iBATIS IN ACTION

SAMPLE CHAPTER

Clinton Begin
Brandon Goodin
Larry Meadors

MANNING

iBATIS in Action

by Clinton Begin
Brandon Goodin
Larry Meadors

Chapter 1

brief contents

1 The iBATIS philosophy

This chapter covers

- iBATIS history
- Understanding iBATIS
- Database types

Structured Query Language (SQL) has been around for a long time. It's been over 35 years since Edgar F. Codd first suggested the idea that data could be normalized into sets of related tables. Since then, corporate IT has invested billions of dollars into relational database management systems (RDBMSs). Few software technologies can claim to have stood the test of time as well as the relational database and SQL. Indeed, after all this time, there is still a great deal of momentum behind relational technology and it is a cornerstone offering of the largest software companies in the world. All indicators suggest that SQL will be around for another 30 years.

iBATIS is based on the idea that there is value in relational databases and SQL, and that it is a good idea to embrace the industrywide investment in SQL. We have experiences whereby the database and even the SQL itself have outlived the application source code, and even multiple versions of the source code. In some cases we have seen that an application was rewritten in a different language, but the SQL and database remained largely unchanged.

It is for such reasons that iBATIS does not attempt to hide SQL or avoid SQL. It is a persistence layer framework that instead embraces SQL by making it easier to work with and easier to integrate into modern object-oriented software. These days, there are rumors that databases and SQL threaten our object models, but that does not have to be the case. iBATIS can help to ensure that it is not.

In this chapter, we will look at the history and rationale for iBATIS, and discuss the forces that influenced its creation.

1.1 A hybrid solution: combining the best of the best

In the modern world, hybrid solutions can be found everywhere. Taking two seemingly opposing ideas and merging them in the middle has proven to be an effective means to filling a niche, which in some cases has resulted in the creation of entire industries. This is certainly true of the automotive industry, as most of the innovation in vehicle designs has come from mixing various ideas. Mix a car with a cargo van and you have the ultimate family minivan. Marry a truck with an all-terrain vehicle, and you have an urban status symbol known as a sport utility vehicle. Cross a hotrod and a station wagon and you have a family car that Dad isn't embarrassed to drive. Set a gasoline engine side by side with an electric motor, and you have the answer for a great deal of the North American pollution problem.

Hybrid solutions have proven effective in the IT industry too. iBATIS is one such hybrid solution for the persistence layer of your application. Over time, various methods have been developed to enable applications to execute SQL against a

database. iBATIS is a unique solution that borrows concepts from several other approaches. Let's start by taking a quick look at these approaches.

1.1.1 Exploring the roots of iBATIS

iBATIS takes the best attributes and ideas from the most popular means of accessing a relational database, and finds synergy among them. Figure 1.1 shows how the iBATIS framework takes what was learned through years of development using different approaches to database integration, and combines the best of those lessons to create a hybrid solution.

The following sections discuss these various approaches to interacting with the database and describe the parts of each that iBATIS leverages.

Structured Query Language

At the heart of iBATIS is SQL. By definition, all relational databases support SQL as the primary means of interacting with the database. SQL is a simple, nonprocedural language for working with the database, and is really two languages in one.

The first is Data Definition Language (DDL), which includes statements like `CREATE`, `DROP`, and `ALTER`. These statements are used to define the structure and design of the database, including the tables, columns, indexes, constraints, procedures, and foreign key relationships. DDL is not something that iBATIS supports directly. Although many people have successfully executed DDL using iBATIS, DDL is usually owned and controlled by a database administration group and is often beyond the reach of developers.

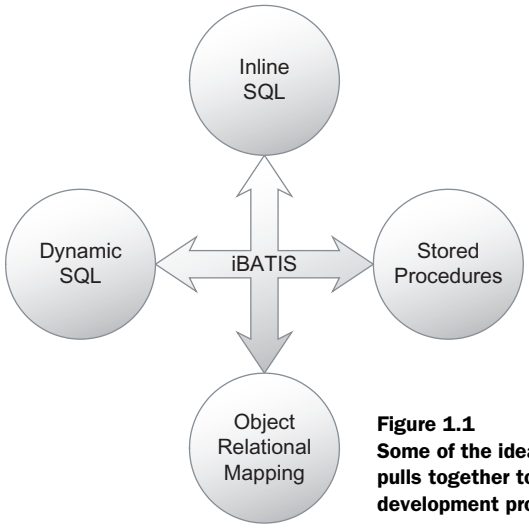

Figure 1.1
Some of the ideas that iBATIS pulls together to simplify the development process

The second part of SQL is the Data Manipulation Language (DML). It includes statements such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. DML is used to manipulate the data directly. Originally SQL was designed to be a language simple enough for end users to use. It was designed so that there should be no need for a rich user interface or even an application at all. Of course, this was back in the day of green-screen terminals, a time when we had more hope for our end users!

These days, databases are much too complex to allow SQL to be run directly against the database by end users. Can you imagine handing a bunch of SQL statements to your accounting department as if to say, "Here you go, you'll find the information you're looking for in the BSHEET table." Indeed.

SQL alone is no longer an effective interface for end users, but it is an extremely powerful tool for developers. SQL is the only complete means of accessing the database; everything else is a subset of the complete set of capabilities of SQL. For this reason, iBATIS fully embraces SQL as the primary means of accessing the relational database. At the same time, iBATIS provides many of the benefits of the other approaches discussed in this chapter, including stored procedures and object/relational mapping tools.

Old-school stored procedures

Stored procedures may be the oldest means of application programming with a relational database. Many legacy applications used what is now known as a *two-tier design*. A two-tier design involved a rich client interface that directly called stored procedures in the database. The stored procedures would contain the SQL that was to be run against the database. In addition to the SQL, the stored procedures could (and often would) contain business logic. Unlike SQL, these stored procedure languages were procedural and had flow control such as conditionals and iteration. Indeed, one could write an entire application using nothing but stored procedures. Many software vendors developed rich client tools, such as Oracle Forms, PowerBuilder, and Visual Basic, for developing two-tier database applications.

The problems with two-tier applications were primarily performance and scalability. Although databases are extremely powerful machines, they aren't necessarily the best choice for dealing with hundreds, thousands, or possibly millions of users. With modern web applications, these scalability requirements are not uncommon. Limitations, including concurrent licenses, hardware resources, and even network sockets, would prevent such architecture from succeeding on a massive scale. Furthermore, deployment of two-tier applications was a nightmare. In addition to the usual rich client deployment issues, complex runtime database engines often had to be deployed to the client machine as well.

Modern stored procedures

In some circles stored procedures are still considered best practice for three-tier and N-tier applications, such as web applications. Stored procedures are now treated more like remote procedure calls from the middle tier, and many of the performance constraints are solved by pooling connections and managing database resources. Stored procedures are still a valid design choice for implementing the entire data access layer in a modern object-oriented application. Stored procedures have the benefit of performance on their side, as they can often manipulate data in the database faster than any other solution. However, there are other concerns beyond simply performance.

