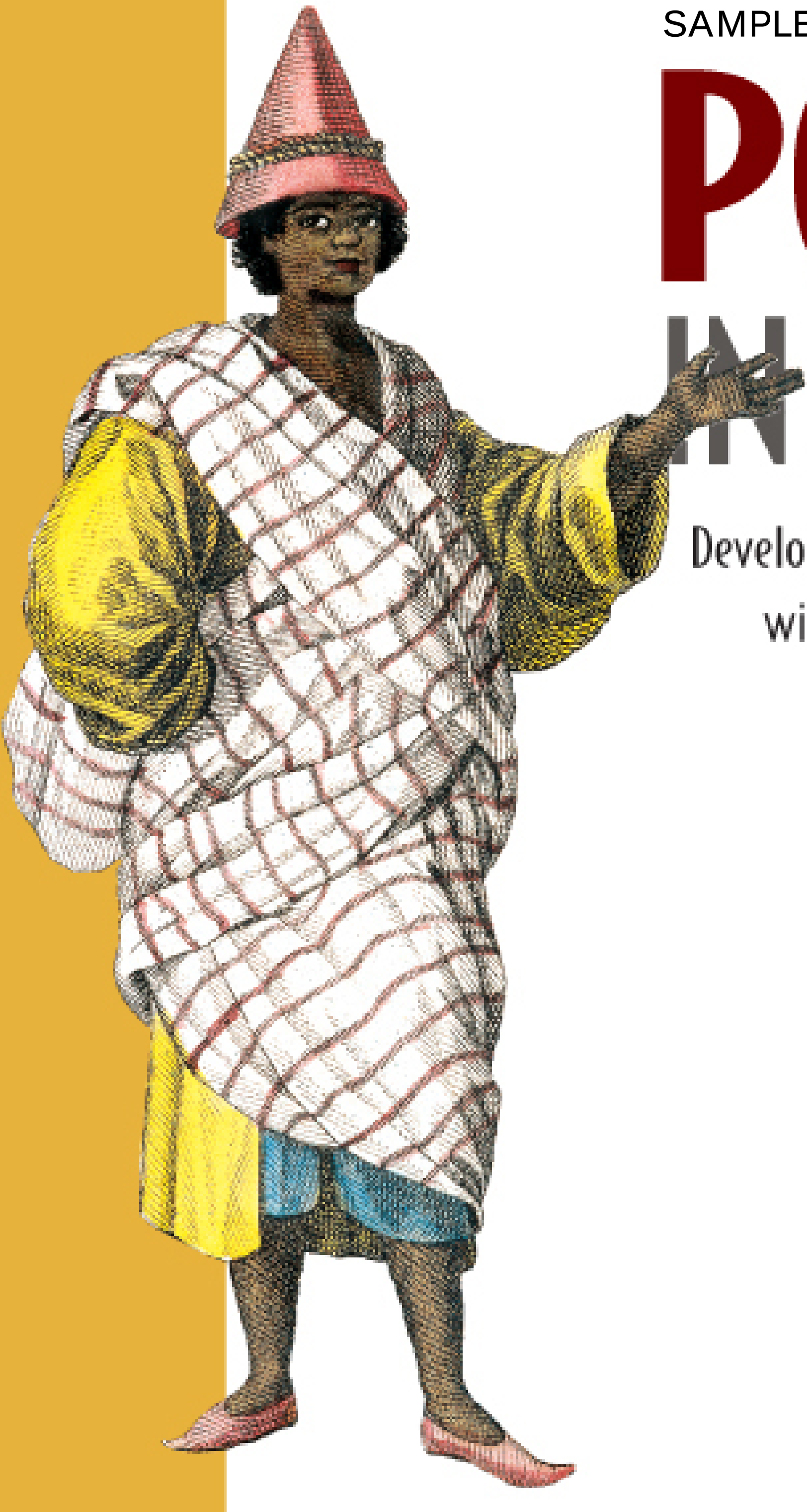
SAMPLE CHAPTER

POJOs IN ACTION

Developing Enterprise Applications with Lightweight Frameworks



Chris Richardson

MANNING

POJOs in Action
by Chris Richardson

Chapter 2

contents

2 J2EE design decisions

This chapter covers

- Encapsulating the business logic
- Organizing the business logic
- Accessing the database
- Handling database concurrency

Now that you have had a glimpse of how POJOs and lightweight frameworks such as Spring and JDO make development easier and faster, let's take a step back and look at how you would decide whether and how to use them. If we blindly used POJOs and lightweight frameworks, we would be repeating the mistake the enterprise Java community made with EJBs. Every technology has both strengths and weaknesses, and it's important to know how to choose the most appropriate one for a given situation.

This book is about implementing enterprise applications using design patterns and lightweight frameworks. To enable you to use them effectively in your application, it provides a decision-making framework that consists of five key questions that must be answered when designing an application or implementing the business logic for an individual use case. By consciously addressing each of these design issues and understanding the consequences of your decisions, you will vastly improve the quality of your application.

In this chapter you will get an overview of those five design decisions, which are described in detail in the rest of this book. I briefly describe each design decision's options as well as their respective benefits and drawbacks. I also introduce the example application that is used throughout this book and provide an overview of how to make decisions about its architecture and design.

2.1 Business logic and database access decisions

As you saw in chapter 1, there are two quite different ways to design an enterprise Java application. One option is to use the classic EJB 2 approach, which I will refer to as the heavyweight approach. When using the heavyweight approach, you use session beans and message-driven beans to implement the business logic. You use either DAOs or entity beans to access the business logic.

The other option is to use POJOs and lightweight frameworks, which I'll refer to as the POJO approach. When using the POJO approach, your business logic consists entirely of POJOs. You use a persistence framework (a.k.a., object/relational mapping framework) such as Hibernate or JDO to access the database, and you use Spring AOP to provide enterprise services such as transaction management and security.

EJB 3 somewhat blurs the distinction between the two approaches because it has embraced POJOs and some lightweight concepts. For example, entity beans are POJOs that can be run both inside and outside the EJB container. However, while session beans and message-driven beans are POJOs they also have heavyweight behavior since they can only run inside the EJB container. So, as you can see, EJB 3 has both heavyweight and POJO characteristics. EJB 3 entity beans are part of the lightweight approach whereas session beans and message-driven beans are part of the heavyweight approach.

Choosing between the heavyweight approach and the POJO approach is one of the first of myriad design decisions that you must make during development. It's a decision that affects several aspects of the application, including business logic organization and the database access mechanism. To help decide between the two approaches, let's look at the architecture of a typical enterprise application, which is shown in figure 2.1, and examine the kinds of decisions that must be made when developing it.

