The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Patterns of Enterprise Application Architecture

Martin Fowler

With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford

Patterns of Enterprise Application Architecture

This page intentionally left blank

Patterns of Enterprise Application Architecture

Martin Fowler

With contributions from David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford

Addison-Wesley

Boston • San Francisco • New York • Toronto • Montreal
London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Library of Congress Cataloging-in-Publication Data

Fowler, Martin, 1963-
Patterns of enterprise application architecture / Martin Fowler.
p. cm.
Includes bibliographical references and index.
ISBN 0-321-12742-0 (alk. paper)
1. System design. 2. Computer architecture. 3. Application software—Development. 4. Business—Data processing. I. Title.

QA76.9.S88 F69 2003
005.1—dc21

2002027743

ISBN 0-321-12742-0
Text printed in the United States on recycled paper at RR Donnelley in Crawfordsville, Indiana.

Seventeenth printing, July 2011

For Denys William Fowler, 1922–2000
in memoriam

—Martin

This page intentionally left blank

Contents

Preface

In the spring of 1999 I flew to Chicago to consult on a project being done by ThoughtWorks, a small but rapidly growing application development company. The project was one of those ambitious enterprise application projects: a back-end leasing system. Essentially it deals with everything that happens to a lease after you've signed on the dotted line: sending out bills, handling someone upgrading one of the assets on the lease, chasing people who don't pay their bills on time, and figuring out what happens when someone returns the assets early. That doesn't sound too bad until you realize that leasing agreements are infinitely varied and horrendously complicated. The business "logic" rarely fits any logical pattern, because, after all, it's written by business people to capture business, where odd small variations can make all the difference in winning a deal. Each of those little victories adds yet more complexity to the system.

That's the kind of thing that gets me excited: how to take all that complexity and come up with a system of objects that can make the problem more tractable. Indeed, I believe that the primary benefit of objects is in making complex logic tractable. Developing a good *Domain Model (116)* for a complex business problem is difficult but wonderfully satisfying.

Yet that's not the end of the problem. Our domain model had to be persisted to a database, and, like many projects, we were using a relational database. We also had to connect this model to a user interface, provide support to allow remote applications to use our software, and integrate our software with third-party packages. All of this on a new technology called J2EE, which nobody in the world had any real experience in using.

Even though this technology was new, we did have the benefit of experience. I'd been doing this kind of thing for ages with C++, Smalltalk, and CORBA. Many of the ThoughtWorkers had a lot of experience with Forte. We already had the key architectural ideas in our heads, and we just had to figure out how

to apply them to J2EE. Looking back on it three years later, the design is not perfect but it has stood the test of time pretty damn well.

That's the kind of situation this book was written for. Over the years I've seen many enterprise application projects. These projects often contain similar design ideas that have proven effective in dealing with the inevitable complexity that enterprise applications possess. This book is a starting point to capture these design ideas as patterns.

The book is organized in two parts, with the first part a set of narrative chapters on a number of important topics in the design of enterprise applications. These chapters introduce various problems in the architecture of enterprise applications and their solutions. However, they don't go into much detail on these solutions. The details of the solutions are in the second part, organized as patterns. These patterns are a reference, and I don't expect you to read them cover to cover. My intention is that you read the narrative chapters in Part 1 from start to finish to get a broad picture of what the book covers; then you dip into the patterns chapters of Part 2 as your interest and needs drive you. Thus, the book is a short narrative book and a longer reference book combined into one.

This is a book on enterprise application design. Enterprise applications are about the display, manipulation, and storage of large amounts of often complex data and the support or automation of business processes with that data. Examples include reservation systems, financial systems, supply chain systems, and many others that run modern business. Enterprise applications have their own particular challenges and solutions, and they are different from embedded systems, control systems, telecoms, or desktop productivity software. Thus, if you work in these other fields, there's nothing really in this book for you (unless you want to get a feel for what enterprise applications are like.) For a general book on software architecture, I'd recommend [POSA].

There are many architectural issues in building enterprise applications. I'm afraid this book can't be a comprehensive guide to them. In building software I'm a great believer in iterative development. At the heart of iterative development is the notion that you should deliver software as soon as you have something useful to the user, even if it's not complete. Although there are many differences between writing a book and writing software, this notion is one that I think the two share. That said, this book is an incomplete but (I trust) useful compendium of advice on enterprise application architecture. The primary topics I talk about are

- Layering of enterprise applications
- Structuring domain (business) logic

- Structuring a Web user interface
- Linking in-memory modules (particularly objects) to a relational database
- Handling session state in stateless environments
- Principles of distribution

The list of things I don't talk about is rather longer. I really fancied writing about organizing validation, incorporating messaging and asynchronous communication, security, error handling, clustering, application integration, architectural refactoring, structuring rich-client user interfaces, among other topics. However, because of space and time constraints and lack of cogitation, you won't find them in this book. I can only hope to see some patterns for this work in the near future. Perhaps I'll do a second volume someday and get into these topics, or maybe someone else will fill these and other gaps.

Of these, message-based communication is a particularly big issue. People who are integrating multiple applications are increasingly making use of asynchronous message-based communication approaches. There's much to be said for using them within an application as well.

This book is not intended to be specific for any particular software platform. I first came across these patterns while working with Smalltalk, C++, and CORBA in the late '80s and early '90s. In the late '90s I started to do extensive work in Java and found that these patterns applied well to both early Java/CORBA systems and later J2EE-based work. More recently I've been doing some initial work with Microsoft's .NET platform and find the patterns apply again. My ThoughtWorks colleagues have also introduced their experiences, particularly with Forte. I can't claim generality across all platforms that have ever been or will be used for enterprise applications, but so far these patterns have shown enough recurrence to be useful.

I have provided code examples for most of the patterns. My choice of language for them is based on what I think most readers are likely to be able to read and understand. Java is a good choice here. Anyone who can read C or C++ can read Java, yet Java is much less complex than C++. Essentially most C++ programmers can read Java but not vice versa. I'm an object bigot, so I inevitably lean to an OO language. As a result, most of the code examples are in Java. As I was working on the book, Microsoft started stabilizing its .NET environment, and its C# language has most of the same properties as Java for an author. So I did some of the code examples in C# as well, although that introduced some risk since developers don't have much experience with .NET and so the idioms for using it well are less mature. Both are C-based languages, so if you can read one

you should be able to read both, even if you aren't deeply into that language or platform. My aim was to use a language that the largest amount of software developers can read, even if it's not their primary or preferred language. (My apologies to those who like Smalltalk, Delphi, Visual Basic, Perl, Python, Ruby, COBOL, or any other language. I know you think you know a better language than Java or C#. All I can say is I do, too!)

The examples are there for inspiration and explanation of the ideas in the patterns. They aren't canned solutions; in all cases you'll need to do a fair bit of work to fit them into your application. Patterns are useful starting points, but they are not destinations.

Who This Book Is For

I've written this book for programmers, designers, and architects who are building enterprise applications and who want to improve either their understanding of architectural issues or their communication about them.

I'm assuming that most of my readers will fall into two groups: those with modest needs who are looking to build their own software and readers with more demanding needs who will be using a tool. For those of modest needs, my intention is that these patterns should get you started. In many areas you'll need more than the patterns will give you, but I'll provide you more of a headstart in this field than I got. For tool users I hope this book will give you some idea of what's happening under the hood and also help you choose which of the tool-supported patterns to use. Using, say, an object-relational mapping tool still means that you have to make decisions about how to map certain situations. Reading the patterns should give you some guidance in making the choices.

There is a third category; those with demanding needs who want to build their own software. The first thing I'd say here is to look carefully at using tools. I've seen more than one project get sucked into a long exercise at building frameworks, which wasn't what the project was really about. If you're still convinced, go ahead. Remember in this case that many of the code examples in this book are deliberately simplified to help understanding, and you'll find you'll need to do a lot tweaking to handle the greater demands you face.

Since patterns are common solutions to recurring problems, there's a good chance that you have already come across some of them. If you've been working in enterprise applications for a while, you may well know most of them. I'm not claiming to present anything new in this book. Indeed, I claim the opposite—this is a book of (for our industry) old ideas. If you're new to this field, I

hope the book will help you learn about these techniques. If you're familiar with the techniques, I hope the book will help you communicate and teach them to others. An important part of patterns is trying to build a common vocabulary, so you can say that this class is a *Remote Facade (388)* and other designers will know what you mean.

Acknowledgments

As with any book, what's written here has a great deal to do with the many people who have worked with me in various ways over the years. Lots of people have helped in lots of ways. Often I don't recall important things people said that went into this book, but I can acknowledge those contributions I do remember.

I'll start with my contributors. David Rice, a colleague of mine at ThoughtWorks, has made a huge contribution—a good tenth of the book. As we worked hard to hit the deadline (while he was also supporting a client), we had several late-night instant message conversations where he confessed to finally seeing why writing a book is both so hard and so compulsive.

Matt Foemmel is another ThoughtWorker, and although the Arctic will need air conditioning before he writes prose for fun, he's been a great contributor of code examples (as well as a very succinct critic of the book.) I was pleased that Randy Stafford contributed *Service Layer (133)* as he's been such a strong advocate for it. I'd also like to thank Edward Hieatt and Rob Mee for their contribution, which arose from Rob's noticing a gap while he was doing his review of the text. He became my favorite reviewer: Not only does he notice something missing, he helps write a section to fix it!

As usual, I owe more than I can say to my first-class panel of official reviewers:

John Brewer	Rob Mee
Kyle Brown	Gerard Meszaros
Jens Coldewey	Dirk Riehle
John Crupi	Randy Stafford
Leonard Fenster	David Siegel
Alan Knight	Kai Yu

I could almost list the ThoughtWorks telephone directory here, for so many of my colleagues have helped this project by talking over their designs and experiences with me. Many patterns formed in my mind because I had the

opportunity to talk with the many talented designers we have, so I have little choice but to thank the whole company.

Kyle Brown, Rachel Reinitz, and Bobby Woolf have gone out of their way to have long and detailed review sessions with me in North Carolina. Their fine-tooth comb has injected all sorts of wisdom, not including this particularly heinous mixed metaphor. In particular I've enjoyed several long telephone calls with Kyle that contributed more than I can list.