Putting business logic in stored procedures is widely accepted as being a bad practice. The primary reason is that stored procedures are more difficult to develop in line with modern application architectures. They are harder to write, test, and deploy. To make things worse, databases are often owned by other teams and are protected by tight change controls. They may not be able to change as fast as they need to to keep up with modern software development methodologies. Furthermore, stored procedures are more limited in their capability to implement the business logic completely. If the business logic involves other systems, resources, or user interfaces, the stored procedure will not likely be able to handle all of the logic. Modern applications are very complex and require a more generic language than a stored procedure that is optimized to manipulate data. To deal with this, some vendors are embedding more powerful languages like Java in their database engines to allow for more robust stored procedures. This really doesn't improve the situation at all. It only serves to further confuse the boundaries of the application and the database and puts a new burden on the database administrators: now they have to worry about Java and C# in their database. It's simply the wrong tool for the job.

A common theme in software development is *overcorrection*. When one problem is found, the first solution attempted is often the exact opposite approach. Instead of solving the problem, the result is an equal number of completely different problems. This brings us to the discussion of inline SQL.

Inline SQL

An approach to dealing with the limitations of stored procedures was to embed SQL into more generic languages. Instead of moving the logic into the database, the SQL was moved from the database to the application code. This allowed SQL statements to interact with the language directly. In a sense, SQL became a feature

of the language. This has been done with a number of languages, including COBOL, C, and even Java. The following is an example of SQLJ in Java:

```
String name;
Date hiredate;

#sql {
      SELECT emp_name, hire_date
      INTO :name, :hiredate
      FROM employee
      WHERE emp_num = 28959
};
```

Inline SQL is quite elegant in that it integrates tightly with the language. Native language variables can be passed directly to the SQL as parameters, and results can be selected directly into similar variables. In a sense, the SQL becomes a feature of the language.

Unfortunately, inline SQL is not widely adopted and has some significant issues keeping it from gaining any ground. First, SQL is not a standard. There are many extensions to SQL and each only works with one particular database. This fragmentation of the SQL language makes it difficult to implement an inline SQL parser that is both complete and portable across database platforms. The second problem with inline SQL is that it is often not implemented as a true language feature. Instead, a precompiler is used to first translate the inline SQL into proper code for the given language. This creates problems for tools like integrated development environments (IDEs) that might have to interpret the code to enable advanced features like syntax highlighting and code completion. Code that contains inline SQL may not even be able to compile without the precompiler, a dependency that creates concerns around the future maintainability of the code.

One solution to the pains of inline SQL is to remove the SQL from the language level, and instead represent it as a data structure (i.e., a string) in the application. This approach is commonly known as Dynamic SQL.

Dynamic SQL

Dynamic SQL deals with some of the problems of inline SQL by avoiding the precompiler. Instead, SQL is represented as a string type that can be manipulated just like any other character data in a modern language. Because the SQL is represented as a string type, it cannot interact with the language directly like inline SQL can. Therefore, Dynamic SQL implementations require a robust API for setting SQL parameters and retrieving the resulting data.

Dynamic SQL has the advantage of flexibility. The SQL can be manipulated at runtime based on different parameters or dynamic application functions. For example, a query-by-example web form might allow the user to select the fields to search upon and what data to search for. This would require a dynamic change to the `WHERE` clause of the SQL statement, which can be easily done with Dynamic SQL.

Dynamic SQL is currently the most popular means of accessing relational databases from modern languages. Most such languages include a standard API for database access. Java developers and .NET developers will be familiar with the standard APIs in those languages: JDBC and ADO.NET, respectively. These standard SQL APIs are generally very robust and offer a great deal of flexibility to the developer. The following is a simple example of Dynamic SQL in Java:

```
String name;
Date hiredate;
String sql = "SELECT emp_name, hire_date"
       + " FROM employee WHERE emp_num = ? ";
Connection conn = dataSource.getConnection();
PreparedStatement ps = conn.prepareStatement (sql);
ps.setInt (1, 28959);
ResultSet rs = ps.executeQuery();
while (rs.next) {
name = rs.getString("emp_name");
hiredate = rs.getDate("hire_date");
}
rs.close();
conn.close();
```

Should be in try-catch block

Without a doubt, Dynamic SQL is not as elegant as inline SQL, or even stored procedures (and we even left out the exception handling). The APIs are often complex and very verbose, just like the previous example. Using these frameworks generally results in a lot of code, which is often very repetitive. In addition, the SQL itself is often too long to be on a single line. This means that the string has to be broken up into multiple strings that are concatenated. Concatenation results in unreadable SQL code that is difficult to maintain and work with.

So if the SQL isn't best placed in the database as a stored procedure, or in the language as inline SQL, or in the application as a data structure, what do we do with it? We avoid it. In modern object-oriented applications, one of the most compelling solutions to interacting with a relational database is through the use of an object/relational mapping tool.

Object/relational mapping

Object/relational mapping (O/RM) was designed to simplify persistence of objects by eliminating SQL from the developer's responsibility altogether. Instead, the SQL is generated. Some tools generate the SQL statically at build or compile time, while others generate it dynamically at runtime. The SQL is generated based on mappings made between application classes and relational database tables. In addition to eliminating the SQL, the API for working with an O/RM tool is usually a lot simpler than the typical SQL APIs. Object/relational mapping is not a new concept and is almost as old as object-oriented programming languages. There have been a lot of advances in recent years that make object/relational mapping a compelling approach to persistence.

Modern object/relational mapping tools do more than simply generate SQL. They offer a complete persistence architecture that benefits the entire application. Any good object/relational mapping tool will provide transaction management. This includes simple APIs for dealing with both local and distributed transactions. O/RM tools also usually offer multiple caching strategies for dealing with different kinds of data to avoid needless access of the database. Another way that an O/RM tool can reduce database hits is by lazy loading of data. Lazy loading delays the retrieval of data until absolutely necessary, right at the point where the data is used.

Despite these features, object/relational mapping tools are not a silver-bullet solution and do not work in all situations. O/RM tools are based on assumptions and rules. The most common assumption is that the database will be properly normalized. As we will discuss in section 1.4, the largest and most valuable databases are rarely normalized perfectly. This can complicate the mappings and may require workarounds or create inefficiencies in the design. No object relational solution will ever be able to provide support for every feature, capability, and design flaw of every single database available. As stated earlier, SQL is not a reliable standard. For this reason, every O/RM tool will always be a subset of the full capabilities of any particular database.

Enter the hybrid.

1.1.2 Understanding the iBATIS advantage

iBATIS is a hybrid solution. It takes the best ideas from each of these solutions and creates synergy between them. Table 1.1 summarizes some of the ideas from each of the approaches discussed earlier that are incorporated into iBATIS.

Table 1.1 Advantages provided by iBATIS, which are the same as those provided by other solutions

Approach	Similar benefit	Solved problems
Stored procedures	iBATIS encapsulates and externalizes SQL such that it is outside of your application code. It describes an API similar to that of a stored procedure, but the iBATIS API is object oriented. iBATIS also fully supports calling stored procedures directly.	Business logic is kept out of the database, and the application is easier to deploy and test, and is more portable.
Inline SQL	iBATIS allows SQL to be written the way it was intended to be written. There's no string concatenation, "setting" of parameters, or "getting" of results.	iBATIS doesn't impose on your application code. No precompiler is needed, and you have full access to all of the features of SQL—not a subset.
Dynamic SQL	iBATIS provides features for dynamically building queries based on parameters. No "query-builder" APIs are required.	iBATIS doesn't force SQL to be written in blocks of concatenated strings interlaced with application code.
Object/relational mapping	iBATIS supports many of the same features as an O/RM tool, such as lazy loading, join fetching, caching, runtime code generation, and inheritance.	iBATIS will work with any combination of data model and object model. There are nearly no restrictions or rules to how either is designed.