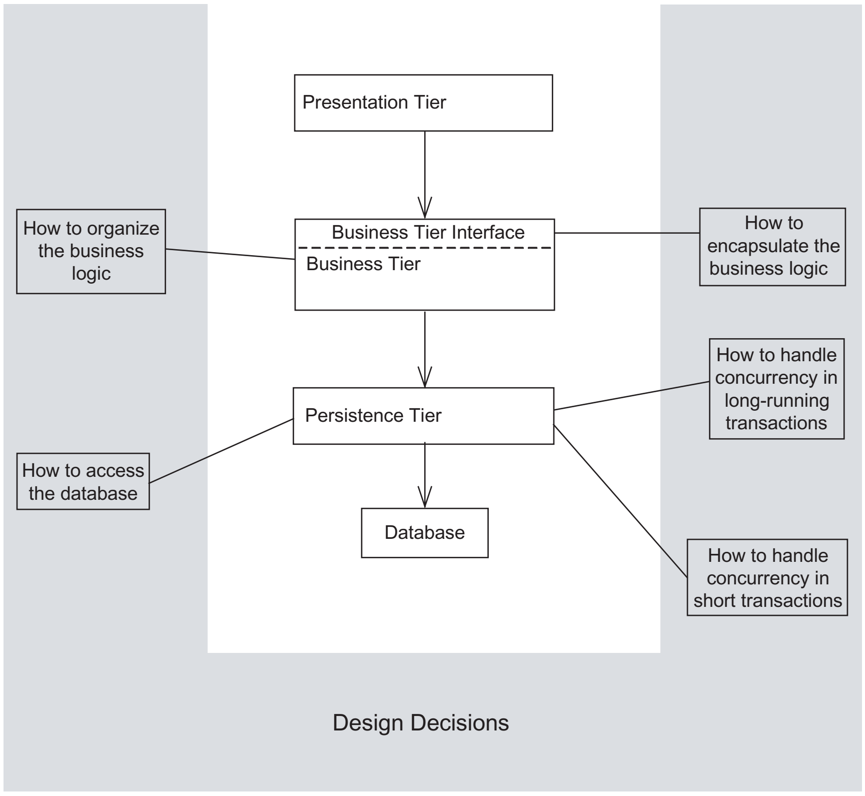

Presentation Tier
How to organize the business logic
Business Tier Interface
Business Tier
How to encapsulate the business logic
How to handle concurrency in long-running transactions
Persistence Tier
How to access the database
Database
How to handle concurrency in short transactions
Design Decisions

Figure 2.1 A typical application architecture and the key business logic and database access design decisions

The application consists of the web-based presentation tier, the business tier, and the persistence tier. The web-based presentation tier handles HTTP requests and generates HTML for regular browser clients and XML and other content for rich Internet clients, such as Ajax-based clients. The business tier, which is invoked by the presentation tier, implements the application's business logic. The persistence tier is used by the business tier to access external data sources such as databases and other applications.

The design of the presentation tier is outside the scope of this book, but let's look at the rest of the diagram. We need to decide the structure of the business tier and the interface that it exposes to the presentation tier and its other clients. We also need to decide how the persistence tier accesses databases, which is the main source of data for many applications. We must also decide how to handle concurrency in short transactions and long-running transactions. That adds up to five decisions that any designer/architect must make and that any developer must know in order to understand the big picture.

These decisions determine key characteristics of the design of the application's business and the persistence tiers. There are, of course, many other important decisions that you must make—such as how to handle transactions, security, and caching and how to assemble the application—but as you will see later in this book, answering those five questions often addresses these other issues as well.

Each of the five decisions shown in figure 2.1 has multiple options. For example, in chapter 1 you saw two different options for three of these decisions. The EJB-based design, which was described in section 1.1, consisted of procedural code implemented by a session bean and used JDBC to access the database. In comparison, the POJO-based design, which was described in section 1.2, consisted of an object model, which was mapped to the database using JDO and was encapsulated with a POJO façade that used Spring for transaction management.

Each option has benefits and drawbacks that determine its applicability to a given situation. As you will see in this chapter, each one makes different trade-offs in terms of one or more areas, including functionality, ease of development, maintainability, and usability. Even though I'm a big fan of the POJO approach, it is important to know these benefits and drawbacks so that you can make the best choices for your application.

Let's now take a brief look at each decision and its options.

2.2 Decision 1: organizing the business logic

These days a lot of attention is focused on the benefits and drawbacks of particular technologies. Although this is certainly very important, it is also essential to think about how your business logic is structured. It is quite easy to write code without giving much thought to how it is organized. For example, as I described in the previous chapter it is too easy to add yet more code to a session bean instead of carefully deciding which domain model class should be responsible for the new functionality. Ideally, however, you should consciously organize your business logic in the way that's the most appropriate for your application. After all, I'm sure you've experienced the frustration of having to maintain someone else's badly structured code.

The key decision you must make is whether to use an object-oriented approach or a procedural approach. This isn't a decision about technologies, but your choice of technologies can potentially constrain the organization of the business logic. Using EJB 2 firmly pushes you toward a procedural design whereas POJOs and lightweight frameworks enable you to choose the best approach for your particular application. Let's examine the options.

2.2.1 Using a procedural design

While I am a strong advocate of the object-oriented approach, there are some situations where it is overkill, such as when you are developing simple business logic. Moreover, an object-oriented design is sometimes infeasible—for example, if you do not have a persistence framework to map your object model to the database. In such a situation, a better approach is to write procedural code and use what Fowler calls the *Transaction Script* pattern [Fowler 2002]. Rather than doing any object-oriented design, you simply write a method, which is called a transaction script, to handle each request from the presentation tier.

An important characteristic of this approach is that the classes that implement behavior are separate from those that store state. In an EJB 2 application, this typically means that your business logic will look similar to the design shown in figure 2.2. This kind of design centralizes behavior in session beans or POJOs, which implement the transaction scripts and manipulate "dumb" data objects that have very little behavior. Because the behavior is concentrated in a few large classes, the code can be difficult to understand and maintain.

The design is highly procedural, and relies on few of the capabilities of object-oriented programming (OOP) languages. This is the type of design you would create if you were writing the application in C or another non-OOP language.

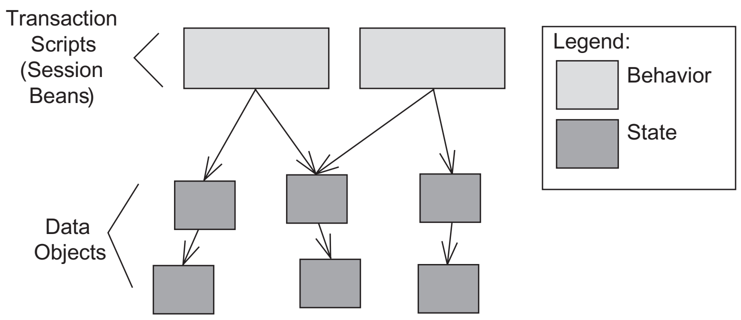


Figure 2.2 The structure of a procedural design: large transaction script classes and many small data objects

Nevertheless, you should not be ashamed to use a procedural design when it is appropriate. In chapter 9 you will learn when it does make sense and see some ways to improve a procedural design.

2.2.2 *Using an object-oriented design*

The simplicity of the procedural approach can be quite seductive. You can just write code without having to carefully consider how to organize the classes. The problem is that if your business logic becomes complex, then you can end up with code that's a nightmare to maintain. Consequently, unless you are writing an extremely simple application you should resist the temptation to write procedural code and instead develop an object-oriented design.