Early in 2000 I prepared a talk for Java One with Alan Knight and Kai Yu that was the earliest genesis of this material. As well as thanking them for their help in that, I should also thank Josh Mackenzie, Rebecca Parsons, and Dave Rice for helping me refine these talks, and the ideas, later on. Jim Newkirk did a great deal in helping me get used to the new world of .NET.

I've learned a lot from the many people working in this field with whom I've had good conversations and collaborations. In particular I'd like to thank Colleen Roe, David Muirhead, and Randy Stafford for sharing their work on the Foodsmart example system at Gemstone. I've also had great conversations at the Crested Butte workshop that Bruce Eckel has hosted and must thank all the people who attended that event in the last couple of years. Joshua Kerievsky didn't have time to do a full review, but he was an excellent patterns consultant.

As usual, I had the remarkable help of the UIUC reading group with their unique brand of no-holds-barred audio reviews. My thanks to: Ariel Gertzenstein, Bosko Zivaljevic , Brad Jones, Brian Foote, Brian Marick, Federico Balaguer, Joseph Yoder, John Brant, Mike Hewner, Ralph Johnson, and Weerasak Witthawaskul.

Dragos Manolescu, an ex-UIUC hitman, got his own group together to give me feedback. My thanks to Muhammad Anan, Brian Doyle, Emad Ghosheh, Glenn Graessle, Daniel Hein, Prabhaharan Kumarakulasingam, Joe Quint, John Reinke, Kevin Reynolds, Sripriya Srinivasan, and Tirumala Vaddiraju.

Kent Beck has given me more good ideas than I can remember. But I do remember that he came up with the name for *Special Case (496)*. Jim Odell was responsible for getting me into the world of consulting, teaching, and writing—no acknowledgment will ever do his help justice.

As I was writing this book, I put drafts on the Web. During this time many people sent me e-mails pointing out problems, asking questions, or talking about alternatives. These people include Michael Banks, Mark Bernstein, Graham Berrisford, Bjorn Beskow, Bryan Boreham, Sean Broadley, Peris Brodsky, Paul Campbell, Chester Chen, John Coakley, Bob Corrick, Pascal Costanza, Andy Czerwonka, Martin Diehl, Daniel Drasin, Juan Gomez Duaso, Don Dwiggins, Peter Foreman, Russell Freeman, Peter Gassmann, Jason Gorman, Dan Green,

Lars Gregori, Rick Hansen, Tobin Harris, Russel Healey, Christian Heller, Richard Henderson, Kyle Hermenean, Carsten Heyl, Akira Hirasawa, Eric Kaun, Kirk Knoernschild, Jesper Ladegaard, Chris Lopez, Paolo Marino, Jeremy Miller, Ivan Mitrovic, Thomas Neumann, Judy Obee, Paolo Parovel, Trevor Pinkney, Tomas Restrepo, Joel Rieder, Matthew Roberts, Stefan Roock, Ken Rosha, Andy Schneider, Alexandre Semenov, Stan Silvert, Geoff Soutter, Volker Termath, Christopher Thames, Volker Turau, Knut Wannheden, Marc Wallace, Stefan Wenig, Brad Wiemerslage, Mark Windholtz, Michael Yoon.

There are many others who gave input whose names I either never knew or can't remember, but my thanks is no less heartfelt.

My biggest thanks is, as ever, to my wife Cindy, whose company I appreciate much more than anyone can appreciate this book.

Colophon

This is the first book that I wrote using XML and related technologies. The master text was written as a series of XML documents using trusty TextPad. I also used a home-grown DTD. While I was working I used XSLT to generate the web pages for the HTML site. For the diagrams I relied on my old friend Visio using Pavel Hruby's wonderful UML templates (much better than those that come with the tool. I have a link on my Web site if you want them.) I wrote a small program that automatically imported the code examples into the output, which saved me from the usual nightmare of code cut and paste. For my first draft I tried XSL-FO with Apache FOP. At the time it wasn't quite up to the job, so for later work I wrote scripts in XSLT and Ruby to import the text into FrameMaker.

I used several open source tools while working on this book—in particular, JUnit, NUnit, ant, Xerces, Xalan, Tomcat, Jboss, Ruby, and Hsql. My thanks to the many developers of these tools. There was also a long list of commercial tools. In particular, I relied on Visual Studio for .NET and on IntelliJ's wonderful Idea—the first IDE that's excited me since Smalltalk—for Java.

The book was acquired for Addison Wesley by Mike Hendrickson who, assisted by Ross Venables, has supervised its publication. I started work on the manuscript in November 2000 and released the final draft to production in June 2002. As I write this, the book is due for release in November 2002 at OOPSLA.

Sarah Weaver was the production editor, coordinating the editing, composition, proofreading, indexing, and production of final files. Dianne Wood was

the copy editor, carrying out the tricky job of cleaning up my English without introducing any untoward refinement. Kim Arney Mulcahy composed the book into the design you see here, cleaned up the diagrams, set the text in Sabon, and prepared the final Framemaker files for the printer. The text design is based on the format we used for *Refactoring*. Cheryl Ferguson proofread the pages and ferreted out any errors that had slipped through the cracks. Irv Hershman prepared the index.

About the Cover Picture

During the couple of years I spent writing this book a more significant construction project was going on in Boston. The Leonard P. Zakim Bunker Hill Bridge (try fitting that name on a road sign) will replace the ugly double-decker that now carries Interstate 93 over the Charles River. The Zakim bridge is a cable-stayed bridge, a style that hasn't been widely used in the U.S. so far, but is very popular in Europe. The Zakim bridge isn't particularly long, but it is the world's widest cable-stayed bridge and also the first U.S. cable-stayed bridge to have an asymmetric design. It's a very beautiful bridge, but that doesn't stop me from teasing Cindy about Henry Petroski's conjecture that we are due for a major failure in a cable-stayed bridge soon.

Martin Fowler, Melrose, Massachusetts, August 2002
http://martinfowler.com

Introduction

In case you haven't realized it, building computer systems is hard. As the complexity of the system gets greater, the task of building the software gets exponentially harder. As in any profession, we can progress only by learning, both from our mistakes and from our successes. This book represents some of this learning written in a form that I hope will help you to learn these lessons quicker than I did, or to communicate to others more effectively than I did before I boiled these patterns down.

In this introduction I want to set the scope of the book and provide some of the background that will underpin its ideas.

Architecture

The software industry delights in taking words and stretching them into a myriad of subtly contradictory meanings. One of the biggest sufferers is "architecture." I tend to look at "architecture" as one of those impressive-sounding words, used primarily to indicate that we're talking something that's important. But I'm pragmatic enough not to let my cynicism get in the way of attracting people to my book. :-)

"Architecture" is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime.

From time to time Ralph Johnson has a truly remarkable posting on a mailing list, and he did one on architecture just as I was finishing the draft of this book. In this posting he brought out the point that architecture is a subjective thing, a shared understanding of a system's design by the expert developers on a

project. Commonly this shared understanding is in the form of the major components of the system and how they interact. It's also about decisions, in that it's the decisions that developers wish they could get right early on because they're perceived as hard to change. The subjectivity comes in here as well because, if you find that something is easier to change than you once thought, then it's no longer architectural. In the end architecture boils down to the important stuff—whatever that is.

In this book I present my perception of the major parts of an enterprise application and of the decisions I wish I could get right early on. The architectural pattern I like the most is that of layers, which I describe more in Chapter 1. This book is thus about how you decompose an enterprise application into layers and how these layers work together. Most nontrivial enterprise applications use a layered architecture of some form, but in some situations other approaches, such as pipes and filters, are valuable. I don't go into those situations, focusing instead on the context of a layered architecture because it's the most widely useful.

Some of the patterns in this book can reasonably be called architectural, in that they represent significant decisions about these parts; others are more about design and help you to realize that architecture. I don't make any strong attempt to separate the two, since what is architectural or not is so subjective.

Enterprise Applications

Lots of people write computer software, and we call all of it software development. However, there are distinct kinds of software out there, each of which has its own challenges and complexities. This comes out when I talk with some of my friends in the telecom field. In some ways enterprise applications are much easier than telecoms software—we don't have very hard multithreading problems, and we don't have hardware and software integration. But in other ways it's much tougher. Enterprise applications often have complex data—and lots of it—to work on, together with business rules that fail all tests of logical reasoning. Although some techniques and patterns are relevant for all kinds of software, many are relevant for only one particular branch.

In my career I've concentrated on enterprise applications, so my patterns here are all about that. (Other terms for enterprise applications include "information systems" or, for those with a long memory, "data processing.") But what do I mean by the term "enterprise application"? I can't give a precise definition, but I can give some indication of my meaning.

I'll start with examples. Enterprise applications include payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading. Enterprise applications don't include automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games.

Enterprise applications usually involve **persistent data**. The data is persistent because it needs to be around between multiple runs of the program—indeed, it usually needs to persist for several years. Also during this time there will be many changes in the programs that use it. It will often outlast the hardware that originally created much of it, and outlast operating systems and compilers. During that time there'll be many changes to the structure of the data in order to store new pieces of information without disturbing the old pieces. Even if there's a fundamental change and the company installs a completely new application to handle a job, the data has to be migrated to the new application.

There's usually **a lot of data**—a moderate system will have over 1 GB of data organized in tens of millions of records—so much that managing it is a major part of the system. Older systems used indexed file structures such as IBM's VSAM and ISAM. Modern systems usually use databases, mostly relational databases. The design and feeding of these databases has turned into a subprofession of its own.

Usually many people **access data concurrently**. For many systems this may be less than a hundred people, but for Web-based systems that talk over the Internet this goes up by orders of magnitude. With so many people there are definite issues in ensuring that all of them can access the system properly. But even without that many people, there are still problems in making sure that two people don't access the same data at the same time in a way that causes errors. Transaction manager tools handle some of this burden, but often it's impossible to hide this from application developers.

With so much data, there's usually **a lot of user interface screens** to handle it. It's not unusual to have hundreds of distinct screens. Users of enterprise applications vary from occasional to regular, and normally they will have little technical expertise. Thus, the data has to be presented lots of different ways for different purposes. Systems often have a lot of batch processing, which is easy to forget when focusing on use cases that stress user interaction.

