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 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 langauge. Database administrations (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.

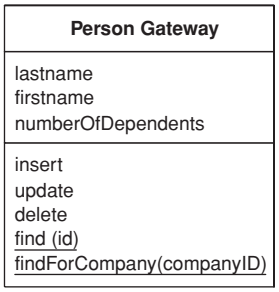

Figure 3.1 *A* Row Data Gateway (152) *has one instance per row returned by a query.*

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)*.

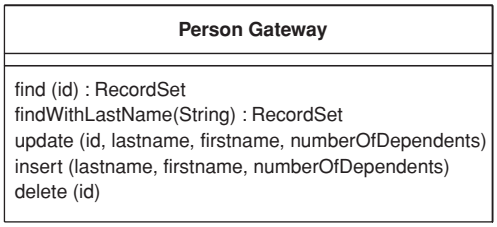

Figure 3.2 *A* Table Data Gateway (144) *has one instance per table.*

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

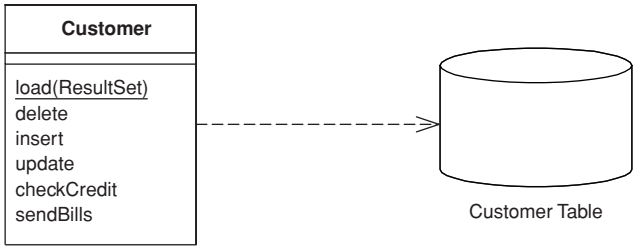

Figure 3.3 *In the* Active Record (160) *a customer domain object knows how to interact with database tables.*

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.

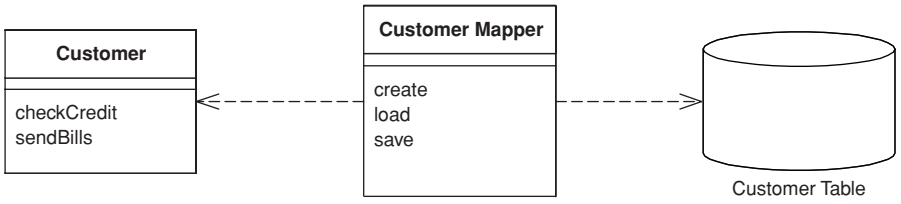

Figure 3.4 *A* Data Mapper (165) *insulates the domain objects and the database from each other.*

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

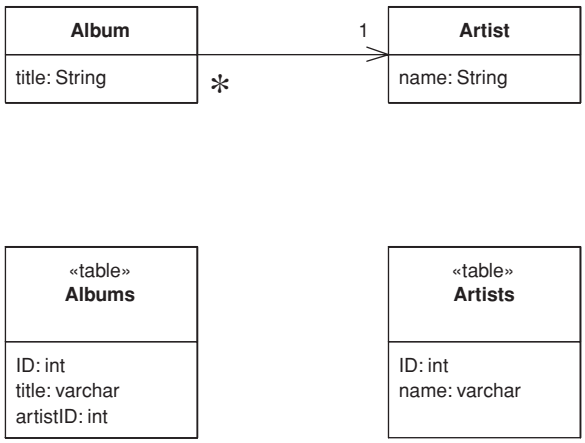

Figure 3.5 *Use a* Foreign Key Mapping (236) *to map a single-valued field.*

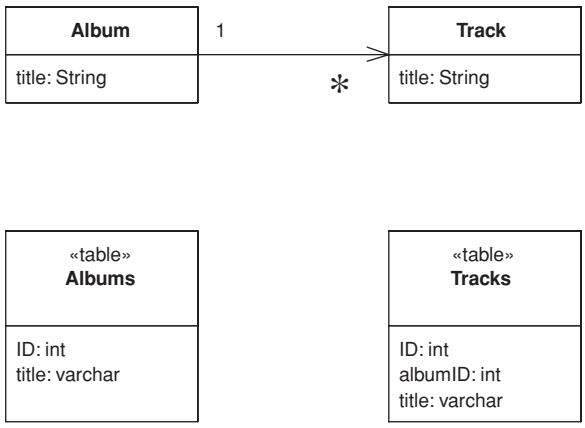

Figure 3.6 *Use a* Foreign Key Mapping (236) *to map a collection field.*

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.

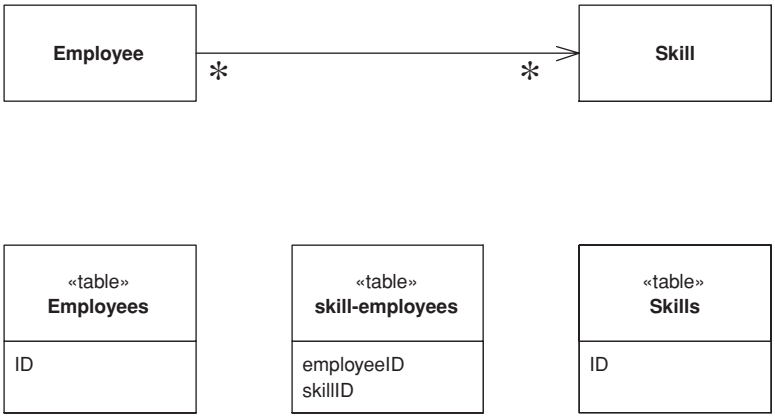

Figure 3.7 *Use an* Association Table Mapping (248) *to map a many-to-many association.*