In an object-oriented design, the business logic consists of an object model, which is a network of relatively small classes. These classes typically correspond directly to concepts from the problem domain. For example, in the money transfer example in section 1.2 the POJO version consists of classes such as `TransferService`, `Account`, `OverdraftPolicy`, and `BankingTransaction`, which correspond to concepts from the banking domain. As figure 2.3 shows, in such a design some classes have only either state or behavior but many contain both, which is the hallmark of a well-designed class.

As we saw in chapter 1, an object-oriented design has many benefits, including improved maintainability and extensibility. You can implement a simple object model using EJB 2 entity beans, but to enjoy most of the benefits you must use POJOs and a lightweight persistence framework such as Hibernate and JDO. POJOs enable you to develop a rich domain model, which makes use of such features as inheritance and loopback calls. A lightweight persistence framework enables you to easily map the domain model to the database.

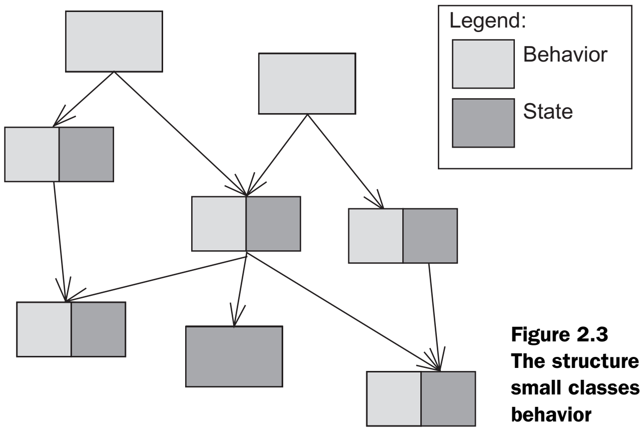


Figure 2.3
The structure of a domain model: small classes that have state and behavior

Another name for an object model is a domain model, and Fowler calls the object-oriented approach to developing business logic the *Domain Model* pattern [Fowler 2002]. In chapter 3 I describe one way to develop a domain model and in chapters 4-6 you will learn about how to persist a domain model with Hibernate and JDO.

2.2.3 Table Module pattern

I have always developed applications using the Domain Model and *Transaction Script* patterns. But I once heard rumors of an enterprise Java application that used a third approach, which is what Martin Fowler calls the *Table Module* pattern. This pattern is more structured than the Transaction Script pattern, because for each database table it defines a table module class that implements the code that operates on that table. But like the Transaction Script pattern it separates state and behavior into separate classes because an instance of a table module class represents the entire database rather individual rows. As a result, maintainability is a problem. Consequently, there is very little benefit to using the Table Module pattern, and so I'm not going to look at it in anymore detail in this book.

2.3 Decision 2: encapsulating the business logic

In the previous section, I covered how to organize the business logic. You must also decide what kind of interface the business logic should have. The business logic's interface consists of those types and methods that are callable by the presentation tier. An important consideration when designing the interface is how much of the business logic's implementation should be encapsulated and therefore not visible to the presentation tier. Encapsulation improves maintainability because by hiding the business logic's implementation details it can prevent changes to it affecting the presentation tier. The downside is that you must typically write more code to encapsulate the business logic.

You must also address other important issues, such as how to handle transactions, security, and remoting, since they are generally the responsibility of the business logic's interface code. The business tier's interface typically ensures that each call to the business tier executes in a transaction in order to preserve the consistency of the database. Similarly, it also verifies that the caller is authorized to invoke a business method. The business tier's interface is also responsible for handling some kinds of remote clients.

Let's consider the options.

2.3.1 EJB session facade

The classic-J2EE approach is to encapsulate business logic with an EJB-based session façade. The EJB container provides transaction management, security, distributed transactions, and remote access. The façade also improves maintainability by encapsulating the business logic. The coarse-grained API can also improve performance by minimizing the number of calls that the presentation tier must make to the business tier. Fewer calls to the business tier reduce the number of database transactions and increase the opportunity to cache objects in memory. It also reduces the number of network round-trips if the presentation tier is accessing the business tier remotely. Figure 2.4 shows an example of an EJB-based session façade.

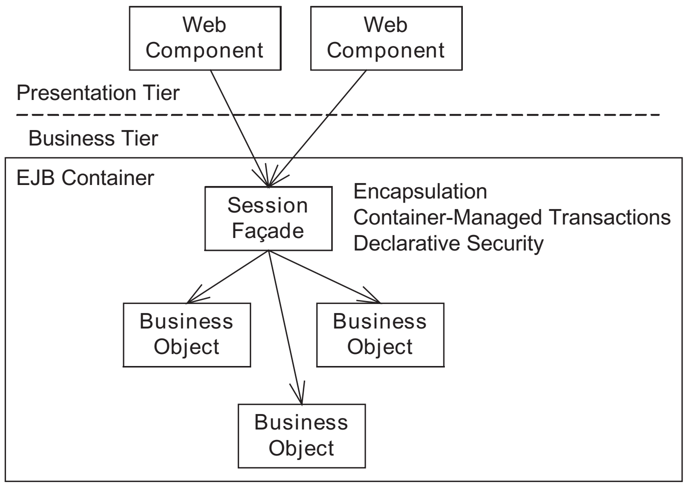


Figure 2.4
Encapsulating the business logic with an EJB session façade

In this design, the presentation tier, which may be remote, calls the façade. The EJB container intercepts the calls to the façade, verifies that the caller is authorized, and begins a transaction. The façade then calls the underlying objects that implement the business logic. After the façade returns, the EJB container commits or rolls back the transaction.

Unfortunately, using an EJB session façade has some significant drawbacks. For example, EJB session beans can only run in the EJB container, which slows development and testing. In addition, if you are using EJB 2, then developing and maintaining DTOs, which are used to return data to the presentation tier, is tedious and time consuming.

2.3.2 POJO façade

For many applications, a better approach uses a POJO façade in conjunction with an AOP-based mechanism such as the Spring framework that manages transactions, persistence framework connections, and security. A POJO facade encapsulates the business tier in a similar fashion to an EJB session façade and usually has the same public methods. The key difference is that it's a POJO instead of an EJB and that services such as transaction management and security are provided by AOP instead of the EJB container. Figure 2.5 shows an example of a design that uses a POJO façade.

The presentation tier invokes the POJO façade, which then calls the business objects. In the same way that the EJB container intercepts the calls to the EJB

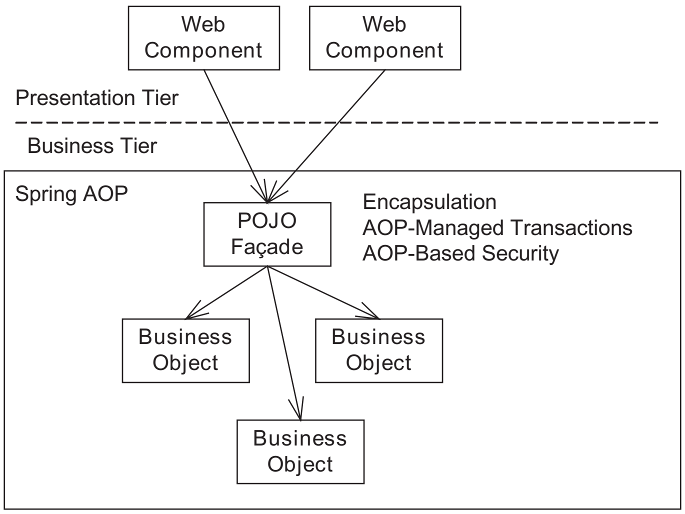


Figure 2.5 Encapsulating the business logic with a POJO façade

façade, the AOP interceptors intercept the calls to the POJO façade and authenticate the caller and begin, commit, and roll back transactions.