Enterprise applications rarely live on an island. Usually they need to **integrate with other enterprise applications** scattered around the enterprise. The various systems are built at different times with different technologies, and even the collaboration mechanisms will be different: COBOL data files, CORBA, messaging systems. Every so often the enterprise will try to integrate

its different systems using a common communication technology. Of course, it hardly ever finishes the job, so there are several different unified integration schemes in place at once. This gets even worse as businesses seek to integrate with their business partners as well.

Even if a company unifies the technology for integration, they run into problems with differences in business process and **conceptual dissonance** with the data. One division of the company may think a customer is someone with whom it has a current agreement; another division also counts those that had a contract but don't any longer; another counts product sales but not service sales. That may sound easy to sort out, but when you have hundreds of records in which every field can have a subtly different meaning, the sheer size of the problem becomes a challenge—even if the only person who knows what the field really means is still with the company. (And, of course, all of this changes without warning.) As a result, data has to be constantly read, munged, and written in all sorts of different syntactic and semantic formats.

Then there's the matter of what comes under the term "business logic." I find this a curious term because there are few things that are less logical than business logic. When you build an operating system you strive to keep the whole thing logical. But business rules are just given to you, and without major political effort there's nothing you can do to change them. You have to deal with a haphazard array of strange conditions that often interact with each other in surprising ways. Of course, they got that way for a reason: Some salesman negotiated to have a certain yearly payment two days later than usual because that fit with his customer's accounting cycle and thus won a couple of million dollars in business. A few thousand of these one-off special cases is what leads to the **complex business "illogic"** that makes business software so difficult. In this situation you have to organize the business logic as effectively as you can, because the only certain thing is that the logic will change over time.

For some people the term "enterprise application" implies a large system. However, it's important to remember that not all enterprise applications are large, even though they can provide a lot of value to the enterprise. Many people assume that, since small systems aren't large, they aren't worth bothering with, and to some degree there's merit here. If a small system fails, it usually makes less noise than a big system. Still, I think such thinking tends to shortchange the cumulative effect of many small projects. If you can do things that improve small projects, then that cumulative effect can be very significant on an enterprise, particularly since small projects often have disproportionate value. Indeed, one of the best things you can do is turn a large project into a small one by simplifying its architecture and process.

Kinds of Enterprise Application

When we discuss how to design enterprise applications, and what patterns to use, it's important to realize that enterprise applications are all different and that different problems lead to different ways of doing things. I have a set of alarm bells that go off when people say, "Always do this." For me much of the challenge (and interest) in design is in knowing about alternatives and judging the trade-offs of using one alternative over another. There is a large space of alternatives to choose from, but here I'll pick three points on this very big plane.

Consider a B2C (business to customer) online retailer: People browse and—with luck and a shopping cart—buy. For such a system we need to be able to handle a very high volume of users, so our solution needs to be not only reasonably efficient in terms of resources used but also scalable so that you can increase the load by adding more hardware. The domain logic for such an application can be pretty straightforward: order capturing, some relatively simple pricing and shipping calculations, and shipment notification. We want anyone to be able access the system easily, so that implies a pretty generic Web presentation that can be used with the widest possible range of browsers. Data source includes a database for holding orders and perhaps some communication with an inventory system to help with availability and delivery information.

Contrast this with a system that automates the processing of leasing agreements. In some ways this is a much simpler system than the B2C retailer's because there are many fewer users—no more than a hundred or so at one time. Where it's more complicated is in the business logic. Calculating monthly bills on a lease, handling events such as early returns and late payments, and validating data as a lease is booked are all complicated tasks, since much of the leasing industry's competition comes in the form of little variations over deals done in the past. A complex business domain such as this is challenging because the rules are so arbitrary.

Such a system also has more complexity in the user interface (UI). At the least this means a much more involved HTML interface with more, and more complex, screens. Often these systems have UI demands that lead users to want a more sophisticated presentation than a HTML front end allows, so a more conventional rich-client interface is needed. A more complex user interaction also leads to more complicated transaction behavior: Booking a lease may take an hour or two, during which time the user is in a logical transaction.We also see a complex database schema with perhaps two hundred tables and connections to packages for asset valuation and pricing.

A third example point is a simple expense-tracking system for a small company. Such a system has few users and simple logic and can easily be made accessible across the company with an HTML presentation. The only data source is a few tables in a database. As simple as it is, a system like this is not devoid of a challenge. You have to build it very quickly and you have to bear in mind that it may grow as people want to calculate reimbursement checks, feed them into the payroll system, understand tax implications, provide reports for the CFO, tie into airline reservation Web services, and so on. Trying to use the architecture for either of the other two example systems will slow down the development of this one. If a system has business benefits (as all enterprise applications should), delaying those benefits costs money. However, you don't want to make decisions now that will hamper future growth. But if you add flexibility now and get it wrong, the complexity added for flexibility's sake may actually make it harder to evolve in the future and may delay deployment and thus delay the benefit. Although such systems may be small, most enterprises have a lot of them so the cumulative effect of an inappropriate architecture can be significant.

Each of these three enterprise application examples has difficulties, and they are different difficulties. As a result you can't come up with a single architecture that will be right for all three. Choosing an architecture means that you have to understand the particular problems of your system and choose an appropriate design based on that understanding. That's why in this book I don't give a single solution for your enterprise needs. Instead, many of the patterns are about choices and alternatives. Even when you choose a particular pattern, you'll have to modify it to meet your demands. You can't build enterprise software without thinking, and all any book can do is give you more information to base your decisions on.

If this applies to patterns, it also applies to tools. Although it obviously makes sense to pick as small a set of tools as you can to develop applications, you also have to recognize that different tools are best for different purposes. Beware of using a tool that is really suited for a different kind of application—it may hinder more than help.

Thinking About Performance

Many architectural decisions are about performance. For most performance issues I prefer to get a system up and running, instrument it, and then use a disciplined optimization process based on measurement. However, some architec-

tural decisions affect performance in a way that's difficult to fix with later optimization. And even when it is easy to fix, people involved in the project worry about these decisions early.

It's always difficult to talk about performance in a book such as this. The reason that it's so difficult is that any advice about performance should not be treated as fact until it's measured on your configuration. Too often I've seen designs used or rejected because of performance considerations, which turn out to be bogus once somebody actually does some measurements on the real setup used for the application.

I give a few guidelines in this book, including minimizing remote calls, which has been good performance advice for quite a while. Even so, you should verify every tip by measuring on your application. Similarly there are several occasions where code examples in this book sacrifice performance for understandability. Again it's up to you to apply the optimizations for your environment. Whenever you do a performance optimization, however, you must measure both before and after, otherwise, you may just be making your code harder to read.

There's an important corollary to this: A significant change in configuration may invalidate any facts about performance. Thus, if you upgrade to a new version of your virtual machine, hardware, database, or almost anything else, you must redo your performance optimizations and make sure they're still helping. In many cases a new configuration can change things. Indeed, you may find that an optimization you did in the past to improve performance actually hurts performance in the new environment.

Another problem with talking about performance is the fact that many terms are used in an inconsistent way. The most noted victim of this is "scalability," which is regularly used to mean half a dozen different things. Here are the terms I use.

Response time is the amount of time it takes for the system to process a request from the outside. This may be a UI action, such as pressing a button, or a server API call.

Responsiveness is about how quickly the system acknowledges a request as opposed to processing it. This is important in many systems because users may become frustrated if a system has low responsiveness, even if its response time is good. If your system waits during the whole request, then your responsiveness and response time are the same. However, if you indicate that you've received the request before you complete, then your responsiveness is better. Providing a progress bar during a file copy improves the responsiveness of your user interface, even though it doesn't improve response time.

Latency is the minimum time required to get any form of response, even if the work to be done is nonexistent. It's usually the big issue in remote systems. If I ask a program to do nothing, but to tell me when it's done doing nothing, then I should get an almost instantaneous response if the program runs on my laptop. However, if the program runs on a remote computer, I may get a few seconds just because of the time taken for the request and response to make their way across the wire. As an application developer, I can usually do nothing to improve latency. Latency is also the reason why you should minimize remote calls.

Throughput is how much stuff you can do in a given amount of time. If you're timing the copying of a file, throughput might be measured in bytes per second. For enterprise applications a typical measure is transactions per second (tps), but the problem is that this depends on the complexity of your transaction. For your particular system you should pick a common set of transactions.

In this terminology **performance** is either throughput or response time—whichever matters more to you. It can sometimes be difficult to talk about performance when a technique improves throughput but decreases response time, so it's best to use the more precise term. From a user's perspective responsiveness may be more important than response time, so improving responsiveness at a cost of response time or throughput will increase performance.

Load is a statement of how much stress a system is under, which might be measured in how many users are currently connected to it. The load is usually a context for some other measurement, such as a response time. Thus, you may say that the response time for some request is 0.5 seconds with 10 users and 2 seconds with 20 users.

Load sensitivity is an expression of how the response time varies with the load. Let's say that system A has a response time of 0.5 seconds for 10 through 20 users and system B has a response time of 0.2 seconds for 10 users that rises to 2 seconds for 20 users. In this case system A has a lower load sensitivity than system B. We might also use the term **degradation** to say that system B degrades more than system A.

Efficiency is performance divided by resources. A system that gets 30 tps on two CPUs is more efficient than a system that gets 40 tps on four identical CPUs.

The **capacity** of a system is an indication of maximum effective throughput or load. This might be an absolute maximum or a point at which the performance dips below an acceptable threshold.

Scalability is a measure of how adding resources (usually hardware) affects performance. A scalable system is one that allows you to add hardware and get a commensurate performance improvement, such as doubling how many serv-

ers you have to double your throughput. **Vertical scalability**, or **scaling up**, means adding more power to a single server, such as more memory. **Horizontal scalability**, or **scaling out**, means adding more servers.

The problem here is that design decisions don't affect all of these performance factors equally. Say we have two software systems running on a server: Swordfish's capacity is 20 tps while Camel's capacity is 40 tps. Which has better performance? Which is more scalable? We can't answer the scalability question from this data, and we can only say that Camel is more efficient on a single server. If we add another server, we notice that Swordfish now handles 35 tps and Camel handles 50 tps. Camel's capacity is still better, but Swordfish looks like it may scale out better. If we continue adding servers we'll discover that Swordfish gets 15 tps per extra server and Camel gets 10. Given this data we can say that Swordfish has better horizontal scalability, even though Camel is more efficient for less than five servers.