Now that you understand the roots of iBATIS, the following sections discuss two of the most important qualities of the iBATIS persistence layer: externalization and encapsulation of the SQL. Together, these concepts provide much of the value and enable many of the advanced features that the framework achieves.

Externalized SQL

One of the wisdoms learned in the last decade of software development has been to design one's systems to correspond to different users of the subsystem. You want to separate out the things that are dealt with by different programming roles such as user interface design, application programming, and database administration. Even if only a single person is playing all of these roles, it helps to have a nicely layered design that allows you to focus on a particular part of the system. If you embed your SQL within Java source code, it will not generally be useful to a database administrator or perhaps a .NET developer who might be working with the same database. Externalization separates the SQL from the application source code, thus keeping both cleaner. Doing so ensures that the SQL is relatively independent of any particular language or platform. Most modern development

languages represent SQL as a string type, which introduces concatenation for long SQL statements. Consider the following simple SQL statement:

```
SELECT
  PRODUCTID,
  NAME,
  DESCRIPTION,
  CATEGORY
FROM PRODUCT
WHERE CATEGORY = ?
```

When embedded in a String data type in a modern programming language such as Java, this gentle SQL statement becomes a mess of multiple language characteristics and unmanageable code:

```
String s = "SELECT"
  + " PRODUCTID,"
  + " NAME,"
  + " DESCRIPTION,"
  + " CATEGORY"
  + " FROM PRODUCT"
  + " WHERE CATEGORY = ?";
```

Simply forgetting to lead the `FROM` clause with a space will cause a SQL error to occur. You can easily imagine the trouble a complex SQL statement could cause.

Therein lies one of the key advantages of iBATIS: the ability to write SQL the way it was meant to be written. The following gives you a sense of what an iBATIS mapped SQL statement looks like:

```
SELECT
  PRODUCTID,
  NAME,
  DESCRIPTION,
  CATEGORY
FROM PRODUCT
WHERE CATEGORY = #categoryId#
```

Notice how the SQL does not change in terms of structure or simplicity. The biggest difference in the SQL is the format of the parameter `#categoryId#`, which is normally a language-specific detail. iBATIS makes it portable and more readable.

Now that we have our SQL out of the source code and into a place where we can work with it more naturally, we need to link it back to the software so that it can be executed in a way that is useful.

Encapsulated SQL

One of the oldest concepts in computer programming is the idea of modularization. In a procedural application, code may be separated into files, functions, and procedures. In an object-oriented application, code is often organized into classes and methods. *Encapsulation* is a form of modularization that not only organizes the code into cohesive modules, but also hides the implementation details while exposing only the interface to the calling code.

This concept can be extended into our persistence layer. We can encapsulate SQL by defining its inputs and outputs (i.e., its interface), but otherwise hide the SQL code from the rest of the application. If you're an object-oriented software developer, you can think of this encapsulation in the same way that you think of separating an interface from its implementation. If you're a SQL developer, you can think of this encapsulation much like you'd think of hiding a SQL statement inside a stored procedure.

iBATIS uses Extensible Markup Language (XML) to encapsulate SQL. XML was chosen because of its general portability across platforms, its industrywide adoption, and the fact that it's more likely to live as long as SQL than any other language and any file format. Using XML, iBATIS maps the inputs and outputs of the statement. Most SQL statements have one or more parameters and produce some sort of tabulated results. That is, results are organized into a series of columns and rows. iBATIS allows you to easily map both parameters and results to properties of objects. Consider the next example:

```
<select id="categoryById"
  parameterClass="string" resultClass="category">
      SELECT CATEGORYID, NAME, DESCRIPTION
      FROM CATEGORY
      WHERE CATEGORYID = #categoryId#
</select>
```

Notice the XML element surrounding the SQL. This is the encapsulation of the SQL. The simple `<select>` element defines the name of the statement, the parameter input type, and the resulting output type. To an object-oriented software developer, this is much like a method signature.

Both simplicity and consistency are achieved through externalizing and encapsulating the SQL. More details of the exact usage of the API and mapping syntax will follow in chapter 2. Before we get to that, it's important to understand where iBATIS fits in your application architecture.

1.2 Where iBATIS fits

Nearly any well-written piece of software uses a layered design. A layered design separates the technical responsibilities of an application into cohesive parts that isolate the implementation details of a particular technology or interface. A layered design can be achieved in any robust (3GL/4GL) programming language. Figure 1.2 shows a high-level view of a typical layering strategy that is useful for many business applications.

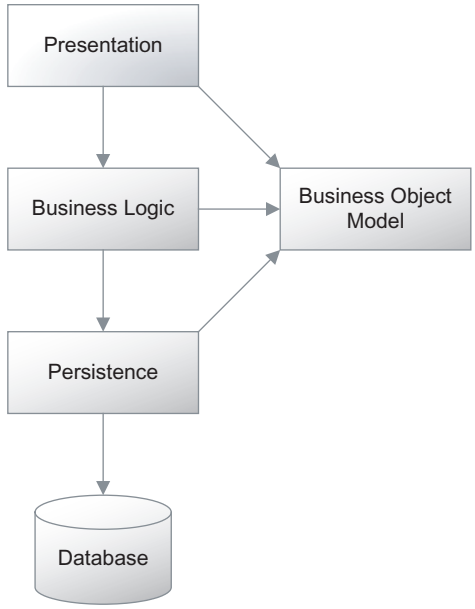

Figure 1.2 A typical layering strategy following the Law of Demeter

You can read the arrows in figure 1.2 as "depends on" or "uses." This layering approach is inspired by the Law of Demeter, which in one form states, "Each layer should have only limited knowledge about other layers: only layers closely related to the current layer."

The idea is that each layer will only talk to the layer directly below it. This ensures that the dependency flows only in one direction and avoids the typical "spaghetti code" that is common of applications designed without layers.

iBATIS is a persistence layer framework. The persistence layer sits between the business logic layer of the application and the database. This separation is important to ensuring that your persistence strategy is not mixed with your business logic code, or vice versa. The benefit of this separation is that your code can be more easily maintained, as it will allow your object model to evolve independently of your database design.

Although iBATIS is heavily focused on the persistence layer, it is important to understand all of the layers of application architecture. Although you separate your concerns so that there are minimal (or no) dependencies on any particular implementation, it would be naive to think that you can be blind to the interaction among these layers. Regardless of how well you design your application, there will be indirect behavioral associations between the layers that you must be aware of. The following sections describe the layers and describe how iBATIS relates to them.

1.2.1 The business object model

The business object serves as the foundation for the rest of the application. It is the object-oriented representation of the problem domain, and therefore the classes that make up the business object model are sometimes called *domain classes*. All other layers use the business object model to represent data and perform certain business logic functions.

