

NHibernate in Action

PIERRE HENRI KUATÉ
TOBIN HARRIS
CHRISTIAN BAUER
GAVIN KING



MANNING

Greenwich
(74° w. long.)

For online information and ordering of this and other Manning books, please visit www.manning.com. The publisher offers discounts on this book when ordered in quantity. For more information, please contact


Special Sales Department
Manning Publications Co.
Sound View Court 3B fax: (609) 877-8256
Greenwich, CT 06830 email: orders@manning.com

©2009 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

© Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15% recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
Sound View Court 3B
Greenwich, CT 06830

Development Editor: Cynthia Kane
Copyeditor: Tiffany Taylor
Typesetter: Gordan Salinovic
Cover designer: Leslie Haimes

ISBN 978-1-932394-92-4

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 – MAL – 14 13 12 11 10 09

contents

foreword xvii
preface xxi
acknowledgments xxiii
about this book xxv

PART 1 DISCOVERING ORM WITH NHIBERNATE 1

1 *Object/relational persistence in .NET* 3

1.1 What is persistence? 5

Relational databases 5 ▪ *Understanding SQL* 6 ▪ *Using SQL in .NET applications* 6 ▪ *Persistence in object-oriented applications* 6 ▪ *Persistence and the layered architecture* 7

1.2 Approaches to persistence in .NET 9

Choice of persistence layer 9 ▪ *Implementing the entities* 11
Displaying entities in the user interface 13 ▪ *Implementing CRUD operations* 14

1.3 Why do we need NHibernate? 15

The paradigm mismatch 15 ▪ *Units of work and conversations* 16
Complex queries and the ADO.NET Entity Framework 18

- 1.4 Object/relational mapping 21
 - What is ORM? 21 ▪ Why ORM? 21*
- 1.5 Summary 23

2 *Hello NHibernate!* 24

- 2.1 “Hello World” with NHibernate 25
 - Installing NHibernate 25 ▪ Create a new Visual Studio project 25*
 - Creating the Employee class 26 ▪ Setting up the database 27*
 - Creating an Employee and saving to the database 27 ▪ Loading an Employee from the database 29 ▪ Creating a mapping file 29*
 - Configuring your application 31 ▪ Updating an Employee 32 ▪ Running the program 33*
- 2.2 Understanding the architecture 33
 - The core interfaces 35 ▪ Callback interfaces 36 ▪ Types 37*
 - Extension interfaces 37*
- 2.3 Basic configuration 38
 - Creating a SessionFactory 38 ▪ Configuring the ADO.NET database access 41*
- 2.4 Advanced configuration settings 44
 - Using the application configuration file 44 ▪ Logging 47*
- 2.5 Summary 48

PART 2 NHIBERNATE DEEP DIVE 49

3 *Writing and mapping classes* 51

- 3.1 The CaveatEmptor application 52
 - Analyzing the business domain 52 ▪ The CaveatEmptor domain model 53*
- 3.2 Implementing the domain model 55
 - Addressing leakage of concerns 55 ▪ Transparent and automated persistence 55 ▪ Writing POCOs 56*
 - Implementing POCO associations 58 ▪ Adding logic to properties 61*
- 3.3 Defining the mapping metadata 63
 - Mapping using XML 63 ▪ Attribute-oriented programming 65*

- 3.4 Basic property and class mappings 66
 - Property mapping overview* 66 ▪ *Using derived properties* 68
 - Property access strategies* 68 ▪ *Taking advantage of the reflection optimizer* 70 ▪ *Controlling insertion and updates* 71 ▪ *Using quoted SQL identifiers* 72 ▪ *Naming conventions* 72 ▪ *SQL schemas* 73 ▪ *Declaring class names* 74 ▪ *Manipulating metadata at runtime* 75
- 3.5 Understanding object identity 76
 - Identity versus equality* 76 ▪ *Database identity with NHibernate* 77 ▪ *Choosing primary keys* 79
- 3.6 Fine-grained object models 81
 - Entity and value types* 81 ▪ *Using components* 82
- 3.7 Introducing associations 86
 - Unidirectional associations* 86 ▪ *Multiplicity* 86 ▪ *The simplest possible association* 87 ▪ *Making the association bidirectional* 88
 - A parent/child relationship* 90
- 3.8 Mapping class inheritance 91
 - Table per concrete class* 92 ▪ *Table per class hierarchy* 93 ▪ *Table per subclass* 95 ▪ *Choosing a strategy* 98
- 3.9 Summary 99

4 Working with persistent objects 100

- 4.1 The persistence lifecycle 101
 - Transient objects* 102 ▪ *Persistent objects* 102 ▪ *Detached objects* 103 ▪ *The scope of object identity* 104 ▪ *Outside the identity scope* 105 ▪ *Implementing Equals() and GetHashCode()* 106
- 4.2 The persistence manager 110
 - Making an object persistent* 110 ▪ *Updating the persistent state of a detached instance* 111 ▪ *Retrieving a persistent object* 112
 - Updating a persistent object transparently* 113 ▪ *Making an object transient* 113
- 4.3 Using transitive persistence in NHibernate 114
 - Persistence by reachability* 115 ▪ *Cascading persistence with NHibernate* 116 ▪ *Managing auction categories* 117
 - Distinguishing between transient and detached instances* 120

4.4 Retrieving objects 121

Retrieving objects by identifier 122 ▪ *Introducing Hibernate Query Language 123* ▪ *Query by Criteria 124* ▪ *Query by Example 124* ▪ *Fetching strategies 125* ▪ *Selecting a fetching strategy in mappings 127* ▪ *Tuning object retrieval 132*

4.5 Summary 133

5 **Transactions, concurrency, and caching 134**

5.1 Understanding database transactions 135

ADO.NET and Enterprise Services/COM+ transactions 136
The NHibernate ITransaction API 137 ▪ *Flushing the session 138* ▪ *Understanding connection-release modes 139*
Understanding isolation levels 140 ▪ *Choosing an isolation level 141* ▪ *Setting an isolation level 143* ▪ *Using pessimistic locking 143*

5.2 Working with conversations 146

An example scenario 146 ▪ *Using managed versioning 147*
Optimistic and pessimistic locking compared 149 ▪ *Granularity of a session 150* ▪ *Other ways to implement optimistic locking 151*

5.3 Caching theory and practice 152

Caching strategies and scopes 153 ▪ *The NHibernate cache architecture 155* ▪ *Caching in practice 159*

5.4 Summary 164

6 **Advanced mapping concepts 166**

6.1 Understanding the NHibernate type system 167

Associations and value types 167 ▪ *Bridging from objects to database 168* ▪ *Mapping types 168* ▪ *Built-in mapping types 169* ▪ *Using mapping types 172*

6.2 Mapping collections of value types 181

Storing value types in sets, bags, lists, and maps 181 ▪ *Collections of components 186*

6.3 Mapping entity associations 189

One-to-one associations 189 ▪ *Many-to-many associations 193*

6.4 Mapping polymorphic associations 200

Polymorphic many-to-one associations 201 ▪ *Polymorphic collections 203* ▪ *Polymorphic associations and table-per-concrete-class 204*

6.5 Summary 205

7 Retrieving objects efficiently 207

- 7.1 Executing queries 208
 - The query interfaces* 208 ▪ *Binding parameters* 211 ▪ *Using named queries* 214 ▪ *Using query substitutions* 215
- 7.2 Basic queries for objects 215
 - The simplest query* 215 ▪ *Using aliases* 216 ▪ *Polymorphic queries* 217 ▪ *Restriction* 217 ▪ *Comparison operators* 218 ▪ *String matching* 220 ▪ *Logical operators* 221 ▪ *Ordering query results* 221
- 7.3 Joining associations 222
 - NHibernate join options* 223 ▪ *Fetching associations* 224 ▪ *Using aliases with joins* 226 ▪ *Using implicit joins* 228 ▪ *Theta-style joins* 229 ▪ *Comparing identifiers* 230
- 7.4 Writing report queries 231
 - Projection* 232 ▪ *Using aggregation* 234 ▪ *Grouping* 234 ▪ *Restricting groups with having* 236 ▪ *Improving performance with report queries* 236 ▪ *Obtaining DataSets* 237
- 7.5 Advanced query techniques 238
 - Dynamic queries* 238 ▪ *Collection filters* 240 ▪ *Subqueries* 242
- 7.6 Native SQL 243
 - Using the ISQLQuery API* 244 ▪ *Named SQL queries* 246 ▪ *Customizing create, retrieve, update, and delete commands* 248
- 7.7 Optimizing object retrieval 249
 - Solving the n+1 selects problem* 249 ▪ *Using Enumerable()* queries 252 ▪ *Caching queries* 253 ▪ *Using profilers and NHibernate Query Analyzer* 255
- 7.8 Summary 255

PART 3 NHIBERNATE IN THE REAL WORLD 257

8 Developing NHibernate applications 259

- 8.1 Inside the layers of an NHibernate application 260
 - Using patterns and methodologies* 261 ▪ *Building and testing the layers* 263 ▪ *The domain model* 263 ▪ *The business layer* 266 ▪ *The persistence layer* 268 ▪ *The presentation layer* 269
- 8.2 Solving issues related to .NET features 270
 - Working with web applications* 271 ▪ *.NET remoting* 271

- 8.3 Achieving goals and solving problems 272
 - Design goals applied to an NHibernate application* 272
 - Identifying and solving problems* 274 ▪ *Use the right tool for the right job* 276
- 8.4 Integrating services: the case of audit logging 277
 - Doing it the hard way* 278 ▪ *Doing it the NHibernate way* 278 ▪ *Other ways of integrating services* 283
- 8.5 Summary 284

9 *Writing real-world domain models* 286

- 9.1 Development processes and tools 287
 - Top down: generating the mapping and the database from entities* 288 ▪ *Middle out: generating entities from the mapping* 292 ▪ *Bottom up: generating the mapping and the entities from the database* 293 ▪ *Automatic database schema maintenance* 294
- 9.2 Legacy schemas 296
 - Mapping a table with a natural key* 297 ▪ *Mapping a table with a composite key* 298 ▪ *Using a custom type to map legacy columns* 302 ▪ *Working with triggers* 303
- 9.3 Understanding persistence ignorance 305
 - Abstracting persistence-related code* 305 ▪ *Applying the Observer pattern to an entity* 307
- 9.4 Implementing the business logic 309
 - Business logic in the business layer* 310 ▪ *Business logic in the domain model* 310 ▪ *Rules that aren't business rules* 312
- 9.5 Data-binding entities 312
 - Implementing manual data binding* 313 ▪ *Using data-bound controls* 314 ▪ *Data binding using NHibernate* 315 ▪ *Data binding using ObjectViews* 315
- 9.6 Filling a DataSet with entities' data 316
 - Converting an entity to a DataSet* 316 ▪ *Using NHibernate to assist with conversion* 317
- 9.7 Summary 317

10 *Architectural patterns for persistence* 319

- 10.1 Designing the persistence layer 320
 - Implementing a simple persistence layer* 321 ▪ *Implementing a generic persistence layer* 326

10.2	Implementing conversations	335
	<i>Approving a new auction</i>	336
	▪ <i>Loading objects on each request</i>	337
	▪ <i>Using detached persistent objects</i>	338
	▪ <i>Using the session-per-conversation pattern</i>	340
	▪ <i>Choosing an approach to conversations</i>	344
10.3	Using NHibernate in an Enterprise Services application	345
	<i>Rethinking DTOs</i>	345
	▪ <i>Enabling distributed transactions for NHibernateHelper</i>	346
10.4	Summary	348
<i>appendix A</i>	<i>SQL fundamentals</i>	349
<i>appendix B</i>	<i>Going forward</i>	352
	<i>index</i>	355

foreword

Somewhere in the middle of 2004, I decided that I needed to take a look at additional ways to deal with persistence, beyond store procedures and code generation using Code Smith. At the time, I was mystified by all the noise around ORM, business objects, and domain-driven design. I had data sets and stored procedures, and I had code generation to make working with them a bit easier, and the world was good. But as I began to deal with more complex applications and attempted to learn from the collective knowledge in the community, I began to see the problems with this approach.

Eventually, I understood the significant problem with my previous method of working with data: I was building procedural applications, where the data was king and the application behavior was, at best, a distant second. This approach doesn't scale well with the complexity of the applications we need to build. Indeed, this programmatic approach has been largely superseded by object-oriented approaches. I see no reason that this shouldn't apply to dealing with data as well.

I can no longer recall what made me decide to focus on NHibernate—it was probably an enthusiastic blog post, come to think of it. But whatever the reason, I made that choice. Four years later, I have yet to regret this decision, and I am proud to state that exactly 100 percent of my projects since then have used NHibernate for persistence. That decision has paid off in many ways.

Two occasions come to mind in particular. The first was a very ... tense meeting with a client, where the client DBA was furious about the need to support SQL Server. That was the client's requirement, but the DBA saw it as an encroachment on his territory, and he didn't like it one bit. In his eyes, DB2 on AS/400 was what the client had

used for the last eon or so, and it should be what they used for the next eon or so. During that meeting, I pulled out my laptop, found the ADO.NET provider for DB2, and configured the application to run against it. I asked the DBA for the credentials of the test database and had the application running against it within 45 minutes. We ended up going for production on SQL Server, but that was the client's choice, not an implementation imperative.

On the second occasion, we had to build a fairly complex multi-tenant HR application on top of a legacy database that was imported from a mainframe and was enough to make a person cry. The table names were numeric (of course, table 200 is the employees table) and were different from one tenant to the next, and the database model was a direct copy of the flat files used in batch processing on the mainframe. Trying to build an application on top of that (and it couldn't be changed) would have been challenging, to say the least. We were able to build a domain model that was mostly free of all the nonsense in the DB layer and map from the DB to the domain model for each tenant. I wouldn't call it simple by any means, but we were able to encapsulate the complexity into a set of mapping files that were maintained by the system integrators (who were the only people who understood what value went where).

In both cases, I managed to get tremendous value out of NHibernate. In the first case, it provided a good reputation and the ability to remove hurdles in working with the client; in the second case, we made the problem simpler by an order of magnitude if not more. The team worked mostly on the UI and the business problems, not on solving persistence issues.

I've been using NHibernate since version 0.4 or 0.5, and I have watched (and had the honor of taking part in) how it has grown from a simple port of Hibernate on Java to have a personality, community, and presence of its own. NHibernate 1.0 gave us parity with Hibernate 2.1, with support for common scenarios, but it was still mostly a port of the Java version. Starting with 1.2, we've seen more and more work being done not only to make NHibernate more friendly to the .Net ecosystem, but also to add features that are unique for NHibernate.

NHibernate 1.0 was a good ORM for the time, looking back at it, but it seems barebones compared to the options that we have now with 1.2 and 2.0.

NHibernate 1.2 added support for generics, stored procedures, multiqueries, write batching, and much more. NHibernate 2.0 is focused on parity with Hibernate 3.2, with events and listeners, stateless sessions, joined and unioned classes, detached queries, and much more. On the horizon is a Linq provider for NHibernate, which is being used in production by several projects and will likely be released as part of NHibernate 2.1.

NHibernate is also able to benefit from the ecosystem that grew around Hibernate, and ports of Hibernate's satellite projects exist for NHibernate. I'll mention NHibernate Search, which lets you integrate your entities with the Lucene.NET search engine; and NHibernate Validator, which gives you a powerful validation framework. NHibernate Contrib contains more examples. But the extensions available for NHibernate go beyond ports of Java projects. Rhino Security is a project that

gives you a complete business-level security package on top of the NHibernate domain model, and it uses NHibernate itself to do that. Several projects provide mapping by convention to NHibernate, and a big community of users are sharing knowledge and issues on a daily basis.

This rich ecosystem didn't happen by accident, it happened because NHibernate is a flexible and adaptable framework; and when you come to understand the way it works and how to utilize its strengths, it will bring significant benefits to your projects. But being flexible and adaptable comes at a cost. Many people find that NHibernate has a steep learning curve. I disagree; but as one of the committers for the project, I'm probably not a good person to judge that particular aspect of NHibernate.

When I started with NHibernate, I got *Hibernate in Action* (Christian Bauer and Gavin King; Manning, 2004) and read it from cover to cover. My goal wasn't to memorize the API; my intent was to understand NHibernate—not just the API and how to use it in simple scenarios, but also the design approach and how NHibernate handles issues. To my joy, *Hibernate in Action* contained exactly that kind of information and has been of tremendous value in understanding and using NHibernate.

But *Hibernate in Action* is a Java book, which is why I was happy to hear (and read) about this book. *NHibernate in Action* is not simply a reproduction of *Hibernate in Action* with different naming conventions. This book has accomplished the task of translating the knowledge and of adapting and extending it. I consider this book to be essential for any developer who wants to be able to do more than the basics with NHibernate. And it certainly helps that the book covers NHibernate-specific features, which do not exist in the Hibernate version.

OREN EINI, A.K.A. AYENDE RAHIEN
NHIBERNATE COMMITTER

preface

For as long as I've been interested in software development, the most challenging and fun aspect has always been problem solving: from the business level to more technical levels, I've routinely spent countless hours thinking about the best solution to my current problem.

After discovering the .NET framework, I investigated how to write business applications. I was particularly worried about how I would load and store information in a database. I tested the then-popular DataSet approach and the low-level ADO.NET API. Although this API was easy to set up, it turned out to be inefficient and inflexible, and it simply felt wrong. Anybody who has written countless plumbing code and SQL queries would understand what I mean. Therefore, I did some research and discovered object/relational mapping (ORM) tools. This was exactly what I was looking for: a non-intrusive, object-oriented persistence approach supporting relational databases. I chose NHibernate after testing numerous alternatives because it fitted that description the best.

I remember downloading and testing NHibernate 0.4. It was surprisingly stable and provided the basic features I needed. More than that, it came with a wonderful community of open source developers. Being able to share my thoughts and having developers willing to help each other was one of my best learning experiences. I eventually shipped my first commercial application using NHibernate 0.7. I've used it in countless other projects, and I think I'll continue to use it in the years to come.

When Manning Publications approached Tobin and me about writing a book on NHibernate, we already had an interest in writing tutorials and helping people on the

NHibernate forum. Nonetheless, writing a book was an intimidating challenge! We learned to write in a simple and readable way for the benefit of the reader. It turned out to be an experience that we recommend anyone try at least once.

Although Java developers have used ORM and written about it for years, this technology is still quite obscure to .NET developers. This book explains not only how to use and extend NHibernate but also the theory behind it. We hope that this book will help enlighten you regarding an indispensable technology that's not so simple to learn.

PIERRE HENRI KUATE

acknowledgments

We'd like to first express our thanks to all the core developers, contributors, and other community members who have helped make NHibernate a first-class open source tool. We'd also like to extend our thanks to those who have made the original Java Hibernate a success. Our thanks to Jim Bolla, Mike Doerfler, Paul Hatcher, Sergey Koshcheyev, Demetris Manikas, Fabio Maulo, Donald Mull, Bill Pierce, Dario Quintana, Ayende Rahien, Peter Smulovics, Michael Third, Kailuo Wang, Kevin Williams, and all the other contributors to NHibernate.

As with any book, this one has required huge quantities of time, effort, and patience. We'd like to thank the Manning Publications team for their incredible expertise and know-how. They've continually endeavored to make the best choices possible for the book and helped bring out the best from its authors. In particular, we'd like to thank publisher Marjan Bace, acquisitions editor Mike Stephens, as well as Tiffany Taylor, Katie Tennant, and Megan Yockey for their invaluable expertise, guidance, and feedback. A special thanks goes to our development editors, Frank Blackwell, Jackie Carter, and Cynthia Kane, who patiently initiated us in the art of book writing.

Our technical proofreaders gave their expert advice on the content of the book as we prepared it for publication. Many thanks to Ayende Rahien for reviewing the manuscript and writing a brilliant foreword. Also, thanks to Mark Monster for the questions, amendments, and suggestions he made to the final version of the manuscript.

The following technical reviewers took time out of their busy schedules to read the manuscript at various stages of development and offered their invaluable feedback, making this a much better book: Sergey Koshcheyev, John Tobler, Dan Hounshell,

Alessandro Gallo, Robi Sen, Paul Wilson, Pete Helgren, Oren Eini, Doug Warren, Jim Geurts, Riccardo Audano, and Armand du Plessis.

Before this book went into print, many people purchased the PDF version of the chapters as they were being written through the Manning Early Access Program (MEAP). We'd like to thank those readers for their comments, support, and suggestions throughout the project, especially Adam Cooper, Darren Maidlow, Morten Mertner, Magnus Salgo, Benjamin VanEvery, Jan Van Ryswyck, Fabio Maulo, Paul Anderson, Damon Wilder Carr, Shane Courtrille, Jim Beveridge, Daren Fox, David Gadd, Jason Whitehorn, Gary Murchison, Muhammad Shehabeddeen, and Thomas Koch.

PIERRE HENRI KUATÉ would like to thank his family for always supporting him, and his friends at the Polelo Research Lab for their encouragement all along the way.

TOBIN HARRIS would like to thank his girlfriend, Georgina Reall, for her support, encouragement, and patience throughout the project! He would also like to thank his sister, Marnie, for her help and endless enthusiasm.

about this book

The NHibernate project was started back in 2003 by Paul Hatcher, and with the tremendous work done by Mike Doerfler and Sergei Koshcheyev, it has steadily become a mature product, popular with thousands of .NET developers.

NHibernate was originally a port of the incredibly popular Java Hibernate project, and object/relational mapping has been very popular with the Java crowd for many years.

A consequence of this popularity is that Java developers have access to a whole heap of books about Hibernate. In fact, the last time I counted I found 15 books dedicated purely to this single tool. New books on Hibernate and related technologies are still appearing regularly.

Until now, .NET developers have had no such luxury for learning NHibernate. This book aims to remedy that problem—we finally have our “manual” written for .NET developers and focusing solely on NHibernate. *NHibernate in Action* is based on the best-selling *Hibernate in Action*, which is considered to be the de facto manual for Java Hibernate. The book is much more than a translation; in fact, much work has gone into making it appeal to the .NET developer while also accommodating API changes, code differences, new features, and the like.

We hope that the arrival of this book is considered good timing. The world of .NET is finally getting excited about object/relational mapping, and we hope this book will help you discover, learn, and enjoy one of the most mature, powerful ORM frameworks available.

Who should read this book

This book is written for developers who work with Microsoft .NET. Both developers and architects should be able to draw great value from this book, regardless of

whether they're new to NHibernate and ORM or they've already gained some experience with it.

For those new to NHibernate, this book assumes no prior knowledge. We also don't expect that you've worked with any object/relational mapping framework before. The idea is that that you can take the knowledge in this book and start building NHibernate solutions with it.

We also anticipate that many reading this book might have used NHibernate on a few projects already, either on its own or as part of another library such as Castle Active Record or Spring.NET. This book will help you if you want to learn a little more about what's going on behind the scenes. It will also help you leverage the great features of NHibernate and understand how to take full advantage of them.

We've done our best to give as much background detail as possible on both the common and the not-so-common usages of NHibernate. We've covered many topics that are barely mentioned in forums and blogs, such as the persistence lifecycle and some of the more exotic mapping capabilities.

Regardless of whether you're new to NHibernate or a seasoned user, we hope this book will teach you new things and increase your enjoyment and success with the tool.

Roadmap

Chapter 1 sets the scene, explaining what persistence is and how it fits into business applications. We take a glimpse at NHibernate, comparing it to other popular approaches such as LINQ to SQL and DataSets. You'll then learn about the fundamental problems posed in object/relational mapping and how NHibernate tackles them.

Chapter 2 puts some code under your nose! Our brief tour takes you from installing NHibernate to building and running a simple application. We then go on to explore the main facilities available in NHibernate, including the APIs for querying, transactions, and customization. We round off with both basic and advanced configuration techniques and show how you can use logging to get a deeper insight into how NHibernate operates behind the scenes.

Chapter 3 will bring you up to speed with the bulk of NHibernate's capabilities. We take a more sophisticated problem—the CaveatEmptor application—and guide you through modeling your domain model, along with mapping it using various types of associations. You'll learn how NHibernate allows mapping with XML and the .NET attributes. We also explain some smarter capabilities, such as flexible property mappings and automatic naming conventions. The chapter also explains the importance of identity in ORM, before building on previous knowledge by explaining more about mapping inheritance and associations.

Chapter 4 gives further insight into some important concepts: entity lifecycle, persistent states, and equality. We look at how this knowledge can be leveraged by NHibernate's APIs. We also look at working with entire object graphs, discussing cascading persistence, batching, lazy fetching, and eager fetching.

Chapter 5 delves into using NHibernate to get tight control over database transactions. We then discuss long-running business transactions and demonstrate how to

achieve automatic versioning and locking. Caching is core to NHibernate, and you'll learn a great deal here about the first- and second-level caches.

Chapter 6 introduces the NHibernate type system and how to implement custom user types. We move on to discuss components, value types, and working with the more advanced associations, indicating some best practices when working with them.

Chapter 7 focuses on efficiently querying NHibernate. We examine both HQL and the `ICriteria` API, giving many code samples for each. You'll see glorious detail for parameter binding, named queries, polymorphic queries, and joins. We also look at how you can run efficient report queries, use collection filters, and use plain SQL rather than HQL. Finally, this chapter looks at solving common performance problems, discussing the `n+1` selects problem and caching.

Chapter 8 offers a look at patterns and practices around NHibernate. We give example code for common practices such as layered applications and unit testing. Also included are some helpful tips for finding bugs in your applications. We also give an example implementation of adding additional services to NHibernate applications, such as audit logging.

Chapter 9 starts by discussing development processes and available tools, explaining the various starting points for an NHibernate application. We also look at code generation and automatic schema maintenance, for evolving domain models and databases in tandem. We then look at working with legacy databases and explain some tried and tested tricks for dealing with things like composite keys and triggers.

Chapter 10 gives more real-world knowledge. We look at refactoring a sample application into layers, with a well-defined persistence layer and a smart domain model. This chapter also introduces the DAO pattern with generics, and a useful NHibernate `Helper` class. Finally, we look at session management for web applications, implementing long-running business conversations, and demonstrating how to implement distributed transactions.

Code conventions and downloads

All source code in listings or in text is in a fixed-width font like this to separate it from ordinary text. Code annotations accompany many of the listings, highlighting important concepts. In some cases, numbered bullets ❶ link to explanations that follow the listing.

The complete example code for the book can be downloaded from the Manning web site at www.manning.com/kuate or www.manning.com/NHibernateinAction.

Author Online

Purchase of *NHibernate in Action* includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the lead author and from other users. To access the forum and subscribe to it, point your web browser to www.manning.com/NHibernateinAction or www.manning.com/kuate. This page provides information on how to get on the forum once you're registered, what kind of help is available, and the rules of conduct on the forum.

Manning's commitment to our readers is to provide a venue where a meaningful dialog between individual readers and between readers and the authors can take place. It's not a commitment to any specific amount of participation on the part of the authors, whose contribution to the AO remains voluntary (and unpaid). We suggest you try asking the authors some challenging questions lest their interest stray!

The Author Online forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

About the authors

PIERRE HENRI KUATÉ is one of the main developers on the NHibernate project team, author of the NHibernate.Mapping.Attributes library, and a major contributor to the NHibernate forum. He was responsible for managing the NHibernate documentation, website, and forum on the Hibernate.org site. He started using NHibernate more than four years ago in commercial development.

TOBIN HARRIS has worked with NHibernate since it was in early beta. He's passionate about tools and practices that help build quality software at high speeds. As an independent consultant and entrepreneur, Tobin works with companies across the globe in various sectors including banking, personal finance, healthcare, software components, and new media. Tobin obtained his degree in software engineering at Leeds Metropolitan University and continues to work and live in Leeds, UK.

CHRISTIAN BAUER is a member of the Hibernate developer team and a trainer, consultant, and product manager for Hibernate, EJB 3.0, and JBoss Seam at JBoss. He is the lead author of Manning's *Hibernate in Action* and *Java Persistence with Hibernate*.

GAVIN KING is a lead developer at JBoss, the creator of Hibernate, and a member of the EJB 3.0 (JSR 220) expert group. He also leads Web Beans JSR 299, a standardization effort involving Hibernate concepts, JSF, and EJB 3.0, and is coauthor with Christian of the two books mentioned above.

About the title

By combining introductions, overviews, and how-to examples, the *In Action* books are designed to help learning and remembering. According to research in cognitive science, the things people remember are things they discover during self-motivated exploration.

Although no one at Manning is a cognitive scientist, we are convinced that for learning to become permanent it must pass through stages of exploration, play, and, interestingly, retelling of what is being learned. People understand and remember new things, which is to say they master them, only after actively exploring them. Humans learn in action. An essential part of an *In Action* guide is that it's example-driven. It encourages the reader to try things out, to play with new code, and explore new ideas.

There is another, more mundane, reason for the title of this book: our readers are busy. They use books to do a job or solve a problem. They need books that allow them

to jump in and jump out easily and learn just what they want just when they want it. They need books that aid them *in action*. The books in this series are designed for such readers.

About the cover illustration

The figure on the cover of *NHibernate in Action* is taken from the 1805 edition of Sylvain Maréchal's four-volume compendium of regional dress customs. This book was first published in Paris in 1788, one year before the French Revolution. Each illustration is finely drawn and colored by hand.

The colorful variety of Maréchal's collection reminds us vividly of how culturally apart the world's towns and regions were just 200 years ago. Isolated from each other, people spoke different dialects and languages. In the streets or the countryside, they were easy to place—sometimes with an error of no more than a dozen miles—just by their dress.

Dress codes have changed everywhere with time and the diversity by region, so rich at the time, has faded away. It is now hard to tell apart the inhabitants of different continents, let alone different towns or regions. Perhaps we have traded cultural diversity for a more varied personal life—certainly for a more varied and fast-paced technological life.

At a time when it is hard to tell one computer book from another, Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by Maréchal's pictures.

Part 1

Discovering ORM with NHibernate

The first part of the book provides insights into what ORM is, why it exists, and how it fits in a typical .NET application. We then introduce NHibernate, using a clear and simple example to help you understand how the various pieces of an NHibernate application fit together.

Object/relational persistence in .NET

This chapter covers

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

Software development is an ever-changing discipline in which 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, more simply, how to load and save data.

Almost endless options are available. You can store data in binary or text files on a disk. You can choose a format such as CSV, XML, JSON, YAML, or SOAP, or invent your own format. Alternatively, you can send data over the network to another

application or service, such as a relational database, an Active Directory server, or a message queue. You may 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 a thorny topic. Relational databases are extremely popular, but many choices, questions, and options still confront you in your daily work. For example, should you use DataSets, or are DataReaders more suitable? Should you use stored procedures? Should you hand-code your SQL or let your tools dynamically generate it? Should you strongly type your DataSets? Should you build a hand-coded domain model containing classes? If so, how do you load data to and save it from the database? Do you use code generation? The list of questions continues.

This topic isn't restricted to .NET. The entire development community has been debating this topic, often fiercely, for many years.

But one approach has gained widespread popularity: *object/relational mapping* (ORM). Over the years, many libraries and tools have emerged to help developers implement ORM in their applications. One of these 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 you 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.

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 explain why tools like NHibernate are needed.

Do I need to read all this background information?

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

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'll 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 issues 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 discuss many of its advantages. Following that, we dig into some complex persistence challenges that make a tool like NHibernate essential. Finally, we define ORM and discuss 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.

1.1 What is persistence?

Persistence is a fundamental concern in application development. If you have some experience in software development, you've already dealt with it. Almost all applications require persistent data. You use persistence to allow data to be stored even when the programs that use it aren't running.

To illustrate, let's say you want to create an application that lets users 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'll soon realize that your application needs to somehow save the contacts somewhere. You're faced with a persistence decision: you 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 may choose a *relational database*, because such databases are widely understood and offer great features for reliably storing and retrieving data.

1.1.1 Relational databases

You've probably already worked with a relational database such as Microsoft SQL Server, MySQL or Oracle. If you haven't, see appendix A. Most developers 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. You can have several applications accessing a single database, some written in .NET, some written in Java or Ruby, and so on. Relational technology provides a way of sharing data between many different applications. Even different components within a single application can independently access a relational database (a reporting engine and a logging component, for example).

Relational technology is a common denominator of many unrelated systems and technology platforms. The relational data model is often the common enterprise-wide representation of *business objects*: a business needs to store information about various things such as customers, accounts, and products (the business objects), and the relational database is usually the chosen central place where they're defined and stored. This makes the relational database an important piece in the IT landscape.

RDBMSs have SQL-based application programming interfaces (APIs). So today's relational database products are called 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 you're using NHibernate. You'll use SQL to tune the performance of your NHibernate application. NHibernate automates many repetitive coding tasks, but your knowledge of persistence technology must extend beyond NHibernate 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 has also written 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 many tools and choices when it comes to making applications work with SQL databases. You might lean on the Visual Studio IDE, taking advantage of its drag-and-drop capabilities: in a series of mouse clicks, you can create database connections, execute queries, and display editable data onscreen. We think this approach is great for simple applications, but the approach doesn't scale well for larger, more complex applications.

Alternatively, you can use SqlCommand objects and manually write and execute SQL to build DataSets. Doing so can quickly become tedious; you want to work at a slightly higher level of abstraction so you can focus on solving business problems rather than worrying about data access concerns. If you're interested in learning more about the wide range of tried and tested approaches to data access, then consider reading Martin Fowler's *Patterns of Enterprise Application Architecture* [Fowler 2003], which explains many techniques in depth.

Of all the options, the approach we take is to write classes—or *business entities*—that can be loaded to and saved from the database. Unlike DataSets, these classes aren't designed to mirror the structure of a relational database (such as rows and columns). Instead, they're concerned with solving the business problem at hand. Together, such 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 persistent; a *transient* object is one that has a limited lifetime that is bounded by the life of the process that instantiated the object. A simple example is a web control object, which exists in memory for only a fraction of a second before it's 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 issues 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 (using DataSets); the application has its own, object-oriented model of the business entities. If a database has ITEM and BID tables, the .NET application defines `Item` and `Bid` classes rather than uses DataTables for them.

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 a SQL stored procedure); it's implemented in .NET. This allows business logic to use sophisticated object-oriented concepts such as inheritance and polymorphism. For example, you 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 may be much better off without a domain model. SQL and ADO.NET are 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.

But 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, because 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 a software system. To explain this, we take a step back and look at 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 well with that design. What is a layered architecture?

A layered architecture splits a system into several groups, where each group contains code addressing a particular problem area. These groups are called *layers*. For example, a user interface layer (also called the presentation layer) might contain all

the application code for building web pages and processing user input. One major benefit of the layering approach is that you can often make changes to one layer without significant disruption to the other layers, thus making systems less fragile and more maintainable.

The practice of layering follows 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 the layer just below it.

Business applications use a popular, proven, high-level application architecture that comprises three layers: the presentation layer, the business layer, and the persistence layer. See figure 1.1.

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

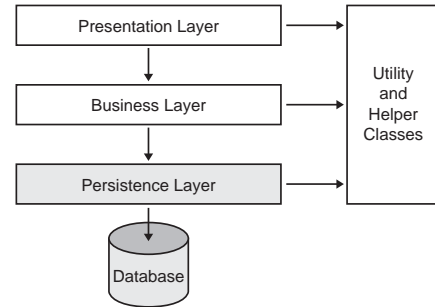


Figure 1.1 Layered architecture highlighting the persistence layer

- *Presentation layer*—The user interface logic is topmost. In a web application, this layer 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. But it's generally agreed that the business layer is responsible for implementing any business rules or system requirements that users would understand as part of the problem domain. In some systems, this layer has its own internal representation of the business domain entities. In others, it reuses the model defined by 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 application data to and retrieving it 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 layers, because they don't obey the rules for interlayer dependency in a layered architecture.

Should all applications have three layers?

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 may be better off without complex objects. SQL and the ADO.NET API are 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.

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. Instead of neatly separating business rules and database-access functions into separate layers, you can put them all in your web/Windows code-behind files. Tools like Visual Studio .NET make it easy and painless to build this kind of application. But 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 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 painful!

1.2 Approaches to persistence in .NET

We've discussed how, in any sizeable application, you need a persistence layer to handle loading and saving data. Many approaches are available when you're building this persistence layer, and each has advantages and disadvantages. Some popular choices are as follow:

- Hand coding
- DataSets
- LINQ-to-SQL
- NHibernate (or similar)
- ADO.NET Entity Framework

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

1.2.1 Choice of persistence layer

In your applications, you'll often want to load, manipulate, and save database items. Regardless of which persistence approach you 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 you have to manipulate data, so you can use a persistence layer to take care of these low-level steps.

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 lets you execute SQL commands that perform the persistence, but the complexity of this process requires that you wrap these commands behind components that understand how your entities should be persisted. These components can also hide the specifics of the database, making your application less coupled to the database and 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, whereas MySQL uses back-ticks. It's possible to hide this detail and let the persistence layer select the right delimiter.

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

HAND-CODED PERSISTENCE LAYER

Hand-coding a persistence layer can involve a lot of work; it's common to first build a generic set of functions to handle database connections, execution of SQL commands, and so on. Then, on top of this sublayer, you have to build another set of functions that save, load, and find your business entities. Things get much more involved if you need to introduce caching, business-rule enforcement, or handling of entity relationships.

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. But it can be a huge undertaking and is often tedious and repetitive work, even when you use code generation.

DATASET-BASED PERSISTENCE LAYER

Visual Studio lets you effortlessly generate your own persistence layer, which you can then extend 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. You have to resort to hand-coding when you need more control, which is usually inevitable (as described in section 1.3).

NHIBERNATE PERSISTENCE LAYER

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

In the context of an auction application (such as eBay), NHibernate lets you easily save an `Item` and its `Bids` by implementing a method like this:

```
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 want you to see how simple the persistence layer is with

NHibernate. You'll start using NHibernate in chapter 2, where you'll discover that it's straightforward to execute persistence operations. All you need to do is write your entities and explain to NHibernate how to persist them. Before moving on to a deeper discussion of NHibernate, let's take a quick look at the newest generation of persistence technologies introduced by Microsoft.

LINQ TO SQL-BASED PERSISTENCE LAYER

Language INtegrated Query (LINQ) was introduced in 2007 by Microsoft. It allows for query and set operations, similar to what SQL statements offer for databases directly within .NET languages like C# and Visual Basic through a set of extensions to these languages. LINQ's ambition is to make queries a natural part of the programming language. LINQ to SQL provides language-integrated data access by using LINQ's extension mechanism. It builds on ADO.NET to map tables and rows to classes and objects.

LINQ to SQL uses mapping information encoded in .NET custom attributes or contained in an XML document. This information is used to automatically handle the persistence of objects in relational databases. A table can be mapped to a class, the table's columns can be mapped to properties of the class, and relationships between tables can be represented by properties. LINQ to SQL automatically keeps track of changes to objects and updates the database accordingly through dynamic SQL queries or stored procedures. Consequently, when you use LINQ to SQL, you don't have to provide the SQL queries yourself most of the time.

LINQ to SQL has some significant limitations when compared to NHibernate. For example, its mapping of classes to tables is strictly one-to-one, and it can't map base class properties to table columns. Although you can create a custom provider in LINQ, LINQ to SQL is a SQL Server-specific solution.

ADO.NET ENTITY FRAMEWORK

The Microsoft ADO.NET Entity Framework is a new approach to persistence, available since .NET 3.5 SP1. At a high level, it proposes to provide a persistence solution similar to NHibernate, but with the full commercial support and backing of Microsoft. This promises to be an attractive option for developers who require a vendor-supported solution. But at the time of this writing, the Entity Framework is early beta software, and its feature set is incomplete.

The ADO.NET Entity Framework 1.0 version supports multiple databases and more complex mapping. But it won't support true "object-first" development, where you design and build, and then generate the database tables from that mapping, until version 2—planned for late 2009 at the earliest. For situations requiring a robust ORM, NHibernate still offers significant advantages.

1.2.2 Implementing the entities

Once you've chosen a persistence-layer approach, you can focus on building the business objects, or entities, that the application will manipulate. These are classes representing the real-world elements that the application must manipulate. For an auction application, `User`, `Item`, and `Bid` are common examples. We now discuss how to implement business entities using each of the three approaches.

HAND-CODED ENTITIES

Returning to the example of an auction application, consider the entities: `User`, `Item`, and `Bid`. In addition to the data they contain, you expect relationships to exist 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 collection like `item.Bids` and a property like `bid.Item`. The object-oriented view is different than the relational view: instead of having primary and foreign keys, you have associations. Object-oriented design also provides other powerful modeling concepts, such as inheritance and polymorphism.

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

But they're difficult and tedious to code. Think about the manual work required to support the persistence of entities inheriting from other entities. It's 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. But without a helpful framework, these features can be time consuming to implement.

ENTITIES IN A DATASET

A `DataSet` represents a collection of database tables, and in turn these tables contain the data of the entities. A `DataSet` stores data about business objects in a fashion similar to a database. You can use a generated-typed `DataSet` to ease the manipulation of data, and it's also possible to insert business logic and rules.

As long as you want to manipulate data, .NET and IDEs provide most features required to work with a `DataSet`. But as soon as you think about business objects as *objects* in the sense of object-oriented design, you 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 isn't possible with `DataSets`. This level of freedom in the design of entities can be achieved only by hand coding them.

ENTITIES AND NHIBERNATE

NHibernate is *non-intrusive*. It's common to use it with hand-coded (or generated) entities. You must provide mapping information indicating how these entities should be loaded and saved. Once this is done, NHibernate can take care of moving your object-oriented entities to and from a relational database.

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

ENTITIES AND LINQ TO SQL

The LINQ to SQL approach looks a lot like the NHibernate way of doing ORM. LINQ to SQL uses POCO objects to represent your application data (the entities). The mapping of those objects to database tables is described either in declarative attributes in code or in an XML document. After the mapping and the classes are complete, the LINQ to SQL framework takes care of generating SQL for database operations.

Once the entities are implemented, you 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 consequences for the way the user interface (UI) is written. For the end user, the UI is one of the most important elements. Whether it's a web application (using ASP.NET) or a Windows application, it must satisfy the needs of the user. A deep discussion of implementing a UI isn't 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 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.

It's worth noting that we expect .NET entity data binding to change soon. Microsoft is beginning to actively push the use of entities in .NET applications as the company promotes the ADO.NET Entity Framework and LINQ to SQL. For this reason, we won't discuss those technologies in this section.

DATASET-BASED PRESENTATION LAYER

Microsoft has added support for data binding with DataSet in most .NET controls. It's easy to bind a DataSet to a control so that its information is displayed and any changes (made by the user) reverberate in the DataSet.

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

PRESENTATION LAYER AND ENTITIES

Data-binding the hand-coded entities typically used in NHibernate applications can be tricky. This is because entities can be implemented in so many different ways. A DataSet is always made of tables, columns, and rows; but a hand-coded entity—a class of your own design—contains fields and methods with no standardized way to access and display them. .NET allows us to data-bind controls to the public properties of any object. This is good enough in simple cases. If you want more flexibility, you must do some hand coding to get entity data into the UI and back again.

Hand coding your own entity/UI bindings is still fairly simple. However, if you find this tedious, then take a look at some of the open source projects designed to tackle this problem for you. “ObjectViews” is one of many projects out there.

Also, don't forget that you're free to fall back to DataSets when you're dealing with edge cases like complex reporting, where DataSets are much easier to manipulate. In fact, at the time of writing, few reporting tools provide good support for entities, so DataSets may be your best option. We discuss this issue in chapter 9.

Using persistence-able information affects the way the UI is designed. Data should be loaded when the UI opens and saved when the UI closes. NHibernate proposes some patterns to deal with this process, as you'll learn in chapter 8.

Now all the layers are in place, and you can work on performing actions.

1.2.4 Implementing CRUD operations

When you're working with persistent information, you're concerned with persisting and retrieving this information. Create, read, update, delete (CRUD) operations 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 a button to view an item. The persistence layer is used to load this item, which is then bound to a form displaying its data.

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

HAND-CODED CRUD OPERATIONS

A hand-coded CRUD operation does exactly what you want because you write the SQL command to execute—but it's repetitive and annoying work. It's 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. But much more work is required for complex queries and manipulating interconnected entities.

CRUD OPERATIONS WITH DATASETS

You know that much 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're also extensible. You can either add methods calling stored procedures or generate SQL commands with few clicks. But if you want to implement anything more complex, you must hand code it; you'll see in the next section that some useful features aren't easy to implement, and the structure of a DataSet may make doing so even harder.

CRUD OPERATIONS USING NHIBERNATE

As soon as you give NHibernate your entities' mapping information, you can execute a CRUD operation with a single method call. This is a fundamental feature of an 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 uncover many optimizations and best practices. For example, when you're

manipulating entities, you can achieve the best performance by delaying persistence to the end of the *transaction*. At this point, you use a single connection to save all entities.

LINQ TO SQL CRUD OPERATIONS

On the surface, executing CRUD operations with LINQ to SQL is similar to using NHibernate—you can load, save, update, and delete objects with simple method calls. LINQ to SQL offers less fine tuning of your CRUD operations for each entity, which can be a good thing or a bad thing depending on the complexity of your project.

Now that we’ve covered all the basic persistence steps and operations, we explore some advanced features that illustrate the advantages of NHibernate.

1.3 Why do we need NHibernate?

So far, we’ve talked about a simple application. In the real world, you 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 walk through some features indispensable to implementing a successful application. First, we give some examples illustrating the fundamental differences between objects and relational database. You’ll also learn 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’ll learn the patterns and features it provides to achieve the best performance. Finally, we cover complex queries; you’ll see that you can use NHibernate to write a powerful search engine.

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

1.3.1 The paradigm mismatch

A database is relational, but we’re using object-oriented languages. There is no direct way to persist an object as a database row. Persistence shouldn’t hinder your ability to design entities that correctly represent what you’re manipulating.

The *paradigm mismatch* (or *object/relational impedance mismatch*) refers to the fundamental incompatibilities that exist between the design of objects and relational databases. Let’s 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 talk 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 auction use case we mentioned in section 1.1.4. Let’s say you want to add an address to a `User` object, not as a string but as another object. How can you persist this user in a table? You can add an `ADDRESS` table, but it’s generally not a good idea (for performance reasons). You can create a *user-defined column type*, but this option isn’t broadly supported and portable. Another option is to merge the address information into the user, but this isn’t a good object-oriented design and it isn’t reusable.

It turns out that the granularity problem isn't difficult to solve. We wouldn't even discuss it if it weren't for the fact that it's visible in so many approaches, including the DataSet. We describe the solution to this problem in section 3.6.

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

THE PROBLEM OF INHERITANCE AND POLYMORPHISM

Object-oriented languages support the notion of *inheritance*, but relational databases typically don't. Let's say that the auction application can have many kinds of items. You could create subclasses like Furniture and Book, each with specific information. How can you persist this hierarchy of entities in a relational database? A Bid can refer to any subclass of Item. It should be possible to run *polymorphic queries*, such as retrieving all bids on books. In section 3.8, we discuss how ORM solutions like 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 section 3.5, .NET *object identity* isn't naturally equivalent to the primary key value. With relational databases, it's recommended that you use a *surrogate key*—a primary key column with no meaning to the user. But .NET objects have an intrinsic identity, which is based either on their memory location or on a user-defined convention (by using the implementation of the Equals() method).

Given this problem, how can you represent associations? Let's look at that next.

PROBLEMS RELATING TO ASSOCIATIONS

In an object model, *associations* represent the relationships between objects. For instance, a bid has a relationship with an item. This association is created using object references. In the relational world, an association is represented by 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. Navigation has no meaning for a relational data model, because you can create arbitrary data associations with table joins and projection. We discuss association mappings in detail in chapters 3 and 6.

If you think about the DataSet in all these problems, you'll realize how rigid its structure is. The information in a DataSet is presented exactly as in the database. To navigate from one row to another, you must manually resolve their relationship by using a *foreign key* to find the referred row in the related table. Let's move from the representation of the entities to how you can manipulate them efficiently.

1.3.2 Units of work and conversations

When users work on applications, they perform distinct unitary operations. These operations can be referred to 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 performed by the user are treated as if they were a single bigger business transaction (a *unit*). You'll learn in this section that NHibernate makes this easier to achieve. Let's take another example to illustrate this concept.

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

The first solution that may come to mind is to persist the rating when the user enters it. This technique is inefficient: the user may change the rating many times, and the persistence will be done separately for each song. (But this approach is safest if you expect the application to crash at any moment.)

Instead, you can let the user rate all the songs and then persist the ratings when the user closes the list. The process of rating these songs is a conversation.

Let's see how it works and what its benefits are.

THE UNIT OF WORK PATTERN

When you're working with a relational database, you may 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 the user (and the user's collection of items); then you can run a query retrieving the item. If you forget to save the user, you'll start getting hard-to-detect bugs.

The Identity Map pattern

NHibernate uses the *Identity Map* pattern to make sure an item's user is the same object as the user you had before loading the item (as long as you're working in the same transaction). You'll learn more about the concept of identity in section 3.5.

Now imagine that you're involved in a complex conversation involving many updates and deletes. If you have to manually track which entities to save or delete, while making sure you load each entity only once, things can quickly become very difficult.

NHibernate follows the Unit of Work pattern to solve this problem and ease the implementation of conversations. (We cover conversations in chapter 5 and implement them in chapter 10.)

You can create entities and associate them with NHibernate; then, NHibernate keeps track of all loading and saving of changes only when required. At the end of the transaction, NHibernate 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 the application's performance. Here are two simple examples.

When working on an item in the auction application, users can add, modify, or delete their bids. It would be painful to manually track these changes one by one.

Instead, you can use NHibernate's *transparent persistence* feature: you ask NHibernate to save all changes in the collection of bids when the item is persisted. It automatically figures out which CRUD operations must be executed.

Now, if you want to modify a `User`, you load, change, and persist it. But what about the collection of items 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 your ability to manipulate the user.

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

Using these features has many implications; we progressively cover them in this book.

CACHING

Tracking entities implies keeping their references somewhere. NHibernate uses a *cache*. This cache is indispensable for implementing the Unit of Work pattern, and it can also make applications more efficient. We cover caching in depth in section 5.3.

NHibernate's identity map uses a cache to avoid loading an entity many times. This cache can be shared by transactions and applications.

Suppose you build a website for the auction application. Visitors may be interested in some items. Without a cache, these items will be loaded from the database each time a visitor wants to see them. With a few lines of code, you can ask NHibernate to cache these items, and then enjoy the performance gain.

1.3.3 *Complex queries and the ADO.NET Entity Framework*

This is the last (but not least) feature related to persistence. In section 1.2.5, we talked about CRUD operations. You've learned about features related to CRUD (all having to do with the Unit of Work pattern). Now we talk about retrieve operations: searching for and loading information.

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

IMPLEMENTING A QUERY ENGINE

If you're familiar with SQL, you know that you can write complex queries using the `SELECT ... FROM ... WHERE ...` construct. But if you work with business objects, you have to transform the results of your SQL queries into entities. We already advertised the benefits of working with entities, so it makes 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 you ask for an entity by its identifier, NHibernate knows how to find it. You 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:

- *Hibernate Query Language* (HQL) is similar to SQL in many ways, but it also has useful object-oriented features. You can query NHibernate using plain old SQL; but as you'll learn, using HQL offers several advantages.
- *Query by Criteria API* (QBC) 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 taste of the power of these APIs, let's build three simple queries. First, here is some HQL. It finds all bids for items where the seller's name starts with the letter *K*:

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

As you can see, this code is easy to understand. If you want to write SQL to do the same thing, you need something more verbose, along these lines:

```
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 use an example derived from one later in the book, in section 8.5.1. This shows a method that lets you 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>();
}
```

This method lets you pass objects that represent the kind of users you 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 an example entity (here, example *User* and example *Item*) is both powerful and elegant, as demonstrated here:

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

You use the `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 so little code! If you're new to this approach, you may find it unbelievable. Don't even try to implement this query using hand-coded SQL.

You'll learn more about queries in chapters 5 and 7. If you aren't fully satisfied by these APIs, you may also want to watch for 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 innovations. You may think this technology will soon replace NHibernate, but this is 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. When querying a relational database, you can do something like this:

```
IEnumerable users = from u in Users
                    where u.Lastname.StartsWith( "K" )
                    order by user.Lastname 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 you a declarative way of working with data, so you can express what you want in simple terms rather than typing lots of for-each loops. You can also benefit from helpful IDE 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 can integrate with this technology and benefit from it. At the time of writing, good progress is being made toward a LINQ to NHibernate project. And Manning Publications has published *LINQ in Action*, a fantastic book by our good friend Fabrice Marguerie.

As mentioned earlier, Microsoft is also working on a framework currently called the ADO.NET Entity Framework, which aims to provide developers with an ORM framework not completely unlike NHibernate. This is a good step forward because Microsoft will promote the `DataSet` less often and begin promoting the benefits of ORM tools. Another project called LINQ over `DataSet` greatly improves `DataSet`'s query capabilities, but it doesn't yet solve many other issues discussed in this chapter.

All these technologies will take time to mature. Many questions remain unanswered, such as, how extensible will this framework be? Will it support most popular RDBMSs or just SQL Server? Will it be easy to work with legacy database schemas? No framework can provide all features, so it must be extendable to let you integrate your own features. (If your particular projects require you to work with legacy databases, you can read section 10.2 to learn about the features NHibernate gives you to work with more exotic data structures.)

Now, let's dig into the theory behind NHibernate.

1.4 Object/relational mapping

You already have an idea of how NHibernate provides object/relational persistence. But you may still be unable to tell what ORM is. We try to answer this question now. After that, we discuss some nontechnical reasons to use ORM.

1.4.1 What is ORM?

Time has proven 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's 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, and concurrency control). All we 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. But if you talk to .NET developers about ORM, it's usually in the context of object/relational mapping.

You learned 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.

1.4.2 Why ORM?

The overall solution for 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 tedious SQL/ADO.NET and manual bridging of the object/relational paradigm mismatch. Despite all this effort, the end result doesn't feel right. We've 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. But an object-oriented purist will model these entities differently than an experienced relational data modeler. You

learned some details of this problem in section 1.3.1. The usual solution 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. No elegant transformation is 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 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 business problems. 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.

Using fewer lines of code makes the system more understandable because 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 often 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 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 interesting question is, what happens when we consider time and budget constraints?

The best way to address this question is to define a means to measure performance and thresholds of acceptability. Then you can find out whether the performance cost of an ORM is unacceptable. Experience has proven 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 lets you take advantage of many performance optimizations with 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 level of portability on your application.

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

1.5 Summary

In this chapter, we've 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 later. We've walked through the layered architecture of a .NET application and the implementation of persistence, exploring four possible approaches.

You now understand the productivity of DataSet, but you also realize how limited and rigid it is. You've learned about many useful features that would be painful to hand code. In addition, you know how NHibernate solves the object/relational mismatch.

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

We glanced at NHibernate's powerful query APIs. After you've started using them, you may never want to go back to SQL.

Finally, you learned what ORM is. 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 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. But we can't wait, and there is no sign that any of these issues will improve soon (a multi-billion-dollar industry isn't agile). ORM is the best solution currently available, and it's a timesaver for developers facing the object/relational mismatch every day.

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

Hello NHibernate!

This chapter covers

- NHibernate in action with a “Hello World” application
- How to architecture an NHibernate application
- Writing and mapping a simple entity
- Configuring NHibernate
- Implementing 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 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. But printing a message to a console window won’t be enough to really demonstrate NHibernate. Instead, your program will store newly created objects in the database, update them, and perform queries to retrieve them from the database.

This chapter forms 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 standalone 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 you can do with instances of the persistent class using NHibernate.

2.1.1 Installing NHibernate

Before you can start coding the “Hello World” application, you must first install NHibernate. You then need to create a new Visual Studio solution to contain the sample application.

NHibernate 1.2.1GA can be downloaded via <http://www.nhforge.org>. Click the “download” tab and locate “NHibernate Core.” From there you can find and download the NHibernate 1.2.1.GA.msi file.

Although the book is written for NHibernate 1.2.1GA, we’re aware that many people are using NHibernate 2.0 Beta. We’ve therefore ensured our first tutorial applies to both of these versions of NHibernate.

Once you’ve downloaded and installed NHibernate, you’re ready to create a new solution and start using it.

2.1.2 Create a new Visual Studio project

For the 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 C# Console Application. Name your project HelloNHibernate. Note that you can also use NHibernate with VB.NET projects, but in this book, we’ve chosen to use C# examples.

The application will need to use the NHibernate library, so the next step is to reference it in your new project. To do this, follow these steps:

- 1 Right-click the project and select Add Reference.
- 2 Click the Browse tab and navigate to the folder where NHibernate is installed. By default, NHibernate resides in the C:\Program Files\NHibernate\bin\net2.0\ folder.
- 3 From the list of assemblies, select NHibernate.dll. Click OK to add this reference to your solution.

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

- 4 Reference the NHibernate library at the top of the Program.cs file with the using NHibernate, using System.Reflection and NHibernate.Cfg statements, as follows:

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

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

Now that your solution is set up, you're ready to start writing your first NHibernate application.

2.1.3 Creating the *Employee* class

The objective of the sample application is to store an `Employee` record in a database and to later retrieve it for display. The application needs a simple persistent class, `Employee`, which represents 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 enter the code from listing 2.1 for the `Employee` entity.

Listing 2.1 `Employee.cs`: A simple persistent class

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

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

The `Employee` class has three fields: the identifier, 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 the identifier field, but this isn't a requirement. NHibernate allows virtually anything for the identifier type, as you'll see later.

Note that you use *public fields* here rather than properties. This is purely to make the sample code shorter; it isn't always considered good practice.

Instances of the `Employee` class may be managed (made persistent) by NHibernate, but they don't have to be. Because the `Employee` object doesn't implement any NHibernate-specific classes or interfaces, you 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 you’ve come to expect from “Hello World” applications: it prints “Hello World, said Fred Bloggs” to the console. It may look like we’re trying to be cute; in fact, we’re demonstrating an important feature that distinguishes NHibernate from some other persistence solutions. The persistent class can be used with or without NHibernate—no special requirements are needed. Of course, you came here to see NHibernate, 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

You 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 you’ll use Microsoft SQL Server 2000 or 2005.

Your first step is 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 your `Employee` data. To do so, delete the previous SQL and replace it with the following

```
USE HelloNHibernate
GO
CREATE TABLE Employee (
    id int identity primary key,
    name varchar(50),
    manager int )
GO
```

Run this code: you’ve created a place to store your `Employee` entities. You’re now ready to see NHibernate in action!

Note that, in chapter 9, we 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!

2.1.5 Creating an Employee and saving to the database

The code required to create an `Employee` and save it to the database is shown in listing 2.2. It comprises two functions: `CreateEmployeeAndSaveToDatabase` and `OpenSession`. You can type these functions into your `Program.cs` file below the `static void Main()` function in the `Program` class.

Listing 2.2 Creating and saving an Employee

```

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()
{
    if(factory == null)
    {
        Configuration c = new Configuration();
        c.AddAssembly(Assembly.GetCallingAssembly());
        factory = c.BuildSessionFactory();
    }
    return factory.OpenSession();
}

static ISessionFactory factory;

```

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

```

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

```

Hold on—the `Id` column isn't being initialized here. You didn't set the `id` field of `message` anywhere, so how can you expect it to get a value? 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.

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

You now have the `Employee` class defined, and have added an `Employee` table to your database. We've also added some code to create an `Employee` instance and save it

to the database using NHibernate. To complete the program, you will next add some code that will load the `Employee` from the database, and print our “Hello World” message. Let’s add that code now.

2.1.6 Loading an Employee from the database

Let’s start by adding code that can retrieve all `Employees` from the database in alphabetical order. Type the code in listing 2.3 below the previous `OpenSession()` function.

Listing 2.3 Retrieving Employees

```
static void LoadEmployeesFromDatabase()
{
    using (ISession session = OpenSession())
    {
        IQuery query = session.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 an 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 (at startup, where possible).

So far, you’ve defined the `Employee` entity, set up the database, and written code to create a new `Employee` and later retrieve it from the `Employee` table. NHibernate has barely entered the picture yet. Next, you’ll write some XML to tell NHibernate about the `Employee` entity and how you want `Employees` saved in the database.

2.1.7 Creating a mapping file

In order for NHibernate to do its magic in any of the code so far, 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 2.4.

Listing 2.4 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>
```

To add this mapping document to your solution, do the following:

- 1 Right-click your HelloNHibernate project in the Solution Explorer, and select Add > New Item.
- 2 Select the XML document type, and name it Employee.hbm.xml.
- 3 Click OK.
- 4 Now highlight the XML file in Solution Explorer and look for the Build Action property in the Properties pane.
- 5 Change it from “Content” to “Embedded Resource.” This is an important step that you shouldn’t miss, because it allows NHibernate to easily find the mapping information.
- 6 Now copy the XML in listing 2.4 into your Employee.hbm.xml file.

The mapping document you’ve just created 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 `manager` property 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 to generate 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 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.

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

2.1.8 Configuring your application

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

- 1 Right-click your `HelloNHibernate` project in the Solution Explorer, and select `Add > New Item`.
- 2 Select `Application Configuration File` from the options. Click `OK` to add an `app.config` file to the project.
- 3 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>
    <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;
      Database=HelloNHibernate;Integrated Security=SSPI;" />
  </nhibernate>
</configuration>
```

That’s quite a lot of XML! 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.

Also note that this configuration is for NHibernate 1.2.1GA. If you’re using NHibernate 2.0 or later, then copy the following XML instead:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="hibernate-configuration"
      type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
  </configSections>

  <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory>
```

```

<property name="connection.provider">
    NHibernate.Connection.DriverConnectionProvider
</property>
<property name="connection.driver_class">
    NHibernate.Driver.SqlClientDriver
</property>
<property name="connection.connection_string">
    Server=(local);database=HelloNHibernate;Integrated Security=SSPI;
</property>
<property name="dialect">
    NHibernate.Dialect.MsSql2000Dialect
</property>
<property name="show_sql">
    false
</property>
</session-factory>
</hibernate-configuration>
</configuration>

```

2.1.9 Updating an Employee

You've added code for saving and loading Employees. Before we run our application, let's finish off by adding one more function to demonstrate how NHibernate can update existing entities. You'll write some code to update the first Employee and, while you're at it, create a new Employee to be the manager of the first, as shown in listing 2.5. Again, type this code below the other functions in Program.cs.

Listing 2.5 Updating an Employee

```

static void UpdateTobinAndAssignPierreHenriAsManager()
{
    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 pierreHenri = new Employee();
            pierreHenri.name = "Pierre Henri Kuate";

            tobin.manager = pierreHenri;
            transaction.Commit();

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

```

Behind the scenes, NHibernate runs 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)
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 detects the modification to the name and manager properties of the first `Employee` (Tobin) and automatically updates the database. You're taking advantage of an NHibernate feature called *automatic dirty checking*: this feature saves you 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` (Pierre Henri) was saved when it was associated with the first `Employee`. This feature is called *cascading save*: it saves you the effort of explicitly making the new object persistent by calling `Save()`, as long as it's reachable by an already persistent object (Tobin). Also, notice that the ordering of the SQL statements isn't the same as the order in which you 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*.

2.1.10 Running the program

Before finally running the example, you 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();
    UpdateTobinAndAssignPierreHenriAsManager();
    LoadEmployeesFromDatabase();

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

```

If you run “Hello World,” it prints

```

Saved Tobin to the database
Updated Tobin and added Pierre Henri

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 take the “Hello World” application. Now that you have some code under your belt, we take a step back and present an overview of NHibernate’s main APIs.

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. But in practice, ORM APIs aren't especially small. Don't worry; 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 because the business layer acts as a client of the persistence layer in a traditionally layered application. Note that some simple applications may not cleanly separate business logic from persistence logic; that's OK—it simplifies the diagram.

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 (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, letting NHibernate support almost any database with an ADO.NET driver.

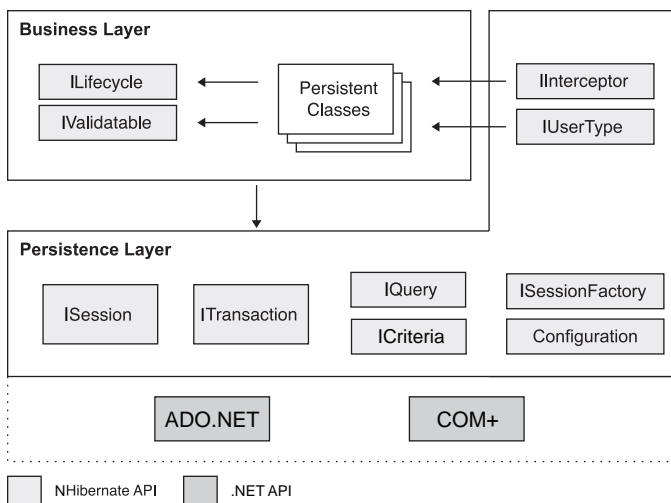


Figure 2.1 High-level overview of the NHibernate API in a layered architecture

In this section, we don't cover the detailed semantics of NHibernate API methods, just the role of each of the primary interfaces. We progressively discuss 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 described in this section are 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 an 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 section 4.2.

ISESSIONFACTORY INTERFACE

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

The `ISessionFactory` certainly isn't 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. But 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 an NHibernate application, it's the first object you'll meet when you begin using NHibernate. Section 2.2 covers the issue of configuring NHibernate in some detail.

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

The ICriteria interface is similar; it lets you create and execute object-oriented criteria queries.

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 dive into how NHibernate is configured and 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 let a persistent object react to events relating to its own *persistence lifecycle*. The persistence lifecycle is encompassed by an object's CRUD operations (when it's created, retrieved, updated, or deleted).

NOTE The original 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 nonportable code. Because these interfaces are deprecated, we don't discuss them in this book.

The `IInterceptor` interface was introduced to let the application 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 discuss an example in chapter 8.

2.2.3 Types

A fundamental and powerful element of the architecture is NHibernate's notion of a `Type`. An NHibernate `Type` object maps a .NET type to a database column type (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 because each RDBMS has a different set of mappings to .NET types.

NHibernate includes 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 let you create your own types. You can also use `IUserCollectionType` to create your own collection types. You can use this feature to handle commonly used application classes such as `Address`, `Name`, and `MonetaryAmount` 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 section 6.1. We now go on to list some of the lower-level interfaces. You may not need to use or understand all of them, but knowing they exist may 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 the following:

- 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 provides. Luckily, you won't need them all. For simple applications, you may need only the `Configuration` and `ISession` interfaces, as shown in the "Hello World" example. But before you can begin to use NHibernate in your applications, you must have some understanding of how NHibernate is configured. That is what we discuss next.

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. Although we concentrate on multitiered web applications in this book, we cover Windows applications when needed.

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

2.3.1 Creating a SessionFactory

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 the previous examples, 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)
        {
            Configuration cfg = new Configuration();
            cfg.Configure();
            cfg.AddInputStream(
                HbmSerializer.Default.Serialize(typeof(Employee)) );
            // OR: cfg.AddXmlFile("Employee.hbm.xml");
            sessionFactory = cfg.BuildSessionFactory();
        }
        return sessionFactory;
    }
}
```

**Done only once
(at first access)**

**When using
NHibernate.Mapping.Attributes**

**When using XML
mapping file**

The location of the mapping file, `Employee.hbm.xml`, is relative to the application's current directory. In this example, you also use an XML file to set all other configuration options (which may have been set earlier by application code or in the application configuration file).

Method chaining

Method chaining is a programming style supported by many NHibernate interfaces (they're 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, but it's 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 and adder methods usually return the receiving object. This would let you rewrite the previous code example as follows:

```
ISessionFactory sessionFactory = new Configuration()
    .Configure()
    .AddXmlFile( "Employee.hbm.xml" )
    .BuildSessionFactory();
```

Notice that you don'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. If you *do* use this coding style, it's better to write each method invocation on a different line. Otherwise, it may 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). The "Hello World" example had only one persistent class. But let's assume you have multiple persistent classes, with an XML mapping file for each. Where should you put these mapping files?

WORKING WITH MAPPING FILES

The NHibernate documentation recommends that the mapping file for each persistent class be 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 you 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's even possible to embed XML mapping files inside .NET assemblies. You have to tell the compiler that each of these files is an embedded resource; most IDEs allow you to specify this option. Then you can use the `AddClass()` method, passing 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's error-prone to use assemblies' names (like `"NHibernate.Auction"`). That's why you use one class's type to directly retrieve the assembly containing the embedded mapping files.

Why does NHibernate say it doesn't know your class?

A common issue when starting to use NHibernate is making sure all your mappings are sent to NHibernate; if you miss one, you'll 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 the `INFO` level (you'll learn how to do that in section 3.3.2). Then read the log to make sure NHibernate read your mappings; you should find a message like *Mapping class: Namespace.YourClass -> YourClass*. If it isn't the case, then check your initialization code to make sure you included the mappings.

If you use `AddAssembly()`, make sure the `hbm.xml` files are embedded in your assembly.

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.

MULTIPLE DATABASES AND SESSION FACTORIES

We've demonstrated the creation of a single `SessionFactory`, which is all that most applications need. If you need another `ISessionFactory` instance—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 you have your `SessionFactory`, you can go on to create sessions, and start loading and saving objects.

CONFIGURATION TECHNIQUES

Of course, there is more to configuring NHibernate than 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 application configuration file (`App.config` or `Web.config`).
- Include `<property>` elements in an 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` file provide the same function: to configure NHibernate. Which file you choose to use depends on your syntax preference. `hibernate.cfg.xml` is the filename chosen by convention. You can use any filename (such as `NHibernate.config`, because `.config` files are automatically protected by ASP.NET when deployed) and provide this filename to the `Configure()` method. It's even possible to mix both options and have different settings for development and deployment.

A rarely used alternative option is to let the application 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 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 Configuring 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.

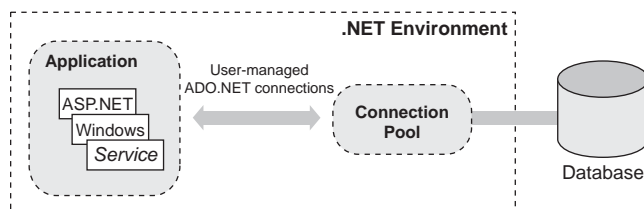


Figure 2.2 Direct access to ADO.NET connections

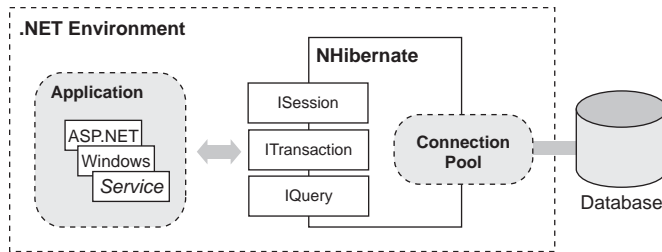


Figure 2.3 NHibernate managing database access

With NHibernate, the picture changes: NHibernate acts as a client of 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.

CONFIGURING NHIBERNATE USING HIBERNATE.CFG.XML

Listing 2.6 uses a file named `hibernate.cfg.xml` to configure NHibernate to access a Microsoft SQL Server 2000 database.

Listing 2.6 Using `hibernate.cfg.xml` to configure NHibernate

```

<?xml version="1.0" ?>
<hibernate-configuration xmlns="urn:hibernate-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:

- `connection.provider` specifies the name of the .NET class implementing the `IConnectionProvider` interface; here, we use the default one.
- `dialect` specifies the name of the .NET class implementing the database `Dialect`. Dialects are how NHibernate can take advantage of database-specific features. Despite the ANSI standardization effort, SQL is implemented differently by various databases vendors. You must specify a `Dialect`. NHibernate includes built-in support for most popular SQL databases, and new dialects may be defined easily.

- `connection.driver_class` specifies the name of the .NET class implementing the ADO.NET Driver. Note when using the partial name of a driver that 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.
- `connection.connection_string` specifies a standard ADO.NET connection string, used to create a database connection.

Note that these names (except the `ConnectionString`) should be fully qualified type names; they aren't here because they're implemented in the `NHibernate.dll` library, which is where the .NET framework looks for non-fully qualified types when NHibernate tries 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's directory. It's 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 your database's ADO.NET data provider isn't yet installed, download and install it; it's usually available from the database vendor website. 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 NHibernate will need.
- 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 `Configure()` method; load the mapped classes (with .NET attributes) using `HbmSerializer.Default.Serialize()` and `AddInputStream()`; and 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're done using NHibernate. Most of the time, you'll do it while closing your application.

There are a few more steps when you use COM+ Enterprise Services; you'll learn more about them in chapter 6. Don't worry; NHibernate code can be easily integrated into COM+ with only a few additions.

You should now have a running NHibernate system. Create and compile a persistent class (the initial `Employee`, for example), add references to `NHibernate.dll`, `log4net`, and `NHibernate.Mapping.Attributes` in 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, and using the convenient XML configuration file instead of plain properties. But if you wish, you may safely skip this section and come back later once you've read more about persistent classes in chapter 3.

2.4 Advanced configuration settings

When you finally have an 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 don'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 need to get started.

But there is one parameter we *must* emphasize at this point. You'll need it continually when 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`, and so on).

2.4.1 Using the application configuration file

You can use the application configuration file to fully configure an `ISessionFactory` instance (as demonstrated in listings 2.7 and 2.8). The application configuration file can contain either configuration parameters using an `<nhibernate>` section, or the same content as `hibernate.cfg.xml` file (using a `<hibernate-configuration>` section). Many users prefer to centralize the configuration of NHibernate this way instead of adding parameters to the `Configuration` in application code.

Listing 2.7 App.config configuration file using `<nhibernate>`

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

  <nhibernate>
```

1 NHibernate
section
declaration


```

<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="initial catalog=nhibernate;Integrated Security=SSPI"
/>
</nhibernate>

<!-- log4net configuration settings here... -->
</configuration>

```

Property specifications ②

③ **Log4net settings could go here**

The NHibernate section is declared ① as a series of key/value entries. The key is the name of the property to set ②. You'll learn about log4net ③ in the next section.

It's recommended that you use a <hibernate-configuration> section, as shown in listing 2.8.

Listing 2.8 App.config configuration file using <hibernate-configuration>

```

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

  <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">
        Initial Catalog=nhibernate;Integrated Security=SSPI
      </property>
    </session-factory>
  </hibernate-configuration>

  <!-- log4net configuration settings here... -->
</configuration>

```

Hibernate configuration section declaration ①

Property specifications ②

Now, ❶ declares a `<hibernate-configuration>`, as in `hibernate.cfg.xml`, which is based on the schema `nhibernate-configuration.xsd`. The value is inside the `<property>` tag ❷.

This way is far more elegant and powerful, because you can also specify assemblies/mapping documents. And you can configure an IDE like Visual Studio to provide IntelliSense inside the `<hibernate-configuration>` section: copy the configuration schema file (`nhibernate-configuration.xsd`) in the subdirectory `\Common7\Packages\schemas\xml\` of the Visual Studio installation directory. You can also configure 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 `hibernate.connection.connection_string_name` property.

Now you can initialize NHibernate as follows:

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

Wait—how does NHibernate know where the configuration file is located?

When `Configure()` is called, NHibernate first searches for the information in the application configuration file and 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 more comfortable than using a programmatic configuration. The fact that you can have the class-mapping files externalized from the application's source (even if it's 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 application configuration file's settings are used.

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

Now that you have a functional NHibernate application, you'll start encountering runtime errors. To ease the debugging process, you 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 generally 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. The NHibernate application can't detect this write-behind (at least, not without resorting to direct ADO.NET calls).

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

We've 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 need to add some information in your application configuration file. The example in listing 2.9 directs all log messages to the console.

Listing 2.9 Basic configuration of log4net

```
<?xml version="1.0" ?>
<configuration>
  <configSections>
    <section
      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 with listing 2.8. With this configuration, you won't see many log messages at runtime.

Replacing the priority value `WARN` with `INFO` or `DEBUG` reveals 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 don't give more details about log4net configuration here; 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 your entities. The next chapter 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 an 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.

Part 2

NHibernate deep dive

This part of the book explains the essential knowledge needed for working with NHibernate. Starting with a complete application, we walk you through the steps needed to design, implement, and optimize NHibernate applications. This section also gives you expertise in some of the less-understood parts of NHibernate, which will help you succeed with even the most complex projects.



Writing and mapping classes

This chapter covers

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

The “Hello World” example in chapter 2 gave a gentle introduction to NHibernate; but we need a more thorough example to demonstrate the needs of real-world applications with complex data models. For the rest of the book, we explore NHibernate using a more sophisticated example application—an online auction system.

We start our discussion of the application by introducing a programming model for persistent classes.

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

You'll then define *mapping metadata* to tell NHibernate how these classes and their properties relate to database tables and columns. We 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 section 1.3.1. For example, how do you map fine-grained objects to simple tables? Or how do you map inheritance hierarchies to tables?

We 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 website <http://caveatemptor.hibernate.org/>. The application will have a console-based user interface. We don't pay much attention to the user interface; we concentrate on the data-access code. In chapter 8, we discuss the changes that would be necessary if you were to perform all business logic and data access from a separate business tier. And in chapter 10, we discuss many solutions to common issues that arise when integrating NHibernate in Windows and web applications.

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 you're building it from scratch. Your first task is *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 may be abstractions of less concrete things the user thinks about (for example, PricingAlgorithm), but even these are usually 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 Computer-Aided Software Engineering (CASE) tool like Microsoft Visio, Sparx Systems Enterprise Architect, or UMLet.

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

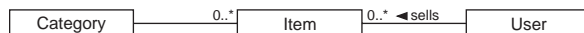


Figure 3.1 A class diagram of a typical online auction object model

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'll refer to this model when you implement your persistent .NET classes.

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

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. Your 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 takes 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 the domain model is shown in figure 3.2. Let's briefly discuss some interesting features of this model.

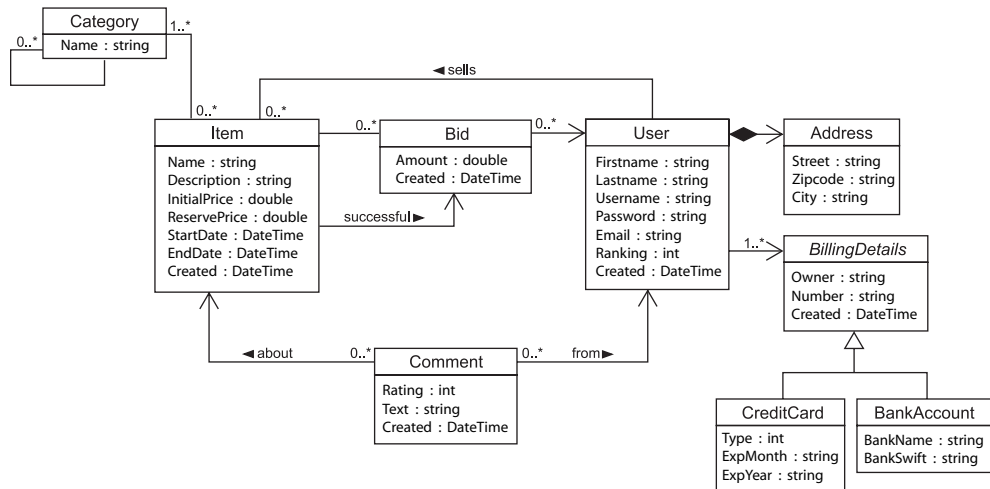


Figure 3.2 Persistent classes of the CaveatEmptor object model and their relationships

Each item may be auctioned only once, so you don't need to make `Item` distinct from the `Auction` entities. Instead, you have a single auction item entity named `Item`. `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`. You do let the user have multiple `BillingDetails`. The various billing strategies are represented as subclasses of an abstract class (allowing future extension).

A `Category` may 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). This 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 *Domain-Driven Design* [Evans 2004].

In this book, we don't have much to say about business rules or the *behavior* of the domain model. This isn't because we consider them unimportant concerns; rather, they're mostly orthogonal to the problem of persistence. It's the *state* of your entities that is persistent. So we concentrate our discussion on how to best represent state in your 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.

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

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

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: *automated* or *transparent*? 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 important, 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. But 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*.

The DataSet doesn't address this problem. It can't be regarded as a domain model mainly because it isn't designed to include business rules.

Much discussion has gone into the topic of persistence, and both NHibernate and DataSets take care of that concern. But 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 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 on—the persistence mechanism.

The Item class, for example, won't 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.
- 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're 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. Your business logic is fully reusable in any other application context. You could easily change to another transparent persistence mechanism.

By this definition of transparent persistence, 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. Transparent persistence should be one of the primary goals of any ORM solution. But 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 `ICollection` 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, “Going forward.”)

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's a back-to-basics approach that essentially consists of using unbound classes in the business layer.¹

When you're 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. Most POCOs are NHibernate-compatible without any changes. The programming model we 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 the example domain model.

Listing 3.1 POCO implementation of the User class

```
[Serializable]
public class User {
    private string username;
    private Address address;
    public User() {}
    public string Username {
        get { return username; }
        set { username = value; }
    }
    public Address Address {
        get { return address; }
        set { address = value; }
    }
    public MonetaryAmount CalcShipCosts(Address from) {
        // ...
    }
}
```

1 Serializable class

2 Class constructor

3 Properties

4 Business method

NHibernate doesn't require persistent classes to be serializable (as this class is 1). But serializability is commonly needed, mainly when you're using .NET remoting.

NHibernate requires a default parameterless constructor for every persistent class 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 your business entities 3. For example, the User's name Username provides access to the private username instance variable (the same is true for Address). NHibernate doesn't require that properties be declared public; it can easily use private ones too. Some properties do

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

something more sophisticated than simple instance variables access (validation, for example), but trivial properties are common. Of course, if you're using C# 3.0, you can take advantage of the auto-implemented properties for these simple cases.

This POCO also defines a business method ④ 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.

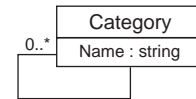


Figure 3.3 Diagram of the `Category` class with an association

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

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

To allow bidirectional navigation of the association, you 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. Here you use 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's good practice to program to collection interfaces rather than concrete implementations, so this restriction shouldn't bother you.

You 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:

Used external library: Iesi.Collections

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

```
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

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, 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 you’d like the `AddChildCategory()` to be the only externally visible mutator method for the child categories, you make the `ChildCategories` property private; you may add more methods to access to `ChildCategories` if required. NHibernate doesn’t care if properties are private or public, so you 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).

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 the `Category` class, as shown in listing 3.2.

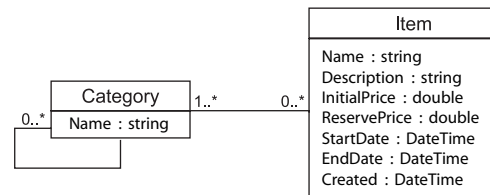


Figure 3.4 Category and the associated Item

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. You 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;
    //...
    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()` method of the `Item` class 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 don'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; these classes 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 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: you can change a property's hidden internal implementation without any changes to the public interface. This lets you abstract a class's internal data structure—the instance variables—from the design of the database.

For example, if your database stores a username 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 an NHibernate *custom type* is probably a better way to handle many of these kinds of situations. But 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 {
            if ( !StringUtil.IsCapitalizedName(firstname) )
                throw new InvalidNameException(value);
            firstname = value;
        }
    }
    //...
}
```

NHibernate will later use your properties to populate the state of an object when loading the object from the database. Sometimes you'd prefer that this validation *not* occur when NHibernate is initializing a newly loaded object. In that case, it may make sense to tell NHibernate to directly access the instance variables (you'll see later that you 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 safe to return a different object from the `get` accessor to the object passed by NHibernate to the `set` accessor. NHibernate compares the objects by value—not by object identity—to determine whether 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); }
}
```

But there is one 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 updates 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); }
}
```

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 shouldn't 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, you've defined a number of classes for your domain model and set up some associations between them. You've also added convenience methods to make working with the model easier, and added some business logic. Your next goal is to be able to load and save objects in the domain model to and from a relational database. We now need to look at setting up the necessary pieces to let NHibernate persist objects—or, more specifically, 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.

It's our job as developers to define and maintain this metadata. We can do this two different ways: attributes and XML mapping files. In this section, you'll learn how to write mapping using these two approaches, and we'll compare them so you can decide which one to use. Let's start with the mapping you're familiar with: 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).

But is XML-based metadata a viable approach? A certain backlash against the over-use 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. 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 isn't 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, an XML Schema Definition (XSD) often isn't provided, preventing auto-completion and validation. Also problematic are XSDs that are too generic, where every declaration is wrapped in a generic extension of a meta element (like the key/value approach).

There is no getting around the need for text-based metadata in ORM. But 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 you can use to generate NHibernate XML mappings.

Let's look at the way you can use XML metadata in NHibernate. We introduced the mapping of the `Category` class in a previous section; now we provide more details about the structure of its XML mapping document shown in listing 3.4.

Listing 3.4 NHibernate XML mapping of the `Category` class

```

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

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

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

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

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

```

Annotations in the diagram:

- 1 Mapping declaration (points to `<hibernate-mapping>`)
- 2 XSD declaration (optional) (points to `xmlns="urn:hibernate-mapping-2.2"`)
- 3 Category class mapping (points to `<class>`)
- 4 Identifier mapping (points to `<id>`)
- 5 Name property mapping (points to `<property>`)
- 6 Reference to parent category (points to `<many-to-one>`)

As you can see, an XML mapping document can be divided into many parts:

- Mappings are declared inside a `<hibernate-mapping>` element ❶. You can include as many class mappings as you like, along with certain other special declarations that we mention later in the book.
- The NHibernate mapping XSD is declared to provide syntactic validation of the XML ❷, and many XML editors use it for auto-completion. But it isn't recommended that you use the online copy of this file, for performance reasons.
- The `Category` class (in the assembly `CaveatEmptor.Model`) is mapped to the table of the same name (`Category`) ❸. Every row in this table represents one instance of type `Category`.
- We haven't discussed the concept of *object identity* much. This complex topic is covered in section 3.5. 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 ④.

- The `Name` property is mapped to a database column of the same name (`Name`) ⑤. 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's possible to explicitly specify the *mapping data type* that NHibernate should use. We take a close look at these types in section 7.1.
- You use an association to link a `Category` to another. Here, it's a *many-to-one* association ⑥. In the database, the `Category` table contains a `ParentCategory` column that is a foreign key to another row in the same table. Association mappings are more complex, so we return to them in 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 to define your mappings; next, we briefly explain how to do this. After that, we look more closely at the nature of the class and property mappings described 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 has provided support for class/member attributes. In chapter 2, we introduced the `NHibernate.Mapping.Attributes` library, which uses attributes directly embedded in the .NET source code to provide all the information NHibernate needs to map classes. All you have to do is to mark up the .NET source code of your 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's easy to use this mapping with NHibernate:

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

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

XML mapping or .NET attributes?

We've introduced mapping using XML mapping files and using `NHibernate.Mapping.Attributes`. Although you can use both at the same time, it's 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're 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. But 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're also easier to manipulate for complex mapping and they can contain some useful information (not directly related to the mapping of the classes). It's common to use XML mapping files when the classes already exist and aren't under our control.

Note that, in few cases, it's 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.

You should now have grasped the basic idea of how both XML mappings and attribute mappings work. Next, we 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'll learn a number of features and tips that will help you write better mappings. NHibernate can "guess" some information to make your mappings shorter. It's 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 an NHibernate type. It maps a .NET property to a table column.

The basic declaration provides many variations and optional settings; for example, it's often possible to omit the type name. For example, if `Description` is a property of (.NET) type `String`, NHibernate uses 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're 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 may need to tell NHibernate more about the database column than just its 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 aren't 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 them based on where the attribute mappings are in the code. The exception is `[Id]`; you must specify the identifier's name when it has one, because it's optional.

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. But it's still preferable to include at least the not-null attribute, because NHibernate can then 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. In particular, a *derived* property takes its value from a SQL expression.

3.4.2 Using derived properties

The value of a derived property is calculated at runtime by evaluating an expression. You define the expression using the `formula` attribute. For example, for a `ShoppingCart` class, you might map a `TotalIncludingTax` property. Because it's 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 a 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 *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 explains 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—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 simple:

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

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

In our “Hello World” example in chapter 2, you used the `field` access strategy in the XML mapping file for the `Employee` entity. The `field` strategy is useful 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's using the field access strategy:

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

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 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, you only want this logic to be executed when your client code sets the property, not during load time. If a class is mapped using a property setter, NHibernate runs the code as it loads the object. Thankfully, there are ways to deal with this situation. NHibernate provides 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 while writing data to it.

If you need even more flexibility than this, you can learn about other access strategies available in the NHibernate reference documentation online. As a sample, if you want NHibernate not to use the getter if you use the standard way of naming fields in C#—camelcase 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 you'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 want to do this for all properties on a class, you can specify this access strategy at the 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`; you name the class implementing your custom strategy in the access

attribute (using its fully qualified name). With `NHibernate.Mapping.Attributes`, you have the alternative of using

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

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, you've learned how to build the classes for your domain model and how to define mapping metadata to tell NHibernate how to persist these classes and their members. NHibernate gives you a wealth of features and flexibility, but essentially we're talking about straightforward ORM capabilities. Next, we look at some under-the-hood 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 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 that can access them without using reflection. A small caveat is that it only works for public properties; you must use the default property-access strategy in your mapping files to get optimal results. Another restriction is that quoted SQL identifiers (section 3.4.6) aren't supported.

NHibernate 1.2 introduces another provider, which only works (and is used by default) under .NET 2.0. This provider injects dynamic methods into your business entities at startup, and it's more powerful because it isn't 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 instantiating the `Configuration`. You can't use the `<hibernate-configuration>` section in your config file (`hibernate.cfg.xml`, `web.config`, and so on) because this is read *after* the `Configuration` object is created (during the call to your `Configuration` object's `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.

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 that implements the interface `NHibernate.Bytecode.IBytecodeProvider`.

The next interesting capability we look at is controlling database inserts and updates for classes and their members. This level of control is 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

You can control whether properties that map to columns 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 entity's `Name` property is 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 classes that you've decided should never be updated after they've been created. An example might be a financial transaction record.)