When building enterprise systems, it often makes sense to build for hardware scalability rather than capacity or even efficiency. Scalability gives you the option of better performance if you need it. Scalability can also be easier to do. Often designers do complicated things that improve the capacity on a particular hardware platform when it might actually be cheaper to buy more hardware. If Camel has a greater cost than Swordfish, and that greater cost is equivalent to a couple of servers, then Swordfish ends up being cheaper even if you only need 40 tps. It's fashionable to complain about having to rely on better hardware to make our software run properly, and I join this choir whenever I have to upgrade my laptop just to handle the latest version of Word. But newer hardware is often cheaper than making software run on less powerful systems. Similarly, adding more servers is often cheaper than adding more programmers—providing that a system is scalable.

Patterns

Patterns have been around for a long time, so part of me doesn't want to regurgitate their history yet another time. Still, this is an opportunity for me to provide my view of patterns and what makes them a worthwhile approach to describing design.

There's no generally accepted definition of a pattern, but perhaps the best place to start is Christopher Alexander, an inspiration for many pattern enthusiasts: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in

such a way that you can use this solution a million times over, without ever doing it the same way twice" [Alexander et al.]. Alexander is an architect, so he was talking about buildings, but the definition works pretty nicely for software as well. The focus of the pattern is a particular solution, one that's both common and effective in dealing with one or more recurring problems. Another way of looking at it is that a pattern is a chunk of advice and the art of creating patterns is to divide up many pieces of advice into relatively independent chunks so that you can refer to them and discuss them more or less separately.

A key part of patterns is that they're rooted in practice. You find patterns by looking at what people do, observing things that work, and then looking for the "core of the solution." It isn't an easy process, but once you've found some good patterns they become a valuable thing. For me their value lies in being able to create a book that serves as a reference. You don't need to read all of this book, or all of any patterns book, to find it useful. You just need to read enough to have a sense of what the patterns are, what problems they solve, and how they solve them. You don't need to know all the details but just enough so that if you run into one of the problems you can find the pattern in the book. Only then do you need to really understand the pattern in depth.

Once you need the pattern, you have to figure out how to apply it to your circumstances. A key thing about patterns is that you can never just apply the solution blindly, which is why pattern tools have been such miserable failures. I like to say that patterns are "half baked," meaning that you always have to finish them off in the oven of your own project. Every time I use a pattern I tweak it a little here and a little there. You see the same solution many times over, but it's never exactly the same.

Each pattern is relatively independent, but patterns aren't isolated from each other. Often one pattern leads to another or one occurs only if another is around. Thus, you'll usually only see *Class Table Inheritance (285)* if there's a *Domain Model (116)* in your design. The boundaries between the patterns are naturally fuzzy, but I've tried to make each pattern as self-standing as I can. If someone says "Use a *Unit of Work (184)*," you can look it up and see how to apply it without having to read the entire book.

If you're an experienced designer of enterprise applications, you'll probably find that most of these patterns are familiar to you. I hope you won't be too disappointed (I did try to warn you in the Preface). Patterns aren't original ideas; they're very much observations of what happens in the field. As a result, we pattern authors don't say we "invented" a pattern but rather that we "discovered" one. Our role is to note the common solution, look for its core, and then write down the resulting pattern. For an experienced designer, the value of the pattern is not that it gives you a new idea; the value lies in helping you commu-

nicate your idea. If you and your colleagues all know what a *Remote Facade (388)* is, you can communicate a lot by saying, "This class is a *Remote Facade*." It also allows you to say to someone newer, "Use a *Data Transfer Object* for this," and they can come to this book to look it up. The result is that patterns create a vocabulary about design, which is why naming is such an important issue.

While most of these patterns are truly for enterprise applications, those in the base patterns chapter (Chapter 18) are more general and localized. I include them because I refer to them in discussions of the enterprise application patterns.

The Structure of the Patterns

Every author has to choose his pattern form. Some base their forms on a classic patterns book such as [Alexander et al.], [Gang of Four], or [POSA]. Others make up their own. I've long wrestled with what makes the best form. On the one hand I don't want something as small as the GOF form; on the other hand I need to have sections that support a reference book. So this is what I've used for this book.

The first item is the name of the pattern. Pattern names are crucial, because part of the purpose of patterns is to create a vocabulary that allows designers to communicate more effectively. Thus, if I tell you my Web server is built around a *Front Controller (344)* and a *Transform View (361)* and you know these patterns, you have a very clear idea of my web server's architecture.

Next are two items that go together: the intent and the sketch. The intent sums up the pattern in a sentence or two; the sketch is a visual representation of the pattern, often but not always a UML diagram. The idea is to create a brief reminder of what the pattern is about so you can quickly recall it. If you already "have the pattern," meaning that you know the solution even if you don't know the name, then the intent and the sketch should be all you need to know what the pattern is.

The next section describes a motivating problem for the pattern. This may not be the only problem that the pattern solves, but it's one that I think best motivates the pattern.

How It Works describes the solution. In here I put a discussion of implementation issues and variations that I've come across. The discussion is as independent as possible of any particular platform—where there are platform-specific sections I've indented them so you can see them and easily skip over them. Where useful I've put in UML diagrams to help explain them.

When to Use It describes when the pattern should be used. Here I talk about the trade-offs that make you select this solution compared to others. Many of

the patterns in this book are alternatives; such *Page Controller (333)* and *Front Controller (344)*. Few patterns are always the right choice, so whenever I find a pattern I always ask myself, "When would I not use this?" That question often leads me to alternative patterns.

The *Further Reading* section points you to other discussions of this pattern. This isn't a comprehensive bibliography. I've limited my references to pieces that I think are important in helping you understand the pattern, so I've eliminated any discussion that I don't think adds much to what I've written and of course I've eliminated discussions of patterns I haven't read. I also haven't mentioned items that I think are going to be hard to find, or unstable Web links that I fear may disappear by the time you read this book.

I like to add one or more *examples*. Each one is a *simple* example of the pattern in use, illustrated with some code in Java or C#. I chose those languages because they seem to be languages that the largest number of professional programmers can read. It's absolutely essential to understand that the example is not the pattern. When you use the pattern, it won't look exactly like this example so don't treat it as some kind of glorified macro. I've deliberately kept the example as simple as possible so you can see the pattern in as clear a form as I can imagine. All sorts of issues are ignored that will become important when you use it, but these will be particular to your own environment. This is why you always have to tweak the pattern.

One of the consequences of this is that I've worked hard to keep each example as simple as I can, while still illustrating its core message. Thus, I've often chosen an example that's simple and explicit, rather than one that demonstrates how a pattern works with the many wrinkles required in a production system. It's a tricky balance between simple and simplistic, but it's also true that too many realistic yet peripheral issues can make it harder to understand the key points of a pattern.

This is also why I've gone for simple independent examples instead of a connected running examples. Independent examples are easier to understand in isolation, but give less guidance on how you put them together. A connected example shows how things fit together, but it's hard to understand any one pattern without understanding all the others involved in the example. While in theory it's possible to produce examples that are connected yet understandable independently, doing so is very hard—or at least too hard for me—so I chose the independent route.

The code in the examples is written with a focus on making the ideas understandable. As a result several things fall aside—in particular, error handling, which I don't pay much attention to since I haven't developed any patterns in

this area yet. They are there purely to illustrate the pattern. They are not intended to show how to model any particular business problem.

For these reasons the code isn't downloadable from my Web site. Each code example in this book is surrounded with too much scaffolding to simplify the basic ideas so they're worth anything in a production setting.

Not all the sections appear in all the patterns. If I couldn't think of a good example or motivation text, I left it out.

Limitations of These Patterns

As I indicated in the Preface, this collection of patterns is by no means a comprehensive guide to enterprise application development. My test for this book is not whether it's complete but merely if it's useful. The field is too big for one mind, let alone one book.

The patterns here are all ones that I've seen in the field, but I'm not going to claim I completely understand all of their ramifications and interrelationships. This book reflects my current understanding, and that understanding has developed as I've been writing the book. I expect it will continue to evolve long after this book has turned into paper. One certainty of software development is that it never stands still.

As you consider using the patterns, never forget that they're a starting point, not a final destination. There's no way that any author can see all the many variations that software projects have. I've written these patterns to help provide a beginning, so you can read about lessons that I, and the people I've observed, have learned from doing and struggling. You'll have your own struggles on top of these. Always remember that every pattern is incomplete and that you have the responsibility, and the fun, of completing it in the context of your own system.

This page intentionally left blank

Chapter 3

Mapping to Relational Databases

The role of the data source layer is to communicate with the various pieces of infrastructure that an application needs to do its job. A dominant part of this problem is talking to a database, which, for the majority of systems built today, means a relational database. Certainly there's still a lot of data in older data storage formats, such as mainframe ISAM and VSAM files, but most people building systems today worry about working with a relational database.

One of the biggest reasons for the success of relational databases is the presence of SQL, a mostly standard language for database communication. Although SQL is full of annoying and complicated vendor-specific enhancements, its core syntax is common and well understood.

Architectural Patterns

The first set of patterns comprises the architectural patterns, which drive the way in which the domain logic talks to the database. The choice you make here is far-reaching for your design and thus difficult to refactor, so it's one that you should pay some attention to. It's also a choice that's strongly affected by how you design your domain logic.

Despite SQL's widespread use in enterprise software, there are still pitfalls in using it. Many application developers don't understand SQL well and, as a result, have problems defining effective queries and commands. Although various techniques exist for embedding SQL in a programming language, they're all somewhat awkward. It would be better to access data using mechanisms that fit in with the application development language. Database administrators (DBAs) also like to get at the SQL that accesses a table so that they can understand how best to tune it and how to arrange indexes.

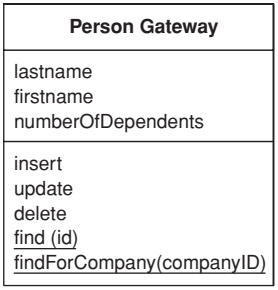

Figure 3.1 *A* Row Data Gateway (152) *has one instance per row returned by a query.*

For these reasons, it's wise to separate SQL access from the domain logic and place it in separate classes. A good way of organizing these classes is to base them on the table structure of the database so that you have one class per database table. These classes then form a *Gateway (466)* to the table. The rest of the application needs to know nothing about SQL, and all the SQL that accesses the database is easy to find. Developers who specialize in the database have a clear place to go.