The POJO façade approach simplifies development by enabling all of the business logic to be developed and tested outside the application server, while providing many of the important benefits of EJB session beans such as declarative transactions and security. As an added bonus, you have to write less code. You can avoid writing many DTO classes because the POJO façade can return domain objects to the presentation tier; you can also use dependency injection to wire the application's components together instead of writing JNDI lookup code.

However, as you will see in chapter 7 there are some reasons not to use the POJO façade. For example, a POJO façade cannot participate in a distributed transaction initiated by a remote client.

2.3.3 Exposed Domain Model pattern

Another drawback of using a façade is that you must write extra code. Moreover, as you will see in chapter 7, the code that enables persistent domain objects to be returned to the presentation tier is especially prone to errors. There is the increased risk of runtime errors caused by the presentation tier trying to access an object that was not loaded by the business tier. If you are using JDO, Hibernate, or EJB 3, you can avoid this problem by exposing the domain model to the presentation tier and letting the business tier return the persistent domain objects back to the presentation tier. As the presentation tier navigates relationships between domain objects, the persistence framework will load the objects that it accesses, a technique known as *lazy loading*. Figure 2.6 shows a design in which the presentation tier freely accesses the domain objects.

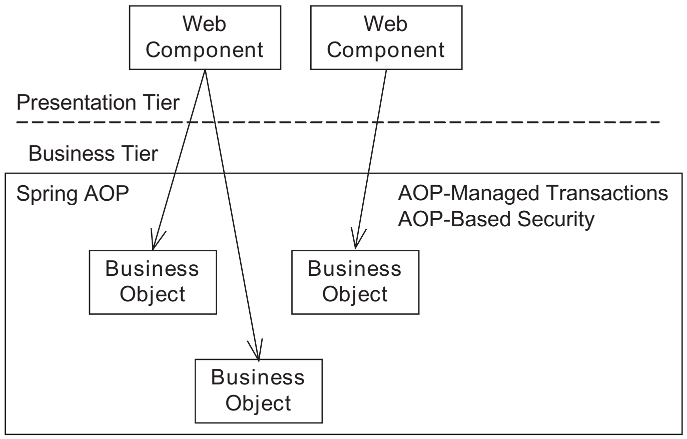


Figure 2.6 Using an exposed domain model

In the design in figure 2.6, the presentation tier calls the domain objects directly without going through a façade. Spring AOP continues to provide services such as transaction management and security.

An important benefit of this approach is that it eliminates the need for the business tier to know what objects it must load and return to the presentation tier. However, although this sounds simple you will see there are some drawbacks. It increases the complexity of the presentation tier, which must manage database connections. Transaction management can also be tricky in a web application because transactions must be committed before the presentation tier sends any part of the response back to the browser. Chapter 8 describes how to address these issues and implement an exposed domain model.

2.4 Decision 3: accessing the database

No matter how you organize and encapsulate the business logic, eventually you have to move data to and from the database. In a classic J2EE application you had two main choices: JDBC, which required a lot of low-level coding, or entity beans, which were difficult to use and lacked important features. In comparison, one of the most exciting things about using lightweight frameworks is that you have some new and much more powerful ways to access the database that significantly reduce the amount of database access code that you must write. Let's take a closer look.

2.4.1 What's wrong with using JDBC directly?

The recent emergence of object/relational mapping frameworks (such as JDO and Hibernate) and SQL mapping frameworks (such as iBATIS) did not occur in a vacuum. Instead, they emerged from the Java community's repeated frustrations with JDBC. Let's review the problems with using JDBC directly in order to understand the motivations behind the newer frameworks. There are three main reasons why using JDBC directly is not a good choice for many applications:

- *Developing and maintaining SQL is difficult and time consuming*—Some developers find writing large, complex SQL statements quite difficult. It can also be time consuming to update the SQL statements to reflect changes in the database schema. You need to carefully consider whether the loss of maintainability is worth the benefits.
- *There is a lack of portability with SQL*—Because you often need to use database-specific SQL, an application that works with multiple databases must have multiple versions of some SQL statements, which can be a maintenance

nightmare. Even if your application only works with one database in production, SQL's lack of portability can be an obstacle to using a simpler and faster in-memory database such as Hypersonic Structured Query Language Database Engine (HSQLDB) for testing.

- *Writing JDBC code is time consuming and error-prone*—You must write lots of boilerplate code to obtain connections, create and initialize `PreparedStatements`, and clean up by closing connections and prepared statements. You also have to write the code to map between Java objects and SQL statements. As well as being tedious to write, JDBC code is also error-prone.

The first two problems are unavoidable if your application must execute SQL directly. Sometimes, you must use the full power of SQL, including vendor-specific features, in order to get good performance. Or, for a variety of business-related reasons, your DBA might demand complete control over the SQL statements executed by your application, which can prevent you from using persistence frameworks that generate the SQL on the fly. Often, the corporate investment in its relational databases is so massive that the applications working with the databases can appear relatively unimportant. Quoting the authors of *iBATIS in Action*, there are cases where "the database and even the SQL itself have outlived the application source code, or even multiple versions of the source code. In some cases, the application has been rewritten in a *different language*, but the SQL and database remained largely unchanged." If you are stuck with using SQL directly, then fortunately there is a framework for executing it directly, one that is much easier to use than JDBC. It is, of course, iBATIS.

2.4.2 Using iBATIS

All of the enterprise Java applications I've developed executed SQL directly. Early applications used SQL exclusively whereas the later ones, which used a persistence framework, used SQL in a few components. Initially, I used plain JDBC to execute the SQL statements, but later on I often ended up writing mini-frameworks to handle the more tedious aspects of using JDBC. I even briefly used Spring's JDBC classes, which eliminate much of the boilerplate code. But neither the homegrown frameworks nor the Spring classes addressed the problem of mapping between Java classes and SQL statements, which is why I was excited to come across iBATIS.

In addition to completely insulating the application from connections and prepared statements, iBATIS maps JavaBeans to SQL statements using XML descriptor files. It uses Java bean introspection to map bean properties to prepared statement placeholders and to construct beans from a `ResultSet`. It also includes support for database-generated primary keys, automatic loading of related objects, caching, and lazy loading. In this way, iBATIS eliminates much of the drudgery of executing SQL statements. As you will see in several chapters, including chapter 9, iBATIS can considerably simplify code that executes SQL statements. Instead of writing a lot of low-level JDBC code, you write an XML descriptor file and make a few calls to iBATIS APIs.

2.4.3 Using a persistence framework

Of course, iBATIS cannot address the overhead of developing and maintaining SQL or its lack of portability. To avoid those problems you need to use a persistence framework. A persistence framework maps domain objects to the database. It provides an API for creating, retrieving, and deleting objects. It automatically loads objects from the database as the application navigates relationships between objects and updates the database at the end of a transaction. A persistence framework automatically generates SQL using the object/relational mapping, which is typically specified by an XML document that defines how classes are mapped to tables, how fields are mapped to columns, and how relationships are mapped to foreign keys and join tables.

