PART 1  OVERVIEW OF POJOS AND LIGHTWEIGHT FRAMEWORKS .................................1
Chapter 1  ■ Developing with POJOS: faster and easier  3
Chapter 2  ■ J2EE design decisions  31

PART 2  A SIMPLER, FASTER APPROACH......................... 59
Chapter 3  ■ Using the Domain Model pattern  61
Chapter 4  ■ Overview of persisting a domain model  95
Chapter 5  ■ Persisting a domain model with JDO 2.0  149
Chapter 6  ■ Persisting a domain model with Hibernate 3  195
Chapter 7  ■ Encapsulating the business logic with a POJO façade  243

PART 3  VARIATIONS ........................................... 287
Chapter 8  ■ Using an exposed domain model  289
Chapter 9  ■ Using the Transaction Script pattern  317
Chapter 10 ■ Implementing POJOS with EJB 3  360
PART 4  DEALING WITH DATABASES AND CONCURRENCY ........................................405

Chapter 11 ➡ Implementing dynamic paged queries 407
Chapter 12 ➡ Database transactions and concurrency 451
Chapter 13 ➡ Using offline locking patterns 488
Developing with POJOs: faster and easier

This chapter covers
- Comparing lightweight frameworks and EJBs
- Simplifying development with POJOs
- Developing an object-oriented design
- Making POJOs transactional and persistent
Sometimes you must use a technology for a while in order to appreciate its true value. A few years ago I had to go out of the country on a business trip, and I didn’t want to risk missing episodes of my favorite show. So, rather than continuing to struggle with the timer function on my VCR, I bought a TiVo box. At the time I thought it was simply going to be a much more convenient and reliable way to record programs. The TiVo box certainly made it easy to record a show, but before long it completely changed how I watched television. In addition to being able to pause live TV, I was able to watch my favorite shows when I wanted and without commercials.

I had a similar experience with plain old Java objects (POJOs), Hibernate, and Spring. I was part of a team developing a server product that had a “classic” Enterprise JavaBeans (EJB) architecture: the business logic consisted of session beans and entity beans. EJB definitely helped by handling infrastructure issues such as transaction management, security, and persistence—but at a high price. For example, we endured long edit-compile-debug cycles caused by having to deploy the components in the application server. We also jumped through all kinds of hoops in order to implement a domain model with entity beans. But somehow we accepted all of this pain as normal.

The final straw was when we were faced with having to support the product on two application servers. Rather than endure the lack of portability of EJB container-managed persistence (CMP) we decided to be adventurous and use a portable persistence mechanism that I was hearing a lot about: Hibernate. Hibernate worked the same way on both application servers and eliminated the need to maintain two separate but equivalent sets of EJB CMP deployment descriptors. But before long we discovered other, much more important benefits of Hibernate. It enabled us to implement a more elaborate POJO domain model in the next version of the product. It sped development by allowing the domain model to be tested without an application server or a database. And soon after we discovered the Spring framework, which enabled us to create a more loosely coupled architecture consisting of easy-to-test POJO services. In hindsight, it’s amazing that we accomplished as much as we did with the old architecture.

*POJOs in Action* describes how POJOs and lightweight technologies such as Spring, Hibernate, and Java Data Objects (JDO) make it easier and faster to develop testable and maintainable applications. You will learn how object-oriented design goes hand in hand with POJOs and how to endow POJOs with the characteristics that enterprise applications require, such as transactions and persistence. It describes how to use Spring for transaction management and Hibernate, JDO, EJB 3, and iBATIS for persistence.
The disillusionment with EJBs

Much of this book focuses on alternatives to EJBs because they frequently offer better characteristics: good object-oriented design, testability, less complexity, easier maintenance, and a raft of other benefits. However, it’s important to remember that EJBs are sometimes the right tool for the job, which is why chapter 10 is about using EJB 3. The key is to be conscious of the options and to make explicit informed decisions rather than slavishly following dogma.

1.1 The disillusionment with EJBs

This book isn’t a screed about why you shouldn’t use “traditional” Java 2 Enterprise Edition (J2EE) architecture and design. It is sometimes the best tool for the job, and later on in this book I describe when you should use it. However, today many developers use it for applications for which it is ill suited. Let’s briefly review the history of EJBs and discover why the Java development community’s initial enthusiasm for them has turned into disillusionment. After that, I will describe an alternative approach to designing an enterprise Java application that uses POJOs.