There are two main ways in which you can use a *Gateway (466)*. The most obvious is to have an instance of it for each row that's returned by a query (Figure 3.1). This *Row Data Gateway (152)* is an approach that naturally fits an object-oriented way of thinking about the data.

Many environments provide a *Record Set (508)*—that is, a generic data structure of tables and rows that mimics the tabular nature of a database. Because a *Record Set (508)* is a generic data structure, environments can use it in many parts of an application. It's quite common for GUI tools to have controls that work with a *Record Set (508)*. If you use a *Record Set (508)*, you only need a single class for each table in the database. This *Table Data Gateway (144)* (see Figure 3.2) provides methods to query the database that return a *Record Set (508)*.

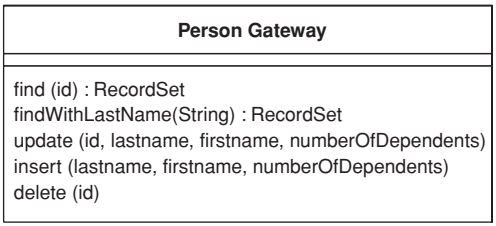

Figure 3.2 *A* Table Data Gateway (144) *has one instance per table.*

Even for simple applications I tend to use one of the gateway patterns. A glance at my Ruby and Python scripts will confirm this. I find the clear separation of SQL and domain logic to be very helpful.

The fact that *Table Data Gateway (144)* fits very nicely with *Record Set (508)* makes it the obvious choice if you are using *Table Module (125)*. It's also a pattern you can use to think about organizing stored procedures. Many designers like to do all of their database access through stored procedures rather than through explicit SQL. In this case you can think of the collection of stored procedures as defining a *Table Data Gateway (144)* for a table. I would still have an in-memory *Table Data Gateway (144)* to wrap the calls to the stored procedures, since that keeps the mechanics of the stored procedure call encapsulated.

If you're using *Domain Model (116)*, some further options come into play. Certainly you can use a *Row Data Gateway (152)* or a *Table Data Gateway (144)* with a *Domain Model (116)*. For my taste, however, that can be either too much indirection or not enough.

In simple applications the *Domain Model (116)* is an uncomplicated structure that actually corresponds pretty closely to the database structure, with one domain class per database table. Such domain objects often have only moderately complex business logic. In this case it makes sense to have each domain object be responsible for loading and saving from the database, which is *Active Record (160)* (see Figure 3.3). Another way to think of the *Active Record (160)* is that you start with a *Row Data Gateway (152)* and then add domain logic to the class, particularly when you see repetitive code in multiple *Transaction Scripts (110)*.

In this kind of situation the added indirection of a *Gateway (466)* doesn't provide a great deal of value. As the domain logic gets more complicated and you begin moving toward a rich *Domain Model (116)*, the simple approach of an *Active Record (160)* starts to break down. The one-to-one match of domain

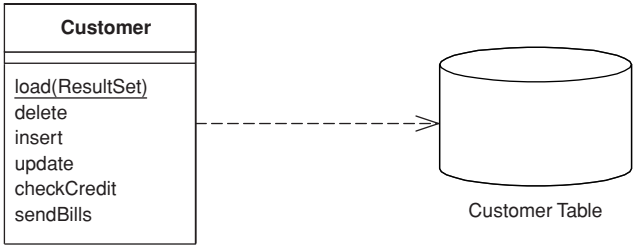

Figure 3.3 *In the* Active Record (160) *a customer domain object knows how to interact with database tables.*

classes to tables starts to fail as you factor domain logic into smaller classes. Relational databases don't handle inheritance, so it becomes difficult to use strategies [Gang of Four] and other neat OO patterns. As the domain logic gets feisty, you want to be able to test it without having to talk to the database all the time.

All of these forces push you to in'direction as your *Domain Model (116)* gets richer. In this case the *Gateway (466)* can solve some problems, but it still leaves you with the *Domain Model (116)* coupled to the schema of the database. As a result there's some transformation from the fields of the *Gateway (466)* to the fields of the domain objects, and this transformation complicates your domain objects.

A better route is to isolate the *Domain Model (116)* from the database completely, by making your indirection layer entirely responsible for the mapping between domain objects and database tables. This *Data Mapper (165)* (see Figure 3.4) handles all of the loading and storing between the database and the *Domain Model (116)* and allows both to vary independently. It's the most complicated of the database mapping architectures, but its benefit is complete isolation of the two layers.

I don't recommend using a *Gateway (466)* as the primary persistence mechanism for a *Domain Model (116)*. If the domain logic is simple and you have a close correspondence between classes and tables, *Active Record (160)* is the simple way to go. If you have something more complicated, *Data Mapper (165)* is what you need.

These patterns aren't entirely mutually exclusive. In much of this discussion we're thinking of the primary persistence mechanism, by which we mean how you save the data in some kind of in-memory model to the database. For that you'll pick one of these patterns; you don't want to mix them because that ends up getting very messy. Even if you're using *Data Mapper (165)* as your primary persistence mechanism, however, you may use a data *Gateway (466)* to wrap tables or services that are being treated as external interfaces.

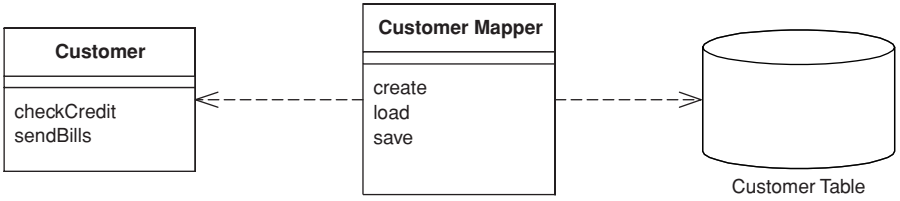

Figure 3.4 *A* Data Mapper (165) *insulates the domain objects and the database from each other.*

In my discussion of these ideas, both here and in the patterns themselves, I tend to use the word "table." However, most of these techniques can apply equally well to views, queries encapsulated through stored procedures, and commonly used dynamic queries. Sadly, there isn't a widely used term for table/view/query/stored procedure, so I use "table" because it represents a tabular data structure. I usually think of views as virtual tables, which is of course how SQL thinks of them too. The same syntax is used for querying views as for querying tables.

Updating obviously is more complicated with views and queries, as you can't always update a view directly but instead have to manipulate the tables that underlie it. In this case encapsulating the view/query with an appropriate pattern is a very good way to implement that update logic in one place, which makes using the views both simpler and more reliable.

One of the problems with using views and queries in this way is that it can lead to inconsistencies that may surprise developers who don't understand how a view is formed. They may perform updates on two different structures, both of which update the same underlying tables where the second update overwrites an update made by the first. Providing that the update logic does proper validation, you shouldn't get inconsistent data this way, but you may surprise your developers.

I should also mention the simplest way of persisting even the most complex *Domain Model (116)*. During the early days of objects many people realized that there was a fundamental "impedance mismatch" between objects and relations. Thus, there followed a spate of effort on object-oriented databases, which essentially brought the OO paradigm to disk storage. With an OO database you don't have to worry about mapping. You work with a large structure of interconnected objects, and the database figures out when to move objects on or off disks. Also, you can use transactions to group together updates and permit sharing of the data store. To programmers this seems like an infinite amount of transactional memory that's transparently backed by disk storage.

The chief advantage of OO databases is that they improve productivity. Although I'm not aware of any controlled tests, anecdotal observations put the effort of mapping to a relational database at around a third of programming effort—a cost that continues during maintenance.

Most projects don't use OO databases, however. The primary reason against them is risk. Relational databases are a well-understood and proven technology backed by big vendors who have been around a long time. SQL provides a relatively standard interface for all sorts of tools. (If you're concerned about performance, all I can say is that I haven't seen any conclusive data comparing the performance of OO against that of relational systems.)

Even if you can't use an OO database, you should seriously consider buying an O/R mapping tool if you have a *Domain Model (116)*. While the patterns in this book will tell you a lot about how to build a *Data Mapper (165)*, it's still a complicated endeavor. Tool vendors have spent many years working on this problem, and commercial O/R mapping tools are much more sophisticated than anything that can reasonably be done by hand. While the tools aren't cheap, you have to compare their price with the considerable cost of writing and maintaining such a layer yourself.

There are moves to provide an OO-database-style layer that can work with relational databases. JDO is such a beast in the Java world, but it's still too early to tell how they'll work out. I haven't had enough experience with them to draw any conclusions for this book.

Even if you do buy a tool, however, it's a good idea to be aware of these patterns. Good O/R tools give you a lot of options in mapping to a database, and these patterns will help you understand when to use the different choices. Don't assume that a tool makes all the effort go away. It makes a big dent, but you'll still find that using and tuning an O/R tool takes a small but significant chunk of work.

The Behavioral Problem

When people talk about O/R mapping, they usually focus on the structural aspects—how you relate tables to objects. However, I've found that the hardest part of the exercise is its architectural and behavioral aspects. I've already talked about the main architectural approaches; the next thing to think about is the behavioral problem.

That behavioral problem is how to get the various objects to load and save themselves to the database. At first sight this doesn't seem to be much of a problem. A customer object can have load and save methods that do this task. Indeed, with *Active Record (160)* this is an obvious route to take.

If you load a bunch of objects into memory and modify them, you have to keep track of which ones you've modified and make sure to write all of them back out to the database. If you only load a couple of records, this is easy. As you load more and more objects it gets to be more of an exercise, particularly when you create some rows and modify others since you'll need the keys from the created rows before you can modify the rows that refer to them. This is a slightly tricky problem to solve.

As you read objects and modify them, you have to ensure that the database state you're working with stays consistent. If you read some objects, it's impor-

tant to ensure that the reading is isolated so that no other process changes any of the objects you've read while you're working on them. Otherwise, you could have inconsistent and invalid data in your objects. This is the issue of concurrency, which is a very tricky problem to solve; we'll talk about this in Chapter 5.

A pattern that's essential to solving both of these problems is *Unit of Work (184)*. A *Unit of Work (184)* keeps track of all objects read from the database, together with all objects modified in any way. It also handles how updates are made to the database. Instead of the application programmer invoking explicit save methods, the programmer tells the unit of work to commit. That unit of work then sequences all of the appropriate behavior to the database, putting all of the complex commit processing in one place. *Unit of Work (184)* is an essential pattern whenever the behavioral interactions with the database become awkward.