In addition, the `dynamic-insert` and `dynamic-update` attributes 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; when NHibernate generates `INSERT` and `UPDATE SQL` for an object, it does so for all properties on the object regardless of whether they've changed since the object was loaded. Enabling either of these settings causes NHibernate to generate 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 lets you 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 always quotes this identifier in the generated SQL. The following property declaration forces NHibernate to generate SQL with the quoted column name "Item Description". NHibernate also knows that Microsoft SQL Server needs the variation [Item Description] and that MySQL requires `Item Description`:

```
<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 you further control when mapping between your domain model and the database schema, by also letting you control naming conventions. We discuss this next.

3.4.7 Naming conventions

Development teams must often follow strict conventions for table and column names in their databases. NHibernate provides a feature that lets you 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 your mapping files. This approach is time consuming and easily forgotten. Instead, you 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 you enable `CENamingStrategy`, this class mapping declaration

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

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

But if you specify a table name

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

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

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

```

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

```

This lets you 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.

But a better way to handle this kind of requirement is to use the concept of a 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. You can specify a default schema using the `hibernate.default_schema` configuration option; doing so offers some small performance benefits.

Alternatively, if your database is like AdventureWorks 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>
```

You can even declare a schema for the whole document:

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

Next, we discuss another useful thing you can do with the `<hibernate-mapping>` element: 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 you name a class in your mapping documents.

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

```
<?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>
```

You don't want to repeat the fully qualified name whenever this or any other class is named in an association, subclass, or component mapping. Instead, you can 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. But this setting is mostly useless when using attribute mapping, because you can specify the .NET type, and `NHibernate.Mapping.Attributes` will write its fully qualified name.

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

NOTE We'll no longer write the XSD information, because it clutters the examples.

Both approaches we've described—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: you can create or manipulate an XML document at runtime before feeding it to the `Configuration` object.

But 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 `Motto` property to the `User` class mapping:

```
PersistentClass userMapping = cfg.getClassMapping(typeof(User));

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

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

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

ISessionFactory sf = cfg.BuildSessionFactory();
```

Get mapping information for User from configuration

Define new column for USER table

Wrap column in value

Define new property of User class

Build new session factory using 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. 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`. But the application may read the `ISessionFactory`'s metamodel by calling `GetClassMetadata()` or `GetCollectionMetadata()`. Here's an 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 the previous examples: the *identifier property mapping*. We 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. You need these concepts if you want to finish mapping the `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.

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 isn't 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. 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 the domain model—it's a persistence-related concern, not part of the business problem. In our examples, the identifier property is always named `id`. 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 may 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 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 let you 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")]
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 CATEGORY table. 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 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).

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 section 4.1.5). 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. Choosing a good primary key (and key-generation strategy) can be more difficult. We discuss these issues next.

3.5.3 Choosing primary keys

You have to tell NHibernate about your preferred strategy for generating a primary key. 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 may satisfy these properties. If a table has only one identifying attribute, it's 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 an 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). 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're unique values generated by the database or application. You can use a number of well-known approaches to generate surrogate keys.

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

You aren't limited to these built-in strategies; you can learn about others by reading NHibernate's reference documentation. You may also create your own identifier generator by implementing NHibernate's `IIdentifierGenerator` interface. It's even

Table 3.1 NHibernate's built-in identifier generator modules

Generator name	Description
<code>native</code>	Picks other identity generators like <code>identity</code> , <code>sequence</code> , or <code>hilo</code> depending on the capabilities of the underlying database.
<code>identity</code>	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 supported.
<code>sequence</code>	Uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi, and Firebird. The identifier returned by the database is converted to the property type using <code>Convert.ChangeType</code> . Any integral property type is thus supported.
<code>increment</code>	At NHibernate startup, reads the table's maximum primary-key column value 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).
<code>hilo</code>	Generates identifiers that are unique only for a particular database. A high/low algorithm is an efficient way to generate identifiers of any integral type, given a table and column (by default <code>hibernate_unique_key</code> and <code>next_hi</code> , respectively) as a source of hi values. See (Ambler 2002) for more information about the high/low approach to unique identifiers. Don't use this generator with a user-supplied connection.
<code>uuid.hex</code>	Uses <code>System.Guid</code> and its <code>ToString(string format)</code> method to generate identifiers of type <code>string</code> . The length of the string returned depends on the configured format. This generation strategy isn't popular, because <code>CHAR</code> primary keys consume more database space than numeric keys and are marginally slower.
<code>guid</code>	Used when the identifier's type is <code>Guid</code> . The identifier must have <code>Guid.Empty</code> as default value. When saving the entity, this generator assigns it a new value using <code>Guid.NewGuid()</code> .
<code>guid.comb</code>	Similar to <code>guid</code> , but uses another algorithm that makes it almost as fast as when using integers (especially when saving in a SQL Server database). The generated values are ordered; you can use a part of these values as reference numbers, for example.

possible to mix identifier generators for persistent classes in a single domain model, but for nonlegacy 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, you'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 section 9.2, “Legacy schemas.”

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.

In crude terms, *fine-grained* means “more classes than tables.” For example, a user may have both a billing address and a home address. In the database, you may 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 your object model, you can use the same approach, representing the two addresses as six string-valued properties of the User class. But it’s much better to 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 EmailAddress class that can add higher-level semantics and behavior. For example, it may 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 advocate 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 section 6.1.)

3.6.2 Using components

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

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 this case, you 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.

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

.NET 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; you 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*.)

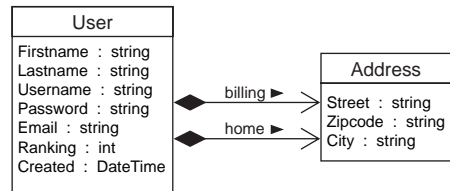


Figure 3.5 Relationships between `User` and `Address` using composition

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">
        <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
        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

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

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

Components may be harder to map with NHibernate. Mapping.Attributes. When you're using a component in many classes with the identical mapping, it's easy to do (far easier than with XML mapping):

<<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

```

[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]
    public Address Location { ... }
}

```

But the User class has two addresses, each mapped to different columns. There are many ways to map them (see 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 User class. This solution isn't elegant. You won't have to deal with this kind of mapping 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; you 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 which kind of mapping the XML should be inserted after; most of the time, it's the type of the attribute 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's the string containing the XML to include.

Notice that in this example, you model the composition association as *unidirectional*. You can't navigate from `Address` to `User`. NHibernate supports both unidirectional and bidirectional compositions, but 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"/>
    <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`. You then call `Address.Owner` to navigate in the other direction.

An 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 discuss various component mappings in chapter 6.)

But classes mapped as components have two important limitations:

- *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's also possible to make a component immutable:

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

When you make entities or components immutable, they may not be updated or deleted. This allows NHibernate to make minor performance optimizations. More important, immutable objects are much simpler to deal with: they can be shared and copied, and you're safe in the knowledge that they can't be changed.

Support for fine-grained classes is just one ingredient of a rich domain model. We now look at creating and mapping associations between the domain model classes.

3.7 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 isn't without pitfalls, especially for new users. In this section, we don't try to cover all the possible combinations; we 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, we need to explain something up front.

3.7.1 Unidirectional associations

When you're using (typed) DataSets, associations are represented as in a database. To link two entities, you have to set the foreign key in one entity to the primary key of the other. There isn't 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's 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 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 aren't bound to any context. In NHibernate applications, the behavior of a nonpersistent instance is the same as the behavior of a persistent instance.

Because associations are so important, we need precise language for classifying them.

3.7.2 Multiplicity

In describing and classifying associations, you'll almost always use the *multiplicity* association. Look at figure 3.7.

The multiplicity consists of two bits of information:

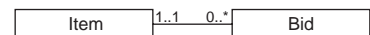


Figure 3.7 Relationship between *Item* and *Bid*

- 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, you can conclude that the association from Bid to Item is a *many-to-one* association. Recalling that associations are directional, you can also call the inverse association from Item to Bid a *one-to-many* association. (There are two more possibilities: *many-to-many* and *one-to-one*; we get back to them in chapter 6.)

In the context of object persistence, we aren't interested in whether "many" means "two" or "maximum of five" or "unrestricted."

3.7.3 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.

You explicitly specify the Item class, which the association refers to. This specification is usually optional, because NHibernate can determine this using reflection.

You specify the not-null attribute because you 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, a many-to-one association may point to a nonexistent entity. The property `not-found` lets you define how NHibernate should react to this situation:

```
<many-to-one ... not-found="ignore|exception" />
```

Using `not-found="exception"` (the default value), NHibernate throws an exception. And `not-found="ignore"` makes NHibernate ignore this association (leaving it null).

3.7.4 Making the association bidirectional

So far so good. But you also need to be able to easily fetch all the bids for a particular item. You need a bidirectional association here, so you have to add scaffolding code to the `Item` class:

```
public class Item {
    //...
    private ISet bids = new HashedSet();
    public ISet Bids {
        get { return bids; }
        set { bids = value; }
    }
    public void AddBid(Bid bid) {
        bid.Item = this;
        bids.Add(bid);
    }
    //...
}
```

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 you specify the same foreign-key column in this collection mapping that you specified in the mapping for the many-to-one association. The table structure for this association mapping is shown in figure 3.8.

Now you 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 your 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, because at this point you've done nothing to indicate that this is a bidirectional association.

You need one more thing in your 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>
  </class>
```

Without the `inverse` attribute, NHibernate would try to execute two different SQL statements, both updating the same foreign-key column, when you manipulate the association between the two instances. By specifying `inverse="true"`, you explicitly tell NHibernate which end of the association it should synchronize with the database. In this example, you 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. If you only call `item.Bids.Add(bid)`, no changes are 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.

You now have a working *bidirectional many-to-one association* (which could also be called a bidirectional one-to-many association, of course).

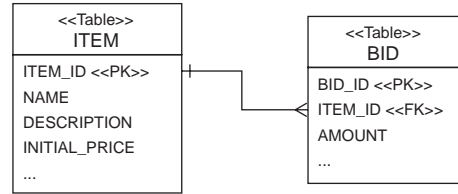


Figure 3.8 Table relationships and keys for a one-to-many/many-to-one mapping

CASCADING SAVES AND DELETES

One final piece is missing. We explore the notion of *transitive persistence* in much greater detail in the next chapter. For now, we introduce the concepts of *cascading save* and *cascading delete*, which you need in order to finish mapping this association.

When you instantiate a new `Bid` and add it to an `Item`, the bid should become persistent immediately. You want to avoid the need to explicitly make a `Bid` persistent by calling `Save()` on the `ISession` interface.

You 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. You 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 you finished? Not quite. You still need to define the lifecycle for both entities in your association.

3.7.5 A parent/child relationship

With the previous mapping, the association between `Bid` and `Item` is fairly loose. You'd 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 the 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 you 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 semantic 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 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, you 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 you 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>
```

You use `cascade="all-delete-orphan"` to indicate the following:

- Any newly instantiated `Bid` becomes persistent if the `Bid` is referenced by a persistent `Item` (as is 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.)

You achieve 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` is removed).

The cascading of operations to associated entities is NHibernate's implementation of transitive persistence. We look more closely at this concept in section 4.3.

We've covered only a tiny subset of the association options available in NHibernate. But you already have enough knowledge to be able to build entire applications. The remaining options are either rare or variations of the associations we've described.

We recommend keeping your association mappings simple and using NHibernate queries for more complex tasks.

So far we've covered how to map classes, components and associations. We now look at an essential capability of NHibernate—mapping inheritance hierarchies.

3.8 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. We end the chapter by exploring this somewhat advanced topic; if you're interested primarily in basic NHibernate usage, feel free to skip to chapter 4.

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.

You can use three different approaches to represent 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 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 you’re starting with a domain model and trying to derive a new SQL schema. But the mapping strategies described are just as relevant if you’re working *bottom up*, starting with existing database tables.

3.8.1 Table per concrete class

Suppose you stick with the simplest approach: you can use exactly one table for each (non-abstract) class. All properties of a class, including inherited properties, can be mapped to columns of this table, as shown in figure 3.9.

The main problem with this approach is that it doesn’t support polymorphic associations well. In the database, associations are usually represented as foreign-key relationships. In figure 3.9, 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 the domain model, because `BillingDetails` is associated with `User`; 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. You might be able to use a SQL `UNION` to improve performance by avoiding multiple round trips to the database. But 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:

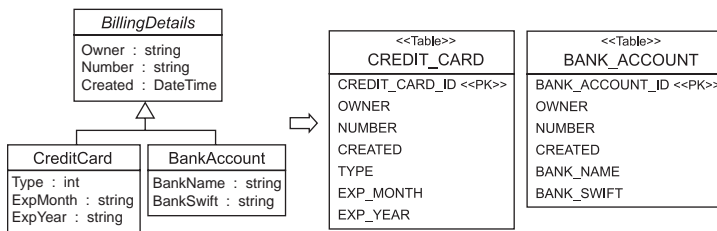


Figure 3.9 Mapping table per concrete class

```

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 may 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: you 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.

3.8.2 Table per class hierarchy

Alternatively, an entire class hierarchy can be mapped to a single table. This table includes 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 3.10.

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 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 can be a serious problem from the point of view of data integrity.

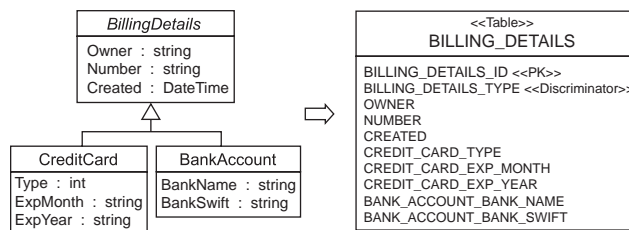


Figure 3.10 Table-per-class hierarchy mapping

In NHibernate, you 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">
    <id
      name="Id"
      column="BILLING_DETAILS_ID"
      type="Int64">
      <generator class="native" />
    </id>
    <discriminator
      column="BILLING_DETAILS_TYPE"
      type="String" />
    <property
      name="Name"
      column="OWNER"
      type="String" />
    ...
    <subclass
      name="CreditCard"
      discriminator-value="CC">
      <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`.
- 2 You 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 are strings—in this case, "CC" (credit card) or "BA" (bank account). NHibernate automatically sets and retrieves 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 any more 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 you want to specify a class in the mapping, you can add "Type" to the element's name; for the attribute `[Discriminator]`, you use `TypeType`. Note that this attribute can be written before any property because it isn't linked to any field/property of the class (if there is more than this attribute on the property, make sure it comes after the `[Id]` and before the other attributes).

NHibernate will 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 uses 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's 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, you use the column `CREDIT_CARD_TYPE` to evaluate the type.

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

3.8.3 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) along with a primary key that is also a foreign key of the base class table. This approach is shown in figure 3.11.

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, you can retrieve the subclass instance from the database by joining the subclass table with the base class table.

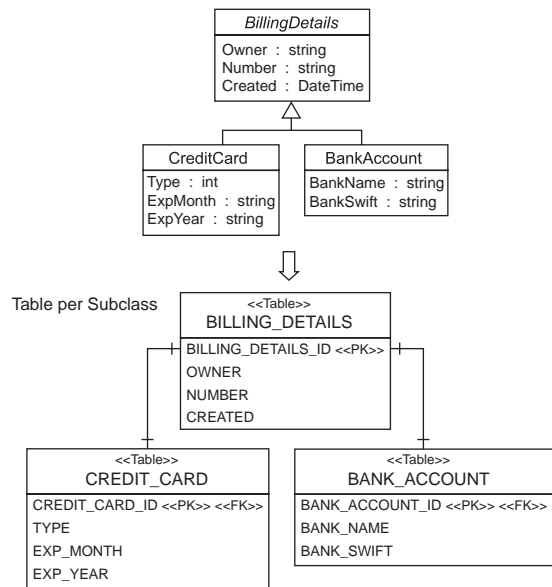


Figure 3.11 Table-per-subclass mapping

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, you 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"
      column="BILLING_DETAILS_ID"
      type="Int64">
      <generator class="native"/>
    </id>
    <property
      name="Owner"
      column="OWNER"
      type="String"/>
    </class>
```

① **BillingDetails root class, mapped to BILLING_DETAILS table**

```

...
<joined-subclass
  name="CreditCard"
  table="CREDIT_CARD">
    <key column="CREDIT_CARD_ID">
      <property
        name="Type"
        column="TYPE" />
    ...
  </joined-subclass>
...
</class>
</hibernate-mapping>

```

Diagram annotations:

- ② **<joined-subclass> element**: points to the `<joined-subclass>` tag.
- ③ **Primary/foreign key**: points to the `<key>` tag.

- ① Again, the root class `BillingDetails` is mapped to the `BILLING_DETAILS` table. 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 are 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 also has 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 = ?

```

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.)

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. The problem should be recast as how to choose an appropriate *combination* of mapping strategies for an application's class hierarchies. A typical domain model design has a mix of interfaces and abstract classes.

3.8.4 *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, 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` to another file and map it like this:

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

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 explored 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.

Finally, we introduced the three well-known inheritance-mapping strategies in NHibernate. We also covered associations and collections mapping; and you implemented and mapped your first parent/child association between persistent classes, using database foreign key fields and the cascading of operations.

With this understanding, you can experiment with NHibernate and handle most common mapping scenarios, and 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.

4

Working with persistent objects

This chapter covers

- The lifecycle of objects in an NHibernate application
- Using the session persistence manager
- Transitive persistence
- Efficient fetching strategy

You now understand how NHibernate and ORM solve the static, structural aspects of the object/relational mismatch introduced in section 1.3.1. More specifically, you learned how object-oriented structures can be mapped to relational database structures to address issues of granularity, identity, inheritance, polymorphism, and associations.

This chapter covers another crucial subject—the dynamic, *behavioral* aspects of the object/relational mismatch. Success with NHibernate will not be guaranteed by simply mapping your domain classes to your databases. You must understand the dynamic nature of the problems that come into play at runtime, and which greatly affect the performance and stability of your applications. In our experience, many developers focus mostly on 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 may be rather formal, but a solid understanding of the *persistence lifecycle* will greatly help you in your application development with NHibernate.

4.1 The persistence lifecycle

Because 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.

But 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 API. 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 may 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*.

Figure 4.1 shows these states and their transitions in a state chart. You can also see the method calls to the persistence manager that trigger

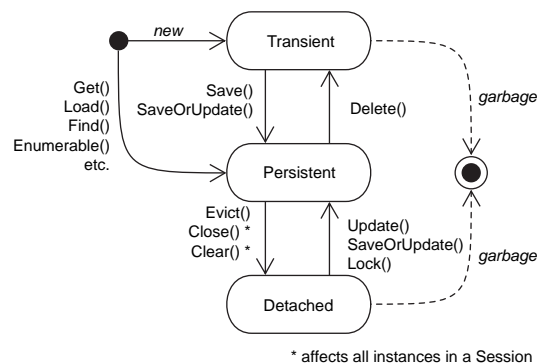


Figure 4.1 States of an object and transitions in an NHibernate application

transitions. We discuss this chart in this section; refer to it later whenever you need an overview.

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

When using NHibernate, simply creating objects using the `new` operator will not make them immediately persistent. At this point, their state is *transient*, which means they aren't associated with any database table row. This is similar to any other object in a .NET application. As you would expect, their state is lost as soon as they're *dereferenced* (no longer referenced by any other object) 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. To transition an object from transient to persistent state, there are two choices. You can `Save()` it using the persistence manager, or create a reference to it from an already-persistent instance and take advantage of transitive persistence (section 4.3).

4.1.2 Persistent objects

A persistent instance is any instance with a *database identity*, as defined in section 3.5. That means a persistent object 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 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 can also occur at other times. For example, NHibernate may synchronize with the database before execution of a query. This ensures that queries are aware of changes made earlier during the transaction.

We call a persistent instance *new* if it has been allocated a primary key value but hasn't 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 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 may bring performance gains, particularly with certain databases. But 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. 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 hand-coded persistence layer. We talk about NHibernate's transaction semantics and the synchronization process, or *flushing*, in more detail in the next chapter.

Finally, you can make a persistent instance transient via a `Delete()` call to the persistence manager API, resulting in the deletion of the corresponding row of the database table.

4.1.3 Detached objects

When a transaction completes and the data is written to the database, the persistent instances associated with the persistence manager still exist in memory. If the transaction was successful, the state of these instances has 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.

But in the case of NHibernate, these instances lose their association with the persistence manager when you `Close()` the `ISession`. Because they're no longer associated with a 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. But 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 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 section 10.3.1.

NHibernate also provides an explicit way of detaching instances: the `Evict()` method of the `ISession`. This method is typically used only for cache management (a performance consideration). It's *not* common 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). 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`). If an object changes state, is its .NET identity guaranteed to be the same in the new state? In a layered application, that may 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 they're 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, only one object instance 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 (.NET CLR).

For a typical web or enterprise application, transaction-scoped identity is preferred. Process-scoped identity offers potential advantages in terms of cache utilization and the programming model for reuse of instances across multiple transactions; but in a pervasively multithreaded application, the cost of always synchronizing shared access to persistent objects in the global identity map is too high. 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 can 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 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 example demonstrates this behavior, with several `Load()` operations in two `ISessions`:

```
ISession session1 = sessionFactory.OpenSession();
ITransaction tx1 = session1.BeginTransaction();
// Load Category with identifier value "1234"
object a = session1.Load<Category>( 1234 );
object b = session1.Load<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 because they were loaded in the same `ISession`. Once outside this boundary, NHibernate doesn't guarantee .NET identity, so `a` and `b2` aren't identical and the message is printed on the console. 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, after it's been modified by the user. For ASP.NET applications this doesn't apply, because you shouldn't keep business objects in memory after the page has been rendered—instead, you reload them on each request, and they don't require reattachment.

When you 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 may 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` may 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 an "old" detached instance. Transient instances (such as the `Bid`) may need to be saved; detached instances (such as the `Item`) may need to be reattached (and later updated in the database).

There are several ways to distinguish between transient and detached instances, but the simplest 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 4.3.4.

If you want to take advantage of NHibernate's support for reassociation of detached instances in your 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 of equality, see section 3.5.1. 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 domain model's persistent classes.

4.1.6 Implementing `Equals()` and `GetHashCode()`

The `Equals()` method is called by application code or, more important, 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`. (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.) But as soon as you have instances from multiple sessions, 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 using the built-in identity equality, 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.

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 must always 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 seemingly 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 (this.Id==null) return false;
        if ( !(other is User) ) return false;
        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, because 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. 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 saves (discussed later in this chapter) useless for sets. We strongly discourage this solution (database identifier equality).

You could fix this problem by assigning an identifier yourself at the creation of the entities and using versioning to distinguish transient and detached instances.

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. Here 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;
    }
}
```

But again, this approach has two problems:

- 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) can be considered equal, unless some combination of properties is guaranteed to be unique (the database columns have a unique constraint). In the case of `User`, there is a unique property: `Username`.

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 may 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, and 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 may have noticed that the `Equals()` and `GetHashCode()` methods always access the properties of the other object via the getter properties. This is important, because 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 you're 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 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 come back to detached objects in more detail in the next chapter.

4.2 The persistence manager

Any transparent persistence tool like NHibernate will include some form of *persistence manager* API, which usually provides services for the following:

- Performing basic CRUD operations
- Executing queries
- Controlling transactions
- Managing the transaction-level cache

The persistence manager can be exposed by several different interfaces (in the case of NHibernate, they include `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 `ISession`; it's your starting point for all the operations just listed. For most of the rest of this book, we refer to the *persistence manager* and the *session* interchangeably; this is consistent with usage in the NHibernate community.

How do you start using the session? At the beginning of a unit of work, you create an instance of `ISession` using the application's `ISessionFactory`. The application may 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. Note that this section explains some of the transitions shown earlier in figure 4.1.

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 = "Mark";
user.Name.Lastname = "Monster";
using( ISession session = sessionFactory.OpenSession() )
using( session.BeginTransaction() ) {
    session.Save(user);
    session.Transaction.Commit();
}
```

First, you instantiate a new transient object `user` as usual. You can also instantiate it after opening an `ISession`; they aren't related yet. You open a new `ISession` using

the `ISessionFactory` referred to by `sessionFactory`, and then you start a new database transaction.

A call to `Save()` makes the transient instance of `User` persistent. It's now associated with the current `ISession`. But no SQL `INSERT` has yet been executed; the `NHibernate ISession` never executes any SQL statement until absolutely necessary.

The changes made to persistent objects must be synchronized with the database at some point. This happens when you `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 `Save()` was called*. You can, of course, modify the object after calling `Save()`, and your changes will be propagated to the database as a SQL `UPDATE`.

Everything between `session.BeginTransaction()` and `Transaction.Commit()` occurs in one database transaction. We haven't discussed transactions in detail yet; we 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. But `NHibernate` does *not* roll back in-memory changes to persistent objects; their state remains exactly as you left it. This is reasonable because 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 has no effect on its persistent representation in the database. When the session is closed, user becomes a *detached* instance. But 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()` forces an update to the persistent state of the object in the database; a 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 with 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. This makes `Update()` a safe way to reassociate objects with a `Session`, because you know changes will be propagated to the database. There is one exception: when you've enabled `select-before-update` in the persistent class mapping. With this option enabled, a call to `Update()` will make NHibernate *determine* whether the object is dirty rather than assuming it is. It does so by executing a `SELECT` statement and comparing the object's current state to the current database state. This is still a "safe" option, even though NHibernate won't force an update if it isn't needed.

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 you're 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 the detached instance hasn't been modified beforehand.

The previous code specifies `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 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 you reassociate them with a `Session`, let's now see what happens when you 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. But 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, you can use .NET 2.0 generics:

```
User user = session.Get<User>(userID);
```

Next, we explain the concept of automatic dirty checking.

4.2.4 Updating a persistent object transparently

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 you retrieve the object from the database with the given identifier. You modify the object, and these modifications are propagated to the database when `Transaction.Commit()` is called. Of course, as soon as you close the `ISession`, the instance is considered detached. Batch updates are also possible because NHibernate has been tweaked to use the ADO.NET 2.0 batching internal feature. Enabling this feature makes NHibernate perform bulk updates; these updates therefore become much faster. All you have to do is define the batch size as an NHibernate property:

```
<property name="hibernate.adonet.batch_size">16</property>
```

By default, the batch size is 0, which means 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 NHibernate may not perform optimistic concurrency checking correctly. For example, if one statement affects two rows and another statement affects no rows (instead of affecting one each), NHibernate will only know that two rows have been affected, and conclude that everything went OK.

4.2.5 Making an object transient

In many use cases, you need persistent (or detached) objects to become transient again, meaning they will no longer have corresponding data in the database. As we 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 removes their persistent state from the database. You can easily do this 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 is 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 is destroyed by the garbage collector if it's no longer referenced by any other object; both the in-memory instance and the persistent database row are removed.

Similarly, detached objects may be made transient. (Detached objects have corresponding state in the database but aren't 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. Using these concepts 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 haven't shown you any exception-handling code so far, but you should be able to figure out the try/catch blocks yourself (as in chapter 2). Map some simple entity classes and components, and then store and load objects in a standalone console application (write a Main method). But 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 3 when we mapped a parent/child relationship between `Bid` and `Item`. In that case, not 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.

More than one model exists for transitive persistence. The best known is *persistence by reachability*, which we 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 4.2.

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 possible only to child categories and not to the parent—for example, you 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; you can re-create any object graph 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 a SQL database and no persistent garbage collector that can detect unreferenced instances. Object-oriented data stores may implement a garbage-collection algorithm similar to the one implemented for in-memory

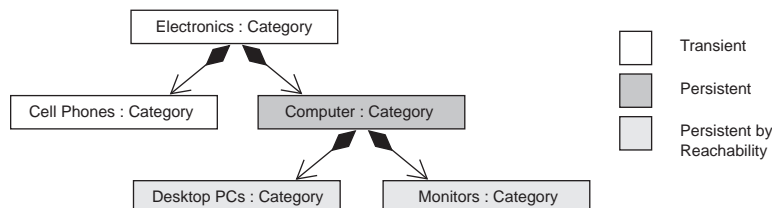


Figure 4.2
Persistence by reachability with a root persistent object

objects by the CLR, but this option isn't available in the ORM world; scanning all tables for unreferenced rows won't perform acceptably.

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.

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. But NHibernate lets you 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` doesn'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"` has NHibernate 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. There is a good reason for this design decision: it lets NHibernate applications 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 is shown in figure 4.3.

Now you 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"
    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 you create a new Category as a child category of Computer (see figure 4.4).

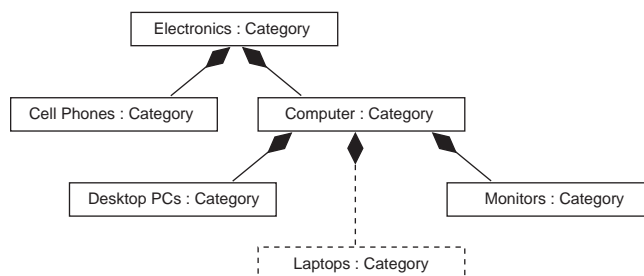


Figure 4.4 Adding a new Category to the object graph

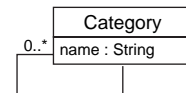


Figure 4.3 Category class with association to itself

You have several ways to create this new Laptops object and save it in the database. You can go back to the database and retrieve the Computer category to which the 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). You 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() ) {
    session.Save(laptops);
    session.Transaction.Commit();
}
```

NHibernate inspects the database identifier property of the parent category of laptops and correctly creates 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.

Because 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 you'd 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, you neither needed nor wanted transitive persistence for the ParentCategory association.

Why do you have cascading operations? You 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 you 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 you mapped the `ChildCategories` association with `cascade="save-update"`, you don't need to. The same code you 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() ) {
    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 you 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);
```

You add a new category as a child of the `Laptops` category and modify 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 reattaches/updates the three detached categories (`laptops`, `laptopAccessories`, and `laptopTabletPCs`) and saves the new child category (`laptopBags`).

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.

You 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() ) {
    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 by 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*

Because 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 assumes 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 an NHibernate `IInterceptor` and return true from `IInterceptor.IsUnsaved()` after checking the instance in your code.

The example domain model uses the primitive type `long` everywhere as the identifier property type. Because it isn't nullable, you have to use the following identifier mapping in all your 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; otherwise, it's the default value of the type (0 for numerical types); because you've chosen `long` as the identifier property type, you can omit the `unsaved-value` attribute in your auction application classes. Technically, NHibernate tries to guess the `unsaved-value` by instantiating an empty object and retrieving default property values from it.

Unsaved assigned identifiers

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 section 10.2. 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 feel for the NHibernate transitive persistence model.

Having focused on how to persist objects with NHibernate, we can now switch perspectives and focus on how you 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 automatically loads (or preloads) 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 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 and having NHibernate take care of mapping the ADO.NET result sets to graphs of persistent objects.

NOTE Using LINQ for NHibernate is another option available. This lets you specify your NHibernate queries using LINQ. At the time of writing, LINQ for NHibernate looks very promising despite the fact it's still a work in progress. We don't cover it in this book, but feel free to investigate further by visiting the NHContrib project website.

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 don't discuss each retrieval method in detail in this section; instead, we focus on the basic fetching strategies and how to tune NHibernate mapping files for the best default fetching performance for all methods. Before we look at the fetching strategies, we provide an overview of the retrieval methods. Note that we mention the NHibernate caching system, but we fully explore it in the next chapter.

Let's start with the simplest case: retrieving 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 here's the code 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:

```
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's 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 tries to load it. On the other hand, `Get()` never returns a proxy because 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. `Load()` may 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.

This behavior has an interesting application. Let's 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();
}
```

You first load a category. NHibernate doesn't 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 you link a new category to the proxy, and you 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!

Now, let's explore arbitrary queries, which are far more flexible than retrieving objects by identifier.

4.4.2 Introducing Hibernate Query Language

Hibernate Query Language (HQL) 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. JPA QL is a subset of HQL. HQL is easy to learn with a 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:

```
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`, you 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, you can provide one using `q.List(myEmptyList)`, and NHibernate will fill it. This is 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:

- Applying restrictions to properties of associated objects related by reference or held in collections (to navigate the object graph using query language).
- Retrieving 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*.
- Ordering the query's results.
- Paginating the results.
- Aggregating with `group by`, `having`, and aggregate functions like `sum`, `min`, and `max`.
- Performing outer joins when retrieving multiple objects per row.
- Calling user-defined SQL functions.
- Performing subqueries (nested queries).

We discuss all these features in chapter 8, 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", "Pierre Henri") );
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 approach, but it can be convenient for some applications. The following code snippet demonstrates an 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 8.

You now know the basic retrieval options in NHibernate. We focus on 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 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 (multiuser, distributed, web, and enterprise applications), object retrieval using many round trips to/from the database is unacceptable. 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 a 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.

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; 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. But it’s 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 a SQL outer join. Performance optimization in NHibernate often involves judicious use of eager fetching for particular transactions. 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 the 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 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 8. A minor caveat: You don't have to understand every option presented in this section immediately; 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; we 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, you 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 you retrieve a `Bid` from the database, the association property may hold an instance of an 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 an NHibernate proxy).

In order to delegate method (and property) invocations, these members need to be virtual. NHibernate 1.2 uses a validator that verifies that proxied entities have a default constructor which isn't private, that they aren't sealed, that all public methods and properties are virtual, and that there is no public field. It's possible to turn off this validator; 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). You can even choose any interface implemented by the `Item` class as the type of the proxy. To do so, declare it using the `proxy` attribute, instead of specifying `lazy="true"`:

```
<class name="Item" proxy="ItemInterface">
```

As soon as you 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's 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 6). 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"`.

If you wanted to re-enable eager fetching for the association, now that proxying is enabled, you would specify

```
<many-to-one name="item" class="Item" outer-join="true">
```

For a one-to-one association (discussed in more detail in chapter 7), lazy fetching is conceptually possible only when the associated object always exists. You 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 can't be null.

To enable batch fetching, you 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 an 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 follow:

- *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 you'll map the item's collection of bids with *lazy="true"*. This option is almost always used for collection mappings (although it's the default since NHibernate 1.2, we'll continue to write it to insist on it):

```
<set name="Bids" lazy="true">
  <key column="ITEM_ID" />
  <one-to-many class="Bid" />
</set>
```

You 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 five *Item* instances have 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 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 7). 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 must 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 you load a Category with this fetching strategy, NHibernate automatically fetches 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 also be fetched eagerly with the same SQL query. The `<many-to-many>` element lets you customize this behavior:

```
<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 now fetches all Items in a Category with a single outer join query when the Category is loaded. But 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 now discuss a global fetching strategy setting: the *maximum fetch depth*. This setting controls the number of outer-joined tables NHibernate uses 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 one-to-many; hence both associations are mapped with collection elements. If you 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. But 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 fetches 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 won't, 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 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 section 8.3.2.

But you may sometimes want 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). But 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.

You 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 section 9.1. But your first choice should be to fetch the complete required graph, 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's possible to tune object retrieval if performance problems are observed. 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. But 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; you need to understand how to analyze and tune your NHibernate applications so 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 3. 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 the cache-hit ratio inside the database? Get your DBA to help you with the performance evaluation; only she 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 use case again and see if any optimizations had side effects for others. With some experience, you'll be able to avoid negative effects and get it right the first time.

This optimization technique isn't practical for more than 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, you've started to think about performance issues, especially issues related to association fetching. The quickest way to fetch a graph of objects is to fetch it from the cache in memory, as we show 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. We discussed the three object states defined by NHibernate: persistent, detached, and transient. Objects transition between these states when you invoke methods of the `ISession` interface, or when you create and remove references from a graph of already persistent instances. This latter behavior is governed by the configurable cascade styles available in NHibernate's model for transitive persistence. This model lets you declare the cascading of operations (such as saving or deletion) on a per-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, 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*.

5

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, the CaveatEmptor application has to perform four tasks:

- 1 Mark the winning (highest amount) bid.
- 2 Charge the seller the cost of the auction.
- 3 Charge the successful bidder the price of the winning bid.
- 4 Notify the seller and the successful bidder.

What happens if you can't bill the auction costs because of a failure in the external credit card system? Your business requirements may state that either all listed actions must succeed or none must succeed. If so, you call these steps collectively a transaction or a 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 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.

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 a *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 may appear at first sight. For example, caching lets you 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. You'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.

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.

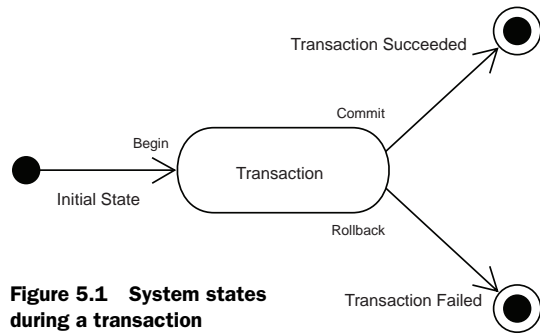


Figure 5.1 System states during a transaction

5.1.1 ADO.NET and Enterprise Services/COM+ transactions

Without Enterprise Services, the ADO.NET 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?

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. Section 10.3 explains how to make NHibernate and Enterprise Services transactions work together.

NHibernate communicates with the database via an ADO.NET `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 don't discuss this option, because its use is discouraged; instead, we 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. Listing 5.1 shows 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 is automatically enlisted in the current distributed transaction, or you have to do it manually if you've disabled automatic transaction enlistment.

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 disposes 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're executing a single query, you can let NHibernate manage the transaction.

It's critically important to make sure 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 application's responsibility.)

NOTE After committing a transaction, the NHibernate session replaces it with a new transaction. This means you should keep a reference to the transaction you're committing if you think you'll need it afterward. This is necessary if you need to call `transaction.WasCommitted.session.Transaction.WasCommitted` always returns false.

Here is another version showing in detail 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 here
        }
    }
    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 shouldn't use this example as a template in your own application, because you should 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 section 8.1.

NOTE You must be aware of one important aspect: 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.

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 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 needs to execute only one SQL `UPDATE`.

NHibernate flushes occur only at the following times:

- 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. But if changes are 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` to the property `session.FlushMode`. The flush modes are as follow:

- `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 open for the shortest amount of time possible. NHibernate is smart enough to open it only when really necessary (opening the session doesn't automatically open the connection). Since NHibernate 1.2, it's also possible to define when the database connection should be closed.

Currently, two options are available. They're 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's 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 tell the session to obtain a new connection when needed.

Obviously, these modes are activated only for connections opened by NHibernate. If you open a connection and send it to NHibernate, you're also responsible for closing this connection.

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`.

Because NHibernate 1.2.0 has some problems dealing with APIs like `System.Transactions`, you must use `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 isn't 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 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 *multiversion concurrency control*, which is generally considered more scalable. We discuss isolation assuming a locking model (most of our observations are also applicable to multiversion 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 required by 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 dangerous, because those changes may 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*—This is 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 wins*.
- *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 need not 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 define the various *transaction isolation levels* and see what problems they prevent.

ISOLATION LEVELS

The standard isolation levels are defined by the ANSI SQL standard. 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. But any transaction may read any row. 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. But 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; another mechanism must prevent 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 your application that can't be reproduced and that you'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 may want to skip this section and come back when it's time to make the decision about 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. What is a sensible database-isolation level to choose in an NHibernate application?

First, 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, because 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), which effectively force a serialized execution of operations in certain situations.

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. Thus read-committed isolation for all database transactions is 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 because 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 may 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 may not be using second-level caching for critical classes, or you may 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 an 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 follow (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 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 may be just a momentary lock, held while the item is being read, or it may 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. But 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`. Suppose you have a transaction that looks like this:

```
ITransaction tx = session.beginTransaction();
Category cat = session.Get<Category>(catId);
cat.Name = "New Name";
tx.Commit();
```

It's 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 loads 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. Here's an example:

```
Item item = ItemDAO.Load(1);
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 scenes, NHibernate executes a `SELECT` to make sure there is a database row with the identifier (and version, if present) of the detached object. It doesn't check all the columns. This isn't a problem when the version is present because it's 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, you should 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. 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 while the database is locking resources, then other transactions may be blocked for that entire duration! All this, of course, also applies to

an NHibernate `ITransaction`, because it's 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 *conversations*.

5.2 Working with conversations

Business processes, which may 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 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 now restate this definition more precisely. Most .NET applications include several examples of the following type of functionality:

- 1 Data is retrieved and displayed on the screen, requiring the first database transaction as data is read.
- 2 The user has an opportunity to view and then modify the data in his own time. (Of course, no database transaction need here.)
- 3 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* and *user transaction* because they're less vague and emphasize the transaction aspect from the user's point of view.

During these long user-based transactions, you can't rely on the database to enforce isolation or atomicity of concurrent conversations. Isolation becomes something your application needs to deal with explicitly—and may even require getting the user's input.

5.2.1 An example scenario

Let's look at an example that uses a conversation. In the 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 you 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. When the second user attempts to save her changes, she receives an error message saying “your updates were lost because someone else updated the record while you were editing it.” The user must start her edits again and hope she has more luck next time she clicks Save! This option is often called *optimistic locking*—the application optimistically assumes there won’t be problems, but it checks and reports if there are.
- *Merge conflicting updates*—The first modification is persisted. When the second user saves, he’s given the option of merging the records. This 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 the 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 you’re using NHibernate, the first option happens by default and 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 optimistic-locking options, then you must add appropriate code to your application. NHibernate can help you implement this using *managed versioning for optimistic locking*, also known as *optimistic offline lock*.

5.2.2 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 to SQL Server’s `TIMESTAMP` column and that database-driven concurrency features aren’t supported.

For NHibernate managed versioning, you must add a new property to your `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, because two concurrent transactions may both load and update the same item all in the same millisecond; in practice, this is unlikely to occur. But 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.

NOTE 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 increments 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 you add or remove a `Bid` from the collection of `bids`, the version of the `Item` will be updated. (Of course, you would make `Bid` an immutable class, because it doesn't make sense to modify bids.)

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 wouldn't contain the value 2, and the row wouldn't

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, you can 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, you can 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’s 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 isn’t incremented).

It’s also possible to implement optimistic locking without a version by writing

```
<class ... optimistic-lock="all">
```

In this case, NHibernate compares the states of all fields to find if any of them as changed. This feature works only for persistent objects; it can’t work for detached objects because NHibernate doesn’t 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 allows, for example, an administrator to change the name of a customer while another administrator changes her phone number at the same time.

It’s 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. But 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 choose one? And why is it called *optimistic*?

5.2.3 Optimistic and pessimistic locking compared

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 only blocks at the end of a unit of work and raises an error.

Both strategies have their places 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, using NHibernate to manage the lock table. Patterns for this can be found on the NHibernate website; but 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 talked about the NHibernate `ISession` not being the same as a transaction. An `ISession` has flexible scope, and you can use it in different ways with database and conversations. This means the *granularity* of an `ISession` is flexible; it can be any unit of work you want it to be.

5.2.4 Granularity of a session

To understand how you can use the NHibernate `ISession`, let's consider its relationship with transactions. Previously, we've 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`. Usually, you open a new `ISession` for each client request (for example, a web browser request) and begin a new `ITransaction`. After executing the business logic, you 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) 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 may, 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).

Suppose your conversation spans two client request/response cycles—for example, two HTTP requests in a web application. You can load the interesting objects in a first `ISession` and later reattach them to a new `ISession` after they've been

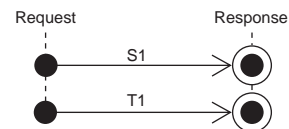


Figure 5.2 Using a one-to-one `ISession` and `ITtransaction` per request/response cycle

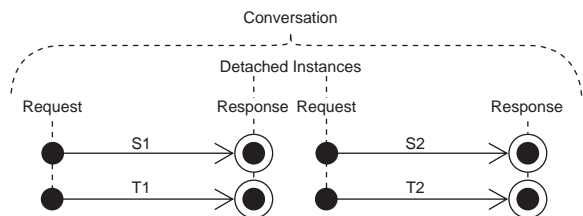


Figure 5.3 Implementing conversations with multiple `ISessions`, one for each request/response cycle

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 may 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, because the objects remain persistent in the context of the one long-running `ISession` (see figure 5.4). Of course, NHibernate is still responsible for performing optimistic locking.

A conversation keeps a reference to the session, although the session can be serialized, if required. The underlying ADO.NET connection must 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 section 10.2.

Finally, there is an important issue you may 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.

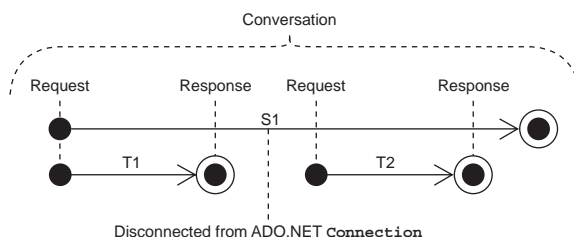


Figure 5.4 Implementing conversations with a long `ISession` using disconnection

5.2.5 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 includes 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).

Note that when you use `optimistic-lock="dirty"`, two concurrent conversations can update the same row as long as they change different columns; the result can be disastrous for the end user. If you have to use this feature, do it cautiously.

We 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; their validity isn't limited to NHibernate applications. This discussion gives you the background to understand why the NHibernate caching system is like it is. We 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 may 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 the disk of the server machine. 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 advantageous when it comes 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 used less often, 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.
- *Process scope*—Shared among many (possibly concurrent) units of work or transactions. Data in the process-scope cache is accessed by concurrently running transactions, obviously with implications on transaction isolation. A process-scope cache may store the persistent instances themselves in the cache, or it may 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, because reading and updating the cache may be only marginally faster than going straight to the database. You must take many parameters into account, so a number of tests and tunings may be required before you make a decision.

Persistence layers may provide multiple levels of caching. For example, a *cache miss* (a cache lookup for an item that isn't contained in the cache) at transaction scope may be followed by a lookup at process scope. If that fails, going to the database for the data may 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. If, during a transaction, the application attempts to retrieve the same object twice, the transaction-scope cache ensures that both look-ups 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 may be returned *by value*. Instead of storing and returning instances, the cache contains tuples of data. Each unit of work first retrieves a copy of the state from the cache (a tuple) and then uses 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 the object identity are no longer the same.

A cluster-scope cache always requires remote communication because it's 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 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're 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 and 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. The decisions to be made are whether to use the second-level cache, 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 among different units of work, multiple threads, or even multiple computers. This may have some nasty side effects on transaction isolation. We now discuss some critical considerations when you're choosing to use a process- or cluster-scope cache.

First, if more than one application is updating the database, then you shouldn't use process-scope caching, or you should use it only for data that changes rarely and may be safely refreshed by a cache expiry. This type of data occurs frequently in content-management applications but rarely in financial applications.

If you're designing your application to scale over several machines, you'll want to build it 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. 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. 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-) scope cache.

A full ORM solution lets you configure second-level caching separately for each class. Good candidate classes for caching are classes that represent

- Data that rarely changes
- Noncritical 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

But 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 are shown in figure 5.5.

The first-level cache is the `ISession`. 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 may 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

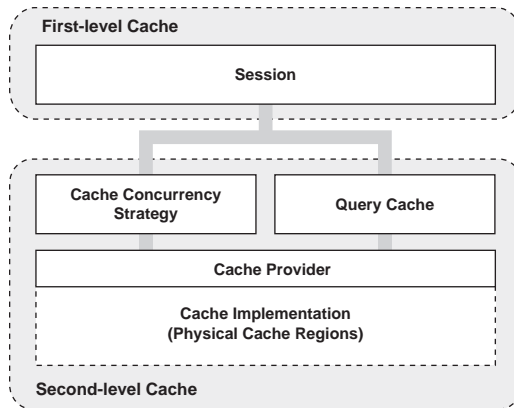


Figure 5.5 NHibernate's two-level cache architecture

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 set that integrates closely with the second-level cache. This is an optional feature. We discuss the query cache in chapter 8, because its usage is closely tied to the 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 important, 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. At most a single object represents 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.

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 is 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. This can be useful in several scenarios.

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 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 section 8.4.5, may be a better solution.

Note that eviction, like save or delete operations, can be automatically applied to associated objects. NHibernate evicts 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 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 (but the algorithm is much, much faster than .NET serialization).

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! 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 have to decide, for each persistent class, which cache concurrency strategy to use, if you want to enable the second-level cache.

Three built-in concurrency strategies are available, 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 nonclustered 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*—Suitable for data that 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 may 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*—Not intended for production use. It only caches in memory and can be set using its provider: `NHibernate.Cache.HashtableCacheProvider` (available in `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 library `NHibernate.Caches.SysCache.dll`. This provider should only be used with ASP.NET Web Applications.
- *Prevalence*—Makes it possible to use the underlying `Bamboo.Prevalence` implementation as a 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/>.

You'll learn about some distributed cache providers 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:

- 1 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.
- 2 Enable your preferred cache provider in the NHibernate configuration, and customize the provider-specific settings.

Let's add caching to the `CaveatEmptor` `Category` and `Item` classes.

5.3.3 Caching in practice

Remember that you don't have to explicitly enable the first-level cache. Let's declare caching policies and set up cache providers for the second-level cache in the `CaveatEmptor` application.

The `Category` has a small number of instances and is rarely updated, and instances are shared among many users, so it's a great candidate for use of the second-level cache. 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, you use the position `-1` because it must come before the other attributes).

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 you navigate to a `Category` or when you load a `Category` by identifier.

You use read-write instead of nonstrict-read-write because `Category` is a highly concurrent class, shared among many concurrent transactions, and it's clear that a read-committed isolation level is good enough. Nonstrict-read-write would probably be an acceptable alternative, because a small probability of inconsistency between the cache and database is acceptable (the category hierarchy has little financial significance).

This mapping is 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, you'll use a read-write concurrency strategy:

```

<class
    name="Category"
    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; make sure that you remove the item from the collection in the cache before deleting it.