1.1.1 A brief history of EJBs

EJB is the Java standard architecture for writing distributed business applications. It’s a framework that provides a large number of useful services and handles some of the most time-consuming aspects of writing distributed applications. For example, EJB provides declarative transactions, which eliminate the need to write transaction management code. The EJB container automatically starts, commits, and rolls back transactions on behalf of the application. Automatically handling transactions was a huge innovation at the time and is still a vital service. In addition, business logic implemented using EJBs can participate in distributed transactions that are started by a remote client. EJBs also provide declarative security, which mostly eliminates the need to write security code, which is another common requirement handled by the application server. Entries in the bean’s deployment descriptor specified who could access a particular bean.

EJB version 1.0 was released in 1998. It provided two types of enterprise beans: session beans and entity beans. Session beans represent either stateless services or a stateful conversation with a client. Entity beans represent data in a database and were originally intended to implement business objects. EJB 1.0 fulfilled its mandate by insulating the application developer from the complexities of building distributed enterprise systems. EJB 2 refined the EJB programming model. It added message-driven beans (which process Java Message Service, or JMS, messages) as well as enhanced entity beans to support relationships managed
by the container. The evolution continues in EJB 3 (described later in this chapter), which simplifies the programming model considerably by enabling POJOs to be EJBs.

### 1.1.2 A typical EJB 2 application architecture

Let’s look at an example of a typical EJB 2 application architecture. Imagine that you work for a bank and you have to write a service to transfer money between two accounts. Figure 1.1 shows how you might use EJB to implement the money transfer service.

The business logic consists of the TransferService EJB and data access objects (DAOs). The TransferService EJB is a session bean that defines the interface that the business logic exposes to the presentation tier. It also implements the business logic.

The TransferService EJB calls the AccountDAO to retrieve the two accounts, and performs any necessary checks and other business logic. For example, it verifies that fromAccount contains sufficient funds and will not become overdrawn. The TransferService EJB calls AccountDAO again to save the updated accounts in the database. It records the transfer by calling TransactionDAO. The TransferService

![Diagram](image-url)

**Figure 1.1** The money transfer service implemented using a typical EJB-based design
EJB returns a TransferResult, which is a DTO that contains the AccountDTOs and their recent transactions. It is used by the presentation tier to display a web page to the customer.

The DAOs, which are implemented using JDBC, provide methods for accessing the database. This application could also use entity beans instead of DAOs to access the database. That is, after all, the role of entity beans within the J2EE architecture. However, for reasons I describe later, EJB 2 entity beans have several drawbacks and limitations. As a result, many J2EE applications use DAOs instead of EJB 2 entity beans.

The class design and their relationships are simple. I haven’t shown the XML deployment descriptors, which are used to configure the EJB, but TransferService is ready to be invoked remotely and to participate in distributed transactions. But despite its apparent simplicity (and sometimes because of it), several serious problems lurk within.

1.1.3 The problems with EJBs

Like many other Java developers, I enthusiastically adopted EJBs and spent several years writing applications whose design was similar to the one you just saw. I was so excited about using the new standard that I thought nothing of abandoning the object-oriented design skills I’d spent the previous decade learning. I was more than happy to write lots of code and XML configuration files just to do the simplest things. I found ways to pass the time while my code deployed. After all, isn’t enterprise application development meant to be challenging?

It is certainly true that some aspects of developing enterprise applications are challenging, such as complex and changing requirements and the need to scale and have high throughput and availability. However, while EJB solves some problems with developing enterprise applications, it does not live up to one of its key goals: making it easy to write applications. Ironically, in order to be a competent EJB developer you need to know how to solve problems that are caused by EJB. An excellent book that tackles the shortcomings of EJB is Bitter EJB by Bruce Tate [Tate 2003]. Other books address the complexity of building effective EJB applications, such as Core J2EE Patterns [Alur 2003] and EJB Design Patterns [Marinescu 2002], which contains patterns to help make sense of EJB and solutions to problems rather than patterns for improving the design of software.

Although these books help developers grapple with EJB and learn how to use it effectively, they don’t directly address the two fundamental problems with EJBs. The first is that EJBs encourage developers to write procedural-style applications. The second problem is that the cumbersome nature of the development process
when using EJBs doesn’t allow developers to take advantage of many of the best practices used for “normal” Java development.