EJB 2 had its own limited form of persistence framework: entity beans. However, EJB 2 entity beans have so many deficiencies, and developing and testing them is extremely tedious. As a result, EJB 2 entity beans should rarely be used. What's more, as I describe in chapter 10 it is unclear how some of their deficiencies will be addressed by EJB 3.

The two most popular lightweight persistence frameworks are JDO[JSR12] [JSR243], which is a Sun standard, and Hibernate, which is an open source project. They both provide transparent persistence for POJO classes. You can develop and test your business logic using POJO classes without worrying about persistence, then map the classes to the database schema. In addition, they both work inside and outside the application server, which simplifies development further. Developing with Hibernate and JDO is so much more pleasurable than with old-style EJB 2 entity beans.

Several chapters in this book describe how to use JDO and Hibernate effectively. In chapter 5 you will learn how to use JDO to persist a domain model. Chapter 6 looks at how to use Hibernate to persist a domain model. In chapter 11 you will learn how to use JDO and Hibernate to efficiently query large databases and process large result sets.

In addition to deciding how to access the database, you must decide how to handle database concurrency. Let's look at why this is important as well as the available options.

2.5 Decision 4: handling concurrency in database transactions

Almost all enterprise applications have multiple users and background threads that concurrently update the database. It's quite common for two database transactions to access the same data simultaneously, which can potentially make the database inconsistent or cause the application to misbehave. In the `TransferService` example in chapter 1, two transactions could update the same bank account simultaneously, and one transaction could overwrite the other's changes; money could simply disappear. Given that the modern banking system is not backed by gold, nor even paper, but just supported by electronic systems, I'm sure you can appreciate the importance of transaction integrity.

Most applications must handle multiple transactions concurrently accessing the same data, which can affect the design of the business and persistence tiers.

Applications must, of course, handle concurrent access to shared data regardless of whether they are using lightweight frameworks or EJBs. However, unlike EJB 2 entity beans, which required you to use vendor-specific extensions, JDO and Hibernate directly support most of the concurrency mechanisms. What's more, using them is either a simple configuration issue or requires only a small amount of code.

The details of concurrency management are described in chapters 12 and 13. In this section, you will get a brief overview of the different options for handling concurrent updates in database transactions, which are transactions that do not involve any user input. In the next section, I briefly describe how to handle concurrent updates in longer application-level transactions, which are transactions that involve user input and consist of a sequence of database transactions.

2.5.1 Isolated database transactions

Sometimes you can simply rely on the database to handle concurrent access to shared data. Databases can be configured to execute database transactions that are, in database-speak, isolated from one another. Don't worry if you are not familiar with this concept; it's explained in more detail in chapter 12. For now the key thing to remember is that if the application uses fully isolated transactions, then the net effect of executing two transactions simultaneously will be as if they were executed one after the other.

On the surface this sounds extremely simple, but the problem with these kinds of transactions is that they have what is sometimes an unacceptable reduction in performance because of how isolated transactions are implemented by the database. For this reason, many applications avoid them and instead use what is termed optimistic or pessimistic locking, which is described a bit later.

Chapter 12 looks at when to use database transactions that are isolated from one another and how to use them with iBATIS, JDO, and Hibernate.

2.5.2 Optimistic locking

One way to handle concurrent updates is to use optimistic locking. Optimistic locking works by having the application check whether the data it is about to update has been changed by another transaction since it was read. One common way to implement optimistic locking is to add a version column to each table, which is incremented by the application each time it changes a row. Each `UPDATE` statement's `WHERE` clause checks that the version number has not changed since it was read. An application can determine whether the `UPDATE` statement succeeded by checking the row count returned by `PreparedStatement.executeUpdate()`. If the row has been updated or deleted by another transaction, the application can roll back the transaction and start over.

It is quite easy to implement an optimistic locking mechanism in an application that executes SQL statements directly. But it is even easier when using persistence frameworks such as JDO and Hibernate because they provide optimistic locking as a configuration option. Once it is enabled, the persistence framework automatically generates SQL `UPDATE` statements that perform the version check. Chapter 12 looks at when to use optimistic locking, explores its drawbacks, and shows you how to use it with iBATIS, JDO, and Hibernate.

Optimistic locking derives its name from the fact it assumes that concurrent updates are rare and that instead of preventing them the application detects and recovers from them. An alternative approach is to use pessimistic locking, which assumes that concurrent updates will occur and must be prevented.

2.5.3 Pessimistic locking

An alternative to optimistic locking is pessimistic locking. A transaction acquires locks on the rows when it reads them, which prevent other transactions from accessing the rows. The details depend on the database, and unfortunately not all databases support pessimistic locking. If it is supported by the database, it is quite easy to implement a pessimistic locking mechanism in an application that executes SQL statements directly. However, as you would expect, using pessimistic locking in a JDO or Hibernate application is even easier. JDO provides pessimistic locking as a configuration option, and Hibernate provides a simple programmatic API for locking objects. Again, in chapter 12 you will learn when to use pessimistic locking, examine its drawbacks, and see how to use it with iBATIS, JDO, and Hibernate.

In addition to handling concurrency within a single database transaction, you must often handle concurrency across a sequence of database transactions.

2.6 Decision 5: handling concurrency in long transactions

Isolated transactions, optimistic locking, and pessimistic locking only work within a single database transaction. However, many applications have use cases that are long running and that consist of multiple database transactions which read and update shared data. For example, one of the use cases that you will encounter later in this chapter is the Modify Order use case, which describes how a user edits an order (the shared data). This is a relatively lengthy process, which might take as long as several minutes and consists of multiple database transactions. Because data is read in one database transaction and modified in another, the application must handle concurrent access to shared data differently. It must use the *Optimistic Offline Lock* pattern or the *Pessimistic Offline Lock* pattern, two patterns described by Fowler [Fowler 2002].

2.6.1 Optimistic Offline Lock pattern

One option is to extend the optimistic locking mechanism described earlier and check in the final database transaction of the editing process that the data has not changed since it was first read. You can, for example, do this by using a version number column in the shared data's table. At the start of the editing process, the application stores the version number in the session state. Then, when the user saves their changes, the application makes sure that the saved version number matches the version number in the database.

In chapter 13 you will learn more about when to use Optimistic Offline Lock pattern and how to use it with iBATIS, JDO, and Hibernate. Because the Optimistic Offline Lock pattern only detects changes when the user tries to save their changes, it only works well when starting over is not a burden on the user. When implementing use cases such as the Modify Order use case where the user would be extremely annoyed by having to discard several minutes' work, a much better option is to use the Pessimistic Offline Lock.

2.6.2 *Pessimistic Offline Lock pattern*

The Pessimistic Offline Lock pattern handles concurrent updates across a sequence of database transactions by locking the shared data at the start of the editing process, which prevents other users from editing it. It is similar to the pessimistic locking mechanism described earlier except that the locks are implemented by the application rather than the database. Because only one user at a time is able to edit the shared data, they are guaranteed to be able to save their changes. In chapter 13 you will learn more about when to use Pessimistic Offline Lock pattern, examine some of the implementation challenges, and see how to use it with iBATIS, JDO, and Hibernate.