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 a *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) 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 system 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 a 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

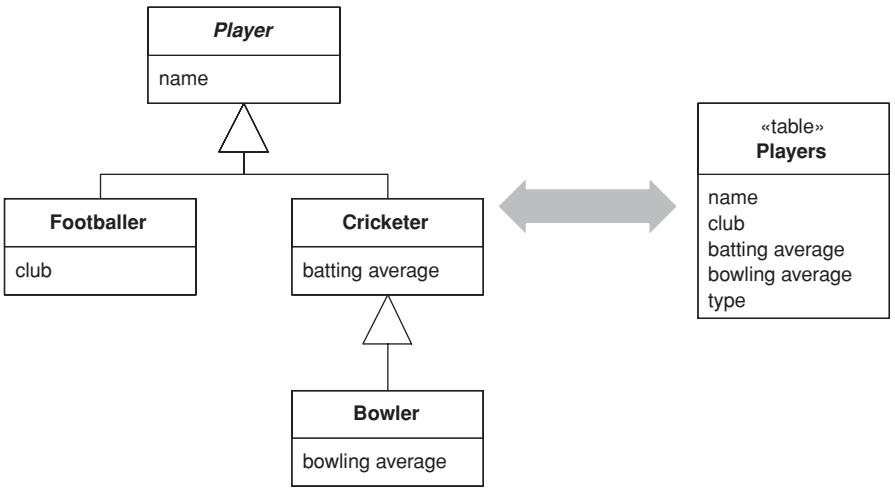

Figure 3.8 Single Table Inheritance (278) *uses one table to store all the classes in a hierarchy.*

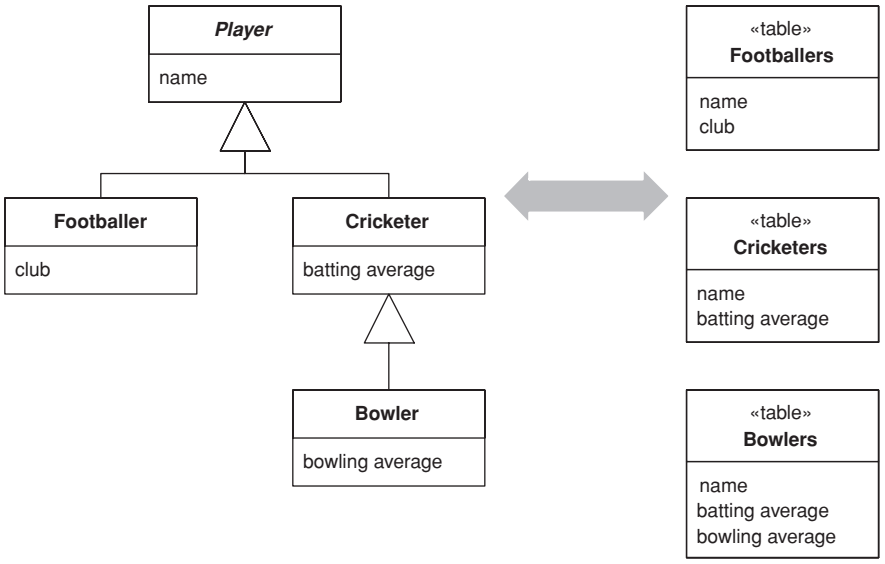

Figure 3.9 Concrete Table Inheritance (293) *uses one table to store each concrete class in a hierarchy.*

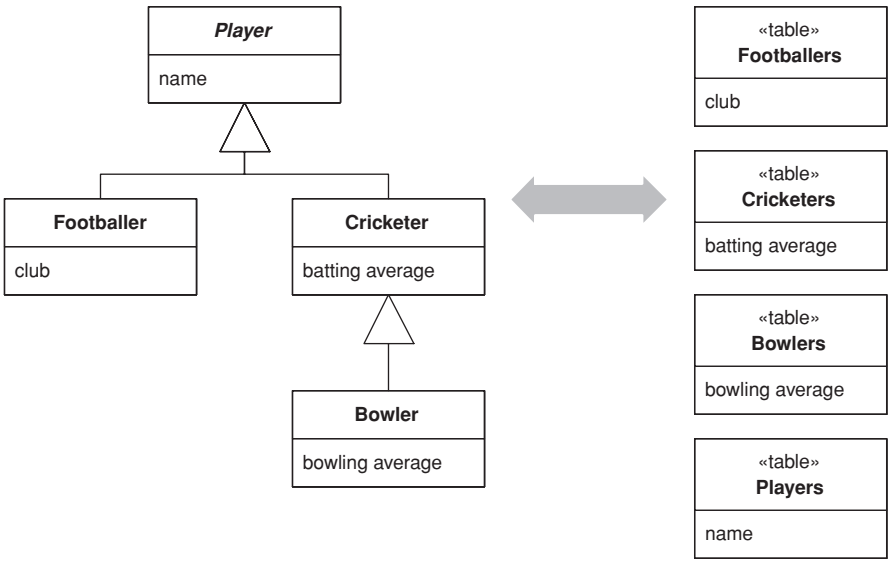

Figure 3.10 Class Table Inheritance (285) *uses one table for each class in a hierarchy.*

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 a 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.