A good way of thinking about *Unit of Work (184)* is as an object that acts as the controller of the database mapping. Without a *Unit of Work (184)*, typically the domain layer acts as the controller; deciding when to read and write to the database. The *Unit of Work (184)* results from factoring the database mapping controller behavior into its own object.

As you load objects, you have to be wary about loading the same one twice. If you do that, you'll have two in-memory objects that correspond to a single database row. Update them both, and everything gets very confusing. To deal with this you need to keep a record of every row you read in an *Identity Map (195)*. Each time you read in some data, you check the *Identity Map (195)* first to make sure that you don't already have it. If the data is already loaded, you can return a second reference to it. That way any updates will be properly coordinated. As a benefit you may also be able to avoid a database call since the *Identity Map (195)* also doubles as a cache for the database. Don't forget, however, that the primary purpose of an *Identity Map (195)* is to maintain correct identities, not to boost performance.

If you're using a *Domain Model (116)*, you'll usually arrange things so that linked objects are loaded together in such a way that a read for an order object loads its associated customer object. However, with many objects connected together any read of any object can pull an enormous object graph out of the database. To avoid such inefficiencies you need to reduce what you bring back yet still keep the door open to pull back more data if you need it later on. *Lazy Load (200)* relies on having a placeholder for a reference to an object. There are several variations on the theme, but all of them have the object reference modified so that, instead of pointing to the real object, it marks a placeholder. Only if you try to follow the link does the real object get pulled in from the database. Using *Lazy Load (200)* at suitable points, you can bring back just enough from the database with each call.

Reading in Data

When reading in data I like to think of the methods as **finders** that wrap SQL select statements with a method-structured interface. Thus, you might have methods such as `find(id)` or `findForCustomer(customer)`. Clearly these methods can get pretty unwieldy if you have 23 different clauses in your select statements, but these are, thankfully, rare.

Where you put the finder methods depends on the interfacing pattern used. If your database interaction classes are table based—that is, you have one instance of the class per table in the database—then you can combine the finder methods with the inserts and updates. If your interaction classes are row based—that is, you have one interaction class per row in the database—this doesn't work.

With row-based classes you can make the find operations static, but doing so will stop you from making the database operations substitutable. This means that you can't swap out the database for testing purposes with *Service Stub (504)*. To avoid this problem the best approach is to have separate finder objects. Each finder class has many methods that encapsulate a SQL query. When you execute the query, the finder object returns a collection of the appropriate row-based objects.

One thing to watch for with finder methods is that they work on the database state, not the object state. If you issue a query against the database to find all people within a club, remember that any person objects you've added to the club in memory won't get picked up by the query. As a result it's usually wise to do queries at the beginning.

When reading in data, performance issues can often loom large. This leads to a few rules of thumb.

Try to pull back multiple rows at once. In particular, never do repeated queries on the same table to get multiple rows. It's almost always better to pull back too much data than too little (although you have to be wary of locking too many rows with pessimistic concurrency control). Therefore, consider a situation where you need to get 50 people that you can identify by a primary key in your domain model, but you can only construct a query such that you get 200 people, from which you'll do some further logic to isolate the 50 you need. It's usually better to use one query that brings back unnecessary rows than to issue 50 individual queries.

Another way to avoid going to the database more than once is to use joins so that you can pull multiple tables back with a single query. The resulting record set looks odd but can really speed things up. In this case you may have a *Gate-*

way (466) that has data from multiple joined tables, or a *Data Mapper (165)* that loads several domain objects with a single call.

However, if you're using joins, bear in mind that databases are optimized to handle up to three or four joins per query. Beyond that, performance suffers, although you can restore a good bit of this with cached views.

Many optimizations are possible in the database. These things involve clustering commonly referenced data together, careful use of indexes, and the database's ability to cache in memory. These are outside the scope of this book but inside the scope of a good DBA.

In all cases you should profile your application with your specific database and data. General rules can guide your thinking, but your particular circumstances will always have their own variations. Database systems and application servers often have sophisticated caching schemes, and there's no way I can predict what will happen for your application. For every rule of thumb I've used, I've heard of surprising exceptions, so set aside time to do performance profiling and tuning.

Structural Mapping Patterns

When people talk about object-relational mapping, mostly what they mean is these kinds of structural mapping patterns, which you use when mapping between in-memory objects and database tables. These patterns aren't usually relevant for *Table Data Gateway (144)*, but you may use a few of them if you use *Row Data Gateway (152)* or *Active Record (160)*. You'll probably need to use all of them for *Data Mapper (165)*.

Mapping Relationships

The central issue here is the different way in which objects and relations handle links, which leads to two problems. First there's a difference in representation. Objects handle links by storing references that are held by the runtime of either memory-managed environments or memory addresses. Relational databases handle links by forming a key into another table. Second, objects can easily use collections to handle multiple references from a single field, while normalization forces all relation links to be single valued. This leads to reversals of the data structure between objects and tables. An order object naturally has a collection of line item objects that don't need any reference back to the order. However, the table structure is the other way around—the line item must

include a foreign key reference to the order since the order can't have a multi-valued field.

The way to handle the representation problem is to keep the relational identity of each object as an *Identity Field (216)* in the object, and to look up these values to map back and forth between the object references and the relational keys. It's a tedious process but not that difficult once you understand the basic technique. When you read objects from the disk you use an *Identity Map (195)* as a lookup table from relational keys to objects. Each time you come across a foreign key in the table, you use *Foreign Key Mapping (236)* (see Figure 3.5) to wire up the appropriate inter-object reference. If you don't have the key in the *Identity Map (195)*, you need to either go to the database to get it or use a *Lazy Load (200)*. Each time you save an object, you save it into the row with the right key. Any inter-object reference is replaced with the target object's ID field.

On this foundation the collection handling requires a more complex version of *Foreign Key Mapping (236)* (see Figure 3.6). If an object has a collection, you need to issue another query to find all the rows that link to the ID of the source object (or you can now avoid the query with *Lazy Load (200)*). Each object that comes back gets created and added to the collection. Saving the collection involves saving each object in it and making sure it has a foreign key to the source object. This gets messy, especially when you have to detect objects added or removed from the collection. This can get repetitive when you get the hang of it, which is why some form of metadata-based approach becomes an obvious move for larger systems (I'll elaborate on that later). If the collection

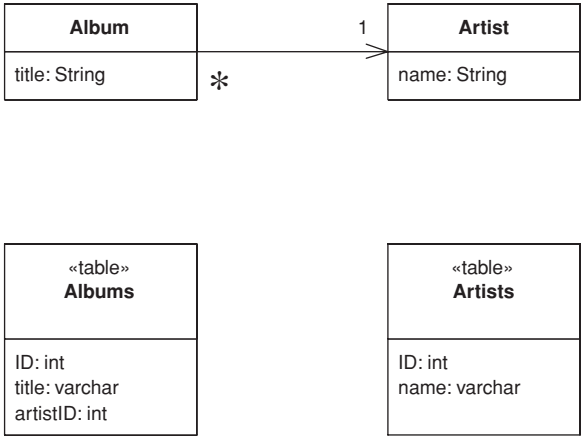

Figure 3.5 *Use a* Foreign Key Mapping (236) *to map a single-valued field.*

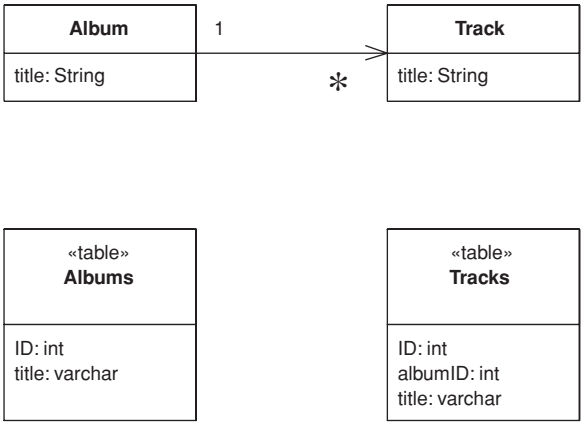

Figure 3.6 *Use a* Foreign Key Mapping (236) *to map a collection field.*

objects aren't used outside the scope of the collection's owner, you can use *Dependent Mapping (262)* to simplify the mapping.

A different case comes up with a many-to-many relationship, which has a collection on both ends. An example is a person having many skills and each skill knowing the people who use it. Relational databases can't handle this directly, so you use an *Association Table Mapping (248)* (see Figure 3.7) to create a new relational table just to handle the many-to-many association.

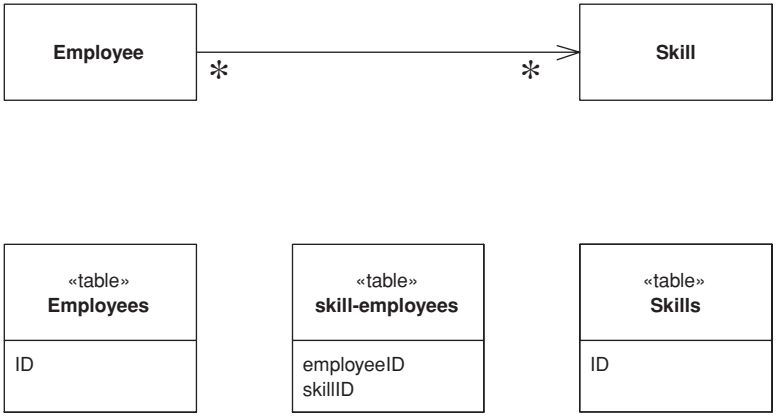

Figure 3.7 *Use an* Association Table Mapping (248) *to map a many-to-many association.*

When you're working with collections, a common gotcha is to rely on the ordering within the collection. In OO languages it's common to use ordered collections such as lists and arrays—indeed, it often makes testing easier. Nevertheless, it's very difficult to maintain an arbitrarily ordered collection when saved to a relational database. For this reason it's worth considering using unordered sets for storing collections. Another option is to decide on a sort order whenever you do a collection query, although that can be quite expensive.