Let's review the five design decisions. These decisions and their options are summarized in table 2.1. In the rest of the book you will learn more about each option, examining in particular its benefits and drawbacks and how to implement it.

Table 2.1 The key business logic design decisions and their options

Decision	Options
Business logic organization	Domain Model pattern Transaction Script pattern Table Module pattern
Business logic encapsulation	EJB Session Façade pattern POJO Façade pattern Exposed Domain Model pattern
Database access	Direct JDBC iBATIS Hibernate JDO
Concurrency in database transactions	Ignore the problem Pessimistic locking Optimistic locking Serializable isolation level
Concurrency in long-running transactions	Ignore the problem Pessimistic Offline Lock pattern Optimistic Offline Lock pattern

Now that you have gotten an overview of the business logic and database access design decisions, let's see how a development team applies them.

2.7 Making design decisions on a project

In this section you will see an example of how a development team goes about making the five design decisions I introduced in this chapter. It illustrates the kind of decision-making process that you must use when choosing between the POJO approach and the heavyweight approach. The team in this example is developing an application for a fictitious company called Food to Go Inc. I describe how the developers make decisions about the overall design of the Food to Go application and decisions about the design of the business logic for individual use cases.

2.7.1 Overview of the example application

Before seeing how the team makes decisions, let's first review some background information about the problem the team is trying to solve, and the application's high-level architecture. This will set the stage for a discussion of how a development team can go about making design decisions. Food To Go Inc. is a company that delivers food orders from restaurants to customers' homes and offices. The founders of Food to Go have decided to build a J2EE-based application to run their business. This application supports the following kinds of users:

- *Customers*—Place orders and check order status
- *Customer service reps*—Handle phone enquiries from customers about their orders
- *Restaurants*—Maintain menus and prepare the orders
- *Dispatchers*—Assign drivers to orders
- *Drivers*—Pick up orders from restaurants and deliver them

The company has put together a team consisting of five developers: Mary, Tom, Dick, Harry, and Wanda. They are all experienced developers who will jointly make architectural decisions in addition to implementing the application. The businesspeople and the development team kick off the project by meeting for a few days to refine the requirements and develop a high-level architecture.

The requirements

After a lot of discussion, they jointly decide on the following scenario to describe how an order flows through the system. The sequence of events is as follows:

1. The customer places the order via the website.
2. The system sends the order (by fax or email) to the restaurant.

3. The restaurant acknowledges receipt of the order.
4. A dispatcher assigns a driver to the order.
5. The system sends a notification to the assigned driver.
6. The driver views the assigned order on a cell phone.
7. The driver picks up the order from the restaurant and notifies the system that the order has been picked up.
8. The driver delivers the order to the customer and notifies the system that the order has been delivered.

In addition to coming up with a scenario that captures the vision of how the application will ultimately work, the developers and businesspeople also break down the application's requirements into a set of use cases. Given that Food to Go has limited resources, the team has decided to use an iterative and incremental approach to developing the application. They have decided to defer the implementation of use cases for dispatches and drivers to later iterations and to tackle the following use cases in the first iteration:

- *Place Order*—Describes how a customer places an order
- *View Orders*—Describes how a customer service representative can view orders
- *Send Orders to Restaurant*—Describes how the system sends orders to restaurants
- *Acknowledge Order*—Describes how a restaurant can acknowledge receipt of an order
- *Modify Order*—Describes how a customer service representative can modify an order

These use cases are used throughout this book to illustrate how to develop enterprise Java applications with POJOs and lightweight frameworks. I describe each of these use cases in a bit more detail later in this chapter, but let's first look at the application's high-level architecture.

The application's architecture

In the kickoff meeting, the team also sketches out the application's high-level architecture, which is shown in figure 2.7. This diagram shows the application's main components and its actors. It has the standard three-layer architecture consisting of the web-based presentation, business, and database access tiers. As you would expect, the application stores its data in a relational database.

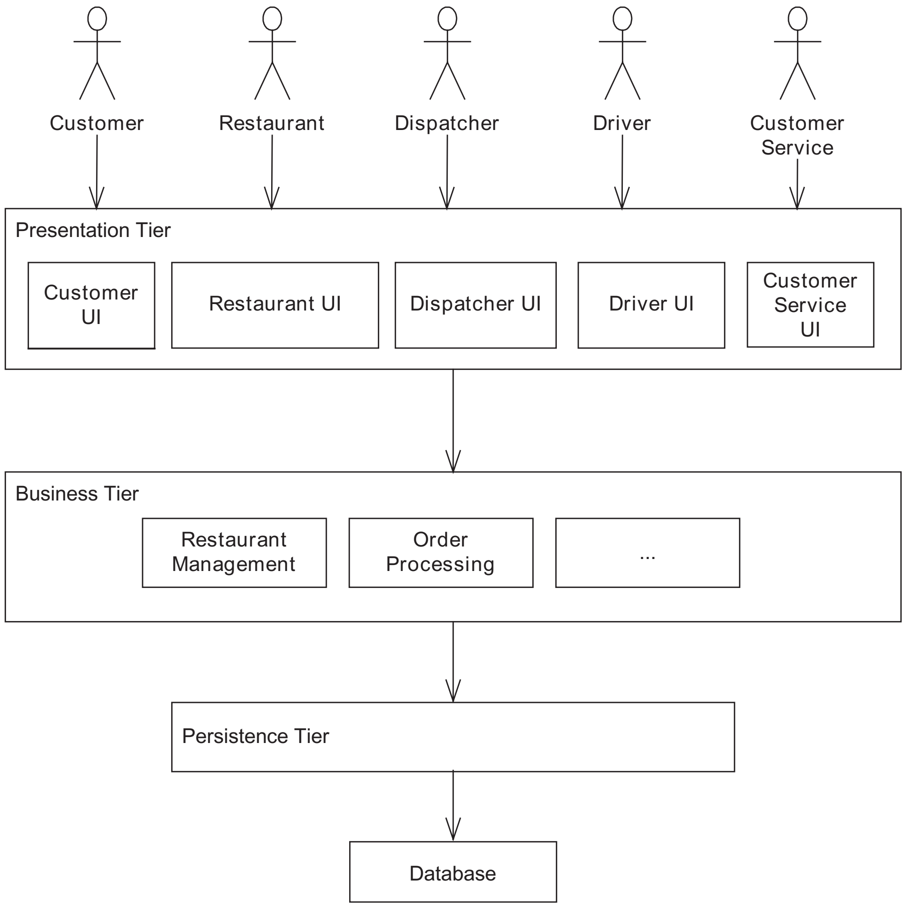


Figure 2.7 High-level architecture of the Food to Go application

The application has a web-based presentation tier that implements the user interface (UI) for the users. The application's business tier consists of various components that are responsible for order management and restaurant management. The application's persistence tier is responsible for accessing the database. The design of the presentation tier is outside the scope of this book, and so we are going to focus on the design of the business and persistence tiers. Let's see how the team makes some critical design decisions.