Application designers usually start with the design of the business object model before anything else. Even if at a very high level, the classes are identified by deriving them from the nouns in the system. For example, in a bookstore application, the business object model might include a class called `Genre` with instances like `Science Fiction`, `Mystery`, and `Children's`. It might also have a class called `Book` with instances such as `The Long Walk`, `The Firm`, and `Curious George`. As the application grows more advanced, classes represent more abstract concepts, like `InvoiceLineItem`.

Business object model classes may contain some logic as well, but they should never contain any code that accesses any other layer, especially the presentation and persistence layers. Furthermore, the business object model should never depend on any other layer. Other layers use the business object model—it's never the other way around.

A persistence layer like iBATIS will generally use the business object model for representing data that is stored in the database. The domain classes of the business object model will become the parameters and return values of the persistence methods. It is for this reason that these classes are sometimes referred to as *data transfer objects (DTOs)*. Although data transfer is not their only purpose, it is a fair name from the perspective of a persistence framework.

1.2.2 The presentation layer

The presentation layer is responsible for displaying application controls and data to the end user. It is responsible for the layout and formatting of all information. The most popular presentation approach in business applications today are web front ends that use HTML and JavaScript to provide a look and feel to the user via a web browser.

Web applications have the advantage of cross-platform compatibility, ease of deployment, and scalability. Amazon.com is a perfect example of a web application that allows you to buy books online. This is a good use of a web application, as it would be impractical to have everyone download an application just to buy books.

Web applications generally break down when advanced user controls or complex data manipulation are a requirement. In such cases, rich clients that use native operating system widgets like tabs, tables, tree views, and embedded objects are preferred. Rich clients allow for a much more powerful user interface, but are somewhat more difficult to deploy and require more care to achieve the level of performance and security a web application can offer. Examples of rich client technologies include Swing in Java and WinForms in .NET.

Recently the two concepts have been mixed into hybrid clients to attempt to achieve the benefits of both web applications and rich clients. Very small rich clients with advanced controls can be downloaded to the users' desktop, perhaps transparently via the web browser. This hybrid-rich client does not contain any business logic and it may not even have the layout of its user interface built in. Instead, the application look and feel and the available business functionality are configured via a web service, or a web application that uses XML as an interface between the rich client and the server. The only disadvantage is that more software is required to both develop and deploy such applications. For example, both Adobe Flex and Laszlo from Laszlo Systems are based on Macromedia's Flash browser plug-in.

Then of course there is the epitome of all hybrid presentation layers, Ajax. Ajax, a term coined by Jesse James Garrett, used to be an acronym for Asynchronous JavaScript and XML, until everyone realized that it need not be asynchronous, or XML. So now Ajax has simply come to mean "a really rich web-based user interface driven by a lot of really funky JavaScript." Ajax is a new approach to using old technology to build very rich and interactive user interfaces. Google demonstrates some of the best examples of Ajax, putting it to good use with its Gmail, Google Maps, and Google Calendar applications.

iBATIS can be used for both web applications, rich client applications and hybrids. Although the presentation layer does not generally talk directly to the persistence framework, certain decisions about the user interface will impact the requirements for your persistence layer. For example, consider a web application that deals with a large list of 5,000 items. We wouldn't want to show all 5,000 at the same time, nor would it be ideal to load 5,000 items from the database all at once if we weren't going to use them right away. A better approach would be to load and display 10 items at a time. Therefore, our persistence layer should allow for some flexibility in the amount of data returned and possibly even offer us the ability to select and retrieve the exact 10 items that we want. This would improve performance by avoiding needless object creation and data retrieval, and by

reducing network traffic and memory requirements for our application. iBATIS can help achieve these goals using features that allow querying for specific ranges of data.

1.2.3 The business logic layer

The business logic layer of the application describes the coarse-grained services that the application provides. For this reason they are sometimes called *service* classes. At a high level, anyone should be able to look at the classes and methods in the business logic layer and understand what the system does. For example, in a banking application, the business logic layer might have a class called `TellerService`, with methods like `openAccount()`, `deposit()`, `withdrawal()`, and `getBalance()`. These are very large functions that involve complex interactions with databases and possibly other systems. They are much too heavy to place into a domain class, as the code would quickly become incohesive, coupled, and generally unmanageable. The solution is to separate the coarse-grained business functions from their related business object model. This separation of object model classes from logic classes is sometimes called *noun-verb separation*.

Object-oriented purists might claim that this design is *less object oriented* than having such methods directly on the related domain class. Regardless of what is *more or less object oriented*, it is a better design choice to separate these concerns. The primary reason is that business functions are often very complex. They usually involve more than one class and deal with a number of infrastructural components, including databases, message queues, and other systems. Furthermore, there are often a number of domain classes involved in a business function, which would make it hard to decide which class the method should belong to. It is for these reasons that coarse-grained business functions are best implemented as separate methods on a class that is part of the business logic layer.

Don't be afraid to put finer-grained business logic directly on related domain classes. The coarse-grained service methods in the business logic layer are free to call the finer-grained pure logic methods built into domain classes.

In our layered architecture, the business logic layer is the consumer of the persistence layer services. It makes calls to the persistence layer to fetch and change data. The business logic layer also makes an excellent place to demarcate transactions, because it defines the coarse-grained business functions that can be consumed by a number of different user interfaces or possibly other interfaces, such as a web service. There are other schools of thought regarding transaction demarcation, but we'll discuss the topic more in chapter 8.

1.2.4 The persistence layer

The persistence layer is where iBATIS fits and is therefore the focus of this book. In an object-oriented system, the primary concern of the persistence layer is the storage and retrieval of objects, or more specifically the data stored in those objects. In enterprise applications persistence layers usually interact with relational database systems for storing data, although in some cases other durable data structures and mediums might be used. Some systems may use simple comma-delimited flat files or XML files. Because of the disparate nature of persistence strategies in enterprise applications, a secondary concern of the persistence layer is abstraction. The persistence layer should hide all details of *how* the data is being stored and how it is retrieved. Such details should never be exposed to the other layers of the application.

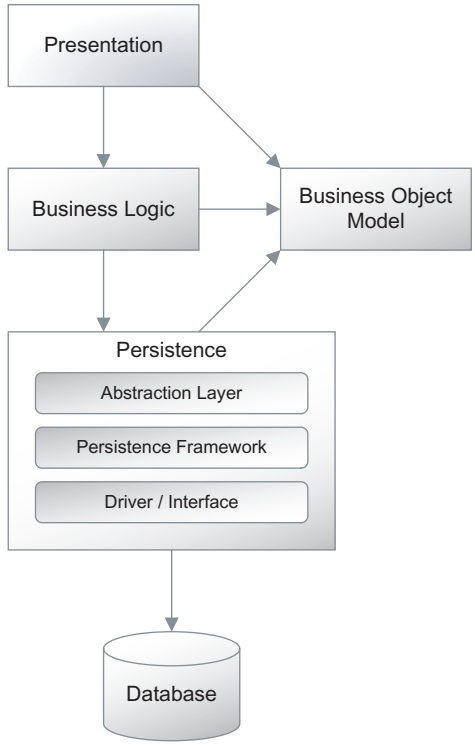

Figure 1.3 Persistence layer zoomed to show internal layered design

To better understand these concerns and how they're managed, it helps to separate the persistence layer into three basic parts: the abstraction layer, the persistence framework, and the driver or interface, as shown in the lower part of figure 1.3.