In some cases referential integrity can make updates more complex. Modern systems allow you to defer referential integrity checking to the end of the transaction. If you have this capability, there's no reason not to use it. Otherwise, the database will check on every write. In this case you have to be careful to do your updates in the right order. How to do this is out of the scope of this book, but one technique is to do a topological sort of your updates. Another is to hardcode which tables get written in which order. This can sometimes reduce deadlock problems inside the database that cause transactions to roll back too often.

Identity Field (216) is used for inter-object references that turn into foreign keys, but not all object relationships need to be persisted that way. Small *Value Objects (486)*, such as date ranges and money objects clearly shouldn't be represented as their own table in the database. Instead, take all the fields of the *Value Object (486)* and embed them into the linked object as an *Embedded Value (268)*. Since *Value Objects (486)* have value semantics, you can happily create them each time you get a read and you don't need to bother with an *Identity Map (195)*. Writing them out is also easy—just dereference the object and spit out its fields into the owning table.

You can do this kind of thing on a larger scale by taking a whole cluster of objects and saving them as a single column in a table as a *Serialized LOB (272)*. LOB stands for "Large OBject," which can be either binary (BLOB) or textual (CLOB—Character Large OBject). Serializing a clump of objects as an XML document is an obvious route to take for a hierarchic object structure. This way you can grab a whole bunch of small linked objects in a single read. Often databases perform poorly with small highly interconnected objects—where you spend a lot of time making many small database calls. Hierarchic structures such as org charts and bills of materials are where a *Serialized LOB (272)* can save a lot of database roundtrips.

The downside is that SQL isn't aware of what's happening, so you can't make portable queries against the data structure. Again, XML may come to the rescue here, allowing you to embed XPath query expressions within SQL calls, although the embedding is largely nonstandard at the moment. As a result *Serialized LOB (272)* is best used when you don't want to query for the parts of the stored structure.

Usually a *Serialized LOB (272)* is best for a relatively isolated group of objects that make part of an application. If you use it too much, it ends up turning your database into little more than a transactional file system.

Inheritance

In the above hierarchies I'm talking about compositional hierarchies, such as a parts tree, which relational systems traditionally do poorly. There's another kind of hierarchy that causes relational headaches: a class hierarchy linked by inheritance. Since there's no standard way to do inheritance in SQL, we again have a mapping to perform. For any inheritance structure there are basically three options. You can have one table for all the classes in the hierarchy: *Single Table Inheritance (278)* (see Figure 3.8); one table for each concrete class: *Concrete Table Inheritance (293)* (see Figure 3.9); or one table per class in the hierarchy: *Class Table Inheritance (285)* (see Figure 3.10).

The trade-offs are all between duplication of data structure and speed of access. *Class Table Inheritance (285)* is the simplest relationship between the classes and the tables, but it needs multiple joins to load a single object, which usually reduces performance. *Concrete Table Inheritance (293)* avoids the joins, allowing you pull a single object from one table, but it's brittle to changes. With any change to a superclass you have to remember to alter all the tables (and the

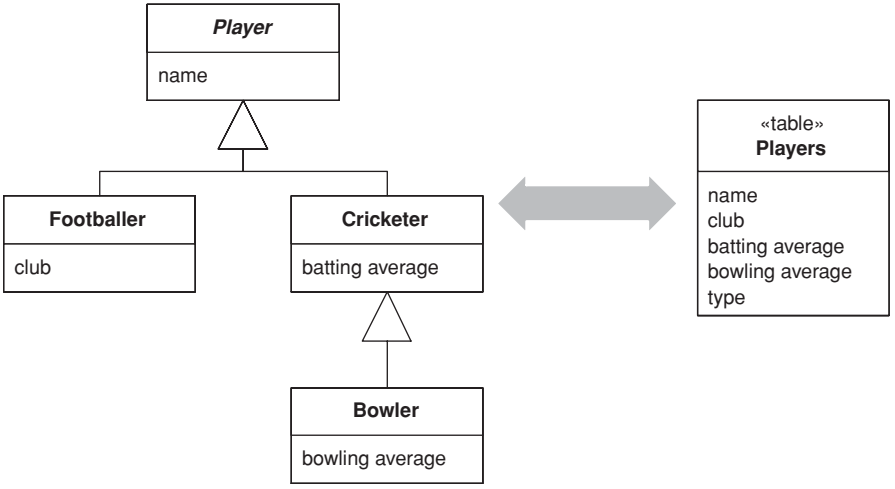

Figure 3.8 Single Table Inheritance (278) *uses one table to store all the classes in a hierarchy.*

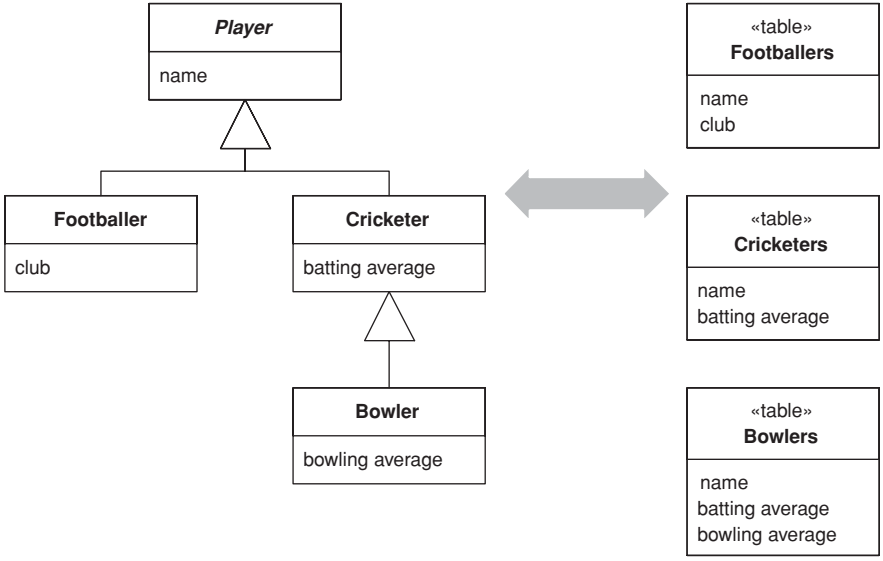

Figure 3.9 Concrete Table Inheritance (293) *uses one table to store each concrete class in a hierarchy.*

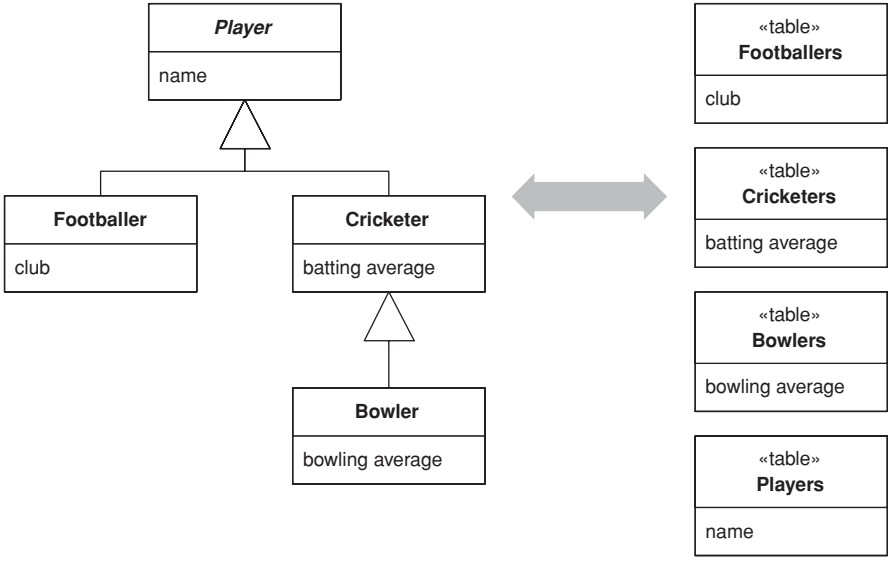

Figure 3.10 Class Table Inheritance (285) *uses one table for each class in a hierarchy.*

mapping code). Altering the hierarchy itself can cause even bigger changes. Also, the lack of a superclass table can make key management awkward and get in the way of referential integrity, although it does reduce lock contention on the superclass table. In some databases *Single Table Inheritance (278)*'s biggest downside is wasted space, since each row has to have columns for all possible subtypes and this leads to empty columns. However, many databases do a very good job of compressing wasted table space. Another problem with *Single Table Inheritance (278)* is its size, making it a bottleneck for accesses. Its great advantage is that it puts all the stuff in one place, which makes modification easier and avoids joins.

The three options aren't mutually exclusive, and in one hierarchy you can mix patterns. For instance, you could have several classes pulled together with *Single Table Inheritance (278)* and use *Class Table Inheritance (285)* for a few unusual cases. Of course, mixing patterns adds complexity.

There's no clearcut winner here. You need to take into account your own circumstances and preferences, much as with all the rest of these patterns. My first choice tends to be *Single Table Inheritance (278)*, as it's easy to do and is resilient to many refactorings. I tend to use the other two as needed to help solve the inevitable issues with irrelevant and wasted columns. Often the best is to talk to the DBAs; they often have good advice as to the sort of access that makes the most sense for the database.

All the examples just described, and in the patterns, use single inheritance. Although multiple inheritance is becoming less fashionable these days and most languages are increasingly avoiding it, the issue still appears in O/R mapping when you use interfaces, as in Java and .NET. The patterns here don't go into this topic specifically, but essentially you cope with multiple inheritance using variations of the trio of inheritance patterns. *Single Table Inheritance (278)* puts all superclasses and interfaces into the one big table, *Class Table Inheritance (285)* makes a separate table for each interface and superclass, and *Concrete Table Inheritance (293)* includes all interfaces and superclasses in each concrete table.

Building the Mapping

When you map to a relational database, there are essentially three situations that you encounter:

- You choose the schema yourself.
- You have to map to an existing schema, which can't be changed.
- You have to map to an existing schema, but changes to it are negotiable.

The simplest case is where you're doing the schema yourself and have little to moderate complexity in your domain logic, resulting in a *Transaction Script (110)* or *Table Module (125)* design. In this case you can design the tables around the data using classic database design techniques. Use a *Row Data Gateway (152)* or *Table Data Gateway (144)* to pull the SQL away from the domain logic.