2.7.2 *Making high-level design decisions*

After identifying some requirements and sketching out a high-level architecture, the team needs to make the high-level design decisions that determine the overall design of the application. In this section, we consider each of the five design decisions that we described earlier and show how a development team might make those decisions. You will learn about the kind of process that you must use when designing your application.

Organizing the business logic

The business logic for this application is responsible for such tasks as determining which restaurants can deliver to a particular address at the specified time, applying discounts, charging credit cards, and scheduling drivers. The team needs to choose between an object-oriented approach or a procedural approach. When making this decision, the team first considers the potential complexity of the business logic. After reviewing the use cases, the team concludes that it could become quite complex, which means that using an object-oriented approach and developing a domain model is the best approach. Even though it is simpler, using a procedural approach to organize the business logic would lead to maintenance problems in the future.

The team also briefly looks at the issue of whether they could use a persistence framework to access the database. Unlike when developing some past applications, they are not constrained by a legacy schema or the requirement to use SQL statements maintained by a database administrator (DBA). Consequently, they are free to use a persistence framework and to implement the business logic using a domain model. However, they also decide that some business logic components can use a procedural approach if they must access the database in ways that are not supported by the persistence framework.

Encapsulating the business logic

In the past the team used EJB-based session façades to encapsulate the business logic and provide transaction management and security. EJB session façades worked reasonably well except for the impact they have on the edit-compile-debug cycle. Eager to avoid the tedium of deploying EJBs, the team is ready to adopt a more lightweight approach. Mary, who has just returned from the TSS Java Symposium 2005, where she spent three days hearing about POJOs, dependency injection, lightweight containers, AOP, and EJB 3, convinces the rest of the team to use the Spring framework instead of EJBs.

Having decided to use Spring, the team must now decide between using POJO façades and the exposed domain model. After spending a lot of time discussing these two options, they decide that the exposed domain model approach is too radical and that they are more comfortable using a POJO façade.

Accessing the database

Because the team has decided to use a domain model, it must pick a persistence framework. It would simply be too much work to persist the domain model without one. On its last project, the team used EJB CMP because, despite its glaring deficiencies, it was at that time the most mature solution. JDO was still in its infancy and the team had not yet heard of Hibernate. However, that was quite some time ago, and since then the team members have all read a few articles about JDO and Hibernate and decide that they are powerful and mature technologies. They are excited that they do not have to use entity beans again. After an animated discussion, the team picks JDO because its company prefers to use standards that are supported by multiple vendors. It hopes, however, to use Hibernate on some other project in the future.

Handling concurrent updates

The Food to Go application, like many other enterprise applications, is a *multiuser* application, which means that multiple transactions will access the same data concurrently. For example, two transactions could attempt to update the same order simultaneously. Therefore, it's essential to have a concurrency strategy. After reviewing the three options—isolated database transactions, optimistic locking, and pessimistic locking—the team picks optimistic locking because they have had experience with it and know that it performs well. Moreover, it is supported by JDO, which means that using it involves a simple configuration option.

Handling offline concurrency

Some of the application's use cases, such as the Modify Order use case, are long-running application transactions where data read in one database transaction is updated in another database transaction. In order to prevent two users from editing the same order simultaneously and overwriting each other's changes, it's important to implement an offline concurrency mechanism. The Optimistic Offline Lock pattern is easier to implement, especially because the application can leverage the optimistic locking mechanism provided by the persistence framework. However, the team decides to use the Pessimistic Offline Lock pattern for the `Order` class because users would be frustrated if they could not save the changes that they made to an order.

Summary of the high-level decisions

The team has made a number of key design decisions. They have decided that the business logic must be primarily organized using a JDO-based domain model, and encapsulated using POJO façades that use detached domain objects as DTOs. Finally, they have decided to use optimistic locking as the database-level concurrency mechanism, the Optimistic Offline Lock pattern as the default offline locking mechanism, and the Pessimistic Offline Lock pattern when necessary. However, these decisions are not completely set in stone, and they agree to revisit them as more is discovered about the application during development. Table 2.2 summarizes the architectural choices and options available to the developers.

Table 2.2 Architectural decisions

Decision	Options
Business logic organization strategy	Domain model with transaction scripts where necessary
Business logic encapsulation strategy	POJO façade
Persistence strategy	JDO for the domain model
Online concurrency strategy	Optimistic locking
Offline concurrency strategy	Optimistic Offline Lock pattern Pessimistic Offline Lock pattern (if required)

Table 2.2 shows the default design decisions the team made when implementing each component of the application. However, a developer working on a particular use case can use a different approach if it is absolutely necessary. For example, she might discover that the business logic for a use case needs to execute SQL directly instead of JDO in order to achieve the necessary performance. Let's look at examples of the decisions that are made when developing individual use cases.

2.7.3 Making use case–level decisions

Mary, Tom, Dick, Harry, and Wanda are each responsible for analyzing one use case and determining the most appropriate option for each design decision. Naturally, they have to work within the constraints imposed by the architecture that they have defined. In addition, even though some business logic components are specifically for a single use case, others are shared by multiple use cases and so it is essential that the developers collaborate closely.

In this section we show how a developer might go about designing the business logic for a use case and direct you to the chapters that will teach you how to implement the chosen options. It's important to remember, however, that the decisions made by each developer in this section are only one of several different ways to implement the use case. Consequently, we also point you to the chapters that describe how to implement alternative approaches. Let's look at each of the use cases and see which options the developer's pick.

The Place Order use case

Mary is responsible for implementing the Place Order use case:

> The customer enters the delivery address and time. The system first verifies that the delivery time is in the future and that at least one restaurant serves the delivery information. It then updates the pending order with the delivery information, and displays a list of available restaurants.
>
> The customer selects a restaurant. The system updates the pending order with the restaurant and displays the menu for the selected restaurant.
>
> The customer enters quantities for each menu item. The system updates the pending order with the quantities and displays the updated pending order.
>
> The customer enters payment information (credit card information and billing address). The system updates the pending order with the payment information and displays the pending order with totals, tax, and charges.
>
> The customer confirms that she wants to place the order. The system authorizes the credit card, creates the order, and displays an order confirmation, which includes the order number.

As you can see, the business logic for this use case is fairly complex, and so it makes sense to implement it using a domain model that is persisted with JDO. Database concurrency isn't an issue because this use case does not update any shared data. The pending order is data that is private to a single user's session and the order, which is shared data, is not updated in this use case once it has been created. After analyzing the use case, Mary makes the decisions shown in table 2.3. In chapter 4, you will learn how to develop a domain model for the Place Order use case; chapter 5 shows you how to persist it with JDO. In chapter 6, we describe

Table 2.3 Mary's decisions

Strategy	Decision	Rationale
Business logic organization	Domain Model pattern	The business logic is relatively complex. There does not appear to be any queries that cannot be handled by the JDO query language.
Database access	JDO	Using the Domain Model pattern.
Concurrency	None	This use case does not update shared data. The order is created at the end of the use case. The pending order is session state and is only updated by this session.

how to persist that domain model with Hibernate, and in chapter 9 you will see how to implement the same business logic using a procedural approach.

The View Orders use case

Tom is responsible for implementing the View Orders use case:

> The customer service representative enters the search criteria. The system displays the orders that match the search criteria. The customer service representative can cancel or modify an order.