Let's take a closer look at each of these three parts.

The abstraction layer

The role of the abstraction layer is to provide a consistent and meaningful interface to the persistence layer. It is a set of classes and methods that act as a façade to the persistence implementation details. Methods in the abstraction layer should never require any implementation-specific parameters, nor should it return any values or throw any exceptions that are exclusive to the persistence implementation. With a proper abstraction layer in place, the entire persistence approach—including both the persistence API and the storage infrastructure—should be able to change without modifications to the abstraction layer or any of

the layers that depend on it. There are patterns that can help with the implementation of a proper abstraction layer, the most popular of which is the *Data Access Objects (DAO)* pattern. Some frameworks, including iBATIS, implement this pattern for you. We discuss the iBATIS DAO framework in chapter 11.

The persistence framework

The persistence framework is responsible for interfacing with the driver (or interface). The persistence framework will provide methods for storing, retrieving, updating, searching, and managing data. Unlike the abstraction layer, a persistence framework is generally specific to one class of storage infrastructure. For example, you might find a persistence API that deals exclusively with XML files for storing data. However, with most modern enterprise applications, a relational database is the storage infrastructure of choice. Most popular languages come with standard APIs for accessing relational databases. JDBC is the standard framework for Java applications to access databases, while ADO.NET is the standard database persistence framework for .NET applications. The standard APIs are general purpose and as a result are very complete in their implementation, but also very verbose and repetitive in their use. For these reasons many frameworks have been built on top of the standard ones to extend the functionality to be more specific, and therefore more powerful. iBATIS is a persistence framework that deals exclusively with relational databases of all kinds and supports both Java and .NET using a consistent approach.

The driver or interface

The storage infrastructure can be as simple as a comma-delimited flat file or as complex as a multimillion-dollar enterprise database server. In either case, a software driver is used to communicate with the storage infrastructure at a low level. Some drivers, such as native file system drivers, are very generic in functionality but specific to a platform. You will likely never see a file input/output (I/O) driver, but you can be sure that it is there. Database drivers, on the other hand, tend to be complex and differ in implementation, size, and behavior. It is the job of the persistence framework to communicate with the driver so that these differences are minimized and simplified. Since iBATIS only supports relational databases, that is what we'll focus on in this book.

1.2.5 The relational database

iBATIS exists entirely to make accessing relational databases easier. Databases are complex beasts that can involve a lot of work to use them properly. The database

is responsible for managing data and changes to that data. The reason we use a database instead of simply a flat file is that a database can offer a lot of benefits, primarily in the areas of integrity, performance, and security.

Integrity

Integrity is probably the most important benefit, as without it not much else matters. If our data isn't consistent, reliable, and correct, then it is less valuable to us—or possibly even useless. Databases achieve integrity by using strong data types, enforcing constraints, and working within transactions.

Databases are strongly typed, which means that when a database table is created, its columns are configured to store a specific type of data. The database management system ensures that the data stored in the tables are valid for the column types. For example, a table might define a column as `VARCHAR(25) NOT NULL`. This type ensures that the value is character data that is not of a length greater than 25. The `NOT NULL` part of the definition means that the data is required and so a value must be provided for this column.

In addition to strong typing, other constraints can be applied to tables. Such constraints are usually broader in scope in that they deal with more than just a single column. A constraint usually involves validation of multiple rows or possibly even multiple tables. One type of constraint is a `UNIQUE` constraint, which ensured that for a given column in a table a particular value can be used only once. Another kind of constraint is a `FOREIGN KEY` constraint, which ensures that the value in one column of a table is the same value as a similar column in another table. Foreign key constraints are used to describe relationships among tables, and so they are imperative to relational database design and data integrity.

One of the most important ways a database maintains integrity is through the use of transactions. Most business functions will require many different types of data, possibly from many different databases. Generally this data is related in some way and therefore must be updated consistently. Using transactions, a database management system can ensure that all related data is updated in a consistent fashion. Furthermore, transactions allow multiple users of the system to update data concurrently without colliding. There is a lot more to know about transactions, so we'll discuss them in more detail in chapter 8.

Performance

Relational databases help us achieve a greater level of performance that is not easily made possible using flat files. That said, database performance is not free and it can take a great deal of time and expertise to get it right. Database performance can be broken into three key factors: design, software tuning, and hardware.

The number one performance consideration for a database is design. A bad relational database design can lead to inefficiencies so great that no amount of software tuning or extra hardware can correct it. Bad designs can lead to deadlocking, exponential relational calculations, or simply table scans of millions of rows. Proper design is such a great concern that we'll talk more about it in section 1.3.

Software tuning is the second-most important performance consideration for large databases. Tuning a relational database management system requires a person educated and experienced in the particular RDBMS software being used. Although some characteristics of RDBMS software are transferable across different products, generally each product has intricacies and sneaky differences that require a specialist for that particular software. Performance tuning can yield some great benefits. Proper tuning of a database index alone can cause a complex query to execute in seconds instead of minutes. There are a lot of parts to an RDBMS, such as caches, file managers, various index algorithms, and even operating system considerations. The same RDBMS software will behave differently if the operating system changes, and therefore must be tuned differently. Needless to say, a lot of effort is involved with tuning database software. Exactly how we do that is beyond the scope of this book, but it is important to know that this is one of the most important factors for improving database performance. Work with your DBA!

Large relational database systems are usually very demanding on computer hardware. For this reason, it is not uncommon that the most powerful servers in a company are the database servers. In many companies the database is the center of their universe, so it makes sense that big investments are made in hardware for databases. Fast disk arrays, I/O controllers, hardware caches, and network interfaces are all critical to the performance of large database management systems. Given that, you should avoid using hardware as an excuse for bad database design or as a replacement for RDBMS tuning. Hardware should not be used to solve performance problems—it should be used to meet performance requirements. Further discussion of hardware is also beyond the scope of this book, but it is important to consider it when you're working with a large database system. Again, work with your DBA!

Security

Relational database systems also provide the benefit of added security. Much of the data that we work with in everyday business is confidential. In recent years, privacy has become more of a concern, as has security in general. For this reason, even something as simple as a person's full name can be considered confidential because it is potentially "uniquely identifiable information." Other information—

for example, such as social security numbers and credit card numbers—must be protected with even higher levels of security such as strong encryption. Most commercial-quality relational databases include advanced security features that allow for fine-grained security as well as data encryption. Each database will have unique security requirements. It's important to understand them, as the application code must not weaken the security policy of the database.

Different databases will have different levels of integrity, performance, and security. Generally the size of the database, the value of the data, and the number of dependents will determine these levels. In the next section we'll explore different database types.

1.3 Working with different database types

Not every database is so complex that it requires an expensive database management system and enterprise class hardware. Some databases are small enough to run on an old desktop machine hidden in a closet. All databases are different. They have different requirements and different challenges. iBATIS will help you work with almost any relational database, but it is always important to understand the type of database you're working with.

Databases are classified more by their relationships with other systems than by their design or size. However, the design and size of a database can often be driven by its relationships. Another factor that will affect the design and size of a database is the *age* of the database. As time passes, databases tend to change in different ways, and often the way that these changes are applied are less than ideal. In this section, we'll talk about four types of databases: application, enterprise, proprietary, and legacy.