If you're using a *Domain Model (116)*, you should beware of a design that looks like a database design. In this case build your *Domain Model (116)* without regard to the database so that you can best simplify the domain logic. Treat the database design as a way of persisting the objects' data. *Data Mapper (165)* gives you the most flexibility here, but it's more complex. If a database design isomorphic to the *Domain Model (116)* makes sense, you might consider an *Active Record (160)* instead.

Although building the model first is a reasonable way of thinking about it, this advice only applies within short iterative cycles. Spending six months building a database-free *Domain Model (116)* and then deciding to persist it once you're done is highly risky. The danger is that the resulting design will have crippling performance problems that take too much refactoring to fix. Instead, build up the database with each iteration, of no more than six weeks in length and preferably fewer. That way you'll get rapid and continuous feedback about how your database interactions work in practice. Within any particular task you should think about the *Domain Model (116)* first, but integrate each piece of *Domain Model (116)* in the database as you go.

When the schema's already there, your choices are similar but the process is slightly different. With simple domain logic you build *Row Data Gateway (152)* or *Table Data Gateway (144)* classes that mimic the database, and layer domain logic on top of that. With more complex domain logic you'll need a *Domain Model (116)*, which is highly unlikely to match the database design. Therefore, gradually build up the *Domain Model (116)* and include *Data Mappers (165)* to persist the data to the existing database.

Double Mapping

Occasionally I run into situations where the same kind of data needs to be pulled from more than one source. There may be multiple databases that hold the same data but have small differences in the schema because of some copy and paste reuse. (In this situation the amount of annoyance is inversely proportional to the amount of the difference.) Another possibility is using different mechanisms, storing the data sometimes in a database and sometimes in messages. You may want to pull similar data from both XML messages, CICS transactions, and relational tables.

The simplest option is to have multiple mapping layers, one for each data source. However, if data is very similar this can lead to a lot of duplication. In this situation you might consider a two-step mapping scheme. The first step converts data from the in-memory schema to a logical data store schema. The logical data store schema is designed to maximize the similarities in the data source formats. The second step maps from the logical data store schema to the actual physical data store schema. This second step contains the differences.

The extra step only pays for itself when you have many commonalities, so you should use it when you have similar but annoyingly different physical data stores. Treat the mapping from the logical data store to the physical data store as a *Gateway (466)* and use any of the mapping techniques to map from the application logic to the logical data store.

Using Metadata

In this book most of my examples use handwritten code. With simple and repetitive mapping this can lead to code that's simple and repetitive—and repetitive code is a sign of something wrong with the design. There's much you can do by factoring out common behaviors with inheritance and delegation—good, honest OO practices—but there's also a more sophisticated approach using *Metadata Mapping (306)*.

Metadata Mapping (306) is based on boiling down the mapping into a metadata file that details how columns in the database map to fields in objects. The point of this is that once you have the metadata you can avoid the repetitive code by using either code generation or reflective programming.

Using metadata buys you a lot of expressiveness from a little metadata. One line of metadata can say something like

```
<field name = customer targetClass = "Customer", dbColumn = "custID", targetTable = "customers"
lowerBound = "1" upperBound = "1" setter = "loadCustomer"/>
```

From that you can define the read and write code, automatically generate ad hoc joins, do all of the SQL, enforce the multiplicity of the relationship, and even do fancy things like computing write orders under the presence of referential integrity. This is why commercial O/R mapping tools tend to use metadata.

When you use *Metadata Mapping (306)* you have the necessary foundation to build queries in terms of in-memory objects. A *Query Object (316)* allows you to build your queries in terms of in-memory objects and data in such a way that developers don't need to know either SQL or the details of the relational

schema. The *Query Object (316)* can then use the *Metadata Mapping (306)* to translate expressions based on object fields into the appropriate SQL.

Take this far enough and you can form a *Repository (322)* that largely hides the database from view. Any queries to the database can be made as *Query Objects (316)* against a *Repository (322)*, and developers can't tell whether the objects were retrieved from memory or from the database. *Repository (322)* works well with rich *Domain Model (116)* systems.

Despite the many advantages of metadata, in this book I've focused on handwritten examples because I think they're easier to understand first. Once you get the hang of the patterns and can handwrite them for your application, you'll be able to figure out how to use metadata to make matters easier.

Database Connections

Most database interfaces rely on some kind of database connection object to act as the link between application code and the database. Typically a connection must be opened before you can execute commands against the database. Indeed, usually you need an explicit connection to create and execute a command. The whole time you execute the command this same connection must be open. Queries return a *Record Set (508)*. Some interfaces provide for disconnected *Record Sets (508)*, which can be manipulated after the connection is closed. Other interfaces provide only connected *Record Sets (508)*, implying that the connection must remain open while the *Record Set (508)* is manipulated. If you're running inside a transaction, usually the transaction is bound to a particular connection and the connection must remain open while it is taking place.

In many environments it's expensive to create a connection, which makes it worthwhile to create a connection pool. In this situation developers request a connection from the pool and release it when they're done, instead of creating and closing the connection. Most platforms these days give you pooling, so you'll rarely have to do it yourself. If you do have to do it yourself, first check to see if pooling actually does help performance. Increasingly environments make it quicker to create a new connection so there's no need to pool.

Environments that give you pooling often put it behind an interface that looks like creating a new connection. That way you don't know whether you're getting a brand new connection or one allocated from a pool. That's a good thing, as the choice to pool or not is properly encapsulated. Similarly, closing the connection may not actually close it but just return it to the pool for some-

one else to use. In this discussion I'll use "open" and "close," which you can substitute for "getting" from the pool and "releasing" back to the pool.

Expensive to create or not, connections need management. Since they're expensive resources to manage, they must be closed as soon as you're done using them. Furthermore, if you're using a transaction, usually you need to ensure that every command inside a particular transaction goes with the same connection.

The most common advice is to get a connection explicitly, using a call to a pool or connection manager, and then supply it to each database command you want to make. Once you're done with the connection, close it. This advice leads to a couple of issues: making sure you have the connection everywhere you need it and ensuring that you don't forget to close it at the end.

To ensure that you have a connection where you need it there are two choices. One is to pass the connection around as an explicit parameter. The problem with this is that the connection gets added to all sorts of method calls where its only purpose is to be passed to some other method five layers down the call stack. Of course, this is the situation to bring out *Registry (480)*. Since you don't want multiple threads using the same connection, you'll want a thread-scoped *Registry (480)*.

If you're half as forgetful as I am, explicit closing isn't such a good idea. It's just too easy to forget to do it when you should. You also can't close the connection with every command because you may be running inside a transaction and the closing will usually cause the transaction to roll back.

Like a connection, memory is a resource that needs to be freed up when you're not using it. Modern environments these days provide automatic memory management and garbage collection, so one way to ensure that connections are closed is to use the garbage collector. In this approach either the connection itself or some object that refers to it closes the connection during garbage collection. The good thing about this is that it uses the same management scheme that's used for memory and so it's both convenient and familiar. The problem is that the close of the connection only happens when the garbage collector actually reclaims the memory, and this can be quite a bit later than when the connection lost its last reference. As a result unreferenced connections may sit around a while before they're closed. Whether this is a problem or not depends very much on your specific environment.

On the whole I don't like relying on garbage collection. Other schemes—even explicit closing—are better. Still, garbage collection makes a good backup in case the regular scheme fails. After all, it's better to have the connections close eventually than to have them hanging around forever.

Since connections are so tied to transactions, a good way to manage them is to tie them to a transaction. Open a connection when you begin a transaction, and close it when you commit or roll back. Have the transaction know what connection it's using so you can ignore the connection completely and just deal with the transaction. Since the transaction's completion has a visible effect, it's easier to remember to commit it and to spot if you forget. A *Unit of Work (184)* makes a natural fit to manage both the transaction and the connection.

If you do things outside of your transaction, such as reading immutable data, you use a fresh connection for each command. Pooling can deal with any issues in creating short-lived connections.

If you're using a disconnected *Record Set (508)*, you can open a connection to put the data in the record set and close it while you manipulate the *Record Set (508)* data. Then, when you're done with the data, you can open a new connection, and transaction, to write the data out. If you do this, you'll need to worry about the data being changed while the *Record Set (508)* was being manipulated. This is a topic I'll talk about with concurrency control.

The specifics of connection management are very much a feature of your database interaction software, so the strategy you use is often dictated by your environment.

Some Miscellaneous Points

You'll notice that some of the code examples use select statements in the form `select * from` while others use named columns. Using `select *` can have serious problems in some database drivers, which break if a new column is added or a column is reordered. Although more modern environments don't suffer from this, it's not wise to use `select *` if you're using positional indices to get information from columns, as a column reorder will break code. It's okay to use column name indices with a `select *`, and indeed column name indices are clearer to read; however, column name indices may be slower, although that probably won't make much difference given the time for the SQL call. As usual, measure to be sure.

If you do use column number indices, you need to make sure that the accesses to the result set are very close to the definition of the SQL statement so they don't get out of sync if the columns are reordered. Consequently, if you're using *Table Data Gateway (144)*, you should use column name indices as the result set is used by every piece of code that runs a find operation on the gateway. As a result it's usually worth having simple create/read/update/delete test

cases for each database mapping structure you use. This will help catch cases when your SQL gets out of sync with your code.

It's always worth making the effort to use static SQL that can be precompiled, rather than dynamic SQL that has to be compiled each time. Most platforms give you a mechanism for precompiling SQL. A good rule of thumb is to avoid using string concatenation to put together SQL queries.

Many environments give you the ability to batch multiple SQL queries into a single database call. I haven't done that for these examples, but it's certainly a tactic you should use in production code. How you do it varies with the platform.

For connections in these examples, I just conjure them up with a call to a "DB" object, which is a *Registry (480)*. How you get a connection will depend on your environment so you'll substitute this with whatever you need to do. I haven't involved transactions in any of the patterns other than those on concurrency. Again, you'll need to mix in whatever your environment needs.

Further Reading

Object-relational mapping is a fact of life for most people, so it's no surprise that there's been a lot written on the subject. The surprise is that there isn't a single coherent, complete, and up-to-date book, which is why I've devoted so much of this one to this tricky yet interesting subject.

The nice thing about database mapping is that there's a lot of ideas out there to steal from. The most victimized intellectual banks are [Brown and Whitenack], [Ambler], [Yoder], and [Keller and Coldewey]. I'd certainly urge you to have a good surf through this material to supplement the patterns in this book.

This page intentionally left blank

Index