, along with `GetMaxBidAmount()` and a method to save items. The full code of the DAO implementation is shown in listing 10.3.

Listing 10.3 A simple DAO abstracting item-related persistence operations

```

public class ItemDAO {
    public static Item FindById(long id) {
        using (ISession session = NHibernateHelper.OpenSession())
            return session.Load<Item>(id);
    }
    public static double GetMaxBidAmount(long itemId) {
        string query = @"select max(b.Amount)
                       from Bid b where b.Item = :item";
        using (ISession session = NHibernateHelper.OpenSession()) {
            IQuery q = session.CreateQuery(query);
            q.SetInt64("itemId", itemId);
            return (double) q.UniqueResult();
        }
    }
}

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    public static Item MakePersistent(Item entity) { |3
        using (ISession session = NHibernateHelper.OpenSession())
            session.SaveOrUpdate(entity);
        return entity;
    }
}

```

This class provides two static methods to perform the operations needed by our `PlaceBid()` method. The `FindById()` method #1 loads items (note that the pessimistic lock isn't available). It is possible to retrieve the highest bid amount using the `GetMaxBidAmount()` method #2 and the `MakePersistent()` method #3 can be used to save items.

Whether `GetMaxBidAmount()` belongs on `ItemDAO` or a `BidDAO` is perhaps a matter of taste; but since the argument is an `Item` identifier, it seems to naturally belong here.

We also need a `UserDAO` with a `FindUserById()` method. You should be able to figure out how to implement it (just replace *Item* by *User* in the previous listing).

Our `PlaceBid()` method is getting cleaner:

```

public void PlaceBid(long itemId, long userId, double bidAmount) {
    try {
        Item item = ItemDAO.FindById(itemId);
        double maxBidAmount = ItemDAO.GetMaxBidAmount(itemId);
        User bidder = UserDAO.FindById(userId);
        item.PlaceBid(bidder, bidAmount, maxBidAmount);

        ItemDAO.MakePersistent(item);
    }
    catch (HibernateException ex) {
        throw new InfrastructureException(
            "Error while accessing the database", ex );
    }
}

```

Notice how much more self-documenting this code is than our first implementation. Someone who knows nothing about NHibernate can still understand immediately what this method does, without the need for code comments.

We also achieved a clear separation of concern. You might be satisfied by this implementation. However, it has several drawbacks.

First, it makes impossible the usage of transparent persistence. This is why we need to explicitly save the item at the end. Moreover, it opens four sessions where a single would be enough. Finally, the implementation of the DAOs has a high level of redundancy for the basic CRUD operations.

These problems can be solved by abstracting the common basic operations and by figuring out a way of making these DAOs share the same session.

Let's jump to the right solution.

10.1.2 Implementing a generic persistence layer

We learnt in the previous section that, although it is easy to implement a simple DAO, there are a number of key issues that require a smarter solution. It should allow all DAOs to share the same session and it should minimize the amount of redundancy.

Let's discover a great solution to the first issue. It is a new feature of NHibernate 1.2.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Using *ISessionFactory.GetCurrentSession()*

The idea behind this feature is that, for a specific action, we generally need a single session. Because this session can be reused for all the operations (even if they are unrelated), it is logical to make this session available to the whole application (that is, to its persistence layer).

Your first impulse could be to create a static session like the session factory defined in listing 11.1. However, it will not work for ASP.NET applications because each Web session should have its own NHibernate session (using a static session would result in a single session for the whole Web application).

It is also possible to open a session and send it to each DAO. In this case, DAOs will no longer have static methods. You would instantiate these DAOs and provide the session as a parameter in their constructors. This solution can work. However, it is very tedious; you will have to pass the NHibernate session everywhere it may be needed.

Instead of solving this problem yourself, you can leverage a new feature of NHibernate 1.2. It is exposed by the method `ISessionFactory.GetCurrentSession()`. This method returns the session instance associated with current *persistence context*, similar to the ASP.NET notion of a HTTP request context. Any components called in the same context will share the same session.

When using this feature, the specific context of your application is abstracted. So your persistence layer will work whether the context is defined by a Web or Windows context.

The first step to enable this feature is to set the context. This is done using the configuration property `current_session_context_class`. For example:

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

This example sets the context to `web`, which is the short name of an implementation included in NHibernate that uses `HttpContext` to track the current session. It is therefore appropriate for ASP.NET applications.

NHibernate 1.2.1 comes with a number of built-in current session context implementations:

Table 10.1 NHibernate's built-in current session context implementations

Short name	Description
Managed_web	This context was the only one available in NHibernate 1.2.0. However, it is now deprecated. You should use <code>web</code> instead.
Call	This context uses the <code>CallContext</code> API to store the current session. Note that, although it works in any kind of application, it isn't recommended for ASP.NET 2.0 applications.
thread_static	When using this context, sessions are stored in a static field marked with <code>[ThreadStaticAttribute]</code> . Therefore, each thread has his own session.
Web	This context uses the <code>HttpContext</code> API to store the current session. It is recommended for Web applications (and only works with them).

It is obviously possible to implement your own contexts. You just have to write a class implementing the extension interface: `NHibernate.Context.ICurrentSessionContext` and set it in the mentioned property. For more details, look at its documentation and the available implementations in the namespace `NHibernate.Context`.

Depending on the implementation of the context, you may have additional work to do. For example, these contexts don't take care of opening and closing the session. You have to do it yourself and bind it to the context (using the class `CurrentSessionContext`).

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Let's see how to use this feature. First, we should add the following method to the class `NHibernateHelper`:

```
public static ISession GetCurrentSession() {
    return SessionFactory.GetCurrentSession();
}
```

Here is what the class `ItemDAO` looks like now:

```
public class ItemDAO {

    public static Item FindById(long id) {
        return NHibernateHelper.GetCurrentSession().Load<Item>(id);
    }

    public static double GetMaxBidAmount(long itemId) {

        string query = @"select max(b.Amount)
                        from Bid b where b.Item = :item";
        IQuery q = NHibernateHelper.GetCurrentSession().CreateQuery(query);
        q.SetInt64("itemId", itemId);
        return (double) q.UniqueResult();
    }

    public static Item MakePersistent(Item entity) {
        NHibernateHelper.GetCurrentSession().SaveOrUpdate(entity);
        return entity;
    }
}
```

The DAO is no longer responsible of opening the `NHibernate` session. This is done at an upper level. In our example, it can be done in our `PlaceBid()` method.

Listing 10.4 provides the implementation of this method.