1.3.1 Application databases

Application databases are generally the smallest, simplest, and easiest databases to work with. These databases are usually the ones that we developers don't mind working with, or perhaps even like working with. Application databases are usually designed and implemented alongside the application as part of the same project. For this reason, there is generally more freedom in terms of design and are therefore more capable of making the design *right* for our particular application. There is minimal external influence in an application database, and there are usually only one or two interfaces. The first interface will be to the application, and the second might just be a simple reporting framework or tool like Crystal Reports. Figure 1.4 shows an application database and its relationships at a very high level.

Figure 1.4 Application database relationships

Application databases are sometimes small enough that they can be deployed to the same server as the application. With application databases there is more infrastructure freedom as well.

With small application databases, it is generally easier to convince companies to buy into using cheaper open source RDBMS solutions such as MySQL or PostgreSQL instead of spending money on Oracle or SQL Server. Some applications may even use an embedded application database that runs within the same virtual environment as the application itself, and therefore does not require a separate SQL at all.

iBATIS works very well as a persistence framework for application databases. Because of the simplicity of iBATIS, a team can get up to speed very quickly with a new application. For simple databases, it's even possible to generate the SQL from the database schema using the administrative tools that come with your RDBMS. Tools are also available that will generate all of the iBATIS SQL Map files for you.

1.3.2 Enterprise databases

Enterprise databases are larger than application databases and have greater external influence. They have more relationships with other systems that include both dependencies, as well as dependents. These relationships might be web applications and reporting tools, but they might also be interfaces to complex systems and databases. With an enterprise database, not only are there a greater number of external interfaces, but the way that the interfaces work is different too. Some interfaces might be nightly batch load interfaces, while others are real-time transactional interfaces. For this reason, the enterprise database itself might actually be composed of more than one database. Figure 1.5 depicts a high-level example of an enterprise database.

Enterprise databases impose many more constraints on the design and use of the database. There is a lot more to consider in terms of integrity, performance, and security. For this reason, enterprise databases are often split up to separate concerns and isolate requirements. If you tried to create a single database to meet all the requirements of an enterprise system, it would be extremely expensive and complex, or it would be completely impractical or even impossible.

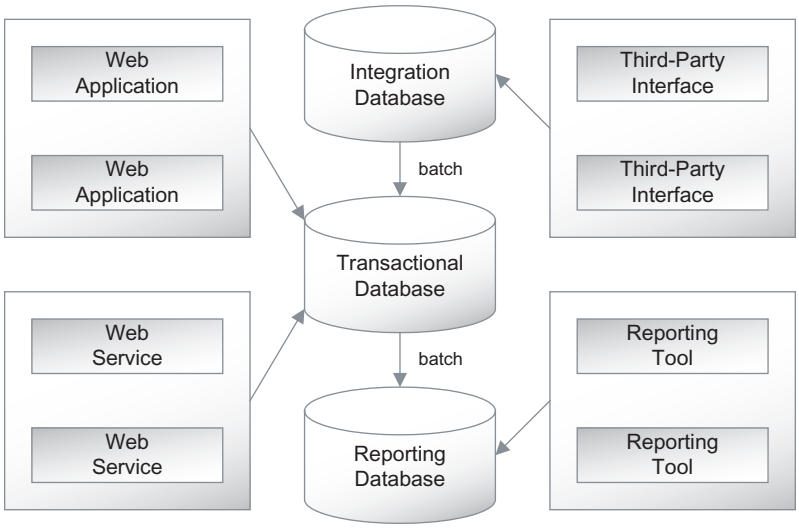

Figure 1.5 An example of enterprise database architecture

In the example depicted by figure 1.5, the requirements have been separated in terms of horizontal and nonfunctional requirements. That is, the databases have been separated into integration concerns, online transactional concerns, and reporting concerns. Both the integration database and the reporting database interface with the transactional system via a batch load, which implies that for this system it is acceptable to have reports that are not exactly up-to-date and that the transactional database only requires periodic updates from third-party systems. The advantage is that the transactional system has a great deal of load lifted from it and can have a simpler design as well. Generally it is not practical to design a database that is efficient for integration, transactions, and reporting. There are patterns for each that ensures the best performance and design. However, it is sometimes a requirement to have near real-time integration and reporting functions. For that reason this kind of design may not work. You might instead find that your enterprise database has to be partitioned vertically by business function.

Regardless of your enterprise database design, it's easy to appreciate the difference between an application database and an enterprise database. It's important to understand the particular limitations of your environment to ensure that your application uses the database effectively and is a good neighbor to other applications that are using the same database.

iBATIS works extremely well in an enterprise database environment. It has a number of features that make it ideal for working with complex database designs and large data sets. iBATIS also works well with multiple databases and does not assume that any type of object is coming from only one database. It also supports complex transactions that require multiple databases to be involved in a single transaction. Furthermore, iBATIS isn't only useful for online transactional systems, but works very well for both implementing reporting and integration systems.

1.3.3 Proprietary databases

If you've been working with software for any length of time, you've no doubt heard of the "build versus buy" debate. That is, should we build our own solution to a business problem, or buy a package that claims to solve the problem already. Often the cost is about the same (otherwise there would be no debate), but the real trade-off is between time to implement and the fit to the problem. Custom-built software can be tailored to an exact fit to business need, but takes more time to implement. Packages can be implemented very quickly, but sometimes don't quite meet every need. For that reason, when a choice is made to buy a package, businesses often decide that they can get the best of both worlds by digging into the proprietary database of the software to "extend" it just for the features that are missing.

We could discuss the horror stories of such a scenario, but it's probably better just to know that such proprietary databases were likely not meant to be touched by third parties. The designs are often full of assumptions, limitations, nonstandard data types, and other warning signs that can be easily read as "Enter at Your Own Risk." Regardless of the warning signs, businesses will do amazing things to save a few dollars. So software developers get stuck with navigating the jungle that is the proprietary database.

iBATIS is an excellent persistence layer for interfacing with proprietary databases. Often such databases allow for read-only access, which you can feel confident about when using iBATIS because you can restrict the kinds of SQL that are run. iBATIS won't perform any magical updates to the database when they aren't expected. If updates are required, proprietary databases are often very picky about how the data is structured. iBATIS allows you to write very specific update statements to deal with that.

1.3.4 Legacy databases

If ever there were a bane of a modern object-oriented developer's existence, it would be the legacy database. Legacy databases are generally the prehistoric remains of what was once an enterprise database. They have all of the complexities,

intricacies, and dependencies of an enterprise database. In addition, they have battle scars from years of modifications, quick fixes, cover-ups, workarounds, bandage solutions, and technical limitations. Furthermore, legacy databases are often implemented on older platforms that are not only outdated but are sometimes totally unsupported. There may not be adequate drivers or tools available for modern developers to work with.

iBATIS can still help with legacy databases. As long as there's an appropriate database driver available for the system you're working with, iBATIS will work the same way it does for any database. In fact, iBATIS is probably one of the best persistence frameworks around for dealing with legacy data, because it makes no assumptions about the database design and can therefore deal with even the most nightmarish of legacy designs.