Tom analyzes this use case and concludes that a key issue is that the order table will contain a large number of rows and will need to be denormalized for efficient access. In addition, the queries will need to be heavily tuned and make use of Oracle-specific features. Consequently, Tom decides that he needs to use SQL queries to retrieve the orders. Table 2.4 summarizes his decisions.

Table 2.4 Tom's decisions

Strategy	Decision	Rationale
Business logic organization	Transaction Script pattern	Simple business logic. Uses iBATIS.
Database access	iBATIS	Heavily optimized SQL queries using Oracle-specific features. Database schema denormalized for efficient access.
Concurrency	None	This use case does not update shared data.

In chapter 11, you will learn about the different ways to implement this use case.

The Send Orders to Restaurant use case

Dick is responsible for implementing the Send Orders to Restaurant use case:

> *X* minutes before the scheduled delivery time, the system either emails or faxes the order to the restaurant.

The business logic for this use case is fairly simple. Dick determines that he can implement this use case using a single database transaction, which finds the orders that need to be sent, sends them to the restaurant, and updates the orders. He also decides that even though the business logic is simple, it fits with the existing domain model. Table 2.5 summarizes his decisions.

Table 2.5 Dick's decisions

Strategy	Decision	Rationale
Business logic organization	Domain Model pattern	Even though the business logic is simple, it fits with the existing domain model.
Database access	JDO	Using the Domain Model pattern.
Concurrency	Optimistic locking	The use case updates orders, which consist of shared data in a single transaction.
Offline concurrency	None	The use case is a single transaction.

Dick forgets that the `Order` class needs to use an offline locking pattern.

Chapter 12 looks at the different ways of implementing this use case.

The Acknowledge Order use case

Harry is responsible for implementing the Acknowledge Order use case:

> The system displays an order that has been sent to the restaurant. The restaurant's order taker accepts or rejects the order. The system displays a confirmation page. The restaurant's order taker confirms that he or she accepts or rejects the order. The system changes the state of the order to "ACCEPTED" or "REJECTED."

Harry determines that the business logic for this use case is quite simple and that he can implement it using the Domain Model pattern. He decides that he must use an offline locking pattern because this use case uses two database transactions: one to read the order, and another to change the status of the order. Table 2.6 lists the design decisions that Harry makes.

Table 2.6 Harry's decisions

Strategy	Decision	Rationale
Business logic organization	Domain Model pattern	Even though the business logic is simple, it fits with the existing domain model.
Database access	JDO	Using the Domain Model pattern.
Concurrency	Optimistic locking	The use case updates orders, which are shared data.
Offline concurrency	Optimistic Offline Lock pattern	The use case reads the order in one transaction and updates it in another. The cost and probability of starting over is small.

Harry also forgets that the `Order` class needs to use an offline locking pattern.

Chapter 13 looks at the different ways of implementing this use case.

The Modify Order use case

Finally, Wanda is responsible for implementing the Modify Order use case:

> The customer service representative selects the order to edit. The system locks and displays the order. The customer service representative updates the quantities and the delivery address and time. The system displays the updated order. The customer service representative saves his changes. The system updates and unlocks the order.

After analyzing the use case, Wanda makes the following decisions. Because the business logic is complex, she decides to implement it using the Domain Model pattern. Furthermore, Wanda thinks that she can reuse a lot of the pending order code from the Place Order use case.

Wanda also decides that she must use an offline concurrency pattern since the business logic consists of multiple database transactions. Because it would be very inconvenient for the user to start over if some other user changed the order while she was editing it, Wanda decides to use the Pessimistic Offline Lock pattern. Table 2.7 summarizes Wanda's decisions.

Table 2.7 Wanda's decisions

Strategy	Decision	Rationale
Business logic organization	Domain Model pattern	Complex business logic.
Database access	JDO	Using the Domain Model pattern.
Concurrency	Optimistic locking	The use case updates orders, which consist of shared data.
Offline concurrency	Pessimistic Offline Lock pattern	The use case reads the order in one transaction and updates it in another. The cost of starting over is high.

Wanda plans to meet with Dick and Harry to reconcile their respective concurrency requirements.

Chapter 13 looks at the different ways of implementing this use case.

2.8 Summary

This chapter describes how the task of designing the business and persistence tiers can be broken down into five main design decisions: organizing business logic; encapsulating business logic; accessing the database; handling database transaction-level concurrency; and handling concurrency in long-running transactions. Each decision has multiple options, and each option has benefits and drawbacks that determine whether it makes sense in a particular situation.

These decisions play a critical role in helping you decide between a POJO approach and a heavyweight EJB 2 approach. Some decisions have POJO options and heavyweight options. For example, you can encapsulate the business logic with a POJO façade or an EJB session façade. Other decisions have options that are made easier by using the POJO approach. For example, as we described in chapter 1, the heavyweight approach favors business logic that is organized procedurally, whereas the POJO approach enables you to use an object-oriented design.

Now that we have reviewed the design decisions and their options, let's examine each one in depth. In the next chapter, we first look at how to implement business logic using the Domain Model pattern.

JAVA

POJOs IN ACTION

Chris Richardson

Developing Enterprise Applications with Lightweight Frameworks

There is agreement in the Java community that EJBs often introduce more problems than they solve. Now there is a major trend toward lightweight technologies such as Hibernate, Spring, JDO, iBATIS, and others, all of which allow the developer to work directly with the simpler Plain Old Java Objects, or POJOs. Bowing to the new consensus, EJB 3 now also works with POJOs.

POJOs in Action describes these new, simpler, and faster ways to develop enterprise Java applications. It shows you how to go about making key design decisions, including how to organize and encapsulate the domain logic, access the database, manage transactions, and handle database concurrency.

Written for developers and designers, this is a new-generation Java applications guide. It helps you build lightweight applications that are easier to build, test, and maintain. The book is uniquely practical with design alternatives illustrated through numerous code examples.

What's Inside

- Leverage the frameworks' strengths, avoid their weaknesses
- Apply enterprise patterns in the lightweight world
- New patterns like POJO Façade and Exposed Domain Model
- Build rich domain models
- How Aspects improve design
- Lightweight testing strategies
- How to be agile

Chris Richardson is a developer and architect with over 20 years of experience. His consulting company specializes in jumpstarting projects and mentoring teams. Chris has been a technical leader at Insignia, BEA, and elswhere. He has a computer science degree from the University of Cambridge in England and lives in Oakland, CA.

“A good way to quickly get up to speed with today's practices for lightweight development.”
—Floyd Marinescu
Founder, InfoQ.com
Creator, TheServerSide.com

“Brings back simplicity to enterprise Java applications.”
—Jonas Bonér
Senior Software Achitect
Terracotta, Inc.

“A valuable guide for lightweight development.”
—Craig Walls
Author, *Spring in Action*

“The author definitely knows what he is talking about.”
—Oliver Zeigermann
J2EE Architect and
Apache committer

“Extremely valuable, plenty of sample code... I enthusiastically recommend it!”
—Brendan Murray
Senior Software Achitect
IBM

AUTHOR ONLINE
Ask the Author

Ebook edition

www.manning.com/crichardson

54495
9 781932 394580
ISBN 1-932394-58-3

MANNING $44.95 US/$60.95 Canada