Listing 10.4 Simple session management using the current session API

```
using NHibernate.Context;

public void PlaceBid(long itemId, long userId, double bidAmount) {
    try {
        using(ISession session = NHibernateHelper.OpenSession())
            using(session.BeginTransaction()) {

                // Attach the session to the context |1
                CurrentSessionContext.Bind(session);

                // Perform the logic as before |2
                Item item = ItemDAO.FindByIdAndLock(itemId);
                double maxBidAmount = ItemDAO.GetMaxBidAmount(itemId);
                User bidder = UserDAO.FindById(userId);
                item.PlaceBid(bidder, bidAmount, maxBidAmount);

                session.Transaction.Commit();
            }
    } catch (HibernateException ex) {
        throw new InfrastructureException(
            "Error while accessing the database", ex );
    }
    finally {
        CurrentSessionContext.Unbind(NHibernateHelper.SessionFactory); |3
    }
}
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

Once the session is opened, we attach it to the current context so that it is available to any object executed in this context #1. After that, we can execute the logic as before #2. Note that pessimistic lock can now be used. At the end of the operation, we must detach the (closed) session from the current context #3.

With this implementation, DAOs can access the same session in a neat and transparent way. It is time to address the other issue in the implementation of a persistence layer: Minimizing redundancy.

Designing DAOs using generics

Whenever you find yourself repeating a similar code repeatedly, it is time to think *inheritance* and *generics*. Note that this section assumes that you have a good understanding of .NET 2.0 generics. In the case that you are still using .NET 1.1, it is possible to adapt the following idea; however, the result will not be as clean.

As you saw when implementing the method `FindById()` for the classes `ItemDAO` and `UserDAO`, only the name of the entity changes for basic CRUD operations. Therefore, it is possible to use generics to abstract this operation.

Here is how the new DAOs may look like:

```
public abstract class GenericNHibernateDAO<T, ID> {
    public T FindById(ID id) {
        try {
            return NHibernateHelper.GetCurrentSession().Load<T>(id);
        }
        catch (HibernateException ex) {
            throw new Exceptions.InfrastructureException(ex);
        }
    }
    ...
}
public class UserDAO : GenericNHibernateDAO<User, long> {
}
```

The class `GenericNHibernateDAO` only needs the types of the entity `T` and its identifier `ID` to implement the method `FindById()`. After that, implementing the class `UserDAO`, simply means inheriting from `GenericNHibernateDAO` and providing these types.

Many other methods can be implemented like that. Before filling the class `GenericNHibernateDAO` with them, let's take a step back and think about the final design that we want.

As far as the business layer is concerned, the persistence layer should provide a set of interfaces to perform all the operations that are needed. This means that the underlining implementation doesn't matter and can be changed as long as the interfaces don't change.

In our design, we will have a `GenericDAO` interface with operations common to all entities and `DAO` interfaces, inheriting from the `GenericDAO` interface, for each entity. These interfaces will all have implementations using `NHibernate`.

Figure 10.1 illustrates this design:

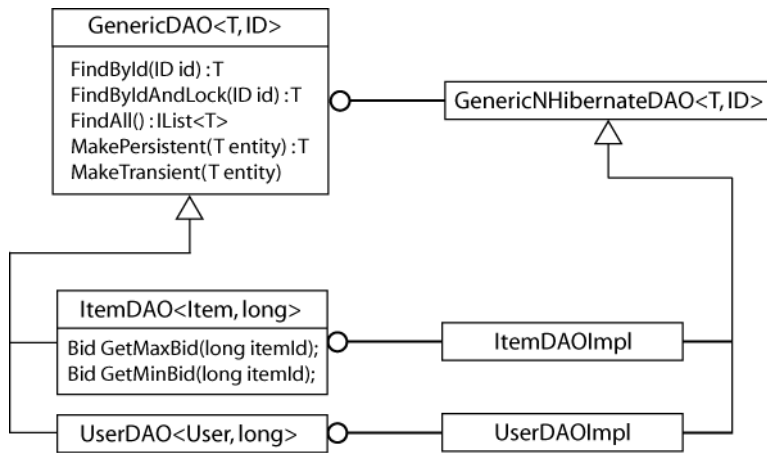


Figure 10.1 Generic DAO interfaces with separated NHibernate implementation

Some interfaces may not have any methods but it is still important to create them because they are strongly typed interfaces and they might be extended in the future.

Finally, we will use the abstract factory pattern as a façade to provide the implementations of the interfaces. This means that the NHibernate classes will be completely hidden from the other layers. It will even be possible to switch the persistence mechanism even at runtime.

Enough theory. Let's take a look at the `GenericDAO` interface:

```

public interface GenericDAO<T, ID> {
    T FindById(ID id);
    T FindByIdAndLock(ID id);

    IList<T> FindAll();

    T MakePersistent(T entity);
    void MakeTransient(T entity);
}
  
```

This interface defines methods to load (the `Find...()` methods), save (using `MakePersistent()`) and delete (using `MakeTransient()`) entities.

Why `MakePersistent()` and `MakeTransient()` instead of `Save()` and `Delete()`?

It is simpler to explain persistence operations using verbs like *save* or *delete*. However, NHibernate is a *state-oriented* framework. This notion was introduced in chapter 5, section 5.1.

For example, when *deleting* an entity, what really happens is that this entity becomes transient. Its row in the database will eventually be deleted, but the entity will not cease to exist (and it can even be *persisted* again).

As these methods are created for the business layer, their names should reflect what happen at that level.

The interfaces inheriting from the `GenericDAO` interface will look like this one:

```

public interface ItemDAO : GenericDAO<Item, long> {

    Bid GetMaxBid(long itemId);
    Bid GetMinBid(long itemId);
}
  
```

Here, we avoid defining a too specific method like `GetMaxBidAmount()`. Now, let's implement these interfaces. First, the `GenericNHibernateDAO`:

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

public abstract class GenericNHibernateDAO<T, ID>: GenericDAO<T, ID> {
    private ISession session;

    public ISession Session {
        get {
            if (session == null)
                session = NHibernateHelper.GetCurrentSession();
            return session;
        }
        set {
            session = value;
        }
    }

    public T FindById(ID id) {
        return Session.Load<T>(id);
    }

    public T FindByIdAndLock(ID id) {
        return Session.Load<T>(id, LockMode.Upgrade);
    }

    public IList<T> FindAll() {
        return Session.CreateCriteria(typeof(T)).List<T>();
    }

    public T MakePersistent(T entity) {
        Session.SaveOrUpdate(entity);
        return entity;
    }

    public void MakeTransient(T entity) {
        Session.Delete(entity);
    }
}

```

In this implementation, we add the property `Session` so that the DAOs can work without needing a session bound to the current context. However, in this case, you must manually provide this session. You may write another class in the persistence layer to take care of that.

The implementation of the other classes is straightforward. Here is the implementation of the `ItemDAO` interface:

```

public class ItemDAOImpl : GenericNHibernateDAO<Model.Item, long>, ItemDAO {
    public virtual Model.Bid GetMinBid(long itemId) {
        IQuery q = Session.GetNamedQuery("MinBid");
        q.SetInt64("itemId", itemId);
        return q.UniqueResult<Model.Bid>();
    }

    public virtual Model.Bid GetMaxBid(long itemId) {
        IQuery q = Session.GetNamedQuery("MaxBid");
        q.SetInt64("itemId", itemId);
        return q.UniqueResult<Model.Bid>();
    }
}

```

Although it has nothing to do with this design, we decided to follow another good practice that is to use named queries. Note that you should wrap all these methods in a `try/catch` statement in case an exception is thrown:

```

try {
    ...
}
catch (HibernateException ex) {
    throw new Exceptions.InfrastructureException(ex);
}

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The usage of this new persistence layer isn't much different than what is done in listing 11.4. The main difference is that these DAOs must be instantiated. We will add another interface to take care of that concern without introducing a dependency to the NHibernate implementation:

```
public abstract class DAOFactory {
    public abstract UserDAO GetUserDAO();
    public abstract ItemDAO GetItemDAO();
}
```

Its implementation is simply:

```
public class NHibernateDAOFactory : DAOFactory {
    public override UserDAO GetUserDAO() {
        return new UserDAOImpl();
    }
    public override ItemDAO GetItemDAO() {
        return new ItemDAOImpl();
    }
}
```

The last step is to instantiate this class at the initialization of the application:

```
DAOFactory daoFactory = new NHibernateDAOFactory();
```

Everything is ready for the new `PlaceBid()` method. Here is the interesting part:

```
ItemDAO itemDAO = daoFactory.GetItemDAO();
Item item = itemDAO.FindByIdAndLock(itemId);
double maxBidAmount = itemDAO.GetMaxBid(itemId).Amount;
User bidder = daoFactory.GetUserDAO().FindById(userId);
item.PlaceBid(bidder, bidAmount, maxBidAmount);
```

The rest of this method remains as in listing 11.4. Interestingly, this code represents exactly what we would like our `PlaceBid()` method to look like (the rest is just plumbing).

More importantly, if we agree that managing the NHibernate session is a persistence layer concern, it shouldn't appear in this method. We are going to make one last refactoring to take care of this issue.

Session management for Web Applications

When processing a complex request, many methods (and classes) of the business layer may get involved. If these methods follow the approach of our previous implementation of the `PlaceBid()` method, it will result in opening many separated sessions.

Aside the obvious performance issue, it means that the business logic will span many database transactions; which implies that, if a latter transaction fails, the previous ones will not be rolled back and the database will be left in a inconsistent state.

Another issue is that, after executing the business logic, or even between two calls to the business layer, the manipulated entities are detached. This means that lazy loading is disabled. Therefore, another layer, like the presentation layer, cannot transparently lazy load the collections of these entities.

Why can't NHibernate open a new connection (or session) if it has to lazy-load associations?

First, we think it's a better solution to fully initialize all required objects for a specific use case using eager fetching (this approach is less vulnerable to the n+1 selects problem). Furthermore, opening new database connections (and ad hoc database transactions!) implicitly and transparently to the developer exposes the application to transaction isolation issues. When do you close the session and

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

end the ad hoc transaction—after each lazy association is loaded? We strongly prefer transactions to be clearly and explicitly demarcated by the application developer. If you want to enable lazy fetching for a detached instance, you can use `Lock()` to attach it to a new session.

Thankfully, this problem is easily solved by using a session that stays open for the whole request lifetime. In this section, we will take the example of a Web request.

Through the processing of the Web request, the rendering of the web page, the computing of the persistence layer and any other component, the same session will be used. This session will be closed at the very end of this process. This means that any un-initialized association or collection will be successfully initialized when accessed at any point of the process.

However, don't get lazy and let NHibernate load data on-demand; it will eventually kill the performance of your application. You should always eagerly load the data you know that you will need. For more details, read the section 7.7.1 of chapter 7.

The solution to our problem is to open and attach a NHibernate session at the beginning of the Web request. ASP.NET allows us to implement the interface `IHttpModule` in order to execute some code at beginning and at the end of a web request. Let's leverage this feature.

Listing 10.5 Web module managing NHibernate sessions

```
using NHibernate.Context;

public class NHibernateCurrentSessionWebModule : IHttpModule {

    public void Init(HttpApplication context) {
        context.BeginRequest += new EventHandler(Application_BeginRequest);
        context.EndRequest += new EventHandler(Application_EndRequest);
    }
    public void Dispose() {
    }
    private void Application_BeginRequest(object sender, EventArgs e) { |1

        ISession session = NHibernateHelper.OpenSession();
        session.BeginTransaction();
        CurrentSessionContext.Bind(session);
    }
    private void Application_EndRequest(object sender, EventArgs e) { |2

        ISession session = CurrentSessionContext.Unbind(
            NHibernateHelper.SessionFactory );

        if (session != null)
            try {
                session.Transaction.Commit();
            }
            catch(Exception ex) {
                session.Transaction.Rollback();
                Server.Transfer("...", true); // Error page
            }
            finally {
                session.Close();
            }
    }
}
```

At the beginning of a request #1, we open and attach a session and at its end #2, we detach and close the session. We also commit the changes that happen through the request's lifetime.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

The following code must be added to the `Web.config` file:

```
<configuration>
  <system.web>
    <httpModules>
      <add name="NHibernateCurrentSessionWebModule"
          type="NHIA.NHibernateCurrentSessionWebModule" />
    </httpModules>
  </system.web>
</configuration>
```

This code is required to register our web module so that ASP.NET uses it.

Now, our `PlaceBid()` method is just like we wanted it to be:

```
public void PlaceBid(long itemId, long userId, double bidAmount) {
    try {
        ItemDAO itemDAO = daoFactory.GetItemDAO();
        Item item = itemDAO.FindByIdAndLock(itemId);
        double maxBidAmount = itemDAO.GetMaxBid(itemId).Amount;
        User bidder = daoFactory.GetUserDAO().FindById(userId);
        item.PlaceBid(bidder, bidAmount, maxBidAmount);
    } catch (Exception ex) {
        throw new BusinessException("Placing the bid failed.", ex);
    }
}
```

We have a session that lives as long as the Web request. It may be enough or it may not: What if the operation to execute requires many Web requests to be completed?

10.2 Implementing conversations

Now that we have implemented our persistence layer, we need to discuss more about its usage. When using a command-oriented framework (for example ADO.NET), each API call is meant to retrieve or change data: Add/update/delete rows in a database. However, NHibernate is state-oriented: each API call is meant to change the state of an entity (as illustrated by the figure 5.1 of chapter 5).

Changing the state of an entity may lead to the execution of a SQL command immediately, when flushing the session or never. The difference between command and state is very important when implementing operations that span many user requests. These operations are called conversations.

We discussed the notion of *conversation* in chapter 5, section 5.2, “Working with conversations.” We also discussed how NHibernate helps detect conflicts between concurrent conversations using managed versioning. We didn’t discuss how conversations are used in NHibernate applications, so we now return to this essential subject.

There are three ways to implement conversations in an application that uses NHibernate: using a *long session*, using *detached objects*, and doing it the *hard way*. We’ll start with the hard way; it will help you understand the benefit of the two other ways. First, we need a use case to illustrate these ideas.

10.2.1 Approving a new auction

Our auction has an approval cycle. A new item is created in the *Draft* state. The user who created the auction may place the item in *Pending* state when the user is satisfied with the item details. System administrators may then approve the auction, placing the item in the *Active* state and beginning the auction. At any time before the auction is approved, the user or any administrator may edit the item details. Once the auction is approved, no user or administrator may edit the item. It’s essential that the approving administrator sees the most recent revision of the item details before approving the auction and that an auction can’t be approved twice. Figure 10.2 shows the item approval cycle.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

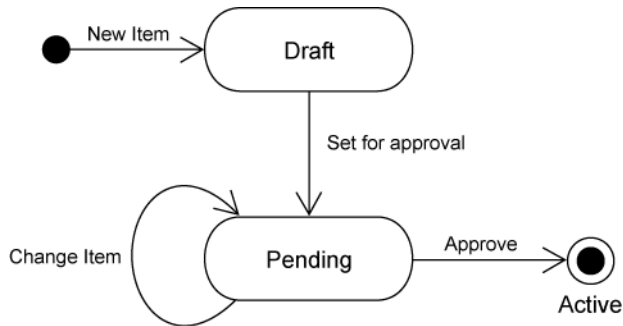


Figure 10.2 State chart of the item approval cycle in CaveatEmptor

The conversation is auction approval, which spans two user requests. First, the administrator selects a pending item to view its details; second, the administrator approves the auction, moving the item to the Active state. The second request must perform a version check to verify that the item hasn't been updated or approved since it was retrieved for display.

As usual, the business logic for approving an auction should be implemented by the domain model. In this case, we add an `Approve()` method to the `Item` class:

```

public void Approve(User byUser) {
    if ( !byUser.IsAdmin )
        throw new PermissionException("Not an administrator.");

    if ( state.equals != ItemState.Pending )
        throw new BusinessException("Item not pending.");

    state = ItemState.Active;
    approvedBy = byUser;
    approvalDatetime = DateTime.Now;
}
  
```

However, it's the code that calls this method that we're interested in.

Are conversations really transactions?

Most books define transaction in terms of the ACID properties: atomicity, consistency, isolation, and durability. Is a conversation really a transaction by that definition? Consistency and durability don't seem to be a problem, but what about atomicity and isolation? Our example is both atomic and isolated, since all update operations occur in the last request/response cycle (that is, the last database transaction). However, our definition of a conversation permits update operations to occur in any request/response cycle. If a conversation performs an update operation in any but the final database transaction, it isn't atomic and may not even be isolated. Nevertheless, we feel that the term transaction is still appropriate, since systems with this kind of conversation usually have functionality or a business process that allows the user to compensate for this lack of atomicity (allowing the user to roll back steps of the conversation manually, for example).

Now that we have our use case, let's look at the different ways we can implement it. We'll start with an approach we don't recommend.

10.2.2 Doing it the hard way

The hard way to implement conversations is to discard all persistent instances between each request. The justification for this approach is that, since the database transaction is ended, the persistent instances are no longer guaranteed to be in a state that is consistent with the database. The longer the administrator spends deciding whether to approve the auction, the greater the risk that some other user has edited the auction details and that the `Item` instance now holds stale data.

Suppose our first request executed the following code to retrieve the auction details:

```
public Item ViewItem(long itemId) {
    return itemDAO.findById(itemId);
}
```

This line of thinking would advise us to discard the returned `Item` after displaying it, storing only the identifier value for use in the next request. It seems superficially reasonable that we should retrieve the `Item` instance again at the start of the second request. We could then be certain that the `Item` held non-stale data for the duration of the second database transaction.

There is one problem with this notion: The administrator already used the possibly stale data to arrive at the decision to approve! Reloading the `Item` in the second request is useless, since the reloaded state *will not be used for anything*—at least, it can't be used in deciding whether the auction should be approved, which is the important thing.

In order to ensure that the details that were viewed and approved by the administrator are still the current details during the second database transaction, we must perform an explicit *manual version check*. The following code demonstrates how this could be implemented by the business layer:

```
public void ApproveAuction(long itemId,
                          int itemVersion,
                          long adminId) {

    Item item = itemDAO.findById(itemId);

    if ( itemVersion != item.Version )
        throw new StaleItemException();

    User admin = userDAO.findById(adminId);
    item.Approve(admin);
}
```

In this case, the manual version check isn't especially difficult to implement.

Are we justified in calling this approach *hard*? In more complex cases involving relationships, it's tedious to perform all the checks manually for all objects that are to be updated. These manual version checks should be considered noise—they implement a purely systemic concern not expressed in the business problem.

More important, the previous code snippet contains other unnecessary noise. We already retrieved the `Item` and `User` in previous requests. Is it necessary to reload them in each request? It should be possible to simplify our control code to the following:

```
public ApproveAuction(Item item, User admin) {
    item.Approve(admin);
}
```

Doing so not only saves three lines of code, but is also arguably more object oriented—our system is working mainly with domain model instances instead of passing around identifier values. Furthermore, this code would be quicker, since it saves two SQL `SELECT` queries that uselessly reload data. How can we achieve this simplification using NHibernate?

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

10.2.3 Using detached persistent objects

Suppose we kept the `Item` as a detached instance (this approach is common in Windows applications). We could reuse it in the second database transaction by re-associating it with the new NHibernate session using either `Lock()` or `Update()`. Let's see what these two options look like.

In the case of `Lock()`, we adjust the `ApproveAuction()` method to look like this:

```
public void ApproveAuction(Item item, User admin) {
    try {
        NHibernateHelper.GetCurrentSession()
            .Lock(item, LockMode.None);
    } catch (HibernateException ex) {
        throw new InfrastructureException(ex);
    }
    item.Approve(admin);
}
```

The call to `ISession.Lock()` re-associates the item with the new NHibernate session and ensures that any subsequent change to the state of the item is propagated to the database when the session is flushed (for a discussion of the different `LockModes`, see chapter 6, section 6.1.8, "Using pessimistic locking").

Since `Item` is versioned (if we map a `<version>` property), NHibernate will check the version number when synchronizing with the database, using the mechanism described in chapter 6, section 6.2.1, "Using managed versioning." You therefore don't have to use a pessimistic lock, as long as it would be allowed for concurrent transactions to read the item in question while the approval routine runs.

Of course, it would be better to hide NHibernate code in a new DAO method, so we add a new `Lock()` method to the `ItemDAO`. This allows us to simplify the `ApproveAuction()` method to

```
public ApproveAuction(Item item, User admin) {
    itemDAO.Lock(item, false); // false = don't be pessimistic
    item.Approve(admin);
}
```

Alternatively, we could use `Update()`. For our example, the only real difference is that `Update()` may be called after the state of the item has been modified, which would be the case if the administrator made changes before approving the auction:

```
public ApproveAuction(Item item, User admin) {
    item.Approve(admin);
    itemDAO.MakePersistent(item);
}
```

Again, NHibernate will perform a version check when updating the item.

Is this implementation, using detached objects really any simpler than the hard way? We still need an explicit call to the `ItemDAO`, so the point is arguable. In a more complex example involving associations, we'd see more benefit, since the call to `Lock()` or `Update()` might cascade to associated instances. Moreover, let's not forget that this implementation is more efficient, avoiding the unnecessary `SELECTs`.

Nevertheless, we're still not satisfied. Is there a way to avoid the need for explicit re-association with a new session? One way would be to use the same NHibernate session for both database transactions, a pattern we described in chapter 6 as *session-per-conversation* or *long session*.

10.2.4 Using the session-per-conversation pattern

A *long session* is a NHibernate session that spans a whole conversation, allowing reuse of persistent instances across multiple database transactions. This approach avoids the need to re-associate detached instances created or retrieved in previous database transactions.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

A session contains two important kinds of state: It holds a cache of persistent instances and an ADO.NET IDbConnection. We've already stressed the importance of not holding database resources open across multiple requests. Therefore, the session needs to release its connection between requests, if you intend to keep it open for more than one request.

As explained in chapter 5, section 651.4, "Understanding connection release modes", NHibernate 1.2 keeps the connection open from the first time it is needed to the moment the transaction is committed. Therefore, committing the transaction at the end of each request is enough to close the connection.

In the listing 10.5, we used the web current session context to store the session. This strategy uses HttpContext. However, this context doesn't outlive the Web request. Therefore, we need a different solution.

Note that, in the case of a Windows application, this is not a problem. The current session context (CallContext for example) will have the same lifespan as the application. The only work that is required is to correctly delimitate the conversation.

In an ASP.NET application, it is possible to keep the NHibernate session in the ASP.NET session state so that it stays available from one request to another. We are going to use this approach in the following example.

Let's write a new Web module called NHibernateConversationWebModule. It will store the NHibernate session between requests instead of discarding it. It will also handle the re-attaching of the session to the new context.

Listing 10.6 The NHibernateConversationWebModule for conversations

```
public class NHibernateConversationWebModule : IHttpModule {
    const string NHibernateSessionKey = "NHIA.NHibernateSession";           |1
    const string EndOfConversationKey = "NHIA.EndOfConversation";           |2
    public static void EndConversationAtTheEndOfThisRequest() {
        HttpContext.Current.Items[EndOfConversationKey] = true;
    }
    public void Init(HttpApplication context) {                               |3
        context.PreRequestHandlerExecute +=
            new EventHandler(OnRequestBeginning);
        context.PostRequestHandlerExecute +=
            new EventHandler(OnRequestEnding);
    }
    public void Dispose() {
    }
    private void OnRequestBeginning(object sender, EventArgs e) {           |4
        // Continuing a conversation?
        ISession currentSession =
            (ISession)HttpContext.Current.Session[NHibernateSessionKey]; |5
        if (currentSession == null) {                                       |6
            // New conversation
            currentSession = NHibernateHelper.OpenSession();
            currentSession.FlushMode = FlushMode.Never;                   |7
        }
        CurrentSessionContext.Bind(currentSession);                       |8
        currentSession.BeginTransaction();
    }
    private void OnRequestEnding(object sender, EventArgs e) {             |9
        // Unbinding Session after processing
        ISession currentSession =                                         |10
            CurrentSessionContext.Unbind(NHibernateHelper.SessionFactory);
```

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>


```

// End or continue the long-running conversation?
if (HttpContext.Current.Items[EndOfConversationKey] != null) { |11
    currentSession.Flush(); |12
    currentSession.Transaction.Commit();
    currentSession.Close();
    HttpContext.Current.Session[NHibernateSessionKey] = null;
}
else { |13
    currentSession.Transaction.Commit();
    HttpContext.Current.Session[NHibernateSessionKey] = currentSession;
}
}
}
}

```

As we will be storing the NHibernate session in a map, we need a key #1 to define its location. As the business logic is responsible of ending the conversation, it needs to call the `EndConversationAtTheEndOfTheRequest()` method #2 to set a value in the current context that will be used at the end of the request. The initialization of the module #3 registers events for the beginning and the end of requests. Note that we are not using the same events as in listing 11.5 because these one allow us to access the ASP.NET session state.

When beginning a new request #4, if a conversation is already running, we extract its detached NHibernate session from the ASP.NET session state #5. Otherwise #6, we start a new conversation by opening a new session. We will explain why we set its flush mode to never #7 in the next section. Once we have the NHibernate session of the running conversation, we bind it to the current context #8 and we begin a new transaction. At this point, the conversation is ready to be used anywhere inside this web request.

When the time to end the request comes #9, we detach its NHibernate session from the current context #10. If the value to end the conversation was set #11, we manually flush the session #12 to process all the changes done in the conversation, we commit these changes, we close the session, and we remove it from the ASP.NET session state. If the conversation is just suspended #13, we commit the transaction to close its database connection and we store the session in the ASP.NET session state. This conversation will resume when the next request starts.

This implementation is not complete because the exception-handling part is missing. Refer to the source code of `CaveatEmptor` for an example. Basically, these methods should be inside a `try/catch` statement. When catching an exception, we should rollback the current transaction and detach then close the current session. There is also another issue that we will cover in the next section.

Now, let's see how we would use this conversation module (don't forget to register it). In our example, the administrator is viewing, and then approving an auction. Therefore, we will have a conversation that spans two web requests: The first to view the action and the second to approve it. The following ASP.NET web page display the auction's item when loading (first request) and it provides a button to approve this auction (second request):

```

public partial class ApproveItem : System.Web.UI.Page {
    ...
    const string ItemKey = "NHIA.ItemKey";

    protected void Page_Load(object sender, EventArgs e) {
        if (!IsPostBack) {
            long itemId = long.Parse(Context.Request.QueryString["Id"]);

            // First request implicitly start the conversation
            Item item = itemDAO.FindById(itemId);
            Session[ItemKey] = item;
            editItemName.Text = item.Name; // ... Show the item
            btnApprove.Click += new EventHandler(btnApprove_Click);
        }
    }
}

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

    }
    protected void btnApprove_Click(object sender, EventArgs e) {
        // Second request uses the running conversation and ends it
        Item item = (Item) Session[ItemKey];
        item.Approve(loggedUser);
        NHibernateConversationWebModule.EndConversationAtTheEndOfThisRequest();
        Context.Response.Redirect("Default.aspx");
    }
}

```

In this example, we store the item in the ASP.NET session state between the requests. Don't forget to enable it.

In the case of Windows application, the implementation of a conversation would be simpler because we can store everything locally. It is also possible to mimic this example by using the `HttpContext` class.

It is also possible to support canceling a conversation. All you have to do is replace the `EndConversationAtTheEndOfTheRequest()` method by two methods: One to accept the changes and another one to cancel them. Then, you will have to distinguish these values when ending the conversation. Canceling a conversation is done by closing the session without flushing it.

There is an exception to this solution. In order to understand this problem, we need to explain some theory behind our implementation of conversations. Then, we will explain how to deal with this problem.

Guaranteeing atomicity and compensating changes

As you saw in listing 10.6, we set the flush mode of the opened session to *never*. It is time to explain the reason behind that.

As explained before, a conversation is supposed to behave like a database transaction. One of its requirements is to be atomic. In order to guarantee the atomicity of a conversation, all the changes done by the requests should be only committed when ending the conversation.

However, by default, committing a transaction makes NHibernate commit the detected changes. This is because the session is flushed at that moment: The session collects all the changes done since it is open and executes the corresponding SQL commands.

The solution to change this behavior is to set the NHibernate session to `FlushMode.Never` and explicitly flush it at the end of the conversation. All changes are held in memory (actually, in the NHibernate session) until the explicit flush. Note that queries won't be aware of these un-flushed changes and might return stale data.

Now the exception: When saving an entity whose identifier is generated by the database (for example, when using native or identity), NHibernate must save this entity immediately in order to retrieve its identifier. The reason behind this behavior is that the method `ISession.Save()` must return this identifier. Depending on your database's behavior, this saving might have additional side effects.

If any permanent change like that happens in an early request and that the conversation must be canceled, it will be necessary to execute some compensation actions to revert these changes. This is also the case when the NHibernate session throws an exception: The conversation must be cancelled and the session closed.

You may take a look at the `IInterceptor` API (used in chapter 9, section 9.4) to keep track of these permanent changes and revert them if necessary. Alternatively, you may simply avoid these generators.

Note that, you may want conversations to persist changes at each request. It may allow you to provide a recovery feature in case of system failure, for example. However, remember that you will also have to provide compensation actions whenever a conversation can be cancelled.

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

There's one final potential complication relating to the long session approach. NHibernate `ISession` implementation isn't thread-safe. So, if an environment allows multiple requests from the same user to be processed concurrently, it's possible that these concurrent requests could obtain the same NHibernate `ISession` instance. This would result in unpredictable behavior. This problem also affects the previous approach, where we used detached objects, since detached objects also aren't thread-safe. Indeed, this problem affects any application that keeps mutable state in a non thread-safe cache.

Since this is a generic problem, we'll leave it to you to find an appropriate solution if you are confronted with it. A good solution for some applications might be to reject any new request if a request is already being processed for the same user. Other applications might need to serialize requests from the same user. Note that this is not a problem for Web applications using the ASP.NET session state because concurrent requests (from the same user) are automatically serialized; the second requests wait that the first completes.

Now that we have covered three different ways to deal with conversations, you might be confused when trying to choose one for your application. *When* are each of the three conversation approaches we've discussed relevant?

10.2.5 Choosing an approach to conversations

You can probably guess from the fact that we called something the hard way that we don't think it's a good technique. We wouldn't use this approach in our own applications. However, if your architecture specifies that the web tier should never access the domain model directly (and so the domain model is completely hidden from the presentation layer behind an intermediate DTO abstraction layer), and if you're unable to keep state associated with the user because you are using a stateless framework, then you have essentially no other choice. It's possible to build NHibernate applications this way, and NHibernate was designed to support this approach. At least this approach frees you from having to consider the difference between persistent and detached instances, and it eliminates the possibility of `LazyInitializationExceptions` thrown by detached objects.

Currently, most NHibernate applications choose the detached objects approach, with a new session per database transaction. In particular, this is the method of choice for an application where business logic and data access execute in the Data Access Layer but where the domain model is also used in the presentation tier, avoiding the need for tedious DTOs. This approach is even being used successfully in Windows applications. We're inclined to think that there are many cases in which it isn't the best approach, however.

There are many cases (most of them quite complex) in which we'd use the long session approach in a Windows application. So far, we've found this approach difficult to explain, and it isn't well understood in the NHibernate community. We suppose this is because the notion of a conversation isn't well understood in the developer community, and most developers aren't used to thinking about problems in terms of conversations. We hope this situation changes soon, because this idea is useful even if you don't use the long session approach.

The next step is to see how we can take this code and adapt it to run in Enterprise Services application. Obviously, we'd like to change as little as possible. We've been arguing all along that one advantage of POCOs and transparent persistence is portability between different runtime environments. If we now have to rewrite all the code for placing a bid, we're going to look a bit silly.

10.3 Using NHibernate in an Enterprise Services application

By *Enterprise Services application*, we mean an application that takes advantage of the distributed transaction service of .NET Enterprise Services. Chapter 5 contains a brief explanation of the steps required to make a NHibernate application participate in a distributed transaction. Now, we are going to implement this approach.

Before that, we must cover an issue linked to inter-process requests. Whenever you have physically separated components/tiers, you must minimize the communication between them. This is important because latency is added by every inter-process request, increasing the application response time and

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

reducing concurrency due to the need for either more database transactions or longer transactions. Moreover, it influences the scalability of your application.

Therefore, it's essential that all data access related to a single user request occur within a single request to the persistence layer. This means you can't use a lazy approach, where the presentation layer pull data as needed. Instead, the business layer must accept responsibility for fetching all data that will be needed subsequently by the presentation layer.

The ubiquitous *data-transfer object* (DTO) pattern provides a way of packaging together the data that the presentation layer will need. A DTO is a class that holds the state of a particular entity; you can think of a DTO as a POCO without any business methods. But as DTOs tend to duplicate entities, we naturally find ourselves questioning the need for DTOs.

10.3.1 Rethinking data transfer objects

DTOs are commonly used to totally separate the presentation tier from the domain model. There are certain reasonable arguments in favor of this approach. However, you shouldn't mistake these arguments for the real reason why DTOs are so useful.

The idea behind the DTO pattern is that fine-grained remote access is slow and un-scalable. It also happened to be useful when the domain model can not be made serializable. In this case, another object must be used to package and carry the state of the business objects between tiers.

There are now twin justifications for the use of DTOs: first, DTOs implement *externalization* of data between tiers; second, DTOs enforce *separation* of the presentation tier from the business logic tier. Only the second justification applies to us, and the benefit of this separation is questionable when weighed against its cost. We won't tell you to never use DTOs (in other places, we're sometimes less reticent). Instead, we'll list some arguments for and against use of the DTO pattern in an application that uses NHibernate and ask you to carefully weigh these arguments in the context of your own application.

It's true that the DTO removes the direct dependency of the presentation tier on the domain model. If your project partitions the roles of .NET developer and web designer, this might be of some value. In particular, the DTO lets you flatten domain model associations, transforming the data into a format that is perhaps more convenient for presentation purposes (it may greatly ease *data binding*). However, in our experience, it's normal for all layers of the application to be highly coupled to the domain model, with or without the use of DTOs. We don't see anything wrong with that, and we suggest that it might be possible to embrace the fact.

The first clue that something is wrong with DTOs is that, contrary to their title, they aren't objects at all. DTOs define state without behavior. This is immediately suspect in the context of object-oriented development. Even worse, the state defined by the DTO is often identical to the state defined in the business objects of the domain model—the supposed separation achieved by the DTO pattern could also be viewed as mere *duplication*.

The DTO pattern exhibits two of the code smells described in [Fowler 1999]. The first is the *shotgun change* smell, where a small change to some system requirement requires changes to multiple classes. The second is the *parallel class hierarchies* smell, where two different class hierarchies contain similar classes in a one-to-one correspondence. The parallel class hierarchy is evident in this case—systems that use the DTO pattern have `Item` and `ItemDTO`, `User` and `UserDTO`, and so on. The shotgun change smell manifests itself when we add a new property to `Item`. We must change not only the presentation tier and the `Item` class, but also the `ItemDTO` and the code that assembles the `ItemDTO` instance from the properties of an `Item` (this last piece of code is especially tedious and fragile).

Of course, DTOs aren't all bad. The code we just referred to as “tedious and fragile”—the *assembler*—does have some value even in the context of NHibernate. DTO assembly provides you with a convenient point at which to ensure that all data the presentation tier will need are fully fetched

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

before returning control to the presentation tier. If you find yourself wrestling with NHibernate LazyInitializationExceptions in the presentation tier, one possible solution is to try the DTO pattern, which naturally imposes extra discipline by requiring that all needed data are copied explicitly from the business objects (we don't find that we need this discipline, but your experience may vary).

Finally, DTOs may have a place in data transfer between loosely coupled applications (our discussion has focused on their use in data transfer between tiers of the *same* application). However, typed DataSet seems to be better adapted to this problem.

A typed DataSet can be seen as a special kind of DTO. There are definitively good reasons to use DataSet: There is an extensive toolset available and many existing libraries use DataSet. However, writing custom classes as DTO gives a better control over the design of your application even if it is tedious to write.

We won't use DTOs in the CaveatEmptor application. Now that we have covered the potential issue which may occur when dealing with physically separated tiers, we can go back to distributed transactions.

10.3.2 Implementation of a distributed transactions enabled NHibernateHelper class

There are few changes which must be applied to the previous NHibernateHelper class in order to enable distributed transactions. First, we must add a reference to the System.EnterpriseServices assembly and change the class definition like this:

```
[Transaction(TransactionOption.Supported)]
public class NHibernateHelper : ServicedComponent {
    ...
}
```

We must also add the following methods to simply the management of the transaction: BeginTransaction(), CommitTransaction() and RollbackTransaction(). They will be used to create, commit/rollback and close the distributed transaction. Here are the two first methods:

```
public static void BeginTransaction() {
    ServiceConfig sc = new ServiceConfig();
    sc.Transaction = TransactionOption.RequiresNew;
    ServiceDomain.Enter(sc);
}

public static void CommitTransaction() {
    try {
        ContextUtil.SetComplete();
        ServiceDomain.Leave();
    }
    catch(HibernateException ex) {
        throw new InfrastructureException(ex);
    }
}
```

In order to take part to the distributed transaction, the NHibernate session's transaction must be enlisted. Note that we have to use the .NET Reflection here because the method used is not part of the IDbConnection interface.

```
private static void TryEnlistDistributedTransaction(ISession session) {
    if (ContextUtil.IsInTransaction) {
        IDbConnection conn = session.Connection;
        MethodInfo mi = conn.GetType().GetMethod(
            "EnlistDistributedTransaction",
            BindingFlags.Public | BindingFlags.Instance );

        if (mi != null)
            mi.Invoke( conn,
                new object[] {
                    (System.EnterpriseServices.ITransaction)

```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

```

        ContextUtil.Transaction } );
    }
}

```

Note that, in the case that the `EnlistDistributedTransaction()` method isn't available, this code will silently fail to enlist the transaction. If it represents an error in your use-case, you should throw an exception. You can even avoid the reflection if you know which type of database connection is used.

Assuming that the application must always try to enlist the distributed transaction, you can change the implementation of the method `OpenSession()` to:

```

public static ISession OpenSession() {
    ISession session = SessionFactory.OpenSession();
    TryEnlistDistributedTransaction(session);
    return session;
}

```

All that is left to do is to update the code to use these new methods. Also, don't forget to register the resulting COM+ assembly (which must be signed) before using it. The tool used for this operation is the command-line executable `RegSvcs`.

We've come to the end of our discussion on the persistence layer. Let's review what we cover in this chapter.

10.4 Summary

This chapter focused on the design of the persistence layer. We first introduced the `NHibernateHelper` class that is useful to abstract the initialization of NHibernate. We also showed how to move from a monolithic method mixing all the concerns to a neat architecture with a clear separation between the layers.

We illustrated a "smart" domain model by implementing business logic in the `CaveatEmptor Item` class. This was the first step of our series of refactoring.

We used the DAO pattern to create a facade for the persistence layer, hiding NHibernate internals from the other layers. We also introduced the `ISessionFactory.GetCurrentSession()` API and the notion of context. We used this feature to significantly improve our DAOs by making them share the same session without having to pass it as a parameter and without using a global static session.

After that, we leveraged the .NET 2.0 generics to reduce the redundancies in the persistence layer. We also designed our persistence layer so that the other layers are completely unaware of the persistence framework that is used. We even made it possible to switch from one implementation to another by changing a single line of code.

We also explained how to make a session live for a whole web request. This is useful to guarantee that a single session is used for the whole processing and that lazy loading will always work transparently.

Chapter 5 introduced the notion of conversation (also called application/business transactions). In this chapter, we provided three ways of implementing them: The *hard-way*, which may be your only choice in some constrained environments; the approach using detached persistent objects, useful in stateless environment and the approach using long-living sessions. This last approach is still relatively unknown. However, it is very powerful, especially in a rich environment.

This chapter ends with improvement of the `NHibernateHelper` class in order to support Enterprise Services transactions. We discussed the potential latency issue and usefulness of the DTO pattern. Although this pattern can be useful when decoupling the domain model is important, we agree that such situation is rare and that the cost of maintaining the DTO is too high.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

At this point, you should have all the technical knowledge that you may need to leverage the features of NHibernate. It is a powerful tool, but it requires a deep understanding of its behavior to be correctly used.

Note that you are not done reading this book yet. You will discover, in the appendixes, few more tools and ideas that will improve your experience working with NHibernate.

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

A

SQL fundamentals

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

This book assumes a basic understanding of relational databases and the *Structured Query Language* (SQL). Moreover, it will be easier for you to learn some of the advanced features of NHibernate if you already have a sound knowledge of SQL. This appendix gives a brief overview of the fundamentals of SQL. Therefore, we highly recommend that you find a book to learn more about it. There are some recommendations in chapter 1, section 1.1.1.

A table, with its rows and columns, is a familiar sight to anyone who has worked with an SQL database. Sometimes you'll see tables referred to as relations, rows as tuples, and columns as attributes. This is the language of the relational data model, the mathematical model that SQL databases (imperfectly) implement.

The relational model allows you to define data structures and constraints that guarantee the integrity of your data (for example, by disallowing values that don't accord with your business rules). The relational model also defines the relational operations of restriction, projection, Cartesian product, and relational join [Codd 1970]. These operations let you do useful things with your data, such as summarizing or navigating it.

Each of the operations produces a new table from a given table or combination of tables. SQL is a language for expressing these operations in your application (therefore called a data language) and for defining the base tables on which the operations are performed.

You write SQL DDL statements to create and manage the tables. We say that DDL defines the database schema. Statements such as CREATE TABLE, ALTER TABLE, and CREATE SEQUENCE belong to DDL.

You write SQL DML statements to work with your data at runtime. Let's describe these DML operations in the context of tables from the CaveatEmptor application.

In CaveatEmptor, we naturally have entities like *item*, *user*, and *bid*. We assume that the SQL database schema for this application includes an ITEM table and a BID table, as shown in figure A.1. The data-types, tables, and constraints for this schema have been created with SQL DDL (CREATE and ALTER operations).

ITEM		
ITEM_ID	NAME	INITIAL_PRICE
1	Foo	2.00
2	Bar	50.00
3	Baz	1.00

BID		
BID_ID	ITEM_ID	AMOUNT
1	1	10.00
2	1	20.00
3	2	55.50

Figure A.1 The ITEM and BID tables of an auction application

Insertion is the operation of creating a new table from an old table by adding a row. SQL databases perform this operation in place, so the new row is added to the existing table:

```
insert into ITEM values (4, 'Fum', 45.0)
```

An SQL update modifies an existing row:

```
update ITEM set INITIAL_PRICE = 47.0 where ITEM_ID = 4
```

A *deletion* removes a row:

```
delete from ITEM where ITEM_ID = 4
```

The real power of SQL lies in querying data. A single query might perform many relational operations on several tables. Let's look at the basic operations.

First, *restriction* is the operation of choosing rows of a table that match a particular criterion. In SQL, this criterion is the expression that occurs in the *where* clause:

```
select * from ITEM where NAME like 'F%'
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

Projection is the operation of choosing columns of a table and eliminating duplicate rows from the result. In SQL, the columns to be included are listed in the `select` clause. You can eliminate duplicate rows by specifying the `distinct` keyword:

```
select distinct NAME from ITEM
```

A *Cartesian product* (also called a *cross join*) produces a new table consisting of all possible combinations of rows of two existing tables. In SQL, you express a Cartesian product by listing tables in the `from` clause:

```
select * from ITEM i, BID b
```

A relational *join* produces a new table by combining the rows of two tables. For each pair of rows for which a *join condition* is true, the new table contains a row with all field values from both joined rows. In ANSI SQL, the `join` clause specifies a table join; the join condition follows the `on` keyword. For example, to retrieve all items that have bids, you join the `ITEM` and the `BID` table on their common `ITEM_ID` attribute:

```
select * from ITEM i inner join BID b on i.ITEM_ID = b.ITEM_ID
```

A join is equivalent to a Cartesian product followed by a restriction. So, joins are often instead expressed in theta style, with a product in the `from` clause and the join condition in the `where` clause. This SQL theta-style join is equivalent to the previous ANSI-style join:

```
select * from ITEM i, BID b where i.ITEM_ID = b.ITEM_ID
```

Along with these basic operations, relational databases define operations for aggregating rows (`GROUP BY`) and ordering rows (`ORDER BY`):

```
select b.ITEM_ID, max(b.AMOUNT)
from BID b
group by b.ITEM_ID
having max(b.AMOUNT) > 15
order by b.ITEM_ID asc
```

SQL was called a *structured* query language in reference to a feature called *subselects*. Since each relational operation produces a new table from an existing table or tables, an SQL query might operate on the result table of a previous query. SQL lets you express this using a single query, by nesting the first query inside the second:

```
select *
from (
    select b.ITEM_ID as ITEM, max(b.AMOUNT) as AMOUNT
    from BID b
    group by b.ITEM_ID
)
where AMOUNT > 15
order by ITEM asc
```

The result of this query is equivalent to the previous one.

A subselect may appear anywhere in an SQL statement; the case of a subselect in the `where` clause is the most interesting:

```
select * from BID b where b.AMOUNT >= (select max(c.AMOUNT) from BID c)
```

This query returns the largest bid in the database. `where` clause subselects are often combined with *quantification*. The following query is equivalent:

```
select * from BID b where b.AMOUNT >= all(select c.AMOUNT from BID c)
```

Please post comments or corrections to the Author online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=295>

An SQL restriction criterion is expressed in a sophisticated expression language that supports mathematical expressions, function calls, string matching, and even more sophisticated features such as full text searches:

```
select * from ITEM i
  where lower(i.NAME) like '%ba%'
         or lower(i.NAME) like '%fo%'
```

There are many other operations in SQL. You may take a look at the chapter 8 to have an idea of the operations that you want to learn.

D

Going Forward

Please post comments or corrections to the Author online forum at
<http://www.manning-sandbox.com/forum.jspa?forumID=295>

You have reached the end of this book. What is left for us to do is give you some guidance and advice to get started and to master NHibernate.

In this appendix, we enumerate the requirements to use NHibernate. Then, we give you a roadmap to progressively master NHibernate and keep yourself up-to-date. Finally, we encourage you to discover the internals of NHibernate, contribute and help improve NHibernate.

This book assumes that you have some experience of .NET programming. Therefore, before starting to use NHibernate, you should already have the .NET framework and an Integrated Development Environment (IDE) like Visual Studio or SharpDevelop. Note that you can also use Mono (<http://www.mono-project.com/>) which runs on other operating systems like Linux.

NHibernate binaries, source code and documentation are available on its SourceForge website: <http://sourceforge.net/projects/nhibernate/>. SourceForge.net is a website providing free hosting to Open Source Software (OSS) development projects.

Before NHibernate 1.2, there were two packages available: `nhibernate` containing the core binaries with its source code and the documentation and `NHibernateContrib` containing optional useful add-ons for NHibernate. Now, they are merged. Moreover, you will need NHibernate 1.2 or later to take advantage of .NET 2.0 generics and nullables.

You can use NHibernate with most popular database systems. You can find the complete list here: <http://www.hibernate.org/361.html>.

This is all you need to start using NHibernate. The next step is to practice using NHibernate, from the simple Hello World example of chapter 2 to a more complex application like `CaveatEmptor`.

We highly encourage you to develop your own applications to test the core features of NHibernate and then integrate the advanced features that interest you. At this point, this book will serve you as a reference book.

You should make sure that you understand the mapping of entities and their persistence lifecycle and the way NHibernate sessions work. Take also special care of the way you use NHibernate caching. Finally, the three last chapters explain how important the architecture of the application is.

You will often encounter some problems while using NHibernate. In most cases, their source is the misuse of some features due to a lack of understanding. For more advices on problem solving techniques, read chapter 9, section 9.3. Note that a well-thought design helps a lot to avoid and solve problems, so take some time to think about the features that you want to use (algorithms) and the architecture of your application (layers, separation of concern, etc).

In case you are still unable to overcome your difficulties, feel free to ask for help on the NHibernate forum: <http://forum.hibernate.org/viewforum.php?f=25>. Make sure that you explain your problem in details, with the logs and error messages.

NHibernate is constantly evolving. Therefore, once you feel comfortable using it, you should keep yourself up-to-date because some new features can improve your applications' capabilities and performance.

The best way to do that is to regularly read and participate on the NHibernate forum. It is also a great place to share your point of view on various NHibernate related problems, get feedback about them, and learn how other people solve them.

There are also many informative and useful resources (documentations, samples, open source projects) available on the Internet. You can find a comprehensive list here: <http://www.hibernate.org/365.html>.

NHibernate has a bug-tracking website: <http://jira.nhibernate.org/>. You can register on this website and report the bugs that you encounter (in case you are not sure that it is a bug, use the NHibernate forum first); you can also request new features.

Since NHibernate is an OSS, its source code is freely available. Instead of using the compiled library, use the source code to gain more details when debugging your application.

Moreover, feel free to modify it whenever you need to add a new feature that your application needs or to fix an existing bug. Don't forget to make this addition publicly available so that other NHibernate user can use and even improve it. You can do that by submitting a *patch* in the bug-tracking website (it is a file containing the changes that you made in the source code).

The bug-tracking website also gives you an idea of the features and bug fixes that are already available in the current version of NHibernate (which the NHibernate developers are working on). The source code of this version is hosted by Sourceforge.net in a SVN repository. If you are interested in using this version, read "Getting Started with the NHibernate Source Code" at <http://www.hibernate.org/428.html> and go to http://sourceforge.net/svn/?group_id=73818. Note that, although this version is generally stable, it can temporally become unstable by time to time.

After starting to use the SVN version of NHibernate, there is only one last step to embrace NHibernate completely: Joining the development list. It is a mailing list used by the developers of NHibernate to discuss its evolution. You can start by registering and reading the archive here: <http://lists.sourceforge.net/mailman/listinfo/nhibernate-development>.

This is the end of the book and, hopefully, the beginning of a wonderful experience with NHibernate.

Bon voyage!