1.4 How iBATIS handles common database challenges

On modern software projects databases are often considered legacy components. They have a history of being difficult to work with for both technical and nontechnical reasons. Most developers probably wish that they could simply start over and rebuild the database entirely. If the database is to remain, some developers might just wish that the DBAs responsible for it would take a long walk off a short pier. Both of these cases are impractical and unlikely to ever happen. Believe it or not, databases are usually the way they are for a reason—even if the reason isn't a good one. It may be that the change would be too costly or there may be other dependencies barring us from changing it. Regardless of why the database is challenged, we have to learn to work effectively with all databases, even challenged ones. iBATIS was developed mostly in response to databases that had very complex designs or even poor designs. The following sections describe some common database challenges and how iBATIS can help with them.

1.4.1 Ownership and control

The first and foremost difficulty with databases in a modern enterprise environment is not technical at all. It is simply the fact that most enterprises separate the ownership and responsibility for the database from the application development teams. Databases are often owned by a separate group within the enterprise altogether. If you're lucky, this group may work with your project team to help deliver the software. If you're unlucky, there will be a wall between your project team and the database group, over which you must volley your requirements and hope that they are received and understood. It's a sad truth, but it happens all the time.

Database teams are often difficult to work with. The primary reason is that they are under enormous pressure and are often dealing with more than one project. They often deal with multiple and sometimes even conflicting requirements. Administration of database systems can be difficult and many companies consider it a mission-critical responsibility. When an enterprise database system fails, corporate executives will know about it. For this reason, database administration teams are cautious. Change control processes are often much stricter for database systems than they are for application code. Some changes to a database might require data migration. Other changes may require significant testing to ensure that they don't impact performance. Database teams have good reasons for being difficult to work with, and therefore it's nice to be able to help them out a bit.

iBATIS allows a lot of flexibility when it comes to database design and interaction. DBAs like to be able to see the SQL that is being run and can also help tune complex queries, and iBATIS allows them to do that. Some teams that use iBATIS even have a DBA or data modeler maintain the iBATIS SQL files directly. Database administrators and SQL programmers will have no problem understanding iBATIS, as there is no magic happening in the background and they can see the SQL.

1.4.2 Access by multiple disparate systems

A database of any importance will no doubt have more than one dependent. Even if it is simply two small web applications sharing a single database, there will be a number of things to consider. Imagine a web application called Web Shopping Cart, which uses a database that contains Category codes. As far as Web Shopping Cart is concerned, Category codes are static and never change, so the application caches the codes to improve performance. Now imagine that a second web application called Web Admin is written to update Category codes. The Web Admin application is a separate program running on a different server. When Web Admin updates a category code, how does Web Shopping Cart know when to flush its cache of Category codes? This is a simple example of what is sometimes a complex problem.

Different systems might access and use the database in different ways. One application might be a web-based e-commerce system that performs a lot of database updates and data creation. Another might be a scheduled batch job for loading data from a third-party interface that requires exclusive access to the database tables. Still another might be a reporting engine that constantly stresses the database with complex queries. One can easily imagine the complexity that is possible.

The important point is that as soon as a database is accessed by more than one system, the situation heats up. iBATIS can help in a number of ways. First of all, iBATIS is a persistence framework that is useful for all types of systems, including

transactional systems, batch systems, and reporting systems. This means that regardless of what systems are accessing a given database, iBATIS is a great tool. Second, if you are able to use iBATIS, or even a consistent platform like Java, then you can use distributed caches to communicate among different systems. Finally, in the most complex of cases, you can easily disable iBATIS caching and write specific queries and update statements that behave perfectly, even when other systems using the same database do not.

1.4.3 Complex keys and relationships

Relational databases were designed and intended to follow a set of strict design rules. Sometimes these rules are broken, perhaps for a good reason, or perhaps not. Complex keys and relationships are usually the result of a rule being broken, misinterpreted, or possibly even overused. One of the relational design rules requires that each row of data be uniquely identified by a primary key. The simplest database designs will use a meaningless key as the primary key. However, some database designs might use what is called a *natural* key, in which case a part of the real data is used as the key. Still more complex designs will use a composite key of two or more columns. Primary keys are also often used to create relationships between other tables. So any complex or erroneous primary key definitions will propagate problems to the relationships between the other tables as well.

Sometimes the primary key rule is not followed. That is, sometimes data doesn't have a primary key at all. This complicates database queries a great deal as it becomes difficult to uniquely identify data. It makes creating relationships between tables difficult and messy at best. It also has a performance impact on the database in that the primary key usually provides a performance-enhancing index and is also used to determine the physical order of the data.

In other cases, the primary key rule might be overdone. A database might use composite natural keys for no practical reason. Instead the design was the result of taking the rule too seriously and implementing it in the strictest sense possible. Creating relationships between tables that use natural keys will actually create some duplication of real data, which is always a bad thing for database maintainability. Composite keys also create more redundancy when used as relationships, as multiple columns must be carried over to the related table to uniquely identify a single row. In these cases flexibility is lost because both natural keys and composite keys are much more difficult to maintain and can cause data-migration nightmares.

iBATIS can deal with any kind of complex key definition and relationship. Although it is always best to design the database properly, iBATIS can deal with tables with meaningless keys, natural keys, composite keys, or even no keys at all.

1.4.4 Denormalized or overnormalized models

Relational database design involves a process of eliminating redundancy. Elimination of redundancy is important to ensure that a database provides high performance and is flexible and maintainable. The process of eliminating redundancy in a data model is called *normalization*, and certain levels of normalization can be achieved. Raw data in tabular form generally will contain a great deal of redundancy and is therefore considered denormalized. Normalization is a complex topic that we won't discuss in great detail here.

When a database is first being designed, the raw data is analyzed for redundancy. A database administrator, a data modeler, or even a developer will take the raw data and normalize it using a collection of specific rules that are intended to eliminate redundancy. A denormalized relational model will contain redundant data in a few tables, each with a lot of rows and columns. A normalized model will have minimal or no redundancy and will have a greater number of tables, but each table will have fewer rows and columns.

There is no perfect level of normalization. Denormalization does have advantages in terms of simplicity and sometimes performance as well. A denormalized model can allow data to be stored and retrieved more quickly than if the data were normalized. This is true simply because there are fewer statements to issue, fewer joins to calculate, and generally less overhead. That said, denormalization should always be the exception and not the rule. A good approach to database design is to begin with a "by the book" normalized model. Then the model can be denormalized as needed. It is much easier to denormalize the database after the fact than it is to renormalize it. So always start new database designs with a normalized model.

It is possible to overnormalize a database, and the results can be problematic. Too many tables create a lot of relationships that need to be managed. This can include a lot of table joins when querying data, and it means multiple update statements are required to update data that is very closely related. Both of these characteristics can have a negative impact on performance. It also means that it's harder to map to an object model, as you may not want to have such fine-grained classes as the data model does.

Denormalized models are problematic too, possibly more so than overnormalized models. Denormalized models tend to have more rows and columns. Having too many rows impacts performance negatively in that there is simply more data to search through. Having too many columns is similar in that each row is bigger and therefore requires more resources to work with each time an update or a query is performed. Care must be taken with these *wide* tables to ensure that only columns

that are required for the particular operation are included in the update or query. Furthermore, a denormalized model can make efficient indexing impossible.

iBATIS works with both denormalized models and overnormalized models. It makes no assumptions about the granularity of your object model or database, nor does it assume that they are the same or even remotely alike. iBATIS does the best job possible of separating the object model from the relational model.