The shortcomings of procedural design

There are two main ways to organize business logic: procedural or object-oriented. The procedural approach organizes the code around functions that manipulate separate simple data objects. In procedural architectures, data structures are populated, passed as parameters to functions, and returned to the caller. The relationship between the data and the operations is very loosely defined, and wholly maintained by the developer. Prior to object-oriented languages, this style of programming dominated software development, and was featured in C, Pascal, and other languages.

By contrast, the object-oriented approach organizes code around objects that have state and behavior and that collaborate with other objects. The data structures and operations are defined in one language construct, co-locating the data and the operations on the data. The relationship (and state) between the data and the operations is maintained by the language. An object-oriented design is easier to understand, maintain, extend, and test than a procedural design.

Despite the benefits of an object-oriented design, most J2EE applications, including the one shown in figure 1.1, are written in a procedural style. In our example, all of the business logic is concentrated in the TransferService EJB, which consists of the transfer() method and possibly one or more helper methods. None of the objects manipulated by the TransferService EJB implement any business logic. These objects exist to provide plumbing and services to the TransferService EJB. The DAOs are wrappers around JDBC, and the remaining objects (including the entity beans) are simple data objects. Even though this business logic is written in Java, which is an object-oriented language, this design fits the definition of procedural code exactly.

The procedural design style isn’t a problem if the business logic is simple, but business logic has a habit of growing in complexity. As the requirements change and the business logic has to implement new features, the amount of code in the EJB steadily increases. For example, in order to add a new kind of overdraft policy you would have to add yet more code to the TransferService EJB to implement that new policy. Even if each enhancement only adds a few lines of code, EJBs that started out quite simple over time can grow into large complex beasts, such as the ones that I encountered on one early J2EE project that were each many hundred of lines of code.
EJBs like these that contain large amount of code cause several problems. The lack of modularity makes them difficult to understand and maintain because it’s hard to find your way around long methods and large classes. They can be extended to support new requirements only by adding more code, which makes the problem worse. Complex EJBs are also very difficult to test because they lack the subcomponents to test in isolation. But if this procedural design style has these problems, why is it so common in J2EE application?

**Why J2EE encourages developers to write procedural code**

There are a couple of reasons why J2EE developers often write procedural-style code rather than developing an object model. One reason is that the EJB specification makes it seductively easy. Although the specification does not force you to write this type of code, it lays down a path of least resistance that encourages stateless, procedural code. When implementing new behavior, you don’t have to worry about identifying classes and assigning responsibilities as you would if you were designing a real object model. Instead, you can write a new session bean method or add code to an existing one.

The second reason why J2EE developers write procedural-style code is that it is encouraged by the EJB architecture, literature, and culture, which place great emphasis on EJB components. EJB 2 components are not suitable for implementing an object model. Session beans and message-driven beans are monolithic, heavyweight classes that cannot be used to implement a fine-grained object model. Nor can they represent business objects that are stored in a database. The best way to use them in an application is to encapsulate an object model: the *Session Façade* and *Message Façade* patterns.

EJB 2 entity beans, which are intended to represent business objects, have numerous limitations that make it extremely difficult to use them to implement a persistent object model. This is why I didn’t use them in our earlier example. EJB 2 entity beans support some kinds of relationships, but not inheritance. Entity beans do not support recursive calls or “loopback” calls, which are common in an object model and occur when object A calls object B, which calls object A. We’ll discuss other limitations of entity beans in a moment. Entity beans have so many limitations that it’s amazing that developers have used them successfully. This is a fundamental problem with the preferred J2EE architecture. Each framework creates a path of least resistance for its use. It is possible to diverge from the path, but it goes against the grain of the framework and takes a great deal of effort. The path of least resistance in J2EE and EJB leads inexorably toward procedural code.
As a result, it has been difficult to do any true object-oriented development in a J2EE application. Furthermore, this procedural design style is so ingrained in the J2EE culture that it has even carried over into newer, non-EJB ways of developing J2EE applications. Some developers still view persistent objects simply as a means to get data in and out of the database and write procedural business logic. They develop what Fowler calls an “anemic domain model” [Fowler Anemic]. Just as