A collection cache holds only the identifiers of the associated item instances. If you require the instances themselves to be cached, you must enable caching of the `Item` class. A read-write strategy is especially appropriate here. Your 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 you have to map the collection using read-write, because new bids may be made at any time (and it's critical that you 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, you 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.

Let's set the cache provider, expiration policies, and physical properties of the cache. You 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`.

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 you'll choose a cache provider.

SETTING UP A LOCAL CACHE PROVIDER

You 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, you choose SysCache as your second-level cache.

Now, you 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 that 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; 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 you can 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. You therefore define a high expiration value (10 hours) and give it a high priority so the categories stay in the cache as long as possible.

Bids, on the other hand, are small and immutable, but there are many of them; you must configure SysCache to carefully manage the cache memory consumption. You 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 five minutes or if the cache is full (because 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.

USING A DISTRIBUTED CACHE

SysCache and Prevalence are excellent cache providers if your application is deployed on a single machine. But enterprise applications supporting thousands of concurrent

users may require more computing power, and scaling your application may 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 to 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 your cache configuration, you can 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. You may be able to have each node in the cluster use a local instance of SysCache, and carefully choose sufficiently short expiration values.

But 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 following:

- *MemCache*—Released with NHibernate. It uses memcached, a distributed cache system available under Linux, so you can use it with Mono (or use the *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*—A commercial distributed cache provider. Its website is <http://www.alachisoft.com/ncache/>.
- *Microsoft Velocity*—A commercial distributed cache, currently in CTP2 (at technology preview stage).

We don't dig into the details of these distributed cache providers. Distributed caching is a complex topic; we recommend that you read some articles about this topic and test these providers.

Note that some distributed cache providers work only with some cache concurrency strategies. A nice trick can help you avoid checking your mapping files one by one. Instead of placing a `[Cache]` attribute in your entities or placing `<cache>` elements in your mapping files, you 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>
```

You enable read-only caching for `Bid` and read-write caching for the `Bids` collection in this example. But be aware of one important caveat: at the time of this writing,

NHibernate will run into a conflict if you also have `<cache>` elements in the mapping file for `Item`. You 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 may 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 may be better to set the NHibernate configuration option `hibernate.cache.use_minimal_puts` to `true`. When this setting is enabled, NHibernate adds an item to the cache only after checking to ensure the item isn't already 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 loads the cache provider and starts using the second-level cache only 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 `ISessionFactory` 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 evict only a particular collection role:

```
SessionFactory.Evict( typeof(Category) );
```

You'll rarely need these control mechanisms; use them with care, because they don't respect any transaction isolation semantics of the usage strategy.

5.4 Summary

This chapter was dedicated to transactions (fine-grained and coarse-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. But 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 delve into some of the more advanced capabilities of NHibernate. The next chapter will start by discussing some of the advanced NHibernate mapping concepts that will enable you to handle the most demanding persistence requirements.



Advanced mapping concepts

This chapter covers

- The NHibernate type system
- Custom mapping types
- Collection mappings
- One-to-one and many-to-many associations

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 these topics by turning to the more exotic collection and association mappings that allow you to handle trickier use cases. It's worth noting that these more exotic mappings should only be used with careful consideration; 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 advise you about when you may or may not want to use the advanced features as they're discussed.

Some of these features require you to have a more in-depth understanding of NHibernate's type system, particularly the distinction between entity and value types. That is where we begin.

6.1 Understanding the NHibernate type system

We first distinguished between entity and value types back in section 3.6.1. In order to give you a better understanding of the NHibernate type system, we elaborate 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’s possible (but less common) to create value types that are more complex. For example, a value type like an address can contain several fields.

How do you 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 don’t. The entity instances may therefore be in any of the three persistent lifecycle states: transient, detached, or persistent. But 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 apply to them, but never to value-type instances. To illustrate, consider figure 6.1.

`TotalAmount` is an instance of value type `Money`. Because value types are completely bound to their owning entities, `TotalAmount` is saved only when the `Order` is saved.

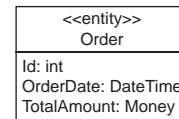


Figure 6.1 An order entity with a `TotalAmount` value type

6.1.1 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, the `Money` value type may have a property called `Currency` that is an association to a `Currency` entity, as shown in figure 6.2.

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 affects only the entity that owns it. For example, changing the `TotalAmount` of one `Order` can’t 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 now look at how the relational model sees value types and entities, and how NHibernate bridges the gap.

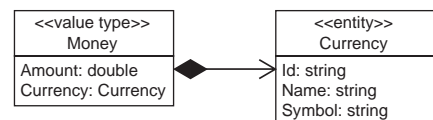


Figure 6.2 The `Money` value type with an association to a `Currency` entity

6.1.2 *Bridging from objects to database*

You may be aware that a database architect sees the world of value types and entities slightly differently from 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.

If all tables represent entities in the database, does that mean you have to map all tables to entities in your .NET domain model? What about those value types you want in your 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 you don't end up with an unwanted entity in your 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. But over time, we've become suspicious of them; these "hidden" entities often end up needing exposure in applications as business requirements evolve. The many-to-many association table, for example, often has columns added as the application matures, so the relationship itself becomes an entity. You may 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.

6.1.3 *Mapping types*

So far, we've discussed the differences between value types and entities as seen from the object-oriented and relational-database perspectives. You know that mapping entities is straightforward—entity classes are 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 this example, imagine that the `Email` field is a .NET string, and the `EMAIL` column is a SQL `varchar`. You want to tell NHibernate how to carry out this conversion, which is where NHibernate mapping types come in. In this case, you specified the mapping type `"String"`, which 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 `DateTime`.

6.1.4 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. But 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 have to define your own custom value types. We get to that topic a little later in this chapter.

We 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 6.1 map .NET primitive types to appropriate DbTypes.

You've probably noticed that your database doesn't support some of the DbTypes listed in table 6.1. But ADO.NET provides a partial abstraction of vendor-specific SQL

Table 6.1 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 —one character
AnsiChar	System.Char	DbType.AnsiStringFixedLength —one character
Boolean	System.Boolean	DbType.Boolean
Guid	System.Guid	DbType.Guid
PersistentEnum	System.Enum (an enumeration)	DbType for the underlying value
TrueFalse	System.Boolean	DbType.AnsiStringFixedLength —either 'T' or 'F'
YesNo	System.Boolean	DbType.AnsiStringFixedLength —either 'Y' or 'N'

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	Mapping type	Additional name
Binary	binary	Int16	short
Boolean	boolean	Int32	int
Byte	byte	Int32	integer
Character	character	Int64	long
CultureInfo	locale	Single	float
DateTime	datetime	String	string
Decimal	big_decimal	TrueFalse	true_false
Double	double	Type	class
Guid	guid	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.

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

Table 6.3 Date and time types

Mapping type	.NET type	System.Data.DbType
DateTime	System.DateTime	DbType.DateTime—ignores milliseconds
Ticks	System.DateTime	DbType.Int64
TimeSpan	System.TimeSpan	DbType.Int64
Timestamp	System.DateTime	DbType.DateTime—as specific as the database supports

`System.DateTime` or `System.TimeSpan`. Because they have different purposes, the choice should be easy.

OBJECT MAPPING TYPES

All of the .NET types in tables 6.1 and 6.3 are value types (derived from `System.ValueType`). This means 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 types

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.

`BinaryBlob` and `StringClob` are mainly supported by SQL Server. They can have a large size and are fully loaded in memory. This can be a performance killer if you use

Table 6.5 Binary and large object types

Mapping type	.NET type	System.Data.DbType
Binary	System.Byte[]	DbType.Binary
BinaryBlob	System.Byte[]	DbType.Binary
StringClob	System.String	DbType.String
Serializable	Any System.Object marked with <code>SerializableAttribute</code>	DbType.Binary

these types to store large objects—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.

Certainly, `<property>` isn't the only NHibernate mapping element that has a type attribute.

Table 6.6 Other CLR-related types

Mapping type	.NET type	System.Data.DbType
CultureInfo	System.Globalization.CultureInfo	DbType.String —five characters for culture
Type	System.Type	DbType.String holding the Assembly Qualified Name

6.1.5 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.)

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 you 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. You 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();
```

These constants are also useful for programmatic manipulation of the NHibernate mapping metamodel, 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 look this next and explain how the mapping type system is 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, because new type conversions would be impossible. In order to avoid that, NHibernate provides a powerful feature called *custom mapping types*.

NHibernate provides two user-friendly interfaces that applications can use when defining new mapping types. The first is `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.

The `Bid` class defines an `Amount` property, and the `Item` class defines an `InitialPrice` property; both are monetary values. So far, you've only used a `System.Double` to represent the value, mapped with `Double` to a single `DbType.Double` column.

Suppose you want to support multiple currencies in the auction application and that you have to refactor the existing domain model for this change. One way to implement this change is to add new properties to `Bid` and `Item`: `AmountCurrency` and `InitialPriceCurrency`. You can then map these new properties to additional `VARCHAR` columns with the built-in `String` mapping type. If you have currency stored in 100 places, this is a lot of changes. We hope you never use this approach!

CREATING AN IMPLEMENTATION OF IUserType

Instead, you 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() { ... }
}

```

You also make life simpler by making `MonetaryAmount` an immutable class, meaning it can't be changed after it's instantiated. You have to implement `Equals()` and `GetHashCode()` to complete the class—but there is nothing special to consider here aside from the facts that they must be consistent, and `GetHashCode()` should return mostly unique numbers.

You'll use this new `MonetaryAmount` to replace the `Double`, as defined on the `InitialPrice` property for `Item`. You'll benefit from using this new class in other places, such as the `Bid.Amount`.

The next challenge is in mapping the new `MonetaryAmount` properties to the database. Suppose you're working with a legacy database that contains all monetary amounts in USD. The new class means the application code is no longer restricted to a single currency, but it will take time for the database team to make the changes. Until this happens, you'd like to store just the `Amount` property of `MonetaryAmount` to the database. Because you can't store the currency yet, you'll convert all `Amounts` to USD before you save them and from USD when you load them.

The first step in handling this is to tell NHibernate how to handle the `MonetaryAmount` type. To do so, you create a `MonetaryAmountUserType` class that implements the NHibernate interface `IUserType`. The 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
                        ){6
    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) {7
    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)
{
    //...8
}
}

```

The `SqlTypes` property tells NHibernate what SQL column types to use for DDL schema generation 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 the legacy data model has only a single `Double`.

`ReturnedType` tells NHibernate what .NET type is mapped by this `IUserType` ².

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 ⁴. Because `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` property ⁵ tells NHibernate that this type is immutable.

The `NullSafeGet()` method 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.

The `NullSafeSet()` method writes the property value to the ADO.NET `IDbCommand` ⑦. 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 ⑧. If you were to implement it, it would have code that converts between various currencies.

You can map the `InitialPrice` property of `Item` 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` can 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 can perform validation, it can read and write data to and from an Active Directory, or it can 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, you'd represent both the amount and currency of the monetary amounts in the database, so you're not limited to storing USD. You can use an `IUserType` for this, but it's limited; if an `IUserType` is mapped with more than one property, you can't use those properties in your HQL or Criteria queries. The NHibernate query engine won't know anything about the individual properties of `MonetaryAmount`. You can access the properties in your C# code (`MonetaryAmount` is 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, you can use the `ICompositeUserType` interface. This interface exposes the properties of the `MonetaryAmount` to NHibernate.

CREATING AN IMPLEMENTATION OF ICOMPOSITEUSERTYPE

To demonstrate the flexibility of custom mapping types, you won't have to change the `MonetaryAmount` domain model class at all—you'll 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 {           ❶
    get { return new string[] { "Value", "Currency" }; }
}
public NHibernate.Type.IType[] PropertyTypes {           ❷
    get { return new NHibernate.Type.IType[] {
        NHibernateUtil.Double, NHibernateUtil.String }; }
}
public object GetPropertyValue(object component, int property) {           ❸
    MonetaryAmount amount = (MonetaryAmount) component;
    if (property == 0)
        return amount.Value;
    else
        return amount.Currency;
}
public void SetPropertyValue(object comp, int property,
    object value) {           ❹
    throw new Exception("Immutable!");
}
public object Assemble(object cached,           ❺
    NHibernate.Engine.ISessionImplementor session, object owner) {
    return cached;
}
public object Disassemble(object value,           ❻
    NHibernate.Engine.ISessionImplementor session) {
    return value;
}
}

```

The implementation of `ICompositeUserType` has its own properties, defined by `PropertyNames` ❶. Similarly, the properties each have their own type, as defined by `PropertyTypes` ❷.

The `GetPropertyValue()` method ❸ returns the value of an individual property of the `MonetaryAmount`. Because `MonetaryAmount` is immutable, you can't set property values individually ❹. This isn't a problem because this method is optional.

The `Assemble()` method is called when an instance of the type is read from the second-level cache ⑤. The `Disassemble()` method is called when an instance of the type is written to the second-level cache 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 you 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, you 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, you've expanded the buffer between the .NET object model and the SQL database schema with your custom composite type. Both representations can now handle changes more robustly.

If implementing custom types seems complex, relax; you'll 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. The decision to use a custom mapping type is often a matter of taste.

You can use a few more interfaces to implement custom types; they're introduced in the next section.

OTHER INTERFACES TO CREATE CUSTOM MAPPING TYPES

You may find that the interfaces `IUserType` and `ICompositeUserType` don't let you easily add more features to your custom types. In this case, you'll need to use some of the other interfaces that are in the `NHibernate.UserTypes` namespace: `IParameterizedType`, `IEnhancedUserType`, `INullableUserType`, and `IUserCollectionType`.

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's 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 the 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 an identifier or a discriminator type. To create a type that can be used as version, you must implement the `IUserVersionType` interface.

The `INullableUserType` interface lets you interpret non-null values in a property as null in the database. When you use `dynamic-insert` or `dynamic-update`, fields identified as null aren't 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 others because it's meant to implement user-defined collection types: `IUserCollectionType`. For more details, see the implementation `NHibernate.Test.UserCollection.MyListType` in the NHibernate source code.

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 couldn't be null; but this is no longer the case in .NET 2.0. Let's say you 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't be null. And you don't want to use some magic value to represent the null state. With .NET 2.0 (and 3.5), you can write

```
public class User
{
    //...
    private DateTime? dismissDate;
    public DateTime? DismissDate
    {
        get { return dismissDate; }
        set { dismissDate = value; }
    }
    //...
}
```

You omit other properties and methods because you focus on the nullable property. 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 that allow primitive types to be null. For the previous case, you can use the `Nullables.NullableDateTime` class:

```
using Nullables;
[Class]
public class User {
    //...
    private NullableDateTime dismissDate;
    [Property]
    public NullableDateTime DismissDate
```

```

    {
        get { return dismissDate; }
        set { dismissDate = value; }
    }
    //...
}

```

The mapping is straightforward:

```

<class name="Example.Person, Example">
    ...
    <property name="DateOfBirth"
        type="Nullables.NHibernate.NullableDateTimeType,
            Nullables.NHibernate" />
</class>

```

It's 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 when you use the `NHibernate.Mapping.Attributes` library, this operation is automatic; that's why you 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 the `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 the `CaveatEmp` application, users can give comments about other users. Instead of using a simple `int` property for the rating, you create an enumeration:

```

public enum Rating {
    Excellent,
    Ok,
    Low
}

```

You then use this type for the `Rating` property of the `Comment` class. In the database, ratings are represented as the type of the underlying value. In this case (and by default), it's `Int32`. And that's all you have to do. You may specify `type="Rating"` in your mapping, but it's optional; `NHibernate` can use reflection to find this.

One problem you may 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. You have to use a bind parameter and set the

rating value for the comparison dynamically (which is what you 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));
```

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 the 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're 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` can have a collection of `Bids` in the `CaveatEmpor` application. Collections aren't limited to entity types, and 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 strings 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'll 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 your 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 your system. In this case, it's reasonable to model the image as a value type. `Item` has a collection of images that `NHibernate` considers to be part of the `Item`, without their own persistence lifecycle. In this example scenario, let's assume that images are stored as files on the filesystem rather than as BLOBs in the database, and that you store filenames in the database to record what images each `Item` has. We 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`, you 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>
```

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, you have two entities. But NHibernate hides this fact so you 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 6.3 for a table schema example.

It doesn't seem likely that you would allow the user to attach the same image more than once, but suppose you did. What kind of mapping would be appropriate then?

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 6.3 Table structure and example data for a collection of strings

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. You use an `<idbag>` mapping to attach a surrogate key column to the collection table, much like the synthetic identifiers you 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 ISet 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 6.4.

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 6.4 Table structure using a bag with a surrogate primary key

If you're wondering why you use `<idbag>` rather than `<bag>`, bear in mind that we'll be discussing bags shortly. First, we'll discuss another common approach to storing 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 you 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. (You don't have to change the Item class; the types you 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 6.5.

Alternatively, you could use a .NET array instead of a list. NHibernate supports this usage, and the details of an array mapping are virtually identical to those of a list. But we strongly recommend against the use of arrays, because arrays can't be lazily initialized (there is no way to proxy an array at the CLR level). Here is the mapping:

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 6.5 Tables for a list with positional elements

```
<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 your 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 6.6 for a graphical view of the tables.

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 6.6 Tables for a map, using strings as indexes and elements

This Map is unordered. What if you want to always sort your 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 a SQL query with an `order by` clause.

Let's make the 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"`, you 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. Here's an 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 maps use `System.Collections.SortedList` in their 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 may 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">
  <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 a SQL order by clause. In this case, you 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; use this functionality carefully because it doesn't perform well with large numbers of elements.

In a real system, it's likely that you'll need to keep more than just the image name and filename; you'll probably need to create an `Image` class to store this extra information. Of course, you could map `Image` as an entity class; but because we've already concluded that this isn't absolutely necessary, let's see how much further you 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 you can map collections of components.

6.2.2 Collections of components

The `Image` class defines the properties `Name`, `Filename`, `SizeX`, and `SizeY`. It has a single association, with its parent `Item` class, as shown in figure 6.7.

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 an 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`.

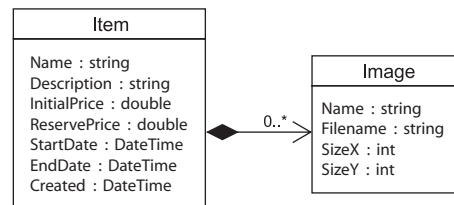


Figure 6.7 Collection of `Image` components in `Item`

WRITING THE COMPONENT CLASS

First, you implement the `Image` class. This is a POJO, with nothing special to consider. As you know from chapter 4, component classes don't have an identifier property. But you 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; but 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 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 can 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 the instances apart despite the fact that 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`. Because these columns will all appear in a composite primary key, they can't 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 lets you get around the not-null restriction.

BIDIRECTIONAL NAVIGATION

So far, the association from `Item` to `Image` is unidirectional. If you decide to make it bidirectional, you'll give the `Image` class a property named `Item` that is a reference back to the owning `Item`. In the mapping file, you 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="SIZE_X" not-null="true"/>
    <property name="SizeY" column="SIZE_Y" 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.

You still have the issue of having to declare all properties as `not-null`, and it would be nice if you could avoid this. Let's look at how you 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="SIZE_X"/>
    <property name="SizeY" column="SIZE_Y"/>
  </composite-element>
</idbag>

```

This time, the primary key is the `ITEM_IMAGE_ID` column. NHibernate now doesn't require that you must implement `Equals()` and `GetHashCode()`, nor do you need to declare the properties with `not-null="true"`. They may be nullable in the case of an `idbag`, as shown in figure 6.8.

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.

You can even remove the `Name` property from the `Image` class and again use the image name as the key of a map:

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 6.8 Collection of `Image` components using a bag with a surrogate key

```

<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="SIZE_X"/>
    <property name="SizeY" column="SIZE_Y"/>
  </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. A composite element with a many-to-one association is useful, and we come back to this kind of mapping later in this chapter.

We're finally finished with value types, and we hope you have an in-depth understanding of what is possible and where you can be able to make use of them. Next, we look at advanced entity-association mapping techniques. The simple parent/child association you 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 are needed only in special cases. But having an awareness of the available techniques will 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. 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 you'll rarely use a many-to-many mapping in your 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 the sample model. Component mappings are usually

the simplest way to represent one-to-one relationships, because 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 you 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, you 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, because *many* doesn't seem to be a good description of either end of a one-to-one association! But from NHibernate's point of view, there isn't much difference between the two kinds of foreign-key associations. You 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 you create an association from a persistent User. The cascade="all" cascade would also make sense for this association, because deletion of the User should result in deletion of the Address.

The database schema still allows duplicate values in the BILLING_ADDRESS_ID column of the USER table, so two users can have a reference to the same address. To make this association truly one-to-one, you 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 in figure 6.9.

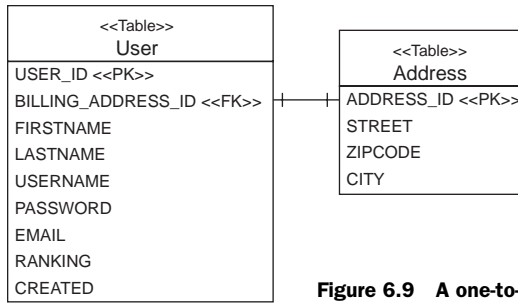


Figure 6.9 A one-to-one association with an extra foreign-key column

But what if you want this association to be navigable from Address to User in .NET? To achieve this, you add a property named User that points to the Address class, and map it like so in your 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, you create the association between the two objects as follows:

```

Address address = new Address();
address.Street = "73 Nowhere Street";
address.City = "Pretoria";
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, you also have to map the HomeAddress property of User. This is easy enough; you 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, you can't make both the BillingAddress and HomeAddress associations bidirectional, because you don't know if a particular address is a billing address or a home address. More specifically, you'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. You 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 exactly one instance of `Address` exists 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 you try to solve this problem, let's see how you 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 you can no longer map both the billing and home address—you can map 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, you must ensure that newly saved instances of `Address` are assigned the same identifier value as their `User`. You use a special NHibernate identifier-generation strategy called `foreign`:

```
<class name="Address" table="ADDRESS" lazy="false">
    <id name="Id" column="ADDRESS_ID">
        <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 you 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 6.10.

The code to create the object association is unchanged for a primary-key association; it's the same code you used earlier for the many-to-one mapping style.

Just one remaining entity association multiplicity remains for us to discuss: many-to-many.

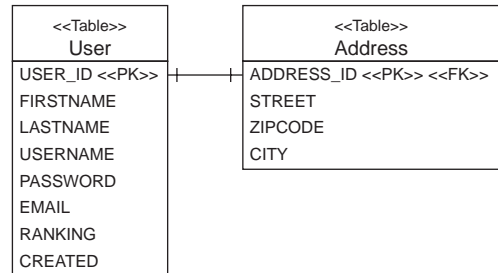


Figure 6.10 The tables for a one-to-one association with shared primary-key values

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 6.11.

As we explained earlier in this section, we avoid the use of many-to-many associations because there is almost always other information that must be attached to the links between associated instances; 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.

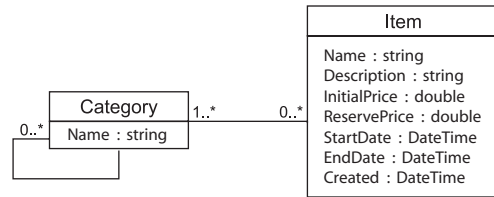


Figure 6.11 A many-to-many valued association between Category and Item

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, you 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 6.12.

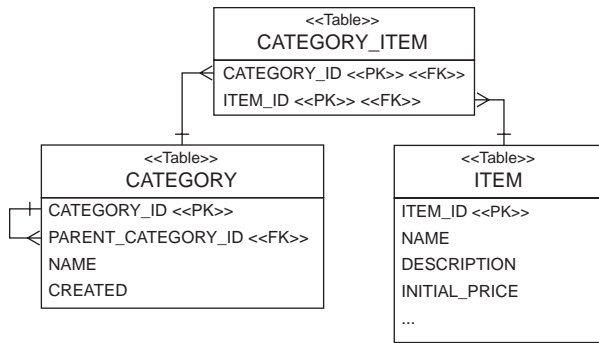


Figure 6.12 Many-to-many entity association mapped to an association table

You 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**; duplicate links are therefore allowed (the same *Item* can be added to a particular *Category* twice).

Other variations you can use are the indexed map and 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>

```

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 you mapped a bidirectional one-to-many association in section 3.6, we explained why one end of the association must be mapped with `inverse="true"`. Feel free to review that section, because it's 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, you may 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 may 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>
```

You can reuse this mapping for the `Category` end of the bidirectional association. You 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>
```

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 suits your needs well. Note that `cascade="all"`, `cascade="delete"`, and `cascade="delete-orphan"` are also valid options.

"all-delete-orphans" aren't meaningful for many-to-many associations, because an instance with potentially many parents shouldn't be deleted when just one parent is deleted.

Another thing to consider is the kinds of collections that 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 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, because 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 you need to record some information each time you add an `Item` to a `Category`. For example, you may need to store the date and the name of the user who added the item to this category. You use a C# class to represent this information:

```
public class CategorizedItem {
  private string username;
  private DateTime dateAdded;
  private Item item;
  private Category category;
  //...
}
```

This code omits the properties and `Equals()` and `GetHashCode()` methods, but they would be necessary for this component class.

You map the `Items` collection on `Category` as shown next. 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>
```

You use the `<many-to-one>` element to declare the association to `Item`, and you 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 you 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 6.13.

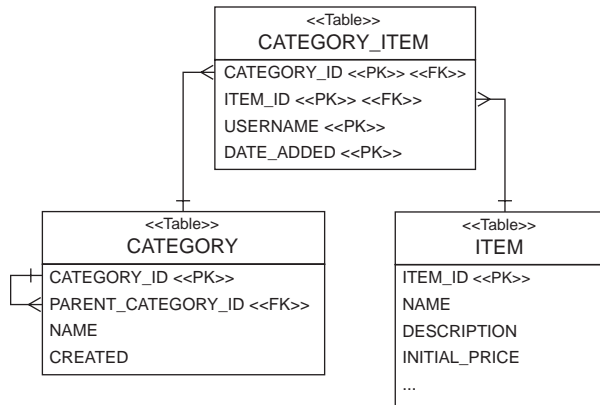


Figure 6.13 Many-to-many entity association table using a component

Rather than mapping just the `Username`, you may want to keep an actual reference to the `User` object. In this case, you 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>

```

This is a fairly exotic beast! If you find yourself with a mapping like this, you should ask whether it may 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. You 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`; you 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 the example, first you 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>

```

You rename the `<set>` element to `<bag>`, making no other changes. Note 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 `ILists` 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 ignores the index of the elements and doesn't update the position column.

But if your parent/child relationship is unidirectional only where navigation is possible only from parent to child, you can use an indexed collection type because the "many" end is no longer inverse. Good uses for unidirectional one-to-many associations are uncommon in practice, and you don't have one in the auction application. You may remember that you started with the `Item` and `Bid` mapping in chapter 4, making it first unidirectional, but you 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, you model this as a collection named `Items` in the `Category` class; you don't have to change anything if you don't use an indexed collection. If `Items` is implemented as an `ISet`, you 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 6.14.

The other side of the association, the `Item` class, has no mapping reference to `Category`. You can now also use an indexed collection in the `Category`—for example, after you change the `Items` property to `List`:

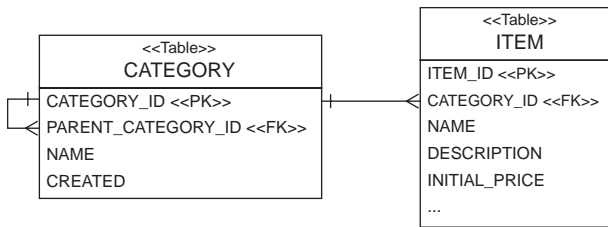


Figure 6.14 A standard one-to-many association using a foreign-key column

```

<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` table must be nullable. An `Item` could be saved without knowing anything about a `Category`—it’s a standalone entity! This is a consistent model and mapping, and you may 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? What you want is polymorphic behavior; 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 first consider a many-to-one association to a class that may 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 guide you through setting up polymorphic collections and then 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 you don't have many `BillingDetails` per `User`, but only one, as shown in figure 6.15.

The user needs a unidirectional association to some `BillingDetails`, which can be `CreditCard` details or `BankAccount` details. You 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 because `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—you 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 you navigate the association in a second transaction, NHibernate automatically retrieves the `CreditCard` instance:

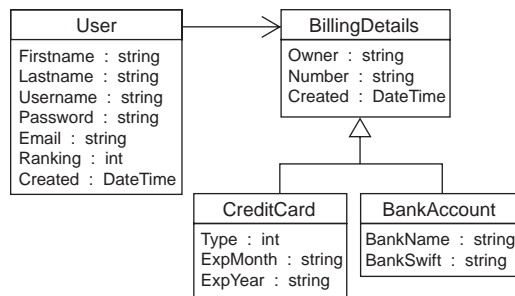


Figure 6.15 The user has only one billing information object.

```

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, in `user.BillingDetails.Pay(paymentAmount)`, the call is against the appropriate subclass.

You must watch out for one thing: if `BillingDetails` was mapped with `lazy="true"`, NHibernate would proxy the `BillingDetails` association. In this case, you wouldn't be able to perform a typecast to the concrete class `CreditCard` at run-time, 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 `bd` is a `CreditCard`; all it knows is that `bd` 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)
    .UniqueResult();
CreditCard cc = (CreditCard) user.BillingDetails;
expiryDate = cc.ExpiryDate;

```

`BillingDetails` is fetched eagerly in this case, 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 the Cave-atEmptor application. If `User` owns many `BillingDetails`, you use a bidirectional one-to-many. In `BillingDetails`, you have the following:

```
<many-to-one name="User"
    class="User"
    column="USER_ID"/>
```

In the `Users` mapping, you 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, the `user.AddBillingDetails(cc)` function ensures that the association is set at both ends by calling `BillingDetails.Add(cc)` and `cc.User=this`.

You 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(...)` calls 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` isn't mentioned explicitly in the mapping file, but only in the C# definition of the subclasses.

6.4.3 *Polymorphic associations and table-per-concrete-class*

In section 3.8.1, 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 you 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. You have CREDIT_CARD and BANK_ACCOUNT tables but no BILLING_DETAILS table. You need two pieces of information in the USER table to uniquely identify the associated CreditCard or BankAccount:

- The name of the table in which the associated instance resides
- 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. You use an 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. You can use any value you like as a type discriminator. For example, you 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>
```

The `<meta-value>` elements are optional; if you omit them, NHibernate uses the fully qualified names of the classes. An example of this table structure is shown in figure 6.16.

Here is the first major problem with this kind of association: you can't add a foreign-key constraint to the BILLING_DETAILS_ID column, because some values refer to

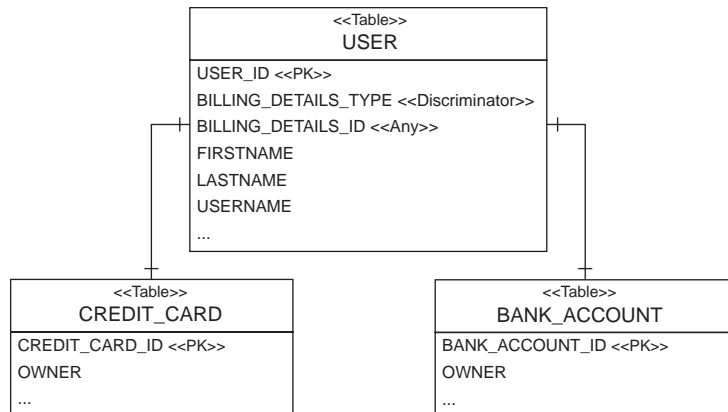


Figure 6.16 Using a discriminator column with an <any> association

the **BANK_ACCOUNT** table and others to the **CREDIT_CARD** table. You need to come up with some other way to ensure integrity. A database trigger may 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. You 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 and gives more detailed information that will let you build more efficient and flexible queries. We'll also cover some of the more advanced topics that surround object retrieval, such as report queries, fetching associations, and caching.



Retrieving objects efficiently

This chapter covers

- NHibernate query features
- HQL, criteria, and native SQL
- Advanced, reporting, and dynamic queries
- Runtime fetching and query optimization

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 may be concerned that ORM will take away some of the expressiveness and flexibility 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 your database's native SQL dialect.

In 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));
```

This chapter covers in-depth techniques using all three methods. You can also use this chapter as a reference; some sections are written in a less verbose style but show many small code examples for different use cases.

Before we continue, we briefly introduce another method of querying NHibernate: LINQ-to-NHibernate. This allows you to write LINQ queries to query NHibernate like this:

```
from cat in session.Linq<Category>()
where cat.Name.StartsWith("Laptop")
select cat
```

LINQ-to-NHibernate is an exciting and welcome addition. Unfortunately, at the time of writing it is still very much in Beta, so we can't cover it in detail here. If you want to try using LINQ-to-NHibernate, you can find the source code in the NHContrib project at <http://nhcontrib.wiki.sourceforge.net/>. We recommend downloading the source code and examining the unit tests to see up-to-date examples of how it can be used.

Let's continue with our investigations into HQL, the Criteria API, and direct SQL queries. We start our exploration by showing you how queries are executed with NHibernate. Rather than focus on the queries themselves, we focus on the various techniques for creating and running queries using NHibernate. Following that, we 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 can 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 an HQL query as follows:

```
IQuery hqlQuery = session.CreateQuery("from User");
```

This query is now set up to fetch all user objects in the databases. You can 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'll learn more about running SQL queries in section 8.5.4. Finally, here's how you 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 back a list of objects. Notice that the root entity type you want the query to return is specified as `User`. We study criteria queries in detail later.

A note about CaveatEmptor

Some queries in this chapter won't work with the `CaveatEmptor` code that accompanies this book. This is because certain techniques illustrated here require variations in the way classes and their mappings are defined.

Now we 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. 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 the next or previous pages by clicking appropriate links on the page. When writing data access code for this scenario, you need 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 here:

```
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 you wanted to get some results for the next page?

```
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 tenth object, you 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 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 technique:

```

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 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 you've created queries and set up pagination, we look at how you get 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 you want only a single instance to be returned. For example, if you want to find the highest bid, you can get that instance by reading it from the result list by index: result[0]. Alternatively, you can 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 your query returns only 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. You can use this technique to optimize loading in specific cases, as discussed in section 7.7.

Why not use `ISession.Find()` instead of `IQuery.List()`?

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()`.

But the 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 when constructing queries is binding parameters. The `IQuery` interface lets you achieve this in a flexible manner, as we discuss next.

7.1.2 Binding parameters

Allowing developers to bind values to queries is an important feature for any data access library because it permits you to construct queries that are both maintainable and secure. We demonstrate the types of parameter binding available in NHibernate; but first, let's 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 plainly and simply *bad*! You may know why: it can potentially leave your application open to SQL injection attacks. In such an attack, a malicious user attempts to trick your application into running the user's own SQL against the database, in order to cause damage or circumnavigate application security. If that user typed this `searchString`

```
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 the application isn't checking the values passed in from the user interface. Because of this, the quote characters aren't escaped, and users can inject their own SQL. Users may 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, you can easily avoid this problem by using parameters. With parameters, your query may look like this:

```
string queryString =
    "from Items I where i.Description like :searchString"
```

When you use parameters, queries and parameters are sent to the database separately, so the database can ensure they're dealt with securely and efficiently.

Another reason to use parameters is that they help NHibernate be more efficient. NHibernate keeps track of the queries you execute; when parameters are used, it needs to keep only one copy of the query in memory, even if the query is run thousands of times with different parameters each time.

Now you understand the importance of parameters. How do you use them in your NHibernate queries? There are two approaches to parameter binding: named parameters and positional parameters. We discuss these in turn.

USING NAMED PARAMETERS

Using named parameters, you can rewrite the earlier query as follows:

```
string queryString =
    "from Item item where item.Description like :searchString";
```

The colon followed by a parameter name indicates a named parameter. Then you 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, you use the `SetString()` method of the `IQuery` interface to bind it to the named parameter (`searchString`).

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();
```

This code is cleaner, much safer, and performs better, because a single compiled SQL statement can be reused if only bind parameters change.

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 that uses named parameters, but it's also much more vulnerable to breakage if you change the query string slightly:

```
string queryString = @"from Item item
    where item.Date > ?
    and item.Description like ?";
```

Every change of the bind parameters' positions requires a change to the parameter-binding code. This leads to fragile and maintenance-intensive code. We recommend that you avoid positional parameters:

```
string userSearch =
    @"from User u where u.Username like :searchString
    or u.Email like :searchString";
IList result = session.CreateQuery(userSearch)
    .SetString("searchString", searchString)
    .List();
```

Notice how the named parameter may appear multiple times in the query string.

BINDING ARBITRARY ARGUMENTS

You'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();
```

In addition, a generic method 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 may 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 you have a POCO with `Seller` and `Description` properties, you can use the `SetProperties()` method to bind the query parameters. For example, you can 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";
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, because 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();
```

But the result of this code is almost certainly not what you intended. The resulting SQL will contain a comparison like `username = null`, which always evaluates to null in SQL ternary logic. Instead, you 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 begin considering complex queries that must be split over multiple lines, it starts to get unwieldy.

7.1.3 *Using named queries*

We don't like to see HQL string literals scattered all over C# code unless they're necessary. NHibernate lets you store query strings outside of your code, a technique that is called *named queries*. This approach 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. You use the name of the query 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, you execute the named query `FindItemsByDescription` after binding a string argument to a named parameter. The named query is defined in mapping metadata, such as 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 may 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 may 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 handcoded 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's 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 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 you can also use this feature to rename SQL functions.

We've now wrapped up our discussion of creating and running queries. It's time to focus on the queries themselves. The next section covers HQL, starting with simple queries and moving on to 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.

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
```

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), set up the NHibernate configuration, and then view the result of HQL queries you type interactively. More details are provided in section 7.7.4.

Using the `ICriteria` interface, it looks like this:

```
ICriteria c = session.CreateCriteria(typeof(Bid));
```

Both generate the following SQL behind the scenes:

```
select B.BID_ID, B.AMOUNT, B.ITEM_ID, B.CREATED from BID B
```

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 like the temporary variable declaration in the following C# code:

```
for ( int i = 0; i < allQueriedBids.Count; i++ ) {
    Bid bid = (Bid) allQueriedBids[i];
    //...
}
```

You assign the alias `bid` to queried instances of the `Bid` class, allowing you 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). 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 You never write HQL keywords in uppercase; you never write SQL keywords in uppercase either. It looks ugly and antiquated—most modern terminals can display both uppercase and lowercase characters. 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 you'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've 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. In this case, the concrete objects are of the subtypes of `BillingDetails`: `CreditCard` and `BankAccount`. If you only want instances of a particular subclass, you can 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
```

This technique also works for interfaces. The following query returns all serializable persistent objects (those implementing the interface `ISerializable`):

```
from System.ISerializable
```

Criteria queries also support polymorphism:

```
session.CreateCriteria(typeof(BillingDetails)).List();
```

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

You usually don't want to retrieve all instances of a class when you run a query. Instead, you want to express some constraints on the property values of your objects, so only a subset of objects is retrieved. This is called *restriction*, and in both HQL and SQL, you achieve it 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 you use an object-oriented notion: just as in C#, `u.Email` may not be abbreviated to plain `Email`.

For a criteria query, you 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();
```

You 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, you'll write this 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 as follows:

```
select U.USER_ID, U.FIRSTNAME, U.LASTNAME, U.USERNAME, U.EMAIL
from USER U
where U.EMAIL = 'foo@hibernate.org'
```

It's common to have a restriction that should always be used; most of the time, it's used to ignore deprecated data. You may, for example, have an `Active` property and write the following:

```
select User u where u.Email = 'foo@hibernate.org' and u.Active = 1
```

But this is dangerous, because you may forget the restriction; a better solution, in this case, is to change your mapping:

```
<class name="User" where="ACTIVE=1">
```

Now, you can write

```
from User u where u.Email = 'foo@hibernate.org'
```

This query generates the following SQL query:

```
select U.USER_ID, U.FIRSTNAME, U.LASTNAME, U.USERNAME, U.EMAIL
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 contain only users whose `ACTIVE` value is 1.

This approach can be useful, but we recommend considering filters for most scenarios (see 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.

What is ternary logic?

A row is included in a 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's 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 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 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:

```
session.CreateCriteria(typeof(User))
    .Add( Expression.IsNull("Email") )
    .List();
```

You also need to be able to find users who 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 by using a substring match expression:

```
from User u where u.Firstname not like "%Foo S%"
```

For criteria queries, wildcard searches may either use the same wildcard symbols or specify a `MatchMode`. NHibernate provides the `MatchMode` as part of the `ICriteria` query API; you use it to write 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. But it does 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%'
```

We return to 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. 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, you may 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:

```
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. You've combined various restriction expressions with logical operators. But you've focused on single persistent classes—that is, you'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 you're querying databases, sometimes you want to combine data in two or more relations. This is achieved using a *join*. For example, you may 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					
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 7.1 The `ITEM` and `BID` tables are obvious candidates for a join operation.

When most people hear the word *join* in the context of SQL databases, they think of 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

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

If you join the tables ITEM and BID with an inner join, using their common attributes (the ITEM_ID column), you 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 you want all items, and null values instead of bid data when there is no corresponding bid, you use a (left) outer join, as shown in figure 7.3.

You can think of a table join as working in this way: 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 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 retrieves all bids and null if a bid has no item—certainly not a sensible query in this situation.)

In SQL, the join condition is usually specified explicitly; it isn't possible to use the name of a foreign-key constraint to specify how two tables are to be joined. Instead, you 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 you name this association in your 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:

- An ordinary join in the `from` clause
- A fetch join in the `from` clause
- A theta-style join in the `where` clause
- An implicit association join

We discuss all of these options in this chapter. Because the ordinary and fetch `from` clause joins have the clearest syntax, we discuss these first.

When you're working with NHibernate, there are usually several reasons to use a join, and it's important to note that NHibernate lets you differentiate between the purposes for joining. Let's put this in the context of a short example. If you're querying Items, there are three possible reasons why you may be interested in joining the Bids:

- You want to retrieve Items returned on the basis of some criterion that should be applied to their Bids. For example, you may want all Items that have a bid of more than \$100; this requires an inner join.
- You're running a query where you're mainly interested in only the Items without any special criterion for Bids. You may or may not want to access the Bids for an item, but you want the option for NHibernate to lazily load them when you first access the collection.
- You want to execute an outer join to load all Items along with their Bids in the same SELECT (*eager fetching*).

Remember that your default preference should be to map all associations lazily; an eager, outer-join fetch query can be used to override this default fetching strategy at runtime. We 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%'
```

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: you use this syntax only because you want eager initialization of the bids collections in a single SQL select.

You 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%'
```

You 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 you can rewrite the previous examples using `join fetch`. Although this looks straightforward to use, you must consider and remember a couple of things.

First, 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)`.

NHibernate currently limits you to fetching just one collection eagerly. This is a reasonable restriction, because fetching more than one collection in a single query would be a Cartesian product result. This restriction may be relaxed in a future version of NHibernate, but we 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 must be transported between database and application can easily grow into the megabyte range, and most of it is 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.

If you fetch a collection, NHibernate doesn't return a distinct result list. For example, an individual `Item` may 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 HashedSet(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 you want to do this using an HQL from clause join, you 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. You then use both aliases to express your 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! 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 you don't want the Bids in the query result, you can specify a select clause in HQL. This clause is optional (it isn't optional in SQL), so you only have to use it when you aren't satisfied with the result returned by default. You 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%'
  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. You assign aliases in the from clause and use them in the where and the optional select clauses. The select clause in HQL is much more powerful; we discuss it in detail later in this chapter.

ICRITERIA JOINS

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();
```

You'll 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 you 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; 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")
        .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, you've used simple qualified property names like *bid.Amount* and *item.Description* in your HQL queries. HQL supports multipart property path expressions for two purposes:

- Querying components
- Expressing implicit association joins

The first use is straightforward:

```

from User u where u.Address.City = 'Bangkok'

```

You 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. Here's an example:

```

from Bid bid where bid.Item.Description like '%part%'

```

This results 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` was many-to-one (instead of the current many-to-many), you 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
```

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 you express the same query like this:

```
from Bid bid
join bid.Item item
  where item.Category.Name like 'Laptop%'
  and item.SuccessfulBid.Amount > 100
```

You 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 lets you 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 you 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. You 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];
}

```

You 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 can't 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 the example application; you use it to identify people bidding for their own items.

You may also want to compare a foreign key value to a query parameter—for example, to find all `Comments` from a `User`:

```
User givenUser = LoadUser(1)
IQuery q =
    session.CreateQuery("from Comment c where c.FromUser = :user");
q.SetEntity("user", givenUser);
IList results = q.List();
```

Alternatively, sometimes you may 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 section 3.5.2).

These queries are 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
```

But you can now use the identifier value as a query parameter:

```
long userId = 1;
IQuery q =
    session.CreateQuery("from Comment c where c.FromUser.id = :id");
q.SetInt64("id", userId);
IList results = q.List();
```

You may 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%'
```

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 change our focus and discuss features of HQL that are used mainly for analysis and reporting functionality.

7.4 Writing report queries

Report 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 may only retrieve their names and prices. If this is the only information you need for a report screen, you 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; 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 report 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 you only want the `Items`, you can use this query instead:

```
select item from Item item join item.Bids bid where bid.Amount > 100
```

Or, if you're displaying a list page to the user, it may be adequate to 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[]` has 3 elements. Also, because it's a report query, the objects in the result aren't `NHibernate` entities and therefore aren't 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
}
```

If you're used to working with domain objects, this example will seem ugly and verbose. `NHibernate` gives you another approach: dynamic instantiation.

USING DYNAMIC INSTANTIATION

If you find working with arrays of values a little cumbersome, `NHibernate` lets you use dynamic instantiation and define a class to represent each row of results. You 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 you'd write just for your report screen; note that you 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 its own mapping file. But in order for NHibernate to "see" it, you 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, `Item` descriptions aren't unique, so the following query may return the same description more than once:

```
select item.Description from Item item
```

It's difficult to see how it can 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 isn't limited to database-dependent functions. You can use it with other, more generic (or standardized) SQL functions as well:

```
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 you 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: you 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) that any `select` clause that contains an aggregate function must contain only aggregate functions.

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
```

In this example, the `u.Lastname` isn't inside an aggregate function; you use it to group the result. You also don't need to specify the property you'd like to count in HQL. The generated SQL will automatically use the primary key if you 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 you 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), you 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`, you 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 only for 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 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 you define a class called `ItemBidSummary` with a constructor that takes a long, a string, and an int, you 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 are always 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 an 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 track changes to these returned objects.

Reporting queries result in faster release of allocated memory, because 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 we urge you to wait until you find an actual case where you have a real performance problem before using this optimization.

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 because NHibernate doesn't return `DataSets`, you have to find a solution.

The most common workaround 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 because 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 code similar to that required for DTOs, but it becomes tedious when you're 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 discuss 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, you can 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",
            '%' + 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. But for search screens with many optional search criteria, there is an even better way.

First, observe that as you add new search criteria, the parameter list of `FindUsers()` grows. It would be better to capture the searchable properties as an object. Because all the search properties belong to the `User` class, why not use an instance of `User`?

QBE (Query by Example) 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`.

You've significantly simplified the code again. The nicest thing about NHibernate Example queries is that an `Example` is just an ordinary `ICriterion`. 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, you 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);
    return GetSession().CreateCriteria(typeof(User))
        .Add( exampleUser )
        .CreateCriteria("Items")
        .Add( Expression.IsNull("SuccessfulBid") )
        .List<User>();
}

```

Even better, you 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 hand-coded SQL/ADO.NET. It's 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, you may have an Item and wish to retrieve all bids for that item, ordered by the amount of the bid. You 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, because 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.

You can 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. You 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 aren't 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`. Because 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();
```

But usually you'll 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 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", DateTime.Now.AddDays(-7)
        .List();

```

Again, this query doesn't initialize the Bids collection of the User.

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 may be a nice future extension. (You may remember from section 3.4.2 that a derived property mapping is a select clause subselect.) Note that some database 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 may 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. But 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:

- any
- all
- some (a synonym for any)
- 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
                                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()`, `max-index()`, `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, because queries with subqueries can often be rewritten using only joins and aggregation. But 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 may 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 you'll stumble on other examples.

In these relatively rare cases, you're free to resort to using the ADO.NET API directly. But 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 lets you 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 latter case, it's 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 can use the methods of `ISQLQuery` to provide more details about your query.

A SQL query can return scalar values from individual columns, a 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, paging, and so on.

SCALAR AND ENTITY QUERIES

The simplest native queries are scalar queries. Here's an example:

```
Ilist results = session.CreateSQLQuery("SELECT * FROM ITEM")
    .AddScalar("ITEM_ID", NHibernateUtil.Int64)
    .AddScalar("NAME", NHibernateUtil.String)
    .AddScalar("CREATED", NHibernateUtil.Date)
    .List();
```

This query won't return `Item` objects; instead, it returns 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 you 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 you must specify the columns of both tables. You must also specify the alias `"item"` in `AddEntity()` in order to join its `Seller`. Here is how you can 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 is enough to write it and take care of the ambiguous column name `ITEM_ID`. But unfortunately, this won't work in NHibernate; it doesn't recognize `i.ITEM_ID` because it isn't specified like that in the mapping file or attributes.

It's time to introduce a new trick that addresses this problem. We demonstrate this in the next section, which explains how to retrieve many entity types in a single query.

MULTIPLE-ENTITY 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 a 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. You need 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. But when you're using your own SQL, you don't want the user to have to understand all this. Instead, you can specify native SQL queries 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))
    .List<Item>();

```

Each placeholder specifies an HQL-style property name. And you 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 column 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 you 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 returns tuples of entities; as usual, NHibernate represents a tuple as an instance of object[].

As with HQL, it's usually recommended that you 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, you can see that the <return> element replaces the method AddEntity(). <return-join> is used for associations and collections, and <return-scalar> returns 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>
<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 returns tuples containing an item with its seller and bids and a computed scalar value (diff). Note that you can also refer to `<resultset>` elements 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, you also use the `{ }` syntax for columns you don't want to customize. In section 6.1.2, you defined an item's initial price as a composite user type; this example shows how to load it.

Because 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. Suppose you create a stored procedure like this (using a SQL Server database):

```
CREATE PROCEDURE FindItems_SP AS
  SELECT ITEM_ID, NAME, INITIAL_PRICE, INITIAL_PRICE_CURRENCY, ...
  FROM ITEM
```

You can call it using 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>
  ...
  </return>
  exec FindItems_SP
</sql-query>
```

Unlike the previous example, you must map all properties here; this is obvious, because NHibernate can't inject its own column aliases.

Note that the stored procedure must return a resultset to be able to work with NHibernate. Another limitation is that associations and collections aren't 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, and delete commands

In most cases, the commands generated by NHibernate to save your entities are acceptable. But it may happen that you need to perform a specific operation and override NHibernate's generated SQL. NHibernate lets you specify the SQL statements for create, retrieve, 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 uppercase). 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; 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. Also, for NHibernate 1.2, it isn't possible to supply named parameters such as `ITEM_ID = :id` for these insert/update SQL queries.

Retrieve commands are defined as named queries. For example, here is a query to load an item with a pessimistic lock:

```
<sql-query name="LoadItem">
  <return alias="item" class="Item" lock-mode="upgrade"/>
  SELECT {item.*}
  FROM ITEM
  WHERE ITEM_ID = ?
  FOR UPDATE
</sql-query>
```

Then it must be referenced in the mapping:

```
<class name="Item">
  ...
  <loader query-ref="LoadItem"/>
</class>
```

It's 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's 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, you 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 should usually make.

The fetch joins, part of the runtime fetching strategies, as introduced in this chapter, deserve some extra attention. But some design issues can't be resolved by tuning; they 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 an 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 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 you now wish to find the maximum bid for each item. The following code would be one way to do this:

```
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 you 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 may 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 that doing so is suboptimal. But 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 may be to enable batch fetching. You change your 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 the 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 you 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 you 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, you may need to do more complex processing on the bids than merely calculating the maximum amount. It's better to do this processing in the .NET application.

You 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 a 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 you may be using a criteria query.)

This mapping avoids the problem as far as this transaction is concerned; you'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 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 section 4.4.6), but you'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 ) );
}

```

You disabled batch fetching and eager fetching at the mapping level; the collection is lazy by default. Instead, you 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. You implement this by adding the results to a `HashSet` (from the library `java.util.Collections`) and then iterating the set.

You've now 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, you 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 consists 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, you can query categories, asking NHibernate to fetch their items. Then you 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 you wouldn't enable caching for this kind of data).

The `n+1 selects` problem may appear whenever you 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 section 4.4.7 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()`.

Next, we examine a third query API method. 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 a 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 you expect that categories are already cached in the session or second-level cache, then you only need the identifier value (the key to the cache). This will reduce the amount of data you have to fetch from the database. The following SQL would be slightly more efficient:

```
select CATEGORY_ID from CATEGORY where NAME like ?
You 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. You 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. But 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 won't happen often that the user chooses the same criterion many times. In this case, the cached query results will be underused, and you'd be better off not enabling caching for that query.

Note that the query cache doesn't cache the entities returned in the query result set, just the identifier values. But NHibernate does 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" results 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. 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 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, the following:

```
<add key="hibernate.cache.use_query_cache" value="true" />
```

But 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);
```

But even this doesn't give you sufficient granularity. 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:

- The default query cache region, null
- Each named region
- 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 whether 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 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; it may be hard to select the optimal approach. Profiler tools can help you test the performance of these options; use them as often as possible.

When you're working with HQL queries, you may wonder whether the generated SQL is optimal. A tool called NHibernate Query Analyzer lets you dynamically write and execute queries on your domain model. It displays the generated SQL query in real time and displays the result of your query when you execute it. It's also helpful when you're learning HQL. For more details, refer to the documentation at <http://www.ayende.com/projects/nhibernate-query-analyzer.aspx>.

7.8 Summary

We don't expect that you know everything about HQL and criteria after reading this chapter once. But 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. 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 round trips with many small queries; but at the same time, you 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.

Part 3

NHibernate in the real world

In real-world projects, you need patterns, tools, and processes that scale well with the projects' complexity. NHibernate's power and flexibility allows it to be adapted to almost any solution architecture. Because of this flexibility, it's not always obvious how to make the best use of NHibernate in your projects. In this part of the book, we aim to remedy that, helping you make the best choices by demonstrating many techniques, tips and best practices needed to design and implement persistence layers with NHibernate in the real world.

8

Developing NHibernate applications

This chapter covers

- Implementing layered applications
- Solving issues when setting up .NET applications using NHibernate
- Achieving design goals
- Solving debugging and performance problems
- Using integrating services like audit logging

At this point, you may be thinking, “I know all about NHibernate features, but how do I fit them together to build a full NHibernate application?” It’s time for us to answer that question and to show you how to apply the knowledge you’ve gained to implement applications as part of a real-world development process.

We discussed the architecture of an NHibernate application in section 1.1.5. This provided the bird’s-eye view, but you need to get from that point to working with executable code.

We discuss the development process layer by layer, showing the internals of each layer and how each should handle its designated responsibilities. We also discuss how layers should communicate with each other.

Because this book focuses on NHibernate, the domain model and persistence layers draw most of our attention. However, using NHibernate requires design decisions throughout all the application layers, so we provide details where we feel they will help.

The complexity of defining and building a layered application depends on the complexity of the problem at hand. For instance, the “Hello World” application in chapter 2 is trivial, but building an application as complex as the CaveatEmptor example is more challenging. By following the advice given in this chapter, you should be able to find your way through any difficult patches.

This chapter begins by focusing on the implementation of an NHibernate application. First, we rediscover the classic architecture of an NHibernate application. We talk about its layers and their purposes and briefly about their implementations. After that, we discuss some issues relating specifically to .NET applications, including web applications, security, and remoting. Finally, we explore approaches to troubleshooting and bug-fixing in your NHibernate applications. This part also serves as a map for the next two chapters; it will provide references to upcoming sections for more details.

The last part of this chapter is about services: how to integrate loosely coupled components with an NHibernate application. You’ll learn how to use the `IInterceptor` interface to efficiently integrate services like audit logging. We also discuss some other alternatives.

Let’s start with the architecture of an NHibernate application and its implementation.

8.1 *Inside the layers of an NHibernate application*

Chapter 1 presented an overview of the layered architecture of an NHibernate application. In this chapter, we look at the code that belongs in these layers.

If you recall, disciplined layering is important because it helps you achieve separation of concerns, making code more readable and maintainable by grouping code that does similar things. Layering also carries a price: each extra layer increases the amount of code required to implement a piece of functionality—and more code makes the functionality more difficult to change.

We don’t try to form any conclusions about the correct number of layers to use (and certainly not about what those layers should be) because 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 exist only if it’s required, because it increases the complexity and costs of development.

In this section, we go through the layers introduced in chapter 1, explaining their roles and discussing their implementations. That way, you’ll progressively learn how to develop an NHibernate application. These layers are as follow:

- The business layer (with the domain model)
- The persistence layer
- The presentation layer

Only two of these layers matter to the end user: the business layer matters because it embodies the problem the application is supposed to solve, and the presentation layer matters because it lets the user issue commands to the application and see the results.

The user doesn't care about the persistence layer directly, but it's obviously essential to most applications. It's useful to remember that persistence is a *service* that the application uses (like those presented in section 8.4), and 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'll see that they all exist to augment the domain model in some way. This is why we say that the development process of an NHibernate application is *domain-centric*; this approach is called *domain-driven development* (DDD). Because testing is part of the development process, we also discuss testing these layers and see how you can apply *test-driven development* (TDD).

Before we go any further, an introduction to patterns, DDD, and TDD is required.

8.1.1 Using patterns and methodologies

The breadth of this chapter requires us to mention many patterns and practices related to software development. Explaining each in depth is beyond the scope of this book, but we give a brief introduction to each pattern as we encounter it. If you're in unfamiliar territory, we encourage you to follow any references we give to get a deeper understanding of the topic.

If you haven't studied patterns before, here's the general idea. Many problems, although different in their formulation, are solved in similar ways. Thus it's possible to formulate a general solution that applies to many similar problems. In software, many professionals have documented the most useful patterns they've observed in the field, allowing them to communicate and share their knowledge with others. Hundreds of software-design patterns have been documented over the years, and throughout this section we give references to books and articles we consider important.

Let's quickly look at one popular and famous pattern: the *Singleton pattern*. You may have heard of it. The Singleton pattern ensures that a class has only one instance in an application and provides a global point of access to that instance. If you don't want people creating more than one instance of your Shopping Basket class, you can use the Singleton pattern to prevent them from doing so. This pattern was well documented in the book *Design Patterns: Elements of Reusable Object-Oriented Software*, published more than 10 years ago [Gamma et al 1995]. Because the Singleton pattern has proven so useful, it continues to be used daily in many open source and commercial .NET projects, including the .NET framework itself.

Learning patterns offers many benefits. They're well tested, time-proven methods that let you leverage the experience of other professionals. They give you a common vocabulary for efficiently communicating aspects of your software with others. You can learn more about patterns in the books *Design Patterns* and *Patterns of Enterprise Application Architecture* [Fowler 2003] and *Head First Design Patterns* [Eric Freeman et al 2004].

In addition, you may want to look up the many books and papers related to the Pattern Language of Programs (PloP) conferences (<http://hillside.net/plop/2008/>).

DOMAIN-DRIVEN DEVELOPMENT

The practice of DDD is complementary to NHibernate development. DDD is an increasingly popular approach to building software; the development process is heavily focused on the domain model. Establishing a good domain model is a fine art, and DDD asks that developers and business folk find common terminology to use in both conversation and code. In a DDD solution, the domain model should fully express the problem in terms that both developers and business people understand.

NHibernate applications are well suited to this domain-model-centric approach. We encourage you to learn more about DDD: see *Domain-Driven Design* [Evans 2004] and *Applying Domain-Driven Design and Patterns* [Nilsson 2006].

TEST-DRIVEN DEVELOPMENT

TDD is another practice that has gained much popularity over the last decade. Often, developers think that TDD must be about testing code that has already been written, but that's not entirely true. TDD is a practice that encourages developers to write tests before—or at least at the same time as—they write their domain logic and other code. This means thinking of a solution to a problem, writing the tests for this solution, and then implementing the solution to make the tests pass. If you've not tried TDD before, it may be hard for you to instantly understand the magic of this approach. We strongly recommend that you try it; once you're used to the process, the benefits become clear.

Unit testing is a key part of TDD and is different from classic application testing. Traditionally, testing involved having a test team exhaustively test applications by navigating screens and prodding buttons. Unit testing is different because it's repeatable and automated, and it doesn't involve user interaction. Thus unit testing can be done frequently.

When you use unit testing, you write tests to test a single *unit* of code at a time, such as a class or function. Each test is usually small and only needs to interact with the unit of code, rather than an entire application. This means you can run thousands of tests by clicking the “run tests” button and sipping your coffee while the computer does the hard work.

The .NET framework has some popular testing libraries to support TDD and unit testing. The most popular is NUnit (<http://nunit.org/>). Later in this chapter, we briefly demonstrate how you can unit test a domain model and the business layer using NUnit.

Like DDD, TDD is an agile software development practice (http://en.wikipedia.org/wiki/agile_software_development). For more details about NUnit and TDD, read *Test-Driven Development in Microsoft .NET* [Newkirk et al 2004]. You can also look at tools like ReSharper (<http://www.jetbrains.com/resharper/>) which is a Visual Studio plugin that can help greatly with both TDD and DDD.

With all these methodologies and patterns in mind, let's see how you can use them to develop the layers of an application.

8.1.2 Building and testing the layers

In the following sections we demonstrate how to combine these good practices to build an NHibernate application. A clear separation of concerns allows you to develop the layers of an application almost independently. Some practitioners like to start with the GUI layer, others like to start with the domain model layer, and some developers build all the layers simultaneously. For the purposes of simplicity, we start with the domain model and work our way up to the presentation layer. For each layer, we look at two activities: writing the core classes and writing the unit tests.

8.1.3 The domain model

The domain model is commonly considered part of the business layer. But we define a slight separation: the domain model is the heart of the business. It indicates what the business is doing (the domain) and represents the entities manipulated in the domain by the business (the model).

It's conceptually independent of any other layer, even the business layer, because it doesn't use any class that isn't itself part of the domain model. Without realizing it, you're familiar with the domain model because it primarily includes entities, which you've been implementing since chapter 2!

You can see a more complex example of a domain model in the CaveatEmptor application 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 enumerates the steps of this process along with some of the problems you may encounter; we cover this topic more thoroughly in chapter 9.

There are two ways to build your domain model entities. You can either write them from scratch, or generate them from the mapping files, a database schema, or some other intermediate format. Either way, building the entities is usually easy enough; it's a matter of creating a set of classes with appropriate constructors and properties.

Implementing the entities' behavior and rules (the business logic) can be less straightforward. You must be careful when choosing how and where to implement business rules to prevent things from getting too complicated.

A domain model doesn't exist in isolation, so the other layers may have an influence on how it is designed. For example, they may require the domain model to not be completely persistence ignorant, or that entities expose properties to assist in GUI data binding.

Finally, you may have to communicate with other components/services/applications that aren't able to manipulate your domain model directly. In .NET applications, it's common to have a specific format for sending information to and from other applications. This might be XML, json, DataSets or even Data Transfer Objects (DTOs). In all these cases, you must find a way of converting data between these formats, which isn't always easy.

All these problems require consideration and experimentation. Chapter 9 provides a wide range of options to help you solve them.

TESTING

If your previous applications didn't include tests for your domain model, you should reconsider your position. Remember, your domain model is the heart of your software; but unlike the presentation layer, a broken domain model may not be easily visible.

Note that when you're unit testing your domain model, the units are the entities.

Globally, you can use two kinds of tests: data-integrity tests and logic tests. Let's take a simple example of each and see how you can use NUnit. To make it even more fun, we apply TDD.

Data-integrity tests are simple tests to make sure the entities' content is valid (according to the business logic) and remains valid, no matter what happens. For example, suppose you have a `User` entity in your domain model, and its name is mandatory; and because it's stored in the database, this name must not exceed a certain number of characters (let's say 64).

Listing 8.1 shows three tests to codify this specification.

Listing 8.1 Unit testing an entity

```
using NUnit.Framework;
[TestFixture]
public class UserFixture {
    [Test]
    public void WorkingName() {
        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() {
        User u = new User();
        u.Name = null;
    }
    [Test, ExpectedException( typeof(BusinessException) )]
    public void TooLongName() {
        User u = new User();
        u.Name = "".PadLeft(65, 'x');
    }
}
```

Name property keeps its value

Name can't be null

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 there is a property called `Name` in the class `User` and that it keeps the value you assign to it. You use a random value to prove that the name is stored correctly. The actual test is done using `Assert.AreEqual(...)`; the `Assert` class belongs to NUnit and provides a wide range of methods for 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 section 9.4.2, you may allow invalid values and validate changed entities before saving them.

Remember, it's good practice to keep test fixture classes in a separate test library.

Now, we can move on to the implementation of the part of the `User` class that makes these three tests pass. First, you create the `User` class and its `Name` property; then, you use a `_name` field to keep the value assigned to this property. At this point, the first test passes.

After that, you 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 easy to understand: in the setter of the `Name` property you throw an exception if the name that is about to be set is invalid (null, empty, or too long); you keep this value in the `_name` field.

The second kind of test for an entity is the logic test, which tests any behavior in the entity. For example, when changing her password, the user must provide the old password, and then enter the new password twice.

Here are some tests for this method (they 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 makes sure you can successfully change the password, and the other two make sure you can't change the password with an invalid old password or a new password that's different than the confirmation password.

Here is an implementation of the `ChangePassword()` method that makes these tests pass:

```
public void ChangePassword(string oldPwd, string newPwd,
    string confirmPwd) {
    if( (_password != oldPwd) || (newPwd != confirmPwd) )
        throw new BusinessException("...");
    _password = newPwd;
}
```

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, as you'll do in the next section.

Remember that a domain model can (and should) be efficiently tested. Its autonomy makes it easy not only to implement, but also to test.

Let's continue with the other classes that form the business aspect of the application.

8.1.4 *The business layer*

The business layer acts as a *gateway* that upper layers (such as the presentation layer) must use to manipulate entities. It's common to see .NET applications (mainly those using `DataSet`) directly access the database. You already know that doing so is wrong, and why (if you've forgotten, see section 1.2).

The business layer plays several roles: It's a layer on top of the persistence layer. It performs high-level business logic (that can't be performed by the entities themselves). It may also integrate services like security and audit logging.

The controller (from the model-view-controller [MVC] pattern) can also be considered part of this layer. It pilots the flow of information between the end user (through the view) and the model. But it's a good practice to keep the controllers as a thin layer on top of the *core* of the business layer.

IMPLEMENTATION

Depending on the coupling between the entities, you can write one business class to manage each entity or one for many entities. You may also have business classes for some use cases or scenarios.

When you're implementing CRUD-like operations, try not to mimic the persistence layer. At this level, *save* and *update* aren't business words; but it's easy to find the right words when you consider the problem from a business perspective. For example, when you place a bid on an item, although you're saving this bid in the database (technically speaking), you 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 section 10.1).

Let's change the previous example to illustrate a piece of the business layer. What if you want to allow administrators to change users' passwords (without knowing their current ones)?

In this case, the class `User` can't perform this logic because it doesn't know which user is currently logged in, unless you make this information available using, for example, the Singleton pattern (but this may not be a good idea).

Here is how you can implement the `ChangePassword()` method (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
}
```

This method includes many validations, but note that you don't validate the new password; this is up to the `User.Password` property. You can make the setter of this property internal (provided the domain model and the business layer are in the same library) to make sure the presentation layer can't change the password directly.

Note that instead of receiving an instance of the `User` class, this method can receive the user's identifier and load the user internally. One advantage is that the business layer won't have to check if the user instance has been modified. Another advantage is that the presentation layer can more easily provide an identifier rather than a complete instance. This is especially the case with ASP.NET applications, where you're more likely to have an identifier rather than the entire user instance. Note that it's considered less object-oriented to pass identifiers rather than instances, but this is a worthwhile trade-off in some cases.

Unless you're writing a heavily customizable search engine UI, you should provide methods for all the kinds of queries that the end user can run. Don't let the presentation layer build arbitrary queries; letting it do so makes the presentation layer aware of the persistence layer, which may become a security issue, and it also makes the application less testable. If you have many queries to run, you might consider a more extensible approach that avoids having to create too many query methods.

You may also include some code for audit logging to keep track of what happened and who did it (invaluable when you're debugging an application in production). But in section 8.4, you'll learn another way to deal with this kind of service.

We discuss implementing the business logic in section 9.4. We can't discuss the general implementation because it depends heavily on the application. Just make sure you cleanly separate the business logic 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 provide much detail about the implementation of controllers, because they tend to be platform-dependent.

TESTING

As you may guess, testing the business layer is crucial. It's similar to testing the domain model. If you understand unit testing as illustrated in listing 8.1, you should be able to easily test the previous method, `ChangePassword()`.

But because this layer uses the persistence layer, it may become troublesome. 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.

Tests for the business layer should only be related to the business layer itself. We look at testing the persistence layer in the next section to see what must not be tested here—even tests for the domain model's entities should be separated.

8.1.5 *The persistence layer*

The persistence layer provides CRUD methods for entities. Thanks to NHibernate, it can be implemented as a service (that is, non-intrusively) for the domain model.

But some libraries merge the persistence layer into the domain model. We generally see that as a bad practice, but the reason behind this design choice is simplicity: when you're writing a new application that isn't complex (in terms of layer coupling and integration issues), having a persistence-ignorant domain model isn't necessarily a requirement.

Regardless, we recommend that you build the persistence layer separately and that you hide it behind the business layer. But we agree that it depends on 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's a well-known pattern to implement persistence layers. In the previous code snippet, `UserPersister` can be called `UserDAO`.

Persistence classes have methods for simple CRUD operations and can let the business layer execute custom queries. We explain this approach in section 10.1.2.

The presentation layer (and other upper layers) shouldn't be able to access this persistence layer unless the application is simple, in which case the business layer isn't required (but beware of simple applications evolving and becoming complex). If this layer is in the same library as the business layer, its classes can be made internal. Otherwise, avoid adding a reference in the presentation library to the persistence library.

Also note that the persistence sits between the business layer and the database, so the layer should hide any database semantics. This leads to a more maintainable business layer that isn't concerned with low-level database and persistence issues.

TESTING

As with the domain model, you can test the persistence layer two ways. 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 the NHibernate mapping is correctly written and those entities are correctly loaded and saved. It involves saving an entity with random content, loading it, and making sure the content hasn't changed.

When you test the persistence logic, the logic is represented by any code customizing the way NHibernate works, such as the queries (the where clause, the ordering, and so on). The idea is to make sure you get the data as intended.

But you should avoid testing NHibernate itself. There are specific NHibernate tests for that. Let's take this test as an example:

```
session1.Save( entity );  
Assert.IsNotNull( session2.Get<Entity>(id) );
```

This test is useless because if `Save()` succeeds, then the entity was saved. You'll have a hard time if you don't trust NHibernate and all the libraries to do their jobs. On the other hand, you can make sure the proper lazy-loading and cascading options are enabled.

Notice that you use two different sessions (`session1` and `session2`) in this example because `session1.Get<Entity>(id)` doesn't hit the database; it uses 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, see section 6.3.

Persistence tests are slower than other tests because of the cost of using a database. You can speed them up using in-memory-capable RDBMSs like SQLite. It's also possible to mock the database; but the tests may become less meaningful. (*Mocking* is a technique that lets you fake a component in order to avoid its dependencies. For details, see http://en.wikipedia.org/wiki/Mock_object.)

Because the persistence layer is hidden behind the business layer, you can test it *through* the business layer. Although doing so 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. In ASP.NET it's made up of the ASPX, ASCX, and various code-behind files. It serves as a bridge between the end user and the business layer. Its primary function is to format and display information (the entities); it also receives commands and information from the user to send them to the business layer (or the Controller).

IMPLEMENTATION

Implementing the presentation layer is largely outside the scope of this book, but using NHibernate may have some effect on it. Section 8.2 discusses deployment issues of .NET applications that use 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. In section 9.5, we show you many alternatives to data-bind entities.

To see a typical command-handling method, let's implement a method that can be called when the user clicks 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 sends the information to the business layer and then displays a message. The method plays the role of the Controller; strictly speaking, it should call the Controller, which will then do exactly what is in this method. This is an example of a situation where the code-behind is used to implement the Controller logic; the Controller isn't 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 vary widely depending on the way you organize your business layer.

Note that you should always output errors to a log file. Doing so may save you a lot of work when you're trying to figure out what happened in an application in production.

TESTING

The presentation layer is the most difficult to test automatically, because it's inherently visual. The most common way to test it is to run the application and see if it works.

On the other hand, if you write your application as we've 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's harder to break.

A number of techniques and libraries are available to test the presentation layer, but we don't cover them here. For more information, start by reading http://en.wikipedia.org/wiki/List_of_GUI_testing_tools.

When you're implementing these layers, you may encounter issues when you try 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 constraints in the way they work or because of security restrictions. In

this section, we talk about two specific features: working in a web environment and using .NET remoting.

8.2.1 Working with web applications

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 for how to develop an application with a web interface.

QUICK START

You shouldn't have major issues starting an application using web pages as the presentation layer. To achieve a good separation of concerns, the other layers must not be in the web application or web site; practically speaking, the code-behind of the pages and `App_Code` section must contain only the presentation layer logic (formatting, displaying, retrieving information, and calling the other layers). The other layers can reside in one or more separate libraries. This is also true for Windows Forms applications, where GUI code is separated from other business and persistence code.

You may encounter two common issues when implementing the presentation layer of an NHibernate application: data-binding the entities (and their collections) and managing the session to efficiently persist these entities. We discuss data binding in section 9.5 and session management in chapter 10.

CODE ACCESS SECURITY

The .NET framework has a powerful security policy. It allows, for example, web administrators to limit the 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. For this reason, 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.

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 sections 3.4.3 and 3.4.4 for more details.

If your NHibernate configuration properties are in `Web.config`, you must add an extra attribute to the section declaration:

```
<section name="nhibernate" type="..." requirePermission="false" />
```

Setting the `requirePermission` attribute to `false` lets NHibernate 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, database access can be restricted so that only the server knows how to access the database; the client communicates with the server using, for example, a layer of *marshal-by-reference* objects.

The domain model must be serializable. Because NHibernate proxies aren't serializable, you must also return entities with fully initialized associations. In some edge cases, you may consider using Data Transfer Objects (DTOs). For more details, see section 10.3.1.

Once the application is finished, you may ask yourself whether you achieved your initial goals. And now, the maintenance starts (fixing bugs, improving performance, and so on). Let's see how we can help you with these boring and stressful tasks.

8.3 *Achieving goals and solving problems*

Now that we've looked at the development process of an 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're achieved.

A good understanding of the development process is required to fully take advantage of this section. But don't make the mistake of delaying this task when developing an application. You should realize that the costs to change the design of an application increase rapidly as the application is implemented. And in the middle of development, don't hesitate to take a break and look back at your initial goals and how far you are from them.

Assessing an application's status isn't always easy. Developers can also have a hard time understanding what is wrong with their application. Having some problem-solving skills is definitively a plus when you're dealing with complex frameworks like NHibernate.

Finally, remember that a single tool can't do every job. NHibernate is a good framework to solve the ORM mismatch; but it's also complex. We advise you to carefully test NHibernate and your competence before you begin to use it in a mission-critical environment. And learn to choose wisely: use NHibernate when it's the best option, and fall back to other alternatives for other situations.

By the end of this section, you'll have a better understanding of how to deal with these tasks. Let's start with the first one in the development process: achieving your design goals.

8.3.1 *Design goals applied to an NHibernate application*

An NHibernate application is a .NET application using NHibernate. But because of the central role played by NHibernate, you must take into consideration some implications when you're designing an NHibernate application.

The MSDN defines six design goals: availability, reliability, manageability, securability, performance, and scalability. We look at each of them with NHibernate in mind.

AVAILABILITY AND RELIABILITY

Availability and reliability relate to an application's ability to be present and ready for use. Basically, you should aim to make your application bug-free.

NHibernate has an impact on the way your application is tested. Because it isn't 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 section on performance and scalability.

Extensive testing is the first recommendation to achieve the goal of availability and reliability. You should test 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 won't cause your application to crash.

Finally, if your application provides services to external 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 part of these external systems. The most common technique is to carefully validate all inputs you receive from these systems.

MANAGEABILITY AND SECURABILITY

Manageability and securability describe the application's ability 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.

NHibernate encourages the separation of concerns in your application (layered architecture), which increases its manageability. NHibernate is also configurable using XML files, thus allowing production-time changes.

Security wasn't a major concern a few years ago, but it's gaining considerably more attention as systems become more open to the outside world. You should 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. Our general advice is that you implement your application with security in mind and encourage the use of minimal privileges by default.

PERFORMANCE AND SCALABILITY

Performance is the measure of an application's operation under load. Developers tend to consider this goal the most important. It's also the most misunderstood (for example, it's 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 NHibernate is efficiently used (lazy/eager fetching, caching, and so on). As a last resort, NHibernate lets you fall back to classic ADO.NET (the underlying connection is accessible through the `ISession.Connection` property); we aren't against stored procedures for batch processing. Note that NHibernate 1.2.0 is faster than hand-coded data access for classic operations on SQL Server because it uses an unexposed batching feature of the .NET framework.

The next section gives you some tips that can help improve your application's performance. We think performance should be considered throughout a project, rather than leaving it until the end. Of course, we don't recommend you spend time

prematurely optimizing your systems, but it always helps to at least have an understanding of the performance implications of the code you're writing.

Scalability refers to an application's ability to match increasing demand with a proportional increase in resources and cost. You can use many techniques to achieve this goal, including 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 an NHibernate session as late as possible and to close it as soon as possible. You may also consider using a distributed cache (see section 6.3).

FINAL NOTE

You may certainly aim at fulfilling all these design goals; but a design choice that has a positive effect on one goal may at the same time have a negative effect on another.

Remember that it's important to have strict rules to make sure these goals are kept in mind throughout the development process. It's also important to balance the amount of effort you put into optimization with the outcome of the work.

Don't work on a feature more than it's worth, and don't optimize blindly—that is, without any way to know whether the optimization is needed and to measure the improvement that results from it.

8.3.2 Identifying and solving problems

Nothing is more frustrating than getting an error and not understanding where it comes from. But a strict debugging process helps easily fix most errors. Errors in .NET applications are generally thrown exceptions.

Users may also complain when they're using a bug-free application, because the application is too slow. This problem can be frustrating for users, and you can be certain they'll pester you about it.

BUG-SOLVING PROCESS

Before you think about fixing a bug, make sure you have an infrastructure to catch and log it and that the user receives a useful message. This is important in a 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, and so on (everything returned by `exception.ToString()`).

Then try to understand the meaning of this exception (refer to the documentation), and begin investigating its origin. A good technique at this stage is to isolate the problem until its origin and solution become obvious. Practically speaking, this means removing processes until you locate the culprit. If you have a hard time understanding an NHibernate exception, appendix B tells where you can ask for help.

Once you've fully identified the problem, TDD recommends that you write tests that reproduce this problem; these tests will help prevent the problem from coming back unnoticed later. You should also take some time to think about the design of your application, to see if the problem is the symptom of a bigger issue.

Finally, you can fix the bug, making the tests you wrote pass.

IMPROVING PERFORMANCE

Suppose you've received a performance complaint. Here are some common mistakes you may have made and tips to help dramatically improve your application's performance (and make your users happier).

To begin with, you should consider writing performance tests at an early stage, rather than waiting for a user to tell you that your application is too slow. And don't forget to optimize your database (adding the correct indexes, and so on).

Sometimes, developers new to NHibernate create the session factory more often than is required. Remember, creating the session factory is an expensive process, and for most applications, it needs to be done only once: at application startup. Try to give your session factory a lifespan equal to that of your application (by keeping it in a static variable, for example).

Another common mistake, related to the fact that NHibernate makes it so easy to load entities, is to load more information than you need (without knowing it). For example, associations and collections are fully initialized when lazy loading isn't enabled. Even when you're loading a single entity, you may end up fetching an entire object graph. Our general advice is to always enable lazy loading and to write your queries carefully.

A related issue arises when you enable lazy loading: the n+1 select problem. For details, read section 8.6.1. You can spot this issue early by measuring the number of queries executed per page; you can easily achieve that by writing a tool to watch logs from `NHibernate.SQL` at the `DEBUG` level. If this level is higher than a certain limit, you have a problem to solve; do it immediately, before you forget what is going on in this page. You can also measure other performance-killer operations (like the number of remote calls per page) and global performance information (such as the time it takes to process each page).

You should also try to load the information you need using the minimum number of queries (but avoid expensive queries like those involving Cartesian products). Note that it's 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 write optimized queries.

Now, let's talk about a less-well-known issue, related to the way NHibernate works. When you load entities, the NHibernate session keeps a number of pieces of information about them (for transparent persistence, dirty checking, and so on). During committing/flushing, the session uses this information to perform the required operations.

In one specific situation, this process can be a performance bottleneck: when you load a lot of entities but update only few of them, this process is slower than it should be. The session checks all the entities to find those that must be updated. You should avoid this waste by evicting unchanged entities or using another session to save changed entities.

As a last resort, consider using the second-level cache (and the query cache) to hit the database less often and reuse previous results. See section 6.3 for more details about the pros and cons of this feature.

THROWING EXCEPTIONS

It's common in other environments (like C++) to write code like this:

```
if( something goes wrong ) return -1;
```

But .NET guidelines recommend using exceptions. They're much more powerful and harder to miss. It's important to understand this concept because NHibernate relies on exceptions to provide error reports, and your application should do the same. Although you should already be familiar with this concept, we briefly review it and explain how to handle NHibernate exceptions.

Your first step should be to create your own exceptions to provide better information (and thus make you able to better handle them). Use .NET built-in exceptions only when they're meaningful (for example, `ArgumentNullException` when reporting a null argument).

This book uses `BusinessException` when a business rule is broken. You can 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 can 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 user).

If you come across a performance problem when using NHibernate, and you find it difficult to solve, step back and ask yourself whether NHibernate was the right tool for the process.

8.3.3 Use the right tool for the right job

As we explained in chapter 1, ORM and NHibernate are powerful tools. But 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 of use cases 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. Many tasks aren't inherently 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, you can add services using many techniques, each of which has pros and cons. Let's study the case of audit logging.

8.4 Integrating services: the case of audit logging

So far, we've talked about the development of an NHibernate application without taking into account many aspects and features that may be hard to plug into a layered architecture. In this section, we 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 into the main application to add 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's also common to have business components 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's also easy to disable them without significantly affecting the main application. This is why they're often part of the nonfunctional requirements.

Audit logging is the process of recording changes made to 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, you may 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 use the *IInterceptor* interface to implement audit logging. But first, we briefly discuss the

hard way (doing it manually), so you can grasp the problem's level of difficulty. Then we show you how `IInterceptor` makes it much easier.

8.4.1 *Doing it the hard way*

The hard way (the manual way) requires continuous effort through the development process. In the case of audit logging, it means calling the audit logging service each time it's needed.

NOTE *Can't I use database triggers?*—Audit logs are often handled using database triggers, and we think this is an excellent approach. But it's sometimes better for the application to take responsibility, especially if 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` generates and saves a log about this change. `LogType` is an enumeration; it's better than using a string. Note that you may use the *Observer pattern* to remove the dependency on this service; see section 9.3.2 for details about this pattern.

The main advantage of this approach is that a better message can be generated for each operation, because you know exactly what you're doing each time you call this service. The disadvantages are that this approach is verbose (it clutters the code) and, more important, can be forgotten or bypassed. This is unacceptable when you're building a trustworthy audit logging service.

This approach may work much better for other services; so don't discard it completely.

8.4.2 *Doing it the NHibernate way*

Let's see how NHibernate allows you 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 an NHibernate `IInterceptor` that automatically creates the audit trail for you.

CREATING THE MARKER ATTRIBUTE

You first create a marker attribute, `AuditableAttribute`. You 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 can add properties to customize the logging per class (for example, using a localized name instead of the class name). Enabling audit logging for a particular persistent class is now trivial; you 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. The audit-logging service has no other means to extract them, unless you use a big switch statement to cast the object (which is feasible if this service is aware of the domain model).

Instead of an attribute, you can create an `IAuditable` interface. That way, the entities can actively participate to the logging process. You can also do both and choose the best one for each entity.

CREATING AND MAPPING THE LOG RECORD

Now you create a new persistent class, `AuditLogRecord`. This class represents the information you 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, you 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>

```

You mark the class `mutable="false"` because `AuditLogRecords` are immutable. NHibernate will 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 for audit logging to be handled in a central piece of code, transparently to the business logic. You wouldn't manually create a new `AuditLogRecord` and save it whenever an `Item` was modified.

NHibernate offers an extension point so you can plug in an audit-log routine or any similar event listener. This extension is known as an NHibernate `IInterceptor`.

WRITING AN IINTERCEPTOR

You'd prefer that a `LogEvent()` method be called automatically when you call `Save()`. The best way to do this with NHibernate is to implement the `IInterceptor` interface, as shown in listing 8.2.

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();
    private ISet updates = new HashSet();
    public ISession Session {
        get { return this.session; }
        set { this.session = value; }
    }
    public long UserId {
        get { return this.userId; }
        set { this.userId = value; }
    }
    public virtual bool OnSave(object entity,
        object id,
        object[] state,
        string[] propertyNames,
        IType[] types) {
        if ( entity.GetType().GetCustomAttributes(

```

Collections to keep new and modified entities

Opened session must be provided

Required for AuditLogRecord

Collects new entities

```

        typeof(AuditableAttribute), false).Length > 0 )
        inserts.Add(entity);
        return base.OnSave(entity, id, state, propertyNames, types);
    }
    public virtual bool OnFlushDirty(object entity,
        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) {
        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);
            }
        } catch (HibernateException ex) {
            throw new CallbackException(ex);
        } finally {
            inserts.Clear();
            updates.Clear();
        }
    }
}

```

Collects modified entities

Log process

Instead of directly implementing the `IInterceptor` interface, you inherit from `EmptyInterceptor`, which allows you to ignore the methods of this interface that you don't need.

This particular interceptor has two interesting aspects. First, the `session` and `userId` are fields this interceptor needs to do its work, so a client using this interceptor must set both properties when enabling the interceptor. The other interesting aspect is the audit-log routine in `OnSave()` and `OnFlushDirty()`, where you add new and updated entities to collections. The `OnSave()` interceptor method is called whenever NHibernate saves an entity; the `OnFlushDirty()` method is called whenever NHibernate detects a dirty object. The audit logging is done in the `PostFlush()` method, which NHibernate calls after executing the synchronization SQL.

Note that `entity.GetType().GetCustomAttributes()` performs badly (compared to using `IAuditable`), but you can optimize this code by caching all the decorated types.

You use the static call `AuditLog.LogEvent()` (a class and method we discuss next) to log the event. Note that you can't log events in `OnSave()`, because the identifier value of a new entity may not be known at this point. NHibernate is guaranteed to have set all entity identifiers after flushing, so `PostFlush()` is a good place to perform audit logging.

Also note how you use the session: you pass the ADO.NET connection of a given session to the static call to `AuditLog.LogEvent()`. There is a good reason for doing this, as we discuss in more detail. Let's first tie it all together and see how you enable the new interceptor.

ENABLING THE INTERCEPTOR

You need to assign the `IInterceptor` to an NHibernate `ISession` when you first open the session:

```
AuditLogInterceptor interceptor = new AuditLogInterceptor();
using( ISession session =
    sessionFactory.OpenSession(interceptor) ) {
    interceptor.Session = session;
    interceptor.UserId = currentUser.Id;
    using( session.BeginTransaction() ) {
        session.Save(newItem); // Triggers OnSave() of the interceptor
        session.Transaction.Commit(); // Triggers PostFlush()
    }
}
```

You should move the session opening to a helper method to avoid doing this work each time.

Let's get back to that interesting session-handling code inside the interceptor and find out why you pass the `Connection` of the current `ISession` to `AuditLog.LogEvent()`.

USING A TEMPORARY SESSION

It should be clear why you require an `ISession` instance inside the `AuditLogInterceptor`. The interceptor has to create and persist `AuditLogRecord` objects, so a first attempt for the `OnSave()` method can 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);
    }
}
```

You 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. But 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, as shown in listing 8.3.

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)) {
            AuditLogRecord record =
                new AuditLogRecord(logType.ToString(),
                                    tempSession.GetIdentifier(entity),
                                    entity.GetType(),
                                    userId);
            tempSession.Save(record);
            tempSession.Flush();
        }
    }
}
```

Create new session; reuse open connection

Create and save log record

Note that this method never commits or starts any database transactions; all it does is execute 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 handy technique you may also find useful in other scenarios.

The NHibernate way is powerful, simple, and easier to integrate. But there are some kinds of operations that can't 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'll probably use both approaches.

We encourage you to experiment and try different interceptor patterns. The NHibernate website also has examples that use nested interceptors and log a complete history (including updated property and collection information) for an entity.

8.4.3 Other ways of integrating services

The approaches we've covered are common and simple to implement. But more complex applications may require a more loosely coupled approach. In this section, we introduce the Inversion of Control and Dependency Injection patterns. We also give you a hint about how you can use a logging library to merge your logs with NHibernate logs.

INVERSION OF CONTROL AND DEPENDENCY INJECTION

It's outside the scope of this book to cover these patterns in detail; but if you don't know about them, a brief introduction will be helpful. These patterns are designed to avoid high coupling between services that address different concerns.

Let's take an example. In the auction application, ending an auction involves updating the database, sending a notification to the winner, collecting the payment, and dispatching the item. These steps require that you communicate with different services. A high coupling between them may cripple the application's manageability and flexibility. Configuring and changing these services can become difficult.

The Inversion of Control pattern solves this issue by externalizing the binding between the application and the services. 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, you can change the service by 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 8.3.

Many libraries provide these features, and each has pros and cons. Two of the most popular are Castle Windsor (<http://www.castleproject.org/container/>) and Spring.NET (<http://www.springframework.net/>). There are many more, including Structure Map, Ninject and Unity. For more information, see 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 used for maintenance rather than auditing.

It's easy to merge your logs with NHibernate logs using log4net. This library provides various destinations for the logs; it's even possible to send them through email or to save them in a database. But be aware of the performance costs.

If you can't use log4net, you have another option: wrap the log4net library to redirect method calls, so you can switch from log4net to another solution like the Enterprise Library or the `System.Diagnostics` API.

Although logging is a nonfunctional requirement that few users care about, logs are invaluable when you're debugging applications in production. Think twice before deciding you don't need to implement logging.

8.5 Summary

This chapter focused on application development and the integration issues you may encounter when writing NHibernate applications. We first considered the practical implementation of a layered application. We discussed how the domain model and the business layer should be implemented and tested. We then discussed the persistence layer and ended with the presentation layer.

The next objective of this chapter was to help you integrate NHibernate applications into production environments. We talked about the medium-trust issue you may encounter when developing web applications.

After that, we summarized how NHibernate can help you achieve the standard design goals of a .NET application. NHibernate has an impact on the way you design an application, and careful use of its features can greatly improve the quality of your application. We also gave you a few tips that can help you identify and solve bugs and performance issues.

In the last section of the chapter, we considered integrating services in an NHibernate application. We discussed the pros and cons of the hard way and the NHibernate way. We also talked about a few other alternatives.

We implemented audit logging for persistent entities with an implementation of the NHibernate `IInterceptor` interface. The custom interceptor uses a temporary `ISession` trick to track modification events in an audit history table.

You're now ready to dig into the details of implementing the two layers directly related to NHibernate: the domain-model layer and the persistence layer. These topics are covered in the next two chapters.



Writing real-world domain models

This chapter covers

- Domain-model development processes
- Legacy schema mapping
- Understanding persistence ignorance
- Implementing business logic
- Data binding in the GUI
- Obtaining DataSets from entities

Having read this book so far, you should be familiar with what a business entity looks like, what a domain model is, and roughly how a domain model is formed. Our examples have aimed to keep things simple, so we haven't yet introduced you to the processes and techniques that will help you tackle real-world projects.

The first part of this chapter looks at the various starting points of an NHibernate project and then explains how you can leverage automation and generation to help build the other layers. Until now, you've been manually implementing entities by hand. You can save much time by using the tools described here to automatically generate domain-model entities, database schema, and even mapping definitions.

One particularly tricky starting point is a legacy database that you can't change. Fortunately there are many mappings explained in this chapter that are especially useful in that scenario.

Once we're finished describing the processes and tools around NHibernate development, we'll take a closer look at the domain model. Up to this point, this book's examples have involved entities that contain only data. This allowed us to demonstrate how the mappings work for the purposes of saving and loading those entities. But the domain-model pattern encourages you to create a much more behavior-rich domain model that encapsulates business rules, validation, and workflow. Later in this chapter, you'll discover how these things can be achieved.

Another aspect that we briefly touched earlier is *persistence ignorance*; certain projects may require that the domain model has no awareness of NHibernate, instead focusing purely on business concerns. We look at what persistence ignorance means and how you can structure your projects to realize it with NHibernate.

Selecting and applying the techniques presented here, you'll develop a fully functional domain model that is well suited to your needs. The next trick is to get the domain model to collaborate with the other layers, including the GUI. This will be the focus of the two last sections, which explain how entities can be consumed in the presentation layer and how to fill DataSets with the content of an entity to allow compatibility with many GUI and reporting components.

We'll begin by discussing the possible starting points for an NHibernate project and the development processes that may follow.

9.1 Development processes and tools

In the earlier chapters, you always started by defining the domain model before creating the database and setting up your mapping. What if you already have a database in place, or even a mapping file? No rule says that things have to be done in any particular order, so we'll present the different processes available and explain which projects they suit best.

You'll find that once you've created either a database, a domain model, or mapping files, NHibernate provides tools that can be used to generate the other representations. Figure 9.1 shows the input and output of tools used for NHibernate development.

Generally, you have to complete and customize the generated code, but the tools can give you a valuable head start. We'll review these tools and processes in this section, starting with the top-down approach.

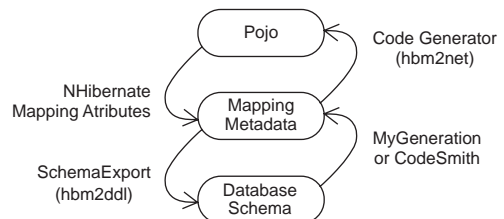


Figure 9.1 Development processes

9.1.1 Top down: generating the mapping and the database from entities

The approach you've been using in this book is commonly called *top-down development*. This is the most comfortable development style when you're starting a new project with no existing database to worry about.

Looking at figure 9.1, the starting point is the Plain Old CLR Object (POCO). When using this approach, you first build your .NET domain model, typically as POCOs. If you've used the `NHibernate.Mapping.Attributes` library to decorate your entities, you can use NHibernate to generate the mapping for you. Alternatively, you can manually write it using an XML editor, as demonstrated throughout this book.

With your entities and mapping file in place, you can let NHibernate's `hbm2ddl` tool generate the database schema for you, using the mapping metadata. This tool is part of the NHibernate library. It isn't a graphical tool; you access the features from your own code by calling methods on the `NHibernate.Tool.hbm2ddl.SchemaExport` class.

When you create your mapping with attributes or XML, you can add elements that help `SchemaExport` create a database schema to your liking. These are optional; without them, NHibernate will attempt to use sensible defaults when creating your databases schema. If you decide to include extra mapping metadata, having the ability to override naming strategies, data types, column sizes, and so on can be useful. Sometimes it's *necessary*, especially if you want your generated schema to follow house rules or your DBA's requirements.

NOTE Using naming strategies was explained in section 3.4.7. This feature lets you change the way entities' names are converted into tables names.

We'll now look at how you can prepare the mapping metadata to control database schema generation.

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)"/> ❶
    <generator class="uuid.hex"/>
  </id>
  <property name="Name" type="String">
    <column name="NAME"
      not-null="true"
      length="255"
      index="IDX_ITEMNAME"/> ❷
  </property>
  <property name="Description">
```

```

        type="String"
        column="DESCRIPTION"
        length="4000" /> ❸
    <property name="InitialPrice"
        type="MonetaryAmount">
        <column name="INITIAL_PRICE" check="INITIAL_PRICE > 0" /> ❹
        <column name="INITIAL_PRICE_CURRENCY" />
    </property>
    <set name="Categories" table="CATEGORY_ITEM" cascade="none">
        <key>
            <column="ITEM_ID" sql-type="char(32)" /> ❺
        </key>
        <many-to-many class="Category">
            <column="CATEGORY_ID" sql-type="char(32)" />
        </many-to-many>
    </set>
    ...
</class>

```

hbm2ddl automatically generates an NVARCHAR typed column if a property (even the identifier property) is of mapping type String. You know the identifier generator `uuid.hex` always generates strings that are 32 characters long; you use a CHAR SQL type and set its size fixed at 32 characters ❶. The nested `<column>` element is required for this declaration because there is no attribute to specify the SQL data type on the `<id>` element.

The `column`, `not-null`, and `length` attributes are also available on the `<property>` element; but because you want to create an additional index in the database, you again use a nested `<column>` element ❷. This index will speed your searches for items by name. If you reuse the same index name on other property mappings, you 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)` ❸.

The custom user-defined type `MonetaryAmount` requires two database columns to work with. You have to use the `<column>` element. The `check` attribute ❹ 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 your association table `CATEGORY_ITEM` would be `NVARCHAR(32)` instead of the more appropriate `CHAR(32)` type ❺.

We've grouped all attributes relevant for schema generation in table 9.1; some of them weren't included in the previous `Item` mapping example.

After you've reviewed (probably together with a DBA) your mapping files and added schema-related attributes, you can create the schema.

Table 9.1 XML mapping attributes for `hbm2ddl`

Attribute	Value	Description
<code>column</code>	<code>string</code>	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. You can change this behavior by implementing a custom <code>INamingStrategy</code> ; see section 3.4.7.
<code>not-null</code>	<code>true/false</code>	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.
<code>unique</code>	<code>true/false</code>	Forces the generation of a single-column UNIQUE constraint. Available for various mapping elements.
<code>length</code>	<code>integer</code>	Can be used to define a "length" of a data type. For example, <code>length="4000"</code> for a <code>string</code> mapped property generates an <code>NVARCHAR(4000)</code> column. Also used to define the precision of decimal types.
<code>index</code>	<code>string</code>	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.
<code>unique-key</code>	<code>string</code>	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. A <code><column></code> element-only attribute.
<code>sql-type</code>	<code>string</code>	Overrides <code>hbm2ddl</code> 's automatic detection of the SQL data type; useful for database specific data types. Be aware that this effectively prevents database independence: <code>hbm2ddl</code> will automatically generate a <code>VARCHAR</code> or <code>VARCHAR2</code> (for Oracle), but it will always use a declared SQL-type instead, if present. Can only be used with the dedicated <code><column></code> element.
<code>foreign-key</code>	<code>string</code>	Names a foreign-key constraint, available for <code><many-to-one></code> , <code><one-to-one></code> , <code><key></code> , and <code><many-to-many></code> mapping elements. Note that <code>inverse="true"</code> sides of an association mapping aren't considered for foreign key naming—only the non-inverse side. If no names are provided, NHibernate generates unique random names.

CREATING THE SCHEMA

The `hbm2ddl` tool is instrumented using an instance of the class `SchemaExport`. Here's an example:

```
Configuration cfg = new Configuration();
cfg.Configure();
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.Create(false, true);
```

This example creates and initializes an 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 that create 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);
    public SchemaExport(Configuration cfg, IDictionary
        connectionProperties);
    public SchemaExport SetOutputFile(string filename);
    public SchemaExport SetDelimiter(string delimiter);
    public void Create(bool script, bool export);
    public void Drop(bool script, bool export);
    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);
}
```

Specifies database-connection properties

Writes generated script to this file

Sets SQL end-of-statement delimiter

Runs create-schema script

Executes drop and create DDL scripts

Runs drop-schema script

Table 9.2 explains the meaning of the parameters of the `Execute()` methods.

Table 9.2 `hbm2ddl.SchemaExport.Execute()` parameters

Parameter	Description
<code>script</code>	Outputs the generated script to the console
<code>export</code>	Executes the generated script against the database
<code>justDrop</code>	Only drops the tables and cleans the database
<code>format</code>	Formats the generated script nicely instead of using one row for each statement
<code>connection</code>	Specifies the opened database connection to use when export is true
<code>exportOutput</code>	Outputs the generated script to this writer

This tool is indispensable when you're applying TDD (explained in 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'll get a fresh, up-to-date database to work on. Note that it's also available as an NAnt task: `NHibernate.Tasks.Hbm2DdlTask`. For more details, read its API documentation.

Using this tool throughout a project requires some thought, because the database is re-created each time—which scraps any data. We'll describe some workarounds in section 9.1.4.

EXECUTING ARBITRARY SQL DURING DATABASE GENERATION

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.

You write these statements in `<database-object>` elements. If they're in the `<create>` sub-element, they're executed when creating the database. Otherwise, they're in the `<drop>` sub-element, and they're executed when dropping the database.

Here's 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 required to create and drop the stored procedure used at the end of section 7.6.2.

Because these SQL statements can be dialect-dependent, it's also possible to use `<dialect-scope>` to specify the dialect for which they must be executed. In the previous example, the code is executed only on SQL Server databases.

The top-down approach is a comfortable route for many developers, especially when working on a new project with no existing database in place. We now look at the middle-out approach, where you generate entities from the mapping file.

9.1.2 Middle out: generating entities from the mapping

Referring back to figure 9.1, this approach starts in the middle box: the mapping documents. These provide sufficient information to deduce the DDL schema *and* to generate working POCOs for your domain model. NHibernate has tools to support this *middle-out development*, so you can take your handwritten NHibernate mapping documents and generate the DDL using `hbm2ddl`, and you can generate the .NET domain-model code using code-generation tools. This approach is appealing primarily when you're migrating from existing NHibernate mappings (used by a .NET application).

When it comes to generating entities from your mapping documents, NHibernate provides a tool called `hbm2net`. It's similar to `hbm2ddl`; but it's available as a separate library along with a console application (`NHibernate.Tool.hbm2net.Console`) and an NAnt task (`NHibernate.Tasks.Hbm2NetTask`).

Before using this tool, you must make sure your mapping provides all the required information, such as 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's the equivalent using its console application:

```
NHibernate.Tool.hbm2net.Console.exe
--config=hbm2net.config --output=DomainModel *.hbm.xml
```

This code generates a C# class for each mapping document in the current directory and saves it in the DomainModel directory. The content of the hbm2net.config file looks like this:

```
<?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 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 isn't as complete as Hibernate's `hbm2java`; refer to the latter's documentation for more details.

Most .NET developers feel more comfortable using top-down development with an attribute library like `NHibernate.Mapping.Attributes`, which gives maximum control; or, they prefer to use *bottom-up development* when there is an existing data model.

9.1.3 Bottom up: generating the mapping and the entities from the database

Bottom-up development begins with an existing database schema and data model. It's depicted as the Database Schema box in figure 9.1 In this case, you use code-generation tools to generate the mapping files and .NET code from the metadata of the database schema.

The following tools are known for their ability to generate NHibernate mapping documents and skeletal POCO persistent classes (data containers with fields and simple implementation of properties, but no logic):

- *MyGeneration*—A free .NET code generator. See <http://www.mygenerationsoftware.com/>. Comes with a simple NHibernate template that you can customize at will.
- *CodeSmith*—Similar to MyGeneration; available in free and commercial editions. See <http://www.codesmithtools.com/>.
- *ActiveWriter*—A work-in-progress Visual Studio add-in; see <http://www.altinoren.com/activewriter/>. Has the benefit of being visually appealing, and provides entities with `ActiveRecord` attributes.

What if I have an existing database and an existing class model?

We call this the *meet-in-the-middle* approach. It isn't shown in figure 9.1, but essentially you have an existing set of .NET classes and an existing database schema. As you can imagine, it's hard to map arbitrary domain models to a given schema, so this should be attempted only if absolutely necessary.

The meet-in-the-middle 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's possible to use `NHibernate.Mapping.Attributes`). This is an incredibly painful scenario that is, fortunately, exceedingly rare.

If you try to use this scenario, don't hesitate to take full advantage of the numerous extension interfaces of NHibernate. They were introduced in section 2.2.4.

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 a SQL schema.

Is it a bad thing to write anemic domain models?

As explained in section 8.1.3, 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 to other layers. In this case, your domain model is said to be *anemic* (see http://en.wikipedia.org/wiki/Anemic_Domain_Model).

This isn't necessarily a bad thing. This approach may work well for simple applications. But 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 (such as the presentation layer).

This type of code generation is generally template-based: you write a template describing your mapping and POJO with placeholders (for the names, types, and so on). The code generator executes this template for each table in the database. This process is intuitive. It's even possible to preserve hand-written regions of code that aren't overwritten when regenerating the classes. Or even better, use partial classes to separate generated code from your hand-written code.

9.1.4 Automatic database schema maintenance

Once you've deployed an application and its database for the first time, you're challenged with rolling out future changes as new versions of your software are released. This is often difficult when working with databases because you need to determine how to safely roll out all the schema changes made during development without losing data (or too much sleep). How do you know which database changes were made during development? And how can you safely apply these to live databases?

UPDATING LIVE DATABASES

NHibernate comes bundled with a tool called `SchemaUpdate`. It's used to modify an existing SQL database schema, dropping obsolete objects and adding new ones as needed. At the time of writing, `SchemaUpdate` isn't ready for use against production databases. It can potentially delete data, and it doesn't support more advanced features such as data transformations and safe column renaming. But the tool is useful during development. It can be great for keeping development and test databases in sync with your domain model, and it's faster than using `SchemaExport`, which creates your database from scratch each time. We want to discuss ways of automatically maintain live databases, so let's look at some other options.

One option is to use a commercial product that can compare live and development database schemas and then generate SQL commands to safely migrate between the two. Some tools can also handle data migration. A few recommended commercial tools include Red Gate's `Sql Compare` and `Data Compare`, `Sql Delta`, and Microsoft's `Data Professional` tools.

If you don't want to take this route, a simple solution is to manually write a SQL migration script as you develop the application and database. In this script, you keep a log of each command used to tweak the database during development. At deployment time, you can then run this migration SQL script against live databases to apply all the updates. This approach has its drawbacks. It isn't cross-database compliant, and despite being simple, it's tedious, and we don't recommend it.

Perhaps one of the best approaches to handling schema changes during both development and live deployment is to use a dedicated *migrations* library. `Migrator` is one open source example (<http://code.macournoyer.com/migrator>), as is `LiquiBase` (<http://www.liquibase.org/>); another is the migrations built into Ruby on Rails, which some people are also using with .NET. Others may also be available.

Database migrations libraries work on the principle that, each time you want to change your development schema, you do so using the migrations library. It automatically keeps track of the changes so they can be applied to any database to bring it up to date.

Here's a simple example:

```
DatabaseSystem.AddUpdate(1.0, 1.1, new string[] {"SQL statements..."});
```

When a database must be updated, this system will read its current version and only execute the changes done since the last update.

These tools usually support rolling back to previous versions. Exploring these tools in full is beyond the scope of this book, but we strongly recommend that you look into them, starting with the ones we've mentioned here.

DEVELOPMENT DATABASES

Schema-maintenance problems also occur during development, even before you roll out to any live databases. A common scenario involves lots of test data, which you want to be sure is inserted correctly each time you change the schema.

Migrations libraries are an excellent choice for achieving this, but you may have decided to generate your database schema using `hbm2ddl` rather than a separate

library. With `hbm2ddl`, you can drop and re-create your test databases regularly during development, and you don't have to worry about adding, renaming, or removing things—the schema is built from scratch whenever needed. But how do you insert test data each time?

One option is to keep a bunch of SQL scripts that insert the test data into the database and run them each time you re-create the database. The downside is that you'll need to update these scripts to match each change of schema, which can be time consuming if you have thousands of insert statements.

Another option is to have a .NET program that creates entities for test purposes and then persists them to the database using NHibernate. Effectively, you're replacing the SQL script with a .NET application. One benefit is that you can lean on refactoring tools to handle changes to class properties, and you won't have to edit cumbersome SQL scripts manually. The *ObjectMother* pattern lends itself well to this approach, where you have an object dedicated to creating test and reference data that can be used by several tests. You can learn more about that at <http://martinfowler.com/bliki/ObjectMother.html>.

So far, this chapter has focused on top-down, middle-out, and bottom-up approaches to developing your application with NHibernate that let you start with an existing domain model, some mapping files, or an existing database schema. We've also introduced you to the concept of migrations and how they can help you manage your evolving database throughout the development process. We'll now look more at the bottom-up scenario discussed in 9.1.3, in which you start the project with an existing database schema. In particular, we'll focus on *legacy* databases whose schema you're often unable to change to fit your needs. This scenario often comes with its own set of problems, so we'll explain how you can tackle some of them when working with legacy schemas.

9.2 Legacy schemas

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 should make as few changes to the existing schema as possible. Every change you make can 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:

- Use of natural (especially composite) keys
- 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 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 interface `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

Your `USER` table has a synthetic primary key, `USER_ID`, and a unique key constraint on `USERNAME`. Here's a portion of the 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 encounter a USER table in a legacy schema, USERNAME is probably the primary key. In this case, you have no synthetic identifier; instead, you 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>

```

You can no longer 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—because it's assigned by the application. Instead, you 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";
user.Firstname = "John";
user.Lastname = "Doe";
session.SaveOrUpdate(user);
System.Console.WriteLine( session.GetIdentifier(user) );
session.Flush();

```

Assigns "john" as primary key

Saves rather than updates

Prints "john", identifier of object

But you 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 doesn't declare a version or timestamp property, it's more difficult to get `SaveOrUpdate()` and cascades to work correctly. You can 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, and 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 your user table consisted of USERNAME and ORGANIZATION_ID. You 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();
user.Username = "john";
user.OrganizationId = 37;
user.Firstname = "John";
user.Lastname = "Doe";
session.SaveOrUpdate(user); // will save, since version is 0
session.Flush();
```

But what object could you use as the identifier when you called `Load()` or `Get()`? It's possible to use an instance of the `User`:

```
User user = new User();
user.Username = "john";
user.OrganizationId = 37;
session.Load(user, user);
```

In this code snippet, `User` acts as its own identifier class. Note that you now have to implement `Equals()` and `GetHashCode()` for this class (and make it `Serializable`). You 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 you implement `Equals()` and `GetHashCode()` correctly, because 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 uppercase/lowercase strings). Composite key classes are also expected to be `Serializable`.

Now you'd remove the `UserName` and `OrganizationId` properties from `User` and add a `UserId` property. You'd use the following mapping:

```
<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" />
  ...
</class>
```

You could save a new instance using this code:

```
User user = new User();
user.UserId = new UserId("john", 42);
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 ORGANIZATION_ID was a foreign key to the ORGANIZATION table, and that you wished to represent this association in your C# model. Our recommended way to do this would be 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 you may of course read it with john.Organization.

An alternative approach would be to use a <key-many-to-one>:

```
<class name="User" table="USER">
  <composite-id name="UserId" class="UserId">
    <key-property name="UserName"
      column="USERNAME" />
    <key-many-to-one name="Organization"
      class="Organization"
      column="ORGANIZATION_ID" />
  </composite-id>
  <version name="Version"
    column="VERSION"
    unsaved-value="0" />
  ...
</class>
```

But 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

Because 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. You 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 a SQL NULL. It's straightforward to use 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 section 6.1, "Creating custom mapping types," carefully.

We'll look at a slightly more interesting problem. So far, your User class has two properties to represent a user's names: Firstname and Lastname. As soon as you add an Initial property, your User class will become messy. Thanks to NHibernate's component support, you can easily improve your model with a single Name property of a new Name C# type (which encapsulates the details).

Also suppose that the database includes a single NAME column. You 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):

```

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` can be a custom type. (This includes associations—an NHibernate `ManyToOneType`, for example, delegates to the identifier type of the associated class, which may be a custom type.)

One further problem often arises in the context of working with legacy data: integrating database triggers.

9.2.4 Working with triggers

There are some reasonable motivations for using triggers even in a brand-new database; 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 you 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, because the value is assigned after the INSERT of the item row. If you need to use the value in your application, you have to tell NHibernate explicitly to reload the object with a new SQL SELECT:

```
Item item = new Item();
//...
NHibernateHelper.BeginTransaction();
ISession session = NHibernateHelper.Session;
session.Save(item);
session.Flush();
session.Refresh(item);
System.Console.WriteLine( item.Created );
NHibernateHelper.CommitTransaction();
NHibernateHelper.CloseSession();
```

Forces insert

Reloads object with SELECT

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. Because no snapshot is available when a detached object is reassociated 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, you 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 a 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. But 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 an 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 may save you many days in the long run—when workarounds and special cases become a burden.

Now that you're finished developing and mapping the data side of the domain model, it's time to dig into its behavior: specifically, how much it's supposed to know about persistence.

9.3 Understanding persistence ignorance

In the description of the layers of an NHibernate application (section 8.1.3), we highlighted the fact that the domain model shouldn't depend on any other layer or service (although this isn't a strict rule). This is important because it influences its portability; the less coupling an entity has, the easier it is to modify, test, and reuse.

This recommendation leads to the notion of *persistence ignorance* (PI). A persistence-ignorant entity has no knowledge of the way it's persisted (it doesn't even know that it can be persisted). Practically speaking, the entity doesn't have methods like `Save()` or static (factory) methods like `Load()`, and it doesn't have any reference to the persistence layer. This is already the case for the entities you've been writing in this book.

Going one step further, we can also say that entities shouldn't have `Identifier` and `Version` properties. The argument is that primary keys and optimistic control have nothing to do with the business domain, and therefore don't belong in the domain model. We usually wouldn't go this far; the convenience of having these properties far outweighs the slight "pollution" of the domain model they introduce.

Note that PI isn't a requirement for all solutions, and you may find it easier to develop solutions without it. However, we do consider PI a good thing to strive for as it creates a less coupled, more testable and maintainable domain model. It's particularly useful when the domain model explicitly requires portability and flexibility.

Now let's see how you can implement an entity that is as free as possible of persistence-related code while still being functional and simple.

9.3.1 Abstracting persistence-related code

A common compromise, at the level of persistence awareness, is to separate persistence-related code from the business code in the implementation of an entity. This can be as trivial as performing a visual separation using a `#region` in your code, to help improve readability. Another option is to create an abstract base class for each entity so that persistent code is separated.

Let's look at how you can implement the latter solution. You'll use `NHibernate.Mapping.Attributes` because it lets the base class abstract the mapping information along with the code. You'll see that the end result can be acceptable as long as you

don't mind inheriting from this base class (if you do mind, 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.

You'll implement an abstract class from which entities can inherit 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. You'll call this class `VersionedEntity`; listing 9.2 shows its implementation.

Listing 9.2 `VersionedEntity` base class abstracting persistence-related code

```
[Serializable]
public abstract class VersionedEntity {
    private Guid id = Guid.NewGuid();
    private int version = 0;
    [Id(Name = "Id", Access = "nosetter.camelcase-underscore")]
    [Generator(1, Class = "assigned")]
    public virtual Guid Id {
        get { return id; }
    }
    [Version(Access = "nosetter.camelcase-underscore")]
    public virtual int Version {
        get { return version; }
    }
    public override string ToString() {
        return GetType().FullName + "#" + Id;
    }
    public override bool Equals(object obj) {
        VersionedEntity entity = obj as VersionedEntity;
        if (entity == null) return false;
        return Id == entity.Id;
    }
    public override int GetHashCode() {
        return Id.GetHashCode();
    }
}
```

Using an assigned `Guid` as identifier ❶ provides many advantages. For example, it simplifies the implementation of `Equals()` and `GetHashCode()`. The version ❷ is used for optimistic concurrency control, explained in section 5.2.1. The implementations of `System.Object` methods ❸ are simple but effective.

Note that you can replace the initialization of the identifier as follows:

```
private Guid id
    = (Guid) new NHibernate.Id.GuidCombGenerator().Generate(null, null);
```

This initialization uses the `guid.comb` identifier generator. You can read about its advantages in table 3.5 of chapter 3, section 3.5.3.

If you don't want to reference `NHibernate` here in your business layer, you can create a private static method in `VersionedEntity` using the same algorithm as the method `GuidCombGenerator.GenerateComb()`. Remember that `NHibernate` is licensed under the LGPL; therefore, all of its source code is publicly available for viewing and customization. (See http://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License.)

IMPLEMENTING PERSISTENCE-ABSTRACTED ENTITIES

When inheriting from this `VersionedEntity` base class, all the basic persistence-related features are neatly taken care of (identifier, the version, and the overloading of `System.Object` methods). But we still have to map our business properties. For that, we have a few choices: XML Mappings or attribute-based mappings. Another option called Fluent NHibernate also looks promising, but because it's a work in progress we won't discuss it here.

You may ask whether the use of `NHibernate.Mapping.Attributes` decreases the persistence ignorance of your entities. After all, mapping attributes is about mapping, which is a persistence concern rather than a domain concern. Do we want all those attributes in our domain models? Like all things, it's a trade-off. The pros and cons of attributes are discussed in section 3.3.2.

Let's use a simple example to illustrate the documentation aspect of these attributes:

```
[Class]
public class User : VersionedEntity {
    private string name;
    [Property(Length = 64, Access = "field.camelcase-underscore")]
    string Name { ... }
}
```

Without the information `Length = 64`, a careless developer may think that names can be unlimited in length—and the user will find that the application truncates a name for an unknown reason.

NOTE You can see that using `VersionedEntity` makes this implementation free of code unrelated to the business domain, without sacrificing functionality.

The fact that the domain model isn't aware of other layers (like the presentation layer) means that it can't directly inform those layers about any events that occur (for example, when a change occurs in the domain model, the GUI may need to be refreshed). Fortunately, a pattern is available to solve this kind of problem.

9.3.2 Applying the Observer pattern to an entity

The Observer pattern lets an object pass information to other objects without knowing about them up front. The object that sends the notifications is called the *subject*, and the objects that receive the notifications are the *observers*. This pattern is often used in a WinForms MVC architecture, as explained in section 8.1.1.

In .NET, you can implement this pattern using events. You add an event to your class, and then the observers must register with the event in order to receive notifications. Most of the time, the registration is done just after the entity is created or loaded.

Let's look at an example that illustrates how to implement this pattern. In the previous example, the class `User` has a property `Name`. If you want to inform the presentation layer when this property changes, this 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, you assume that the entity has access to the presentation layer, which provides a method to call when the entity changes. The problem, in this implementation, is that the entity is tied to the presentation layer—and that’s bad because you can’t use the entity in any other context (for example, when testing).

Here’s 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();
        }
    }
    public event NameChangedEventHandler NameChanged;
    protected virtual void OnNameChanged() {
        if (NameChanged != null)
            NameChanged(this, EventArgs.Empty);
    }
}

```

You first define a delegate for the `NameChanged` event. Then, in the implementation of the property (`Name`), you raise the event after changing the property’s value. The code to raise the event is in the `OnNameChanged()` method. Using a separate method is a recommended guideline from the official .NET documentation, which discusses the implementation and use of events.

The next step is to listen to the event:

```

User user = BusinessLayer.LoadUser(userId);
user.NameChanged += User_NameChanged;

```

In this code, you load a user and register the `NameChanged` event. The method `User_NameChanged()` will be called whenever this property changes.

Note that the .NET framework provides an `INotifyPropertyChanged` interface for this scenario. Here’s an implementation of the `User` class 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. But it has the benefit of working well with other mechanisms in the .NET framework, such as data binding.

You can also use the Observer pattern in many other situations. Here's a security-related example:

```

public class SecurityService {
    public void User_IsAdministratorChanging(object sender, EventArgs e) {
        if ( ! loggedUser.IsAdministrator )
            throw new SecurityException("Not allowed");
    }
}

```

The security service listens to the `User.IsAdministratorChanging` event. This service is able to cancel a modification by throwing an exception if the logged user isn't an administrator.

This section has explained how to avoid cluttering the domain model with unrelated concerns. Now, let's talk about the domain model's primary concern: the business logic.

9.4 Implementing the business logic

In this book, we use the term *business logic* for any code that dictates how entities should behave. It defines what can be done with the entities and enforces business rules on the data they contain. Note that we aren't strictly speaking about the domain model, but about the business layer in general.

The business layer can contain many kinds of business logic. We'll use a case study to cover them all. Let's say you want to implement a subsystem of the *CaveatEmptor* application that lets a user place a bid on an item; when this is done, all other bidders are notified of the new bid via email.

You can do that in the *Bid* or *Item* entity because sending emails isn't their responsibility. The business layer should do it. Here's an example:

```

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 a few details. This method takes the identifier of the `item` and a `bid` object. This might be convenient if you were calling the method from an ASP.NET page where these `item` identifiers would be included in the GET or POST request. In its implementation, `PlaceBid` loads the `item` using a DAO called `ItemDAO`. (For more details about the DAO pattern, see section 10.1.)

After loading the `item`, you 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. But in this scenario you know you'll definitely be using the `Bids` collection, so you can save trips to the database by eagerly loading the collection using the `Load()` method.

Let's dissect this method to see how various kinds of business logic are executed when it's called.

9.4.1 *Business logic in the business layer*

The method `BusinessLayer.PlaceBid()` should contain logic that belongs in the business layer. For example, it's common for the business layer to contain rules related to security and validation:

```

BusinessLayer.PlaceBid(int itemId, Bid bid) {
    If ( loggedUser.IsBanned )
        throw new SecurityException("Not allowed");
    // ...
}

```

Here, before placing the bid, you make sure the logged user isn't banned.

It's also common to use the `IInterceptor` API when a business rule is hooked to the persistence of entities. Read section 8.4 to see how it's done. For complex business rules, you may consider using a rules engine.

The remaining business logic belongs in the domain model.

9.4.2 *Business logic in the domain model*

The business logic in an entity expresses what the entity is supposed to do from a core domain point of view. 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 illustrates the different kinds of business logic. You'll see this method again in chapter 10, where we discuss more architectural decisions.

In this implementation, you start with some guard clauses, and then you perform the action itself (assigning the bid to the item). The guard clauses are the `if` statements before the action takes place. They prevent the user from doing something that goes against the rules of the business. In this example, you use guard clauses to prevent someone placing a lower bid than the current one or placing a bid on an item that has already been sold.

Sometimes, business logic must be executed at a specific time. For example, if some validation logic must be performed before saving an entity, you can add a `Validate()` method to the entity. Suppose you already have code such as this:

```
public string Name {
    get { return name; }
    set {
        if ( string.IsNullOrEmpty(value) )
            throw new BusinessException("Name required.");
        if (name==value) return;
        name = value;
    }
}
```

You can have an additional method that re-checks all the logic without duplicating code:

```
public void Validate() {
    Name = Name;
    Password = Password;
    ...
}
```

By setting the properties in the `Validate()` method, the business rules are re-checked. Note that this trick only works if the validity checks are done in a specific order. Specifically, things like

```
if ( string.IsNullOrEmpty(value) )
```

must come before

```
if (name==value) return;
```

Once your `Validate()` method has validated each individual property, you can add validations that work on multiple properties. For example, in a holiday booking application, you might check that an outbound flight date comes before an inbound flight date.

When you're implementing the domain model's business logic, you must be careful to avoid unwanted dependencies. You can often move these up to an upper layer, such as the business layer. This is better explained with an example:

```
public string Password {
    if ( CryptographicService.IsNotAStrongPassword(value) )
        throw new BusinessException("Not strong enough.");
}
```

```

        password = value;
        MailingService.NotifyPasswordChange(this);
    }

```

Here you can see the domain model doing too much; it shouldn't depend on cryptographic and mailing services. Instead, these concerns are better suited to the business layer.

There are also some *rules* that shouldn't be implemented in the domain model, or anywhere else in the application for that matter.

9.4.3 Rules that aren't business rules

Some rules shouldn't 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 generally test the code, to make sure that everything went as expected. Here's an example:

```

public void PlaceBid(int itemId, Bid bid) {
    //...
    item.PlaceBid(bid);
    if ( ! item.Bids.Contains(bid) )
        throw new Exception("PlaceBid() failed.");
    //...
}

```

Some would say that this is fine, because the code is testing whether the post-conditions of the action are as expected. In this case, you're testing that the item contains the bid after `PlaceBid` is called. Post-condition checking is common with programmers who employ Design by Contract [Meyer, Bertrand]. We usually adopt a different approach: we put post-conditional checks such as this into a separate *test*.

Let's take another example:

```

[Test]
public void LoadMinAndMaxBids() {
    //...
    min = BidDAO.LoadMinBid(item);
    max = BidDAO.LoadMaxBid(item);
    Assert.LessThan(min, max);

    //...
}

```

This is a unit test for the persistence layer. If the `Assert` fails, it means that there is a bug in its implementation. We cover testing in more detail in section 8.1.

So far, you've implemented the internal structure of the domain model, taking care of its data and its behavior. Now it's time to address some issues related to the environment in which this domain model is used.

9.5 Data-binding entities

The presentation layer allows the end user to display and modify the entities of an NHibernate application. This implies that the data inside your entities is displayed

using .NET GUI controls and that the user's input is sent back to the entities to perform updates to the data.

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 changes to one of them are reverberated to the other. This is called *data binding*. In the context of NHibernate, these objects are called POCOs rather than entities, to emphasize the fact that they don't have any special infrastructure to assist data binding. Developers who are used to DataSets (and the wizards of Visual Studio .NET) many find POCO data binding challenging. DataSets contain special infrastructure to make data binding easier, and POCOs are generally free of this additional infrastructure. Fortunately most .NET GUI controls support basic data binding to POCOs, and .NET provides interfaces that allow you to improve this support.

In this section, we'll discuss a number of alternatives for data binding; these alternatives apply equally to Windows and web applications. We'll first explain how you can interact with .NET GUI controls without using data binding. Then you'll data bind POCOs and learn about a number of extensions that improve these capabilities. You'll also see how NHibernate can help implement data binding. Finally, you'll discover a library that can help you data bind POCOs.

A POCO includes three kinds of data: a simple property (that 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'll ignore the reference to another POCO because it's generally visualized by displaying one of its properties (its name, for example). An additional mechanism (such as a button) is provided to view or edit the related entities.

In order to cover how the simple properties and collections can be data bound, we'll use the example of writing a form to manage users and their billing details (as defined for the auction application in section 3.1.2). This form, shown in figure 9.2, will retrieve the user's information and let you update the billing details.

The interesting aspect of this example is that *BillingDetails* is an abstract class, so the entities in the collection can be either *BankAccount* or *CreditCard* instances. This complication will let us demonstrate the limitations of some approaches to data binding, discussed next.

Note that we don't give a thorough explanation of the .NET APIs you'll use; if you need to learn more about them, refer to the official .NET documentation. You may also want to read *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 simple: you copy the POCO data from/to the GUI. When you need to display something, you take it from the POCO and send it to the GUI:

```
editName.Text = user.Name;
```

Figure 9.2 Domain model bound to a user interface

When you need to process the POCOs, you retrieve any changes in the GUI and apply them back in the POCOs:

```
user.Name = editName.Text;
```

This approach is simple to understand and implement. It's also easy to customize. For example, when displaying an identifier (of the type `integer`), you may decide to display *New* for a transient entity (instead of 0).

It's also straightforward to support polymorphism:

```
if(billingDetails is BankAccount) {
    //...
    editBankName.Text = (billingDetails as BankAccount).BankName;
}
else {
    //...
    editExpYear.Text = (billingDetails as CreditCard).ExpYear.ToString();
}
```

The downside of manual data binding is that it can be tedious to implement, especially for complex objects.

9.5.2 Using data-bound controls

In this case, you rely on the support of built-in data binding for public properties and collections. Here's an example for Name:

```
textBoxName.DataBindings.Add("Text", user, "Name");
```

In this example, the Windows Forms control `textBoxName` is data bound to the property `Name` of the `User` instance. When binding collections, you can use the control's `DataSource` property:

```
dataGridView.DataSource = user.BillingDetails;
```

The `DataGridView` control uses the `BillingDetails` collection as data source. But this solution is limited: for example, it doesn't support polymorphism, which means you can only edit the properties of the class `BillingDetails`. You can't edit the properties of the subclasses `BankAccount` and `CreditCard`.

You can use numerous helper classes and extensions to improve this support: `ObjectDataSource`, `BindingSource`; `BindingList`, `IEditableObject`, `INotifyPropertyChanged`, and so on. We suggest that you look at these APIs to see which ones suit your needs.

If you value simplicity in your domain model and still want to do powerful data binding, you can implement wrapping classes (using the Adapter pattern) that 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 the focus on its business value, and the wrapper provides data-binding capabilities on top of the entity. They also mean you have to do more work, because you have two classes to implement instead of one.

Another benefit of using wrapper classes is that you can add properties for reporting purposes. A common example is to add a `FullName` property that returns the first name and the last name of a `User` as a single string.

9.5.3 Data binding using NHibernate

If you think about the way NHibernate works, you'll realize that it already does a kind of data binding. But instead of binding an object to the GUI, it binds the object to the database. When you load an entity, it fills the entity with data; and when you save the entity, it pushes the data back to the database.

The part of NHibernate responsible for this is the `MetaData` API. You can leverage this API to help automate binding entities to a GUI. The following code is based on that previously shown in section 3.4.10, where we discussed working with `MetaData` in more depth:

```
User user = UserDAO.Load(userId);
NHibernate.IMetadata interface 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 a user's data and generates labels and text boxes to display it. For brevity, we haven't written code to set the position of these controls on the form.

The interface `IMetadata` 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 aren't acceptable in production application. Even with a well-designed algorithm, the resulting layout of the GUI is far from perfect. You may have problems with the formatting of the values (such as dates). There are better controls than `TextBox` for some types of data (for example, `DateTimePicker`). Finally, this approach requires extra work to support references to other POCOs and collections.

It's possible to solve these issues with some effort; and this approach can help when you're 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's largely outside the scope of this book to explore this library, but it's worth mentioning that it supports data binding of both individual POCOs and collections.

At the time of writing, ObjectViews is based on .NET 1.1 and won't evolve further. You can download this library (with a helpful example application) from <http://sourceforge.net/projects/objectviews/>.

9.6 Filling a DataSet with entities' data

DataSets are widely used, mostly by data-centric applications leveraging wizards in tools like Visual Studio .NET to generate code. But they're different than POCOs. If, for some reason, your domain model must communicate with a component using DataSets, you'll have to find a solution to this problem.

Before you begin, remember that you can execute classic ADO.NET code by either opening a database connection yourself, or by using the one NHibernate has. NHibernate's connection can be accessed using the `ISession.Connection` property. In this case, you'll have to be careful not to work with stale data or change something without clearing the related NHibernate second-level cache. You may also consider rewriting the component using DataSets for better consistency. If neither of these options is applicable, you'll have to convert your entities from/to DataSets.

In the following sections, we'll consider going from entities to a DataSet filled with their data. You shouldn't have any problem reversing this process.

9.6.1 Converting an entity to a DataSet

A DataSet is an in-memory data container that mimics the structure of a relational database. Filling it means adding *rows* to its *tables*. It's relatively easy to figure out the code required to fill a DataSet. Here, we assume you're working with a typed DataSet, because they're easier to work with.

Listing 9.3 contains a method that does this work for the `Item` entity. It's complete because it handles the simple properties, the `Seller` reference 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;
    adding.Add(item);
    TypedDataset.ItemRow row;
    if ( dataset.Item.Rows.Contains(item.ItemID) )
        row = dataset.Item.FindByItemID(item.ItemID);           2
    else
        row = dataset.Item.NewItemRow();
    row.ItemID = item.ItemID;
    row.Name = item.Name;           3
    //...
    if ( item.Seller == null )
        row.SetSellerIDNull();
    else {
        if ( ! dataset.User.Rows.Contains(item.Seller.UserID) )           4
            // ...
        }
    }
}
```

```

        FillDataSet(dataset, item.Seller);
        row.SellerID = item.Seller.UserID;
    }
    if ( NHibernateUtil.IsInitialized(item.Bids) )
        foreach (Bid bid in item.Bids)
            FillDataSet(dataset, bid);
    if ( ! dataset.Item.Rows.Contains(item.ItemID) )
        dataset.Item.AddItemRow(row);
    adding.Remove(item);
}

```

You use a collection to keep the list of entities that are currently being added ❶. 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; otherwise, its row must be created ❷.

Filling the simple properties is straightforward ❸. Handling references to other entities is more complex: you must either remove the entity or add it if it isn't in the DataSet yet ❹.

Handling collections requires that you first make sure the collection is already loaded (unless, in this case, you want it to be lazy loaded). Then you add the bids one by one ❺. You add the row to the DataSet if it doesn't already contain the entity ❻. Finally, you remove the entity from this collection ❼ because we've processed it.

If you're using a code generator as explained earlier in this chapter, you may be able to generate this code for all your entities. Doing so will save you a lot of time.

Now let's see how NHibernate can help you achieve the same result more quickly.

9.6.2 Using NHibernate to assist with conversion

If you're working with a non-typed DataSet, you can use an approach similar to the one explained in section 9.5.3: you can extract the class names and the property names and use them as table names and column names.

NHibernate approximates this idea with the `ToString(object entity)` method of the `NHibernate.Impl.Printer` class. Look at it before starting your implementation.

Succeeding in implementing this approach means it's generic enough to work with any entity, because you'll only be manipulating metadata. But it also means the domain model dictates the schema of the DataSet. You can solve this issue by using the mapping between the domain model and the database (because the DataSet schema is generally based on the domain model).

With this ability to communicate with a component using DataSets, you're finished implementing a real-world domain model.

9.7 Summary

Writing real-world domain models can be tricky because of the influence of the environment. We hope this chapter has helped you understand the process.

The first step is to implement the domain model and the database and write the mapping between them. Until this chapter, you wrote them manually; now, you know how to generate them. You even know how to automate the migration of the database as the domain model evolves.

This chapter explained how to handle legacy databases when you're writing the mapping. NHibernate supports the mapping of natural and composite keys. As a last resort, you can implement user types to handle custom situations. It's also possible to work with a database using triggers.

After explaining how to implement and map the domain model's data, we moved to its business logic. We explained what persistence ignorance means and how to write a clean domain model that's free of unwanted dependencies. Then, we explained how the different kinds of business logic should be implemented. We also gave you some advice about errors to avoid.

When you've completed the domain model, you need to display it. This is where data binding comes into play. As you saw, doing it correctly can require quite a bit of work.

We completed this chapter by looking at how you can obtain a `DataSet` from an entity's content. Although this process may require a lot of time at first, it can be automated.

This chapter was just an introduction to the real world of domain models. You may need to do some research to find the perfect answer for your needs, and we hope the resources we've mentioned will keep you busy for a while.

Now, it's time to move to the persistence layer. So far, you've been writing simple, short persistence operations that act on similar entities. Let's step back and look at the architectural issues that accompany writing a functional persistence layer in the real world.

10

Architectural patterns for persistence

This chapter covers

- Designing the persistence layer
- Implementing reusable Data Access Objects
- Implementing conversations
- Supporting Enterprise Services transactions

And so, you've finally arrived at the last chapter. We've touched on many topics along the way, and you should now feel comfortable about using NHibernate to implement persistence in your applications. You should also be roughly familiar with the breadth of features available in NHibernate and understand the flexibility they give you. We've also discussed layered architecture, which will help you build maintainable applications where concerns are neatly separated.

With all 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. We've discussed domain models, but so far we haven't addressed the persistence layer in much depth. In chapter 2, you may recall using simple function calls to load, save, and update your entities. These types of examples are great for quickly

explaining concepts; but in a real-world application, you'll benefit from something more structured and coordinated.

This chapter starts with the presentation of the *Data Access Object* (DAO) pattern. It's a popular pattern that deals with the organization of the persistence layer. We'll take this pattern and demonstrate how you can build a neat, structured persistence layer that is both generalized and reusable.

You'll also learn the basics of session management, which is an important (and somewhat challenging) aspect of working with NHibernate. Following that, we'll return to the interesting topic of conversations (introduced in chapter 5) and show practical examples of the various ways you can implement conversations with NHibernate.

In the real world, you may be using NHibernate as part of a larger system. This chapter will end with a discussion of distributed applications. It will explain how to make an 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 (or Mono). It may run in an ASP.NET application, WPF, WCF, a Windows Forms, or a Console application. It may even be used inside a web service or Windows service.

These environments are similar as far as NHibernate is concerned; only a few changes are required to port an 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 here, 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 needs and goals. For this reason, our examples are written in 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 and maintainable by grouping functionality 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 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) because 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 exist only if it's required, because layers increase the complexity and cost of development. But we 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. The example is based on a console application, but it will be easy to reuse this persistence layer in another (web or Windows) application.

We'll use the simple "place bid" use case from the CaveatEmptor application to demonstrate our ideas. This use case states that when a user places a bid on an item, CaveatEmptor must perform the following tasks, all in a single request:

- 1 Check that when the user enters the bid, the amount is greater than any other bids for the item (you can't bid lower than someone else!).
- 2 Check that the auction hasn't yet ended.
- 3 Create a new bid for the item.

If either of the first two 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 the *business rules*. We also have a nonfunctional requirement: if a failure occurs while accessing the database, the user should be informed that the system is currently unavailable (this is an infrastructure concern).

Let's see how you can implement this functionality, starting with an overly simple approach.

10.1.1 Implementing a simple persistence layer

In the "Hello World" application of chapter 2, the example program contained simple functions for everything related to persistence. This design doesn't scale well, and using it in larger applications would result in a sprawling, disorganized mess of functions for creating, reading, updating, and deleting entities.

In this section, we'll suggest a tidier approach, where you split the persistence layer into a number of classes, each responsible for a specific concern. The first thing we want to tackle is finding a way for the application to obtain new `ISession` instances. For this, you'll write a simple *helper* (or utility) class to handle configuration and initialization of the `ISessionFactory` (see chapter 3) and also to 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; ❶
    static NHibernateHelper() { ❷
        try {
            Configuration cfg = new Configuration();
            SessionFactory = cfg.Configure().BuildSessionFactory(); ❸
        } catch (Exception ex) {
            Console.Error.WriteLine(ex); ❹
            throw new Exception("NHibernate initialization failed", ex);
        }
    }
    public static ISession OpenSession() { ❺
        return SessionFactory.OpenSession();
    }
}
```

The `ISessionFactory` is bound to a static (and readonly) variable ❶. All your threads can share this one constant, because the `ISessionFactory` implementation is

thread-safe. This session factory is created in a static constructor ❷, and this constructor is executed the first time this helper class is accessed.

The `ISessionFactory` is built from a `Configuration` ❸; this is the same process we've demonstrated throughout the book. You catch and log exceptions ❹, but of course you should use your own logging mechanism rather than `Console.Error`. The utility class has one public method: a factory method for new `ISessions` ❺. It's a convenient method to shorten the code required for the most common usage of this class: opening new sessions.

This (trivial) implementation stores the `ISessionFactory` in a static variable. 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. Thus you can safely have as many `ISessionFactory` instances as you like. Despite this freedom, in practice you want as *few* as possible, because 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 common question that arises with NHibernate newcomers), we'll continue with the use-case implementation.

PERFORMING ALL THE OPERATIONS INSIDE THE SAME METHOD

In this section, you'll write the code that implements the "place bid" use case in a single `PlaceBidForItem()` method, shown in listing 10.2. This code can live in an ASP.NET code-behind or a function in a console application. It doesn't matter; let's assume that wherever it is, the containing program will get some user input and pass it to this method.

It's worth noting that this code sample isn't considered a good implementation, but we'll get to that shortly. It does give us a nice starting point for demonstrating the varying degrees of separation you can introduce into your applications.

Listing 10.2 Implementing a simple use case in one method

```
public void PlaceBidForItem(long itemId, long userId, double bidAmount) {
    try {
        using(ISession session = NHibernateHelper.OpenSession())           ❶
        using(session.BeginTransaction()) {
            Item item = session.Load<Item>(itemId, LockMode.Upgrade);      ❷

            if ( item.EndDate < DateTime.Now ) {                          ❸
                throw new BusinessException("Auction already ended.");    ❹
            }
            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.");              ❺
            }
            User bidder = session.Load<User>(userId);
            Bid newBid = new Bid(bidAmount, item, bidder);
            item.AddBid(newBid);
        }
    }
}
```

```

        session.Transaction.Commit(); ❹
    }
} catch (HibernateException ex) { ❺
    throw new InfrastructureException(
        "Error while accessing the database", ex );
}
}

```

You first get a new `ISession` using your utility class ❶. You then start a database transaction. The session and transaction will be closed automatically due to the `using()` statement. If you don't commit the transaction, or if this commit fails for some reason, the transaction will be automatically rolled back.

You load the `Item` from the database using its identifier value ❷ and also ask for a pessimistic lock so the database won't allow another transaction to modify the record while you're working on it. This prevents two simultaneous bids for the same item.

If the end date of the auction is earlier than the current date ❸, you 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 ❹, you check whether the database contains a higher bid for the current item. If a higher bid exists, you display an error message. Otherwise, if all checks are successful, you place the new bid by adding it to the item ❺. Note that you don't have to save it manually by calling `Save()`; it's saved using NHibernate's transitive persistence (cascading from the `Item` to `Bid`).

Committing the database transaction ❻ flushes the current state of the `ISession` to the database. A try-catch block ❼ is responsible for exceptions thrown when rolling back the transaction or closing the session; it's wrapped to abstract NHibernate details.

As we mentioned, this implementation of the `PlaceBidForItem()` method isn't necessarily a good one; it does too much of the hard work itself and takes on the responsibilities of implementing domain logic, enforcing business rules, and carrying out persistence functionality. Essentially, it does the work of a persistence layer, business layer, and domain model combined.

Let's now see if you can improve things by pushing some of that hard work into the domain model.

CREATING A "SMART" DOMAIN MODEL

The current `PlaceBidForItem()` method contains code that implements business logic. Let's move that code to its right place: the `Item` entity.

To do this, give your `Item` entity a `PlaceBid()` method :

```

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.");
    }

    Bid newBid = new Bid(bidAmount, this, bidder);

    this.AddBid(newBid);
    return newBid;
}

```

This code enforces business rules that constrain the state of your business objects, but it doesn't hold any data-access functionality. The motivation here is to encapsulate business logic in the classes of the domain model without worrying about loading and saving data. Hence, you have a separation of concerns.

You may be surprised to see that this new `PlaceBid()` method has a `maxBidAmount` parameter—surely the `Item` entity can find this information for itself. This is a matter of taste, and we'd rather pass this data in from the upper layer than ask the domain object to run queries that may require accessing the persistence layer.

Now that your domain model has taken some responsibility for itself, you can simplify the original `PlaceBidForItem()` method as follows:

```
public void PlaceBidForItem(long itemId, long userId, double bidAmount) {
    try {
        using(ISession session = NHibernateHelper.OpenSession())
        using(session.BeginTransaction()) {
            Item item = session.Load<Item>(itemId, LockMode.Upgrade);
            IQuery q =
                session.CreateQuery(@"select max(b.Amount)
                                   from Bid b where b.Item = :item");
            q.SetEntity("item", item);
            double maxBidAmount = (double) q.UniqueResult();

            User bidder = session.Load<User>(userId);
            item.PlaceBid(bidder, bidAmount, maxBidAmount);
            session.Transaction.Commit();
        }
    } catch (HibernateException ex) {
        throw new InfrastructureException(
            "Error while accessing the database", ex );
    }
}
```

Loads (and locks) item

Retrieves maximum bid amount for item

Retrieves bidder and places bid

Obviously, you were able to reduce this method because some of the work is now delegated to the domain model (`item.PlaceBid`). But this code still contains functionality that is relevant to both the persistence layer *and* the business layer; it's dealing with both saving and loading entities, and deciding *how* to construct queries to coordinate the placing of a bid.

To take things a step further, let's attempt to clearly separate these responsibilities. Remember, as your programs get bigger, combining lots of responsibilities into a single class can lead to software that is difficult to maintain and evolve. Separating responsibilities will make life much easier for you.

Many patterns are available to help you address this separation of concerns. Let's examine one of the most popular.

INTRODUCING THE DATA ACCESS OBJECT PATTERN

Mixing data access code (the 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 façade with higher-level business semantics. There is more than one way to design

this façade—some small applications may use a single class for all persistence operations; some may use a class for each operation—but we prefer the Data Access Object (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* (although they have slightly different meanings). If you’ve read *Domain-Driven Design* [Evans 2004], you may realize that the DAO pattern is similar to the *Repository* pattern described there. DAO tends to be slightly more fine-grained, having one DAO class per entity. We like both these patterns, but for this example it’s simpler to explain DAO rather than the ins and outs of domain-driven development (DDD).

To begin our explanation of the DAO pattern, 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 DAO abstracting item-related persistence operations

```
public class ItemDAO {
    public static Item FindById(long id) {           ❶
        using (ISession session = NHibernateHelper.OpenSession())
            return session.Load<Item>(id);
    }
    public static double GetMaxBidAmount(long itemId) {       ❷
        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();
        }
    }
    public static Item MakePersistent(Item entity) {         ❸
        using (ISession session = NHibernateHelper.OpenSession())
            session.SaveOrUpdate(entity);
        return entity;
    }
}
```

This class provides two static methods to perform the operations needed by your `PlaceBid()` method. The `FindById()` method ❶ loads items. Note that pessimistic locking isn’t an option here because you’re opening and closing sessions in several places, and you may want the lock to span all these operations (something we haven’t allowed for here, but we’ll get to that). To retrieve the highest bid amount, you can use the `GetMaxBidAmount()` method ❷. The `MakePersistent()` method ❸ can be used to save items.

Whether `GetMaxBidAmount()` belongs on an `ItemDAO` or a `BidDAO` is a matter of taste; but because the argument is an `Item` identifier, it seems to naturally belong

here. When you're designing any class interfaces, we encourage you not to be troubled by such decisions because modern refactoring tools make it easy to move responsibilities around if you change your mind.

Your `UserDAO` also needs a `FindUserById()` method. You should be able to figure out how to implement it (replace *Item* with *User* in listing 10.3).

As a result of all this separation, you reap the rewards of a much cleaner `PlaceBidForItem()` method:

```
public void PlaceBidForItem(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 the first implementation. Someone who knows nothing about NHibernate can understand immediately what this method does, without the need for code comments. You've also achieved a clear separation of concerns.

You may be satisfied with this improved implementation, but let's look at some of its drawbacks. First, it makes transparent persistence impossible. This is why you need to explicitly save the item at the end. The implementation also opens four sessions where a single would be enough. Finally, the implementation of several similar DAOs violates the Don't Repeat Yourself (DRY) principle, giving you redundancy for the basic CRUD operations.

These problems can all be solved by abstracting the common basic operations and by figuring out a way to make these DAOs share the same session.

Let's jump to the right solution.

10.1.2 *Implementing a generic persistence layer*

You learned in the previous section that although it's easy to implement a simple DAO, a number of key issues require a smarter solution. An ideal solution would allow all DAOs to share the same session and would minimize the amount of code repetition.

Let's examine a great solution to the first issue of shared sessions, using a feature introduced in NHibernate 1.2.

USING ISESSIONFACTORY.GETCURRENTSESSION()

The idea behind this feature is that, for a specific action, you generally need a single session. Because this session can be reused for all the operations (even if they're unrelated), it's logical to make this session available to the entire application (that is, to its persistence layer).

Your first impulse may be to create a static session like the session factory defined in listing 10.1. But this won't work for ASP.NET applications because each HTTP context should have its own NHibernate session (using a static session results in a single session for the whole web application, which is bad because sessions aren't thread safe).

It's also possible to open a session and send it to each DAO. In this case, the DAOs would no longer have static methods. You'd instantiate these DAOs and provide the session as a parameter in their constructors. This solution could work, but it's tedious having to pass the NHibernate session everywhere it may be needed.

Instead of solving this problem yourself, you can leverage the `ISessionFactory.GetCurrentSession()` method. This method returns the session instance associated with the current persistence context, similar to the ASP.NET notion of an HTTP request context. Any components called in the same context share the same session.

When you use this feature, the specific context of your application is abstracted. Your persistence layer works whether the context is defined by a web or Windows context.

The first step to enable this feature is to set the context. You do so using the configuration property `current_session_context_class`:

```
<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's therefore appropriate for ASP.NET applications.

NHibernate 1.2.1 comes with a number of built-in current session context implementations, listed in table 10.1.

It's obviously possible to implement your own contexts. You have to write a class implementing the extension interface `NHibernate.Context.ICurrentSessionContext` and set it in the mentioned property. For more details, see this extension's documentation and the available implementations in the namespace `NHibernate.Context`.

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. It's now deprecated: 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> . Each thread has its own session.
Web	This context uses the <code>HttpContext</code> API to store the current session. It's recommended for web applications (and only works with them).

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 the session to the context (using the class `CurrentSessionContext`).

Let's see how to use this feature. First, add the following method to the class `NHibernateHelper`:

```
public static ISession GetCurrentSession() {
    return SessionFactory.GetCurrentSession();
}
```

Here's what the `ItemDAO` class 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 for opening the `NHibernate` session. This is done at an upper level. In the example, it can be done in the `PlaceBidForItem()` method; see listing 10.4.

Listing 10.4 Session management using the current session API

```
using NHibernate.Context;
public void PlaceBidForItem(long itemId, long userId, double bidAmount) {
    try {
        using(ISession session = NHibernateHelper.OpenSession())
        using(session.BeginTransaction()) {
            CurrentSessionContext.Bind(session);
            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);
    }
}
```

Once the session is opened, you attach it to the current context so it's available to any object executed in this context ❶. After that, you can execute the logic as before ❷. Note that a pessimistic lock can now be used. At the end of the operation, you must detach the (closed) session from the current context ❸.

With this implementation, DAOs can access the same session in a neat and transparent way. It's 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's time to think *composition*, *inheritance*, and *generics*. Note that this section assumes you have a good understanding of .NET 2.0 generics. If you're still using .NET 1.1, it's possible to adapt the following idea, but the result won't be as clean.

As you saw when implementing the method `FindById()` for the `ItemDAO` and `UserDAO` classes, only the name of the entity changes for basic CRUD operations. Therefore, it's possible to use generics to abstract this operation.

The new DAOs may look like this:

```
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 `GenericNHibernateDAO` class only needs the types of the entity `T` and its identifier `ID` to implement the `FindById()` method. After that, implementing the `UserDAO` class means inheriting from `GenericNHibernateDAO` and providing these types.

Many other methods can be implemented like that. Before you fill the `GenericNHibernateDAO` class with them, let's take a step back and think about the final design.

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 the underlying implementation doesn't matter and can be changed as long as the interfaces don't change.

In your design, you'll 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.

Our experience tells us that even though some interfaces may not have any methods, it's still important to create them because they're likely candidates for future extension.

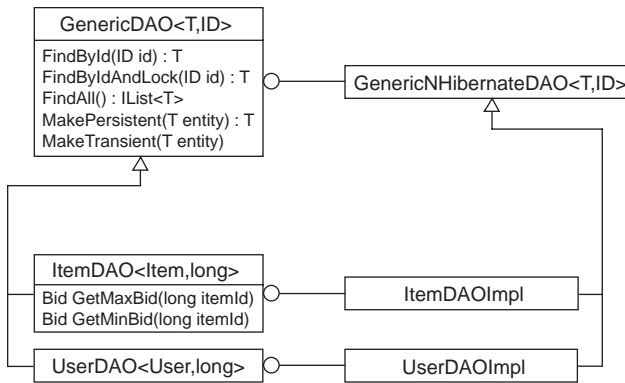


Figure 10.1 Generic DAO interfaces with a separated NHibernate implementation

You'll use the Abstract Factory pattern as a façade to provide the implementations of the interfaces. This means the NHibernate classes will be completely hidden from the other layers. It will even be possible to switch the persistence mechanism at runtime.

Enough theory. Let's 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's simpler to explain persistence operations using verbs like *save* and *delete*. But NHibernate is a state-oriented framework. This notion was introduced in section 5.1.

For example, when you delete an entity, that entity becomes transient. Its row in the database is eventually deleted, but the entity doesn't cease to exist (and it can even be persisted again). Because these methods are created for the business layer, their names should reflect what happens at that level.

The interfaces inheriting from the GenericDAO interface look like this:

```
public interface ItemDAO : GenericDAO<Item, long> {
    Bid GetMaxBid(long itemId);
    Bid GetMinBid(long itemId);
}
```

Here, you avoid defining a too-specific method like GetMaxBidAmount(). Now, let's implement these interfaces. First, GenericNHibernateDAO:

```

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, you add the `Session` property so the DAOs can work without needing a session bound to the current context. But in this case, you must manually provide this session. You can write another class in the persistence layer to take care of that.

The implementation of the other classes is straightforward. Here's 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 and 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);
}

```

Using this new persistence layer isn't much different than what is done in listing 10.4. The main difference is that these DAOs must be instantiated. You 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 as follows:

```

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 `PlaceBidForItem()` method. Here's the interesting part of that method:

```

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 the method remains as in listing 10.4. Interestingly, this code represents exactly what you want the `PlaceBidForItem()` method to look like (the rest is plumbing).

You need to clean up one other issue: the `PlaceBidForItem()` method still takes care of creating sessions and dealing with NHibernate. If you agree that managing the NHibernate session is a persistence-layer concern, it shouldn't appear in this business-layer method. You'll do one last refactoring to take care of this issue by using session management. We'll start by explaining session management for ASP.NET applications, because this is likely to be the most common scenario.

SESSION MANAGEMENT FOR WEB APPLICATIONS

When you process a complex request, many classes and methods of the business layer may be involved. The latest implementation of the `PlaceBidForItem()` method is

currently responsible for opening and closing a session; but what if the program needs to call a similar method called `UpdateItemPopularity()` after calling the `PlaceBidForItem()` method? Following the current strategy, you'd have to open another session in that function, too.

Aside from the obvious performance issue, this means the business logic must involve one database transaction for each method that opens and closes a session. In turn, this implies that if `UpdateItemPopularity()` fails, it won't be possible to fall back to the previous call to `PlaceBidForItem()`, so you risk leaving your database in an inconsistent state.

Another issue is that after executing the business logic in each of these functions, the entities they manipulate become detached from their session as it's closed. Lazy loading is subsequently disabled, thus preventing other layers (like the presentation layer) from transparently lazy loading 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 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, you can easily solve these problems by using a single session that is used for all the high-level functions, and which stays open for the entire request. Rather than make each function in the business layer responsible for session management, you move that responsibility somewhere else. In this section, we'll take the example of an ASP.NET application, where you want to execute a bunch of operations during a single web request.

Your applications may do lots of things during a single web request—they render web pages, load data in the persistence layer, carry out business functions, and so on. In the approach we'll describe, you can use the *same* single session throughout the entire request. This session is opened at the start of the request and closed at the end. This approach provides several benefits: any un-initialized associations or collections are successfully initialized when accessed at any point during the request, and a session is always available for saving and loading entities. Furthermore, entities aren't detached during the request as their session is not closed.

Despite the benefits of this *session-per-request* approach, don't be tempted to get lazy and let NHibernate always load data on demand; it may eventually kill your application's

performance. Always eagerly load the data you know you'll need. For more details, read section 7.7.1.

A simple way of implementing session-per-request is to open and attach an NHibernate session at the beginning of the web request. ASP.NET lets you implement the `IHttpModule` interface in order to execute code at the beginning and end of a web request. Listing 10.5 shows how to 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) { ❶
        ISession session = NHibernateHelper.OpenSession();
        session.BeginTransaction();
        CurrentSessionContext.Bind(session);
    }
    private void Application_EndRequest(object sender, EventArgs e) { ❷
        ISession session = CurrentSessionContext.Unbind(
            NHibernateHelper.SessionFactory );
        if (session != null)
            try {
                session.Transaction.Commit();
            }
            catch(Exception ex) {
                session.Transaction.Rollback();
                Server.Transfer("...", true);
            }
            finally {
                session.Close();
            }
    }
}
```

At the beginning of a request ❶, you open and attach a session. At its end ❷, you detach and close the session. You also commit the changes that happen thorough the request's lifetime.

The following code must be added to the `Web.config` file:

```
<configuration>
  <system.web>
    <httpModules>
      <add name="NHibernateCurrentSessionWebModule"
        type="NHibernateCurrentSessionWebModule" />
    </httpModules>
  </system.web>
</configuration>
```

This code is required to register your web module so ASP.NET uses it.

Now, the `PlaceBidForItem()` method is how you want it to be:

```
public void PlaceBidForItem(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);
    }
}
```

If you want to call some other method (such as `UpdateItemPopularity()`) after this, you can do so using the same session and transaction. You have a session that lives as long as the web request.

Do I really have to write all this infrastructure code?

In this chapter we explain how you can write your own data access objects, HTTP modules for ASP.NET session management, and various other useful infrastructure-related components.

Understanding how to do this is great, but you don't *have* to write this code yourself. Many people have done this for you already! Existing NHibernate libraries take care of much of the hard work, and include the best practices described here. After reading this section, we encourage you to research some of those libraries, including NHibernate Burrow, Castle ActiveRecord, Rhino Tools, and S#arp Architecture.

Even if you decide not to use any of them, you can learn a great deal by browsing the code of these libraries.

You can build almost any application using this approach, although sometimes you may feel you need something more. For example, what if you have a long operation that is completed over several web requests? Is there a better way to handle those longer use cases? We'll look at this next.

10.2 Implementing conversations

You've implemented your persistence layer, and you may think that you're finished. Unfortunately, not quite! There is a common real-world scenario that your persistence layer hasn't accommodated: long-running conversations.

We discussed the notion of *conversations* in section 5.2. Despite discussing the mechanics of these features, we didn't explain how they're used in the context of real NHibernate applications; we now return to this essential subject.

When you're using a command-oriented framework (for example, ADO.NET), each API call is meant to retrieve or change data: it adds/updates/deletes rows in a database. But NHibernate's API takes a different approach: it's state oriented. Rather

than execute a command to cause a direct result in the database, each API call is meant to change the state of an entity (as illustrated in figure 5.1).

Changing the state of an entity *may* lead to the execution of a SQL command immediately, or it may lead to execution some time later when the session is flushed. Alternatively, it may never lead to the execution of SQL, such as when you're using `FlushMode.Never`, and may decide to cancel a set of updates. It's important to be aware of this difference when you're implementing operations that span many user requests (conversations).

In an application that uses NHibernate, you can implement conversations three ways: using a *long session*, using *detached objects*, and *loading objects on each request*. We'll start with the latter; it's by far the simplest and is particularly well suited to web applications. First, you need a use case with which to illustrate these ideas.

10.2.1 Approving a new auction

Your 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 the *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.

The conversation is auction approval, which spans two user requests (and possibly many web requests over many days). 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, you 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;
}
```

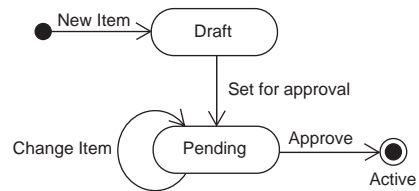


Figure 10.2 State chart of the item-approval cycle in CaveatEmptor

Are conversations really transactions?

Most books define *transaction* in terms of the ACID properties: atomicity, consistency, isolation, and durability. Is a conversation 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, because all update operations occur in the last request/response cycle (that is, the last database transaction). But 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 appropriate, because systems with this kind of conversation usually have functionality or a business process that lets the user compensate for the lack of atomicity (allowing the user to roll back steps of the conversation manually, for example).

This code should give you a feel for what the use case is about. Now we can look at how to implement the conversation that realizes this use case. As we said, you can choose from three approaches when implementing conversations, and we'll start by demonstrating the simplest one.

10.2.2 Loading objects on each request

A simple way to implement conversations is to load all persistent instances at the beginning of a request and discard them at the end (web request or otherwise). If you've worked with ASP.NET applications, you may be used to doing this using Linq to SQL, DataSets, or DataReaders.

We like this approach because it's simple to manage and implement, but it also has a few caveats. Let's use our use case to explain them.

The longer the administrator spends deciding whether to approve the auction, the greater the risk that some other user will edit the auction details, thus making the displayed *Item* out of date (the page is showing stale data). Suppose your first request executed the following code to retrieve the auction details:

```
public Item ViewItem(long itemId) {  
    return itemDAO.FindById(itemId);  
}
```

In a typical web request, you'd load the *Item*, display it on screen, and then discard it. You'd need to store the identifier value somewhere so you could retrieve the same *Item* again for use in the next request, after the administrator clicked the Approve button to approve the *Item*. It seems superficially reasonable that the newly loaded *Item* would hold nonstale data for the duration of the second database transaction, so all is well.

Not so! This notion has one problem: the data might have gone out of date *while* the administrator was looking at it (someone else could have updated it). It's possible

that the administrator based the decision to approve the `Item` on false information. Reloading the `Item` to approve it in another request wouldn't help at all, because the reloaded state *wouldn't be shown on screen and wouldn't be used for anything*—at least, it couldn't be used in deciding whether the auction should be approved, which is the important thing.

To ensure that the entity's state at the time of approval is the same as the entity's state at the time of viewing, you need to perform an explicit *manual version check*. The following code demonstrates how this can 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 difficult to implement. You take note of the version of the record that was loaded in the first request, and then you make sure it's the same when you decide to approve it in the second request.

Despite its simplicity, this load-objects-on-each-request approach doesn't always fit the bill; you may consider one of the other approaches that we tackle in the next section. For example, this approach may not work in a more complex use case that has many relationships and related objects. It would be tedious to perform all the version 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.

Some would argue that the previous code snippet contains other unnecessary noise, too; you already retrieved the `Item` and `User` in previous requests. Is it necessary to reload them in each request? It should be possible to simplify the 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—the system is working mainly with domain model instances instead of passing around identifier values. This code is also quicker, because it saves two SQL `SELECT` queries that uselessly reload data. How can you achieve this simplification using NHibernate?

10.2.3 Using detached persistent objects

Suppose you kept the `Item` as a detached instance (this approach is common in Windows applications). You could reuse it in the second database transaction by reassociating 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()`, you 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()` reassociates 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 section 5.1.8).

Because `Item` is versioned (if you map a `<version>` property), NHibernate checks the version number when synchronizing with the database, using the mechanism described in section 5.2.1. You don't have to use a pessimistic lock, as long as concurrent transactions are allowed 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 you add a new `Lock()` method to the `ItemDAO`. Doing so lets you simplify the `ApproveAuction()` method as follows:

```
public ApproveAuction(Item item, User admin) {
    itemDAO.Lock(item, false);
    item.Approve(admin);
}
```

← “false” means
not pessimistic

Alternatively, you can use `Update()`. For the example, the only real difference is that `Update()` can 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 performs a version check when updating the item.

Is this implementation, using detached objects, any simpler than the load-objects-on-every-request approach? You still need an explicit call to the `ItemDAO`, so the point is arguable. In a more complex example involving associations, you'll see more benefit, because the call to `Lock()` or `Update()` may cascade to associated instances. And let's not forget that this implementation is more efficient, avoiding the unnecessary `SELECTs`.

Nevertheless, we're not satisfied. Is there a way to avoid the need for explicit reassociation with a new session? One way is to use the same NHibernate session for both database transactions, a pattern we described in chapter 5 as *session-per-conversation* or *long session*.

10.2.4 Using the session-per-conversation pattern

A *long session* is an NHibernate session that spans a whole conversation, allowing reuse of persistent instances across multiple database transactions. This approach avoids the need to reassociate detached instances created or retrieved in previous database transactions.

A session contains two important kinds of state: 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. The session needs to release its connection between requests if you intend to reuse it in the many web requests that may span the conversation.

As explained in section 5.1.4, NHibernate 1.2 keeps the connection open from the first time it's needed to the moment the transaction is committed. Committing the transaction at the end of each request is enough to close the connection. You don't want that, because a closed session is useless in subsequent transactions.

Listing 10.5 used the web context to store the session. This strategy uses HttpContext, and the session lives only as long as the request. How do you make your session usable across multiple requests?

The simplest solution is to keep the NHibernate session in the ASP.NET session state, so that it's available from one request to another. You'll use this approach in the following example.

Let's write a new web module called NHibernateConversationWebModule (see listing 10.6). It will store the NHibernate session between requests instead of discarding it. It will also handle the reattaching of the session to the new context.

Listing 10.6 NHibernateConversationWebModule for conversations

```
public class NHibernateConversationWebModule : IHttpModule {
    const string NHibernateSessionKey =
        "NHIA.NHibernateSession"; ❶
    const string EndOfConversationKey = "NHIA.EndOfConversation";
    public static void
        EndConversationAtTheEndOfThisRequest() { ❷
        HttpContext.Current.Items[EndOfConversationKey] = true;
        }
    public void Init(HttpApplication context) { ❸
        context.PreRequestHandlerExecute +=
            new EventHandler(OnRequestBeginning);
        context.PostRequestHandlerExecute +=
            new EventHandler(OnRequestEnding);
        }
    public void Dispose() {
        }
    private void OnRequestBeginning(object sender,
        EventArgs e) { ❹
        ISession currentSession =
            (ISession)HttpContext.Current.Session[NHibernateSessionKey]; ❺
        if (currentSession == null) { ❻
```

```

        currentSession = NHibernateHelper.OpenSession();
        currentSession.FlushMode = FlushMode.Never; ⑦
    }
    CurrentSessionContext.Bind(currentSession); ⑧
    currentSession.BeginTransaction();
}
private void OnRequestEnding(object sender,
    EventArgs e) { ⑨
    ISession currentSession = ⑩
        CurrentSessionContext.Unbind(NHibernateHelper.SessionFactory);

    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;
    }
}
}

```

Because you'll be storing the NHibernate session in a map, you need a key ① to define its location. Because the business logic is responsible for 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 you aren't using the same events as in listing 11.5 because these let you access the ASP.NET session state.

When beginning a new request ④, if a conversation is already running, you extract its detached NHibernate session from the ASP.NET session state ⑤. Otherwise ⑥, you start a new conversation by opening a new session. We'll explain why you set its flush mode to never ⑦ in the next section. Once you have the NHibernate session of the running conversation, you bind it to the current context ⑧ and begin a new transaction. At this point, the conversation is ready to be used anywhere in this web request.

When the time comes to end the request ⑨, you detach its NHibernate session from the current context ⑩. If the value to end the conversation was set ⑪, you manually flush the session ⑫ to process all the changes made in the conversation, you commit these changes, you close the session, and you then remove the NHibernate session from ASP.NET session state. If the conversation is suspended ⑬, you commit the transaction to close its database connection, and you store the session in the ASP.NET session state. This conversation will resume when the next request starts.

This implementation isn't complete because the exception-handling part is missing. Refer to the `CaveatEmptor` source code for an example. Basically, these methods

should be inside a try/catch statement. When catching an exception, you should roll back the current transaction and detach and then close the current session. There is also another issue that we'll cover in the next section.

STARTING, CONTINUING, AND ENDING A CONVERSATION

Now let's see how you can use this conversation module (don't forget to register it). If you recall, in the example the administrator is viewing and then approving an auction. You'll 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 displays 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"]);

            Item item = itemDAO.FindById(itemId);
            Session[ItemKey] = item;
            editItemName.Text = item.Name; // ... Show the item
            btnApprove.Click += new EventHandler(btnApprove_Click);
        }
    }
    protected void btnApprove_Click(object sender, EventArgs e) {
        Item item = (Item) Session[ItemKey];
        item.Approve(loggedUser);

        NHibernateConversationWebModule.EndConversationAtTheEndOfThisRequest();
        Context.Response.Redirect("Default.aspx");
    }
}
```

First request implicitly starts conversation

Second request uses conversation and ends it

In this example, you store the item in the ASP.NET session state between the requests (don't forget to enable the session).

In the case of a Windows application, the implementation of a conversation is simpler because you can store everything locally in memory. It's also possible to mimic this example by using the CallContext class.

CANCELLING A CONVERSATION

You can also support canceling a conversation. All you have to do is replace the `EndConversationAtTheEndOfTheRequest()` method with two methods: one to accept the changes and another one to cancel them. Then you must distinguish these values when ending the conversation. You cancel the conversation by closing the session without flushing it.

There is an exception to this solution. To help you understand this problem, we need to explain some theory behind the implementation of conversations. Then we'll describe how to deal with the problem.

GUARANTEEING ATOMICITY AND COMPENSATING FOR CHANGES

A conversation is supposed to behave like a database transaction, so one of its requirements is to be atomic. In order to guarantee the atomicity of a conversation, all the changes made by the requests should be committed only when ending the conversation.

But by default, committing a transaction makes NHibernate commit the detected changes to the database. The session is flushed at that moment, collecting all the changes made since it was opened and executing the corresponding SQL commands.

The solution to change this behavior is to set the NHibernate session to `FlushMode.Never`, as we've done in listing 10.6.

To propagate any changes to the database, you must explicitly flush the session when you're ready (at the end of the conversation). Until then, all changes are tracked inside the NHibernate session, and no database commands are issued. Note that any queries involving the database aren't aware of these unflushed changes, so they may return stale data.

Now the exception: When you ask the session to `Save()` an entity whose identifier is generated by the database, perhaps using an identity column, NHibernate must save this entity *immediately* in order to retrieve its identifier. This can cause side effects if you have triggers, constraints, and keys set up, for example.

Another related consideration is that if the conversation is then canceled, it will be necessary to revert these changes in the database by deleting the stub record or undoing any database triggers that were fired. If the NHibernate session throws an exception during the conversation; you should perform a similar cleanup, end the conversation, and close the session.

You can look at the `IInterceptor` API (used in section 9.4) to keep track of these permanent changes and revert them if necessary. Alternatively, you can avoid using auto-generated identifiers in your database.

Sometimes, you may want conversations to persist changes at each request. This lets you ensure a recovery mechanism in case of a system failure. In this case, you again have to provide compensation actions whenever a conversation is cancelled, to clean up any database debris.

There's one final potential complication in guaranteeing atomicity with the long session approach: NHibernate's `ISession` implementation isn't thread-safe. If an environment allows multiple requests from the same user to be processed concurrently, it's possible that these concurrent requests can obtain the same NHibernate `ISession` instance. This will result in unpredictable behavior. This problem also affects the previous approach, which uses detached objects, because detached objects also aren't thread-safe. This problem affects *any* application that keeps mutable state in a non-thread-safe cache.

Because this isn't a generic problem, we'll leave it to you to find an appropriate solution if you're confronted with it. It's worth mentioning that the NHibernate.Burrow project may offer some valuable insights, because it's specifically designed to help ASP.NET work with the session-per-conversation pattern. Another good solution is to

reject any new request if a request is already being processed for the same user. Other applications may need to serialize requests from the same user. Note that this isn't a problem for web applications that use the ASP.NET session state: concurrent requests (from the same user) are automatically serialized, so the second request waits until the first completes.

Now that we've covered three different ways to deal with conversations, you may 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*

Our default recommendation for NHibernate web applications is to use the load-objects-on-every-request approach, where you create one NHibernate session per request. It's easy to understand and in many cases easier to implement than the other approaches. It's particularly well suited to architectures in which you're unable to keep state associated with the user because you're using a stateless framework. This approach will especially appeal to those who prefer to avoid using the ASP.NET session. Another good fit for this approach is in architectures where 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 Data Transfer Object (DTO) abstraction layer.

The second most popular approach in NHibernate applications uses detached objects, 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 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. But we think that in many cases, it isn't the best approach.

More complex cases may be suited to the long-session approach, especially Windows Forms applications. 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 widely understood, 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 you can take this code and adapt it to run in an Enterprise Services application. Obviously, you'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 you now have to rewrite all the code for placing a bid, we're going to look 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 an NHibernate application participate in a distributed transaction. Now you'll implement this approach.

First, we must cover an issue linked to interprocess requests. Whenever you have physically separated components/tiers, you must minimize the communication between them. This is important because latency is added by every interprocess request, increasing the application response time and reducing concurrency due to the need for either more database transactions or longer transactions. These issues can strongly influence the scalability of your application.

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 pulls 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 DTO pattern provides a way of packaging the data 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 because DTOs tend to duplicate entities, we naturally find ourselves questioning the need for DTOs.

10.3.1 Rethinking DTOs

DTOs are commonly used to totally separate the presentation tier from the domain model. Certain reasonable arguments can be made in favor of this approach, but you shouldn't mistake these arguments for the real reason why DTOs are useful.

The idea behind the DTO pattern is that fine-grained remote access is slow and unscalable. It's also useful when the domain model can't 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 you, and the benefit of this separation is questionable when weighed against its cost. We won't tell you never to use DTOs (sometimes we're 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 may 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). But 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 may 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. 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 can also be viewed as mere *duplication*.

The DTO pattern exhibits two of the code smells described in *Refactoring: Improving the Design of Existing Code* [Fowler 1999]. The first is the *shotgun-change* smell, where a small change to a 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 you add a new property to `Item`: you 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*—has value even in the context of NHibernate. DTO assembly provides a convenient point at which to ensure that all data the presentation tier needs is fully fetched 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 imposes extra discipline by requiring that all needed data is 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). But typed `DataSets` seem better adapted to this problem.

You can consider a typed `DataSet` as a special kind of DTO. There are definitively good reasons to use `DataSets`: an extensive toolset is available, and many existing libraries use `DataSets`. But writing custom classes as DTOs gives you better control over the design of your application even if the code is tedious to write.

You won't use DTOs in the `CaveatEmptor` application. Now that we've covered the potential issues that may occur when you're dealing with physically separated tiers, we can go back to distributed transactions.

10.3.2 Enabling distributed transactions for `NHibernateHelper`

You must apply a few changes to the previous `NHibernateHelper` class to enable distributed transactions. First, you must add a reference to the `System.EnterpriseServices` assembly and change the class definition like this:

```
[Transaction(TransactionOption.Supported)]
public class NHibernateHelper : ServicedComponent {
    //...
}
```

You must also add the following methods to the management of the transaction: `BeginTransaction()`, `CommitTransaction()`, and `RollbackTransaction()`. They're used to create, commit/roll back, 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 in the distributed transaction, the NHibernate session's transaction must be enlisted. Note that you have to use .NET reflection here because the method used isn't 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)
                    ContextUtil.Transaction } );
    }
}

```

If the `EnlistDistributedTransaction()` method isn't available, this code silently fails to enlist the transaction. If it represents an error in your use case, you should throw an exception. You can even avoid 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 `OpenSession()` method:

```

public static ISession OpenSession() {
    ISession session = SessionFactory.OpenSession();
    TryEnlistDistributedTransaction(session);
    return session;
}

```

All that's left to do is update the code to use these new methods. Don't forget to register the resulting COM+ assembly (which must be signed) before you use it. To do so, you use the command-line executable `RegSvcs`.

10.4 Summary

This chapter focused on the design of the persistence layer. We first introduced the `NHibernateHelper` class, which is useful to abstract the initialization of NHibernate.

We also showed you how to move from a monolithic method that mixes 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 `CaveatEmptorItem` class. This was the first step of a series of refactorings.

You used the DAO pattern to create a façade for the persistence layer, hiding NHibernate's internals from the other layers. We also introduced the `ISessionFactory.GetCurrentSession()` API and the notion of context. You used this feature to significantly improve your DAOs by making them share the same session without having to pass the session as a parameter and without using a global static session.

After that, you leveraged the .NET 2.0 generics to reduce the redundancies in the persistence layer. You also designed the persistence layer so the other layers were unaware of the persistence framework that was used. You even made it possible to switch from one implementation to another by changing a single line of code.

We explained how to make a session live for an entire web request. This is useful to guarantee that a single session is used for all processing and that lazy loading always works transparently.

Chapter 5 introduced the notion of a conversation (also called application/business transaction). In this chapter, we provided three ways of implementing conversations: the load-objects-on-each-request approach, which is simple and suits many web applications; the approach that uses detached persistent objects, which is useful in stateless environments; and the approach that uses long-living sessions. The last approach is relatively unknown; but it's powerful, especially in a rich environment.

This chapter ended by improving the `NHibernateHelper` class to support Enterprise Services transactions. We discussed the potential latency issue and the DTO pattern. Although this pattern can be useful when decoupling the domain model is important, we agree that such a situation is rare and that the cost of maintaining the DTO is too high.

At this point, you should have all the technical knowledge required to leverage the features of NHibernate. It's a powerful tool, but it requires a deep understanding of its behavior to be correctly used.

NHibernate has a growing, enthusiastic community. We hope that you'll enjoy being part of it. The Google `nhusers` forum is frequented by many of the NHibernate developers, so feel free to drop by to find answers to your questions and join in the discussions. Also, we recommend <http://nhforge.org> for its wealth of blog posts and articles. It's also a great place to keep up with new NHibernate developments. Happy NHibernating!

appendix A: SQL fundamentals

This book assumes that you have a basic understanding of relational databases and the Structured Query Language (SQL). 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. We highly recommend that you find a book to learn more about it.

Tables

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.

Relational model

The relational model lets you 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. These operations let you do useful things with your data, such as summarize or navigate 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.

DDL and DML

You write SQL Data Definition Language (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 Data Manipulation Language (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, you naturally have entities like items, users, and bids. Assume that the SQL database schema for this application includes an ITEM table and a BID table. You can create the data types, tables, and constraints for this schema with SQL DDL (CREATE and ALTER operations).

Table operations

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
```

But the real power of SQL lies in querying data.

Queries

A single query may perform many relational operations on several tables. Let's look at the basic operations.

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%'
```

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 from 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 BID tables 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. Because each relational operation produces a new table from an existing table or tables, an SQL query can 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 can 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)
```

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%'
```

SQL also includes many other operations that you'll want to learn about as you become more experienced.

appendix B: *Going forward*

You've reached the end of this book. What remains is for us to give you some guidance and advice that will help you get started and 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 and to contribute to and help improve NHibernate.

What you need

This book assumes that you have some experience with .NET programming. Before you begin 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.

The NHibernate binaries, source code, and documentation are available on the NHibernate SourceForge website: <http://sourceforge.net/projects/nhibernate/>. (SourceForge.net is a website that provides free hosting for open source software (OSS) development projects.)

Before NHibernate 1.2, two packages were available: `nhibernate`, containing the core binaries with source code and documentation; and `NHibernateContrib`, containing optional useful add-ons for NHibernate. Now, the packages have been merged. You'll 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. The complete list is available at <http://www.hibernate.org/361.html>.

This is all you need to start using NHibernate.

Practice makes perfect

The next step is to practice using NHibernate, from the simple “Hello World” example in 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 that point, this book will serve as a reference.

Make sure you understand the mapping of entities and their persistence lifecycle and the way NHibernate sessions work. Also be careful of the way you use NHibernate caching. The book’s three final chapters explain how important the application’s architecture is.

Problem solving

You’ll often encounter problems while using NHibernate. In most cases, their source is the misuse of features due to a lack of understanding. For more advice about problem-solving techniques, see section 8.3. Note that a well-thought-out design will help you avoid and solve problems; take time to think about the features you want to use (algorithms) and the architecture of your application (layers, separation of concerns, and so on).

If you’re still unable to overcome your difficulties, feel free to ask for help on the NHibernate forum: <http://groups.google.com/group/nhusers>. Be sure you explain your problem in detail, including logs and error messages.

Staying up to date

NHibernate is constantly evolving. Once you feel comfortable using it, you should keep yourself up to date, because 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’s also a great place to share your point of view about various NHibernate-related problems, get feedback about them, and learn how other people solve them.

Many other informative and useful resources (documentations, samples, and open source projects) are available on the internet. You can find a comprehensive list at <http://nhforge.org/wikis/>.

NHibernate has a bug-tracking website: <http://jira.nhibernate.org/>. You can register on this website and report any bugs you encounter (if you aren’t sure it’s a bug, use the NHibernate forum first); you can also request new features.

Because NHibernate is OSS, its source code is freely available. Instead of using the compiled library, you can use the source code to gain more details when debugging your application.

Feel free to modify NHibernate whenever you need to, to add a new feature or fix an existing bug. Don’t forget to make this addition publicly available so that other

NHibernate users can use and even improve it. You can do so by submitting a patch at the bug-tracking website (a *patch* is a file that contains the changes you've made in the source code).

The bug-tracking website also gives you an idea of the features and bug fixes that are available in the bleeding-edge version of NHibernate (which the NHibernate developers are working on). The source code of this version is hosted by Sourceforge.net in an SVN repository. If you're 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 from time to time.

After you begin to use the SVN version of NHibernate, you need to take only one final step to embrace NHibernate completely: joining the development list. This mailing list is used by the developers of NHibernate to discuss its evolution. You can register and read the archive at <http://groups.google.com/group/nhibernate-development>.

This is the end of the book and, we hope, the beginning of a wonderful experience with NHibernate. Bon voyage!

Symbols

{ } syntax 247

A

Abstract Factory, in persistence layer design 330

access, property 68–70

ACID, applicability to conversations 337

ActiveWriter 293

Adapter pattern 314

AddAssembly() 40

AddCategory() 61

AddChildCategory() 60
method 119

AddClass() 39

AddXmlFile() 39

ADO.NET 7, 9, 21, 34

connection management 111

connection pool 274

database access

configuration 41–44

DataSet. *See* DataSet

Entity Framework 11

ADO.NET connection 41

ADO.NET Entity

Framework 18–20

ADO.NET IDbConnection. *See*
IDbConnection

aggregate function 234

aggregation 82, 234

alias 216

naming convention 216

using with join 226–228

all quantifier 242

Ambler, Scott 92

analysis 52–53

and, logical operator 221

<any> 204–205

any quantifier 242

API (application programming
interfaces)

APIs

Auditable 278

Criteria 208

FetchMode 224

Interceptor 280

MatchMode 220

UserType 302–303

CompositeUserType 176–178

UserType 173–176

application

architecture 260, 320

layered, design of 320, 336

legacy, porting to

NHibernate 215

application transaction 146

avoiding reassociations with
new sessions 340–344

choosing implementation
approach 344

implementing the hard
way 337–338

use case 336

using detached objects
338–340

using long sessions 340–344

architecture 33–38

coupling 10

layered 7–9

three layers 9

argument, binding

arbitrary 213–214

ArgumentNullException 276

arithmetic expressions, support
for 219

asc, ordering query results 222

ASP.NET 13, 342

applications, session

management 332–335

session, storing NHibernate
session in 340

sharing sessions, and 327

assembler, DTO 346

Assert class 264

association 12, 58, 82, 86–91

bidirectional 58, 85, 88–90

cardinality 60

defined 189

fetching 224–225

foreign key 190–192

inverse 89

joining 225

managed 59

many-to-many 60, 193–200

many-to-one 65, 87

polymorphic 201–202

mapping 189–200

for lazy initialization 250

multiplicity 86–87

one-to-many 87–88, 198–199

one-to-one 189–193

parent/child 90–91

polymorphic

mapping 200–205

table-per-concrete-
class 204–205

polymorphism 92

association (*continued*)
 primary key 192–193
 simplest 87–88
 single point 127–129
 ternary 197
 unidirectional 85–86, 199
 many-to-one 87–88
 association class 193
 association table 130, 193
 association-level cascade
 style 116
 associations, joining 222–231
 asynchronous
 programming 274
 atomic 111
 atomicity, guaranteeing
 343–344
 attribute 349
 Column 290
 foreign-key 290
 index mapping 290
 Length 290
 not-null 290
 sql-type 290
 Unique 290
 unique-key 290
 XML mapping 289
 attribute-oriented
 programming 65–66
 audit log 277
 audit logging 277–284
 automated 278
 manually 278
 Auditable API 278
 AuditLog 278
 AuditLog.LogEvent() 282
 AuditLogRecord class 279
 automated persistence 55–56
 automatic dirty checking
 33, 103, 113
 availability 273
 avg() function 234

B

backtick 72
 <bag> 196
 bag collection 182, 194, 196
 with set semantics 198–199
 batch fetching 127, 250
 batch update 113
 BeginTransaction() 136–137
 behavior entity 54
 bidirectional 88–90
 BindingList 314

BindingSource 314
 BLOB 171
 bottom-up development 293
 bug-solving process 274
 bug-tracking website 353
 business entity 7
 See also domain model
 business key 108–110
 business layer 8, 266–268
 business logic in 310
 implementing 266
 implementing manual
 version checks 338
 testing 268
 business logic 7
 example in domain
 model 336
 implementing 309–312
 tiers of, separating from web
 tiers 345
 business logic layer 8
 business model 52
 business object 5
 See also domain model
 business rule 321
 encapsulating in domain
 model 324
 vs. test 312
 See also business logic
 business transaction 146
 See also conversation
 by value equality 108

C

cache 104, 152
 cluster scope 153
 distributed 162
 expiration policy 254
 first level 156
 managing 157
 miss 153
 policy 157
 process scope 153
 region 161
 second level 157
 controlling 164
 timestamp 254
 transaction scope 153
 cache architecture 155–159
 cache provider
 choosing 159
 Hashtable 159
 local, setting up 161
 MemCache 163
 NCache 163
 Prevalence 159
 SysCache 159
 caching 18, 142, 152–164
 example 159
 good and bad candidates
 for 155
 maintaining consistency
 across clusters 322
 methods used for
 lookups 300
 object identity, and 154
 queries 253–255
 reference data 155
 strategies 153–155
 transaction isolation, and 154
 callback API 36–37
 CallContext API 327
 candidate key 79
 Cartesian product 223, 229, 350
 cascade attribute 90
 cascade style 116–117
 cascading delete 90
 cascading save 33, 90
 CASE 52
 CaveatEmptor 52–54
 domain model 53–54
 changes detection 103, 113
 check constraint 289
 class
 association 193
 coarse-grained 167
 component, writing 187
 DTO 345
 entity 167
 fine-grained 167
 helper 321
 immutable 71
 parallel hierarchy 346
 utility 321
 value type 167
 wrapping 314
 class name qualification 74–75
 ClassToTableName() 73
 CLOB 171
 cluster scope cache 153
 cluster-safe design 322
 coarse-grained transaction 145
 code generation 10
 bottom-up development,
 and 293
 for domain model 12
 code smell 346
 CodeDom provider 70
 CodeGenerator class 292

- CodeSmith 293
- collection
 - bag 182, 194, 196
 - with set semantics 198–199
- columns, avoiding
 - not-null 188
- component 186
 - using for many-to-many association 196–198
- fetching 128, 225
- filtering 240–242
- indexed 199
- indexed map 194
- list 183, 194, 196
- map 184
- nonindexed 198
- ordered 184, 186
- persistent 184
- polymorphic 203
- set 182
- sorted 184
- wrapper 129, 131
- collections
 - comparison by identity 62
 - component 189
 - value-type, mapping 189
- column 67
 - avoiding not-null 188
 - inconvenient type 302–303
- Column attribute 290
- <column> element 289
- ColumnName() 73
- COM+ assembly, distributed
 - NHibernate 348
- command, custom 248
- commit 135
- Commit() 111, 137
- CompareTo() 185
- comparison operators and
 - restriction 218
- component 82–86
 - collections of 186, 190
 - using for many-to-many association 196–198
- composite identifier class
 - 299–301
- composite key 80, 296
 - mapping table with 298–302
 - referencing entity with 301
- <composite-id> mapping 297
- CompositeUserType API
 - 176–178
- composition 82
 - and minimizing redundancy 329
- conceptual view 52
- concurrency
 - database capabilities 7
 - optimistic checking, issue with
 - batching 113
 - reducing 345
- concurrency strategy 158
 - nonstrict-read-write 158
 - read-only 158
 - read-write 158
- concurrent requests, problems
 - with 343
- concurrent transactions 339
- Configuration 34–35
- configuration
 - app.config file 44–46
 - connection string 46
 - database access 42–43, 71
 - NHibernate 31–32, 38–44
 - advanced 44–48
 - reflection 70–71
 - schema 46
 - steps 43
 - techniques 40
- configuration document 31
- Configuration.SetProperty() 41
- Configure() 41
- connection pool 41
- connection string 46
 - securing 273
- connection-release mode 139
- consistency 135
- control logic, separating from
 - data access code 324
- control, data bound 314
- controller, as part of business
 - layer 266
- conversation 16–18
 - approach, choosing 344
 - ASP.NET
 - implementation 342
 - example 146
 - guaranteeing atomicity
 - of 343–344
 - implementing 335–344
 - loading objects on each
 - request 337–338
 - using detached objects
 - 338–340
 - using the session-per-conversation pattern 340–344
 - vs. transaction 337
 - working with 146–152
- core interfaces 35–36
- correlated subquery 242
- count() function 234
- create command, custom 248
- create, read, update, delete (CRUD) 7
 - hand-coded 14–15
 - operations,
 - implementing 14–15
 - with DataSets 14
 - with NHibernate 14
 - with the data access object
 - pattern 325
- CreateAlias() 227
- CreateCriteria() 209, 227
- createQuery() 208
- createSQLQuery() 208
- creating object. *See* persistence
- Criteria API 19
 - comparison operators 218
 - FetchMode 224
 - implicit joins 228–229
 - logical operators 221
 - MatchMode 220
 - nesting 227
 - polymorphic queries 217
 - purpose of 208
 - QBC, and 208
 - QBE, and 239
 - restriction 217
 - results, ordering 221
 - SQL function calls, and 220
 - string matching 220
 - theta-style joins 230
 - where clause, and 219
- criteria query
 - comparison operators 219
 - wildcard search 220
- cross join 350
- cross-cutting concern 55
- CRUD. *See* create, read, update, delete (CRUD)
- current_session_context_class
 - 327
- CurrentSessionContext 328
- custom mapping type 173
- custom type 62
- custom type API 37

D

- data
 - deprecated, ignoring 218
 - externalization 345
- data access approaches 6
- Data Access Object pattern
 - 324–326

- data binding 312–316
 - manual 313
 - using data-bound controls 314
 - using NHibernate 315
 - with ObjectViews 315
- Data Definition Language (DDL) 349
- data integrity, database capabilities 7
- data language 349
- Data Manipulation Language (DML) 350
- data storage. *See* persistence
- Data Transfer Object (DTO) 345
 - assembly 346
 - data transfer, and 346
 - need for, questioning 345–346
 - problems with 346
- Data Transfer Object pattern 345
- database
 - generation, executing arbitrary SQL during 291
 - identity 76–79
 - persistent object, and 102
 - live, updating 295
 - lock table 149
 - schema maintenance, automatic 294–296
 - setup 27
 - subsystem 8
 - systems that work with NHibernate 352
 - trigger 205, 278
 - triggers for 303–305
 - update
 - first commit wins 147
 - last commit wins 147
 - merge conflicting updates 147
- database schema generation 67
- database transaction 135–146
- database-independent application 23
- data-bound control 314
- data-integrity test 264
- DataReaders 337
- DataSet 6–7
 - as entity 12
 - association 16
 - filling with entity data 316–317
 - granularity 16
 - obtaining 237
 - presentation layer 13
 - typed 346
 - using LINQ 20
- DataSets 337
- DataTable, avoiding 7
- DDL schema
 - creating with hbm2ddl 290–291
 - tools for 288
- debugging 274–276
- delete command, custom 248
- Delete() 113
- deleting object. *See* persistence
- Dependency Injection pattern 284
- desc, query ordering 222
- design goal 15, 272–274
 - availability 273
 - manageability 273
 - performance 273
 - reliability 273
 - scalability 273
 - securability 273
- design pattern 7
- detached object 78, 339
 - in conversations 336
 - when to use 344
- development
 - database 295
 - list 354
 - tools 287
- development process 287–296
 - bottom-up 293
 - domain-centric 261
 - meet-in-the-middle 294
 - middle-out 292–293
 - top-down 288–292
- Dialect 37
 - specifying 42
- dirty checking 62
 - automatic 33
- dirty read 140
- Disconnect() 139
- discriminator 94
- discriminator column 205
- distinct 233
- distributed cache 274
- distributed transaction 345
 - enabling 346–348
- domain expert 52
- domain model 6–7, 53, 263–266
 - "smart" 323–324
 - adding logic 61–63
 - association 16, 58–61
 - attached to persistence 103, 106
 - behavior 54
 - business logic in 310
 - component 82–86
 - creating 26–27
 - detaching from persistence 104
 - discriminator 94
 - distinguishing transient and detached instances 106, 120–121
 - fine-grained 81–86
 - granularity 15
 - hand-coding 12
 - identifier 18, 102
 - identity 16, 64, 76–81
 - identity implementation 107
 - identity scope 104–105
 - immutable 71
 - implementation 55–63
 - implementing 166, 173, 263
 - instances, working with 338
 - joined-subclass 94
 - mapping 18
 - mapping to given schemas 294
 - mutable 71
 - ORM without 54
 - persistence lifecycle 101–110
 - proxy 122
 - reattaching to persistence 106
 - referencing 16
 - state 54
 - subclass 94
 - testing 264
 - See also* type
- domain-centric 261
- Domain-Driven Design 325
- domain-driven development (DDD) 261–262
- Dont Repeat Yourself (DRY) 326
- drag and drop 6
- DTO. *See* Data Transfer Object (DTO)
- durability 135
- dynamic instantiation 232

E

- eager fetching 126, 224–225
- elements() function 241, 243
- embedded resource 30

EmptyInterceptor 281
 EnableLike() 239
 Enterprise Library 284
 Enterprise Services application,
 using NHibernate in
 345–348
 entity 167
 applying Observer pattern
 to 307–309
 binding persistent 213
 data binding 312–316
 hand-coding 12
 hidden 168
 implementing 11–13
 lifecycles of 167
 persistence-abstracted 307
 referencing with composite
 key 301
 retrieving multiple types in a
 query 245
 root 209, 228
 See also domain model
 entity framework 20
 entity query 244
 entity.ToString() 279
 EntityNameDAO 268
 Enumerable() method 210, 252
 Enumerable() query 252
 enumerated type 180–181
 enumeration 181
 Environment.BuildBytecode-
 Provider() 71
 equality
 by value 108
 consistency 107
 using business key 108–110
 using database identifier 107
 using versioning 108
 vs. identity 76–77
 See also identity
 Equals() 76, 106–110
 Equals(object o) 77
 equivalence 76
 error, understanding and
 solving 274–276
 Evict() 156, 164
 exception
 .NET, built-in 276
 from external libraries 276
 throwing 276
 understanding 274
 exception.ToString() 274
 Execute() 291
 Expression class 217
 expression, SQL 218

Expression.And() 221
 Expression.Conjunction() 221
 Expression.Disjunction() 221
 Expression.Or() 221
 extension API 37–38

F

failure
 accessing databases 321
 request checks 321
 fetch 224
 fetch attribute 251
 fetching strategy 125–127
 batch 127
 collections and 129–130
 eager 126, 225
 fetch depth 130–131
 global 130–131
 immediate 126
 lazy 126, 225
 runtime association 225
 selecting in mapping 127–132
 field 26
 fine-grained transaction 145
 first commit wins 147
 fixture 264
 fluent interface 39
 flush mode 139
 Flush() 139
 executing trigger 304
 flushing 139
 FlushMode 139
 FlushMode.Never 343
 foreign key 12, 81, 230
 foreign key association 190–192
 foreign-key attribute 290
 formula 68
 from clause 217, 242
 fetch join 224
 implicit, in collection
 filters 240

G

garbage collection 102
 Gateway pattern 325
 generation database 67
 generics, and minimizing
 redundancy 329
 Get() 112, 122
 GetClassMetadata() 76
 GetCollectionMetadata() 76
 GetHashCode() 106–110, 300

getNamedQuery() 214
 global assembly cache (GAC) 43
 granularity of a session 150–151
 group by clause 234
 grouping, database
 capabilities 7

H

HashSet 252
 Hashtable cache provider 159
 having clause 236
 rules governing 236
 hbm2ddl (SchemaExport) 67
 DDL schema generation
 with 290–291
 middle-out development
 with 292
 parameter descriptions 291
 preparing mapping
 metadata 288–289
 top-down development
 with 288
 XML mapping attributes
 for 289
 hbm2net (CodeGenerator) 292
 middle-out development
 with 292
 hbm2net.config file 293
 HbmSerializer.Serialize() 40
 Hello World 25–33
 helper class 321
 helper/utility classes 8
 Hibernate mapping types,
 system 167–181
 Hibernate Query Language
 (HQL) 19, 123–124, 208
 aggregation, using 234
 aliases 216
 and joins 226–228
 basic queries 215–222
 collection filters 240–242
 comparison operators 218
 distinct results, getting 233
 dynamic instantiation 232
 expressing queries with 208
 fetch join and 224–225
 grouping 234
 implicit joins 228–229
 keywords, writing 216
 logical operators 221
 polymorphic queries 217
 projection 232–234
 restricting groups with
 having 236

Hibernate Query Language (HQL) (*continued*)
 restriction 217
 results, ordering 221
 SQL function calls, and 220, 233
 string matching 220
 subqueries 242–243
 theta-style joins 229
 where clause, and 219
 hibernate.cfg.xml 41, 163
 hibernate.connection.release_mode 140
 HibernateException 276
 high coupling, avoiding 284
 HQL. *See* Hibernate Query Language (HQL)
 HTTP request context for ASP.NET session management 327
 HttpContext API 327

I

IAuditable 279, 293
 ICache 37
 ICacheProvider 37
 IClassPersister 37
 ICompositeUserType 34, 37
 IConnectionProvider 37
 ICriteria 34, 110, 124, 208
 introducing 36
 method chaining 210
 pagination 209
 results, ordering 222
 ICriteria API 121
 ICriterion 124, 217
 ICurrentSessionContext 327
 <idbag> 196
 IDbConnection 136
 manual 41
 identifier 18, 77
 native generator 78
 identity 16, 64, 76–81
 .NET 107
 implementation 107
 process-scoped 104
 reference equality 105
 scope 104–105
 transaction-scoped 104
 vs. equality 76–77
 See also equality
 identity map 17
 cache usage 18
 Identity Map pattern 17

id-type attribute 204
 IEditableObject 314
 IEnhancedUserType 178
 Iesi.Collections 59
 Iesi.Collections library 252
 Iesi.Collections.ListSet 186
 Iesi.Collections.SortedSet 185
 ignoreCase() 239
 IHttpModule interface 334
 IIdentifierGenerator 34, 37, 79
 IInterceptor 34, 37, 278, 310
 IInterceptor API 343
 ILifecycle 34, 36
 IList 196
 immediate fetching 126
 immutable 71
 impedance mismatch. *See* paradigm mismatch
 implicit join 228–229
 in quantifier 242
 INamingStrategy 290
 introducing 72
 inconvenient column type 302–303
 index mapping attribute 290
 indexed collection 199
 and inverse= 196
 indexed map collection 194
 indices() function 243
 InfrastructureException 138
 inheritance 7, 12, 16, 91
 mapping 91
 mapping strategy 98
 minimizing redundancy, and 329
 table per class hierarchy 92–95
 table per concrete class 92–93
 table per subclass 92, 95–98
 inner join 222
 INotifyPropertyChanged 308, 314
 InsensitiveLike() operator 239
 insert
 control 71
 dynamic 71
 insertion 350
 instance, detached 106
 instantiation, dynamic 232
 integrating services 277–284
 integrity, guaranteeing
 referential 115
 interceptor
 enabling 282
 writing 280–282
 Interceptor API 280, 282

interfaces, core 35–36
 INullableUserType 178
 inverse attribute 89
 Inversion of Control
 pattern 284
 IParameterizedType 37, 178
 IPropertyAccessor 37
 IProxyFactory 37
 IQuery 34, 56, 110, 208
 binding arbitrary arguments with 213
 introducing 36
 method chaining 210
 pagination 209
 IS NULL 219
 is null operator 214, 219
 ISession 34, 56, 110, 150
 as first-level cache 155
 Close 103
 Delete 103
 Evict 104
 introducing 35
 Save 102
 transparent write-behind 138
 See also persistence
 ISession API
 obtaining new instances of 321
 query shortcuts 211
 ISession.Connection
 property 273, 316
 ISession.CreateSQLQuery() 244
 ISession.Get() 218
 ISession.Load() 218
 ISessionFactory 110, 157
 creating 38–41
 instance, configuring 44
 introducing 35
 naming 46
 ISessionFactory.GetCurrentSession() 326–329
 isolation issues 140
 isolation level 140–141
 choosing 141–143
 read committed 141
 read uncommitted 141
 repeatable read 141
 serializable 141
 setting 143
 ISQLQuery API 244–246
 ISQLQuery instance,
 creating 244
 ISQLQuery.SetResultSetMapping() 247
 iterate() 210

ITransaction 34, 37, 110,
137–138, 146
introducing 36
ITransactionFactory 37
IUserCollectionType 37, 178
IUserType 34, 37
IValidatable 34
IValidatable interface 36

J

JDBC connections 340
join 222–231, 350
 ANSI-style 222
 fetch 224–225
 from clause and 223
 implicit 223, 228–229
 inner 222
 means of expressing 223
 options 223
 outer 223
 table 223
 theta-style 223, 229
 using alias with 226–228
 where clause, and 223
join condition 223
joined-subclass 94

K

key
 candidate 79
 composite 80, 303
 mapping table with
 298–302
 referencing entity with 301
 foreign 190–192, 230
 generation 79
 natural 79
 natural, mapping table
 with 297–298
 natural/primary 297–298
 primary 79, 192–193
 choosing 79–81
 surrogate 79
keyword, case sensitivity 216

L

last commit wins 147
latency 345
layer
 application 260–270, 320
 building and testing 263

business logic. *See* business
 logic layer
interaction 101
persistence, designing
 320–335
persistence. *See* persistence
 layer
presentation. *See* presentation
 layer
 separating business and
 presentation 320
layered architecture 7–9
lazy association, initializing 131
lazy fetching, avoiding 202
lazy loading 17, 121, 126
 always enable 275
 application 123
LazyInitializationException 131
leakage of concerns 55
legacy application, porting to
 NHibernate 215
legacy column, mapping with
 custom type 302–303
legacy data 296–305
legacy database schema
 296–305
 changes needed 296
 composite key mapping
 298–303
 integrating database triggers
 with 303–305
 natural key mapping 297–298
 problems with 296
 required changes 297
Length attribute 290
library migrations 295
lifecycle state
 detached 167
 persistent 167
 transient 167
like operator and wildcard
 searches 220
link table 130, 193
LINQ 20
 over DataSet 20
 to Entities 20
 to NHibernate 20
 to SQL 11
<list> 196
list collection 183, 194, 196
List() 210, 252
Load() 122
<load-collection> 246
loading objects on each request
 in conversations 336

load-objects-on-every-
 request 339
lock mode 144
lock table 149
Lock() 111
lock() 339
locking 140, 143
 optimistic 147
 alternative
 implementations 151
 offline 147–149
 optimistic and pessimistic,
 compared 149
 pessimistic 143–146
LockMode 112, 144
log record, mapping 279
log4net 284
 configuration 47
 See also logging
LogEvent() 278
logging 40, 47–48
 integrating 284
logging library 284
logic test 264–265
logic, ternary 218
logical expression,
 constructing 218
logical operator 221
LogType 278
long session 151, 340
 in conversations 336
 when to use 344
long transaction 146
lost update 140
lower() function 220

M

maintainability 7, 9, 22
MakePersistent() 330
MakeTransient() 330
manageability 273
managed relationship 59
managed versioning, for
 optimistic locking 147–149
many-to-many association
 60, 130
 bidirectional 195–196
 component collections used
 for 196–198
 mapping 193–200
 mappings 168
 tables 168
 unidirectional 193–194

- many-to-one 87
 - mapping 30
- many-to-one association 65, 229
 - implicit join 228
 - polymorphic 201–202
- <map> 196
- map collection 184
- mapping
 - assembly 40
 - association 16, 30, 86–91, 189–200
 - unidirectional 86
 - attributes 65
 - basic 66–76
 - bidirectional many-to-many 195–196
 - choice 66
 - class 40
 - component collection 187
 - composite key 298–303
 - correctness, testing 269
 - creating 29–30
 - data type 65
 - domain model 18
 - entity class 168
 - error 40
 - inheritance 91
 - inheritance strategy 98
 - IntelliSense 46
 - legacy columns 302–303
 - log record 279
 - many-to-many
 - association 168, 193–200
 - metadata 63–66
 - metamodels 173
 - natural key 297–298
 - one-to-many association 198–199
 - one-to-one association 189, 193
 - property 66–68
 - runtime 75–76
 - schema 46
 - strategy 98
 - synthetic identifier 297
 - table per class hierarchy 92–95
 - table per concrete class 92–93
 - table per subclass 92, 95–98
 - ternary association 197
 - unidirectional many-to-many 193–194
 - unidirectional one-to-many 199–200
 - XML 63–65

- mapping attributes, XML
 - injection 84
- mapping document 30
- mapping file
 - Hibernate 215
 - working with 39
- mapping metadata, preparation of 288–289
- mapping type, Hibernate
 - basic 169, 172–181
 - built-in 169–171
 - custom, creating 173
 - date and time 170
 - enumerated 181
 - Java primitive 169
 - JDK 171
 - large objects 171
 - object 171
 - system 167–181
 - using 172–181
- marker attribute 278
- marshal-by-reference object 272
- MatchMode API 220
- max() function 234
- maxelement() function 243
- maximum fetch depth 130
- maxindex() function 243
- medium-trust policy, issues with 271
- meet-in-the-middle
 - development 294
- MemCache distributed cache
 - provider 163
- merge conflicting updates 147
- metadata 63–66
 - manipulating at runtime 75–76
- metamodel 173
- meta-type attribute 204
- method chaining 39, 210
- method, parameter-binding 211–214
- middle-out development 292–293
- migrations library 295
- Migrator 295
- min() function 234
- minelement() function 243
- minindex() function 243
- model, presentation 314
- modeling, object vs. relational 21
- model-view-controller (MVC) 266
- Mono 352

- multilayered architecture 260, 320
- multiplicity 86–87
- multiversion concurrency
 - control 140
- mutable 71
- MyGeneration 293
- MySQL 10
 - See also* relational database management system

N

- n+1 select problem 249–252, 275
- named parameter 212
- named query 214–215
 - used in persistence layer 331
- named SQL query 246–248
 - calling stored procedures 247
- namespace, default 74–75
- naming convention 72–73
- native SQL 243–249
 - queries 121
- natural key 77
 - mapping 297
 - mapping table with 297–298
- navigation, unidirectional 193–194
- NCache distributed cache
 - provider 163
- .NET
 - built-in exceptions 276
 - data binding 313
 - database access 6
 - features, solving issues related to 270–272
 - generics 329
 - reflection 347
 - remoting 271
 - security policy, issues with 271
- NHibernate 4
 - advanced configuration 44–48
 - alternatives 9
 - API 34
 - bug-tracking website 353
 - compatible database systems 352
 - configuration 38–44
 - development list 354
 - downloads 352
 - forum 353
 - identity scope 105

NHibernate (*continued*)
 installing 25
 learning curve 277
 making sure it's the right tool 276
 online resources 353
 reasons to use 15–20
 SourceForge website 352
 starting 43–44
 uses of 320
 NHibernate Query Analyzer 216, 255
 NHibernate.Burrow project 344
 NHibernate.Cache.ICache-ConcurrencyStrategy 158
 NHibernate.Cache.ICache-Provider 159
 NHibernate.ConnectionReleaseMode 139
 NHibernate.Context namespace 327
 NHibernate.SQL 275
 NHibernate.Tasks.
 Hbm2Net-Task 292
 NHibernate.Tool.hbm2ddl.
 SchemaExport 288
 NHibernate.Tool.hbm2net.
 Console 292
 NHibernateHelper class 321
 NHibernateUtil.Initialize() 131
 NHibernateUtil.IsInitialized() 131
 noise 338
 nonindexed collection 198
 non-intrusive 12
 nonstrict-read-write
 concurrency strategy 158
 nosetter.* strategy 69
 not-null attribute 290
 not-null column, avoiding 188
 null operator 219
 null value, testing for 219
 nullable type 179–180

O

object
 creating. *See* persistence
 detached 78
 equality 76–77
 See also identity
 graph 10
 identity 76–81
 making persistent 110–111

retrieving 121–133
 by identifier 122–123
 See also domain model
 object identity and caching 154
 object referencing 16
 object retrieval, optimizing
 p249–255
 object role modeling 21
 object.ReferenceEquals() 76–77
 object/relational impedance
 mismatch 12
 object/relational mapping
 (ORM) 4, 14
 defined 21–23
 problems in 125
 reasons to use 21–23
 triggers combined with 303
 without domain model 54
 object/relational persistence 76
 ObjectDataSource 314
 object-oriented programming
 (OOP) 7, 12
 object-oriented viewpoint 12
 ObjectViews 13
 data binding with 315
 Observer pattern 278
 applying to an entity 307–309
 on clause, specifying join
 condition 223
 one-to-many 87–88
 one-to-many association 198
 one-to-one association
 189–193, 232
 implicit join 228
 OnFlushDirty() 281
 OnSave() 281–282
 OOP. *See* object-oriented
 programming (OOP)
 OpenSession() 28
 operations, grouping 60
 operator
 comparison, and
 restriction 218
 logical 221
 optimistic locking 142, 147
 alternative
 implementations 151
 vs. pessimistic 149
 optimistic offline locking
 147–149
 optimistic-lock attribute 151
 Oracle. *See* relational database
 management system
 order by clause 221
 order-by 185

ordered collection 184
 ordered pair 226
 ORM. *See* object/relational
 mapping (ORM)
 outer join 223
 outer-join attribute 251
 outer-join loading 126

P

pagination 209
 in SQL 210
 paradigm mismatch
 5, 12, 15–16
 parallel class hierarchies
 smell 346
 parameters 212
 binding 211–215
 importance of 211
 positional 212
 parent/child relationship 90–91
 pattern 261–262
 Data Access Object 324–326
 Data Transfer Object 345
 Gateway 325
 Repository 325
 Pattern Language of Programs
 (PLoP) conference 262
 performance 22, 273
 advice on eager loading 334
 improving 275
 with report queries 236
 performance tuning 249
 persistence 5–9
 API 110–114
 approaches in .NET 9–15
 automated 55–56
 automatic dirty checking
 103, 113
 by reachability 115–116
 cascading 116–117
 choices 4
 context 327
 conversation 17
 deleting 103, 113–114
 detaching domain model 104
 dirty checking 62
 hand-coding 10, 14
 ignorance of 101
 lifecycle 36, 101–110
 management 110–114
 mechanism 5
 optimization 132–133
 querying
 using HQL 123–124

persistence (*continued*)
 retrieving 112
 techniques 121–133
 saving 102, 110–111
 state 101
 detached 103–105
 persistent 102–103
 transient 102
 transitive 114–121
 transparency issue 109
 transparent 17, 55–56
 tuning 132–133
 updating 111–112
 transparently 113
 using NHibernate 10
 persistence ignorance 305–309
 persistence layer 8, 15, 268–269
 choosing 9
 designing 320–335
 generic 326–335
 implementing 268, 321–326
 testing 269
 persistence logic, testing 269
 persistence manager
 56, 110–114
 persistence-abstracted
 entity 307
 persistence-related code
 260, 320
 abstracting 305–307
 persistent instance, caches in
 sessions 340
 pessimistic lock 339
 pessimistic locking 143–146
 not available 325
 vs. optimistic 149
 phantom read 141
 placeholder 245
 Plain Old CLR Object
 (POCO) 56, 344
 DTO as 345
 making serializable 345
 persistent class
 generation 293
 POCO. *See* Plain Old CLR
 Object (POCO)
 polymorphic association 201
 and table-per-concrete-
 class 204–205
 polymorphic collection 203
 polymorphic query 217
 polymorphism 7, 12
 association 92
 mapping 200–205
 query 92

portability 56
 positional parameter 212
 PostFlush() 281
 presentation layer 8, 269–270
 DataSet-based 13
 implementing 269
 web pages as 271
 presentation model 314
 Prevalence cache provider 159
 primary key 12, 16, 77, 230
 choosing 79–81
 natural 79
 primary key association 192–193
 primitive types 81
 problem domain 53
 process scope cache 153
 process-scoped identity 103–104
 productivity 22
 profiler 255
 projection 124, 232–234, 350
 property
 access 68–70
 derived 68
 mapping 66–68
 reading and setting state 63
 property accessor 68
 PropertyToColumnName() 73
 proxy 122, 127, 131
 problems with 202
 proxy-safe typecast 202

Q

QBC. *See* Query by Criteria
 (QBC) API
 QBE. *See* Query by Example
 (QBE)
 <qualifyAssembly> 43
 quantification 242, 351
 query
 advanced techniques 238–243
 building with string
 manipulations 238
 by example 238–240
 caching 253–255
 comparisons between
 keys 230
 complex 15, 207
 complex criteria 221
 criteria, comparison
 operators 219
 dynamic 238–240
 entity 244
 Enumerable() 252
 executing 208–215
 externalizing strings to
 mapping metadata 214
 identifier value, comparisons
 with 231
 iterating results 210, 252
 listing results 210
 named 214–215
 native SQL 243–249
 NHibernate API 36
 object reference comparisons
 with 230
 optimizing 215
 parameter binding 211–214
 polymorphic 217
 polymorphism 16, 92
 porting to mapping files 215
 QBE and dynamic 239
 report 231–237
 restriction 217
 results, ordering 221
 retrieving multiple entity
 types 245
 root entities of criteria 209
 scalar 244
 simplest 215
 SQL, named 246–248
 calling stored
 procedures 247
 storing 214
 substitutions 215
 using aliases 216
 using enumerated types
 in 180
 using LINQ 20
 ways of expressing in
 Hibernate 208
 Query APIs
 binding arbitrary arguments
 with 213
 creating a new instance
 of 208–211
 Enumerable() method
 and 252
 List() method and 252
 method-chaining 210
 purpose of 208
 testing 216
 Query by Criteria (QBC)
 API 19, 124, 207
 Query by Example (QBE)
 124, 208, 239
 <query> element 214
 query engine,
 implementing 18–20

R

RDBMS. *See* relational database, management system

read

- committed 141
- dirty 140
- phantom 141
- uncommitted 141
- unrepeatable 141

readability 56

read-only concurrency strategy 158

read-write concurrency strategy 158

reassociation, selective, of detached instances. 106

Reconnect() 139

redundancy, minimizing 329

reference data 155

reflection 65

- optimization 70–71

Refresh() 304

relation, SQL 349

relational database 5, 7

- management system 5
- independence 22

relational model 349

relationship 91

- "has a" 91
- "is a" 91
- between entities 12
- managed 59

See also association

relationship table 130

reliability 273

repeatable read 141

report query 124, 231–237

- aggregation 234
- grouping 234
- improving performances with 236
- projection 232–234
- restricting groups with having 236
- select clause 232

Repository pattern 325

requirePermission attribute 271

restriction 217, 350

- ignoring deprecated data 218
- with comparison operators 218

result, distinct 233

<resultset> 246

retrieve command, custom 248

<return> 246

<return-join> 246

<return-scalar> 246

reusability 7

rich object model 54

roll back 135

Rollback() 136

root entity (criteria queries) 209, 228

Ruby on Rails

- migrations 295

Ruby on Rails migrations 295

S

Save() 110

SaveOrUpdate() 120

saving. *See* persistence

scaffolding code 58

scalability 273

scalar query 244

schema

- creating with hbm2ddl 290–291
- database 288
- maintenance, automatic 294–296
- mapping domain models to given 294

schema definition 63

SchemaExport 292

SchemaUpdate 295

search

- case-insensitive 220
- string-based 220
- wildcard 220

second-level cache 35

securability 273

security, database capabilities 7

select clause

- calling aggregate functions in 234
- calling SQL functions from 233
- changing results with 230
- elements of results 233
- grouping, and 234
- projection, and 232–234
- rules governing 236
- subqueries 242
- using aliases in 226–227

select new 232, 236

select-before-update 304

selective reassociation of detached instances 106

self-documenting code, achieved through refactoring 326

separation of concerns 55, 261, 263

serializability 57

serializable 141

services 277

- integrating 277–284
- loose coupling 277

session 28

- flushing 138–139
- granularity 150–151
- temporary 282–283

See also persistence manager

Session API

- kinds of state contained in 340
- long 340–344

session cache 156

- managing 157

session factory 275

session management for ASP.NET applications 332–335

session per request, when to use 344

Session.Clear() 157

session.Connection property 248

session.FlushMode 139

session.GetIdentifier(entity) 282

Session.Load() 202

session-context implementations built in to NHibernate 327

SessionFactory 35

SessionFactory API

- initialization 321
- statelessness of 322
- storage of 322

session-per-conversation 151

session-per-request 150, 333

session-per-request-with-detached-objects 151

<set> 196

- using for parent/child relationships 199

set 59

set collection 181

SetDefaultAssembly() 75

SetDefaultNamespace() 75

SetEntity() 213

SetEnum() 213

SetFetchMode() 252

SetInt32() 213
 SetMaxResults() 209
 SetParameter() 214
 SetProperties() 214
 SetString() 212
 SetTimestamp() 213
 SharpDevelop 352
 shotgun-change smell 346
 show_sql 44
 silver bullet, ORM 23
 single point association 127–129
 Singleton pattern 261
 size() function 243
 smell
 code 346
 parallel class hierarchies 346
 shotgun-change 346
 software development 3
 some quantifier 242
 sorted collection 184
 sorting database capabilities 7
 Sparx Systems Enterprise Architect 52
 SQL (Structured Query Language) 6, 9
 aggregate functions 234
 arbitrary, executing during database generation 291
 backtick 72
 basic concepts 349
 books 6
 command 14
 dynamic generation 103
 expressing queries with 208
 function 68
 inner joins 222
 keywords, writing 216
 logging 44
 named 246–248
 calling stored procedures 247
 outer joins 223
 pagination 210
 passthrough 243
 query 18
 query hints 207
 quoted identifier 72
 quoted identifiers 70
 relation 349
 schema 73–74
 subselects 242–243
 SQL databases. *See* relational database, management system
 SQL function, calling 233

SQL injection attack 211
 SQL Server 10, 42
 See also relational database, management system
 SQL statement execution, timing of 111
 SqlCommand 6
 sql-type attribute 290
 stale data 337
 StaleObjectStateException 145, 149
 state, entity 54
 state-oriented NHibernate vs ADO.NET 335
 stored procedure 7, 247
 string
 concatenation 220
 matching 220
 string manipulation, building queries with 238
 structured data 7
 subclass 94
 subquery 242–243
 correlated 242
 uncorrelated 242
 subselect 68, 242–243, 351
 sum() function 234
 surrogate key 16, 79, 109
 synthetic identifier 79
 mapping 297
 SysCache cache provider 159
 system transaction 135–146
 System.Collections.IDictionary 41
 System.Collections.SortedList 185
 System.Collections.Specialized.ListDictionary 186
 System.Data.IsolationLevel 143
 System.Diagnostics API 284
 System.EnterpriseServices assembly 346
 System.String 185

T

table
 audit logs 277
 mapping with composite keys 298–302
 mapping with natural keys 297–298
 relationship 130
 table adapter 14

table per class hierarchy 92–95
 table per concrete class 92–93
 and polymorphic associations 204–205
 table per subclass 92, 95–98
 TableName() 73
 tables, mapping with composite keys 303
 ternary association 197
 ternary logic 218, 264
 test
 data integrity 264
 logic 264–265
 testability 56
 test-driven development (TDD) 261–262
 theta-style join 223, 229
 tier, externalizing data between 345
 timestamp cache 254
 toolset, NHibernate hbm2ddl. *See* hbm2ddl (SchemaExport)
 top-down development 288–292
 ToString(object entity) method 317
 Transaction 111
 transaction 15
 boundaries
 declaring with ITransaction methods 137
 marking 136
 coarse-grained 145
 committed 135
 conversation 17
 demarcation 136
 entity 102
 fine-grained 145
 isolation 140
 caching, and 154
 levels 141
 rolled back 135
 system states 136
 transparent write-behind 103, 113
 transaction scope cache 153
 transaction.WasCommitted 137
 transactional write-behind 33
 transaction-scoped identity 104
 transience 113–114
 transient object 6
 transitive persistence 114–121
 transparent persistence 55–56, 344

- transparent transactional
 - write-behind 103, 113
- transparent write behind 138
- trigger, database 278, 303–305
- tuple 349
- type
 - custom 62
 - mapping legacy columns with 302–303
 - mapping 66
 - NHibernate 37
 - primitive 81
 - value 81–82
 - See also* domain model
- type discriminator 93, 204
- type system 167–181
- typesafe enumeration 179–181

U

- UML class diagram 52
- UMLet 52
- uncorrelated subquery 242
- unidirectional many-to-one 87–88
- Unique attribute 290
- unique constraint 290
- unique-key attribute 290
- uniqueResult() 218
- unit of work 16–18

- Unit of Work pattern 17
- unit test 55
- unit testing 262
- unrepeatable read 141
- unsaved-value 297
 - attribute 120
- update
 - command, custom 248
 - control 71
 - dynamic 71
 - lost 140
- Update() 111
- update() 339
- upper() function 220
- user interface. *See* presentation layer
- user transaction 146
- user-defined column type 15
- UserType API 173–176, 302–303
- utility class 321

V

- validation for proxy 127
- value type 81–82
 - collections of, mapping 181–189
 - storing 181–186
- VelocityRenderer 293
- version check, manual 338

- versioning
 - unsaved-value 120
 - usage with equality 108
- Visio 21, 52
- Visual Studio 6, 14
 - creating an NHibernate project 25–26

W

- web application 13, 271
 - security policy, issues with 271
- web page as presentation layer 271
- web tier, separation from business-logic tiers 345
- where clause
 - achieving restriction 217
 - arithmetic expression support by 219
 - calling SQL functions in 220
 - evaluating expressions with 218
 - restricting rows with 236
 - specifying join condition 223
 - subqueries in 242
- Windows application 13
 - implementing conversations 342
- wrapping class 314
- write-behind 47