1.4.5 Skinny data models

Skinny data models are one of the most notorious and problematic abuses of relational database systems. Unfortunately, they're sometimes necessary. A skinny data model basically turns each table into a generic data structure that can store sets of name and value pairs, much like a properties file in Java or an old-school INI (initialization) file in Windows. Sometimes these tables also store metadata such as the intended data type. This is necessary because the database only allows one type definition for a column. To better understand a skinny data model, consider the following example of typical address data, shown in table 1.2.

Table 1.2 Address data in typical model

ADDRESS_ID	STREET	CITY	STATE	ZIP	COUNTRY
1	123 Some Street	San Francisco	California	12345	USA
2	456 Another Street	New York	New York	54321	USA

Obviously this address data could be normalized in a better way. For example, we could have related tables for COUNTRY, STATE and CITY, and ZIP. But this is a simple and effective design that works for a lot of applications. Unless your requirements are complex, this is unlikely to be a problematic design.

If we were to take this data and arrange it in a skinny table design, it would look like table 1.3.

Table 1.3 Address data in a skinny model

ADDRESS_ID	FIELD	VALUE
1	STREET	123 Some Street
1	CITY	San Francisco
1	STATE	California
1	ZIP	12345

Table 1.3 Address data in a skinny model *(continued)*

ADDRESS_ID	FIELD	VALUE
1	COUNTRY	USA
2	STREET	456 Another Street
2	CITY	New York
2	STATE	New York
2	ZIP	54321
2	COUNTRY	USA

This design is an absolute nightmare. To start, there is no hope of possibly normalizing this data any better than it already is, which can only be classified as first normal form. There's no chance of creating managed relationships with COUNTRY, CITY, STATE, or ZIP tables, as we can't define multiple foreign key definitions on a single column. This data is also difficult to query and would require complex subqueries if we wanted to perform a query-by-example style query that involved a number of the address fields (e.g., searching for an address with both street and city as criteria). When it comes to updates, this design is especially poor in terms of performance; inserting a single address requires not one, but five insert statements on a single table. This can create greater potential for lock contention and possibly even deadlocks. Furthermore, the number of rows in the skinny design is now five times that of the normalized model. Due to the number of rows, the lack of data definition, and the number of update statements required to modify this data, effective indexing becomes impossible.

Without going further, it's easy to see why this design is problematic and why it should be avoided at all costs. The one place that it is useful is for dynamic fields in an application. Some applications have a need to allow users to add additional data to their records. If the user wants to be able to define new fields and insert data into those fields dynamically while the application is running, then this model works well. That said, all known data should still be properly normalized, and then these additional dynamic fields can be associated to a parent record. The design still suffers all of the consequences as discussed, but they are minimized because most of the data (probably the important data) is still properly normalized.

Even if you encounter a skinny data model in an enterprise database, iBATIS can help you deal with it. It is difficult or maybe even impossible to map classes to a skinny data model, because you don't know what fields there might be. You'd have better luck mapping such a thing to a hashtable, and luckily iBATIS supports

that. With iBATIS, you don't necessarily have to map every table to a user-defined class. iBATIS allows you to map relational data to primitives, maps, XML, and user-defined classes (e.g., JavaBeans). This great flexibility makes iBATIS extremely effective for complex data models, including skinny data models.

1.5 Summary

iBATIS was designed as a hybrid solution that does not attempt to solve every problem, but instead solves the most important problems. iBATIS borrows from the various other methods of access. Like a stored procedure, every iBATIS statement has a signature that gives it a name and defines its inputs and outputs (encapsulation). Similar to inline SQL, iBATIS allows the SQL to be written in the way it was supposed to be, and to use language variables directly for parameters and results. Like Dynamic SQL, iBATIS provides a means of modifying the SQL at runtime. Such queries can be dynamically built to reflect a user request. From object/relational mapping tools, iBATIS borrows a number of concepts, including caching, lazy loading, and advanced transaction management.

In an application architecture, iBATIS fits in at the persistence layer. iBATIS supports other layers by providing features that allow for easier implementation of requirements at all layers of the application. For example, a web search engine may require paginated lists of search results. iBATIS supports such features by allowing a query to specify an offset (i.e., a starting point) and the number of rows to return. This allows the pagination to operate at a low level, while keeping the database details out of the application.

iBATIS works with databases of any size or purpose. It works well for small application databases because it is simple to learn and quick to ramp up. It is excellent for large enterprise applications because it doesn't make any assumptions about the database design, behaviors, or dependencies that might impact how our application uses the database. Even databases that have challenging designs or are perhaps surrounded by political turmoil can easily work with iBATIS. Above all else, iBATIS has been designed to be flexible enough to suit almost any situation while saving you time by eliminating redundant boilerplate code.

In this chapter we've discussed the philosophy and the roots of iBATIS. In the next chapter we'll explain exactly what iBATIS is and how it works.

JAVA

iBATIS IN ACTION

Clinton Begin • Brandon Goodin • Larry Meadors

Unlike some complex and invasive persistence solutions, iBATIS keeps O/RM clean and simple. It is an elegant persistence framework that maps classes to SQL statements and keeps the learning curve flat. The iBATIS approach makes apps easy to code, test, and deploy. You write regular SQL and iBATIS gives you standard objects for persistence and retrieval. There's no need to change existing database schemas—iBATIS is tolerant of legacy databases (even badly designed ones).

iBATIS in Action is a comprehensive tutorial on the framework and an introduction to the iBATIS philosophy. Clinton Begin and coauthors lead you through the core features, including configuration, statements, and transactions. Because you'll need more than the basics, it explores sophisticated topics like Dynamic SQL and data layer abstraction. You'll also learn a useful skill: how to extend iBATIS itself. A complete, detailed example shows you how to put iBATIS to work. Topics are clearly organized and easily accessible for reference.

What's Inside

- A comprehensive iBATIS tutorial
- Learn iBATIS techniques, patterns, and best practices
- How to build a complete web application
- Inside story from the authoritative source

Clinton Begin is a senior developer at ThoughtWorks Canada and the creator of iBATIS. **Brandon Goodin** is a consultant who has contributed to the iBATIS project since 2003. **Larry Meadors** is a consultant and trainer who has been involved with iBATIS since version 1.x.

"Unique and invaluable, this book will be at my side for years to come."
—Nathan Maves, Senior Java Architect
Sun Microsystems

"This book really shines."
—Benjamin Gorlick
Global Engineered Products, LLC.

"The writing is good, relaxed, and sometimes fun."
—Dick Zetterberg, Transitor AB

"Gets new users going and gives experienced users in-depth coverage of advanced features."
—Jeff Cunningham
The Weather Channel Interactive

"Easy flow, good breakdown of topics, relevant and thorough—valuable for all."
—Rick Reumann
Nielsen Media Research

Ask the Authors


Ebook edition

www.manning.com/begin

ISBN 1-932394-82-6

$44.99 US/$58.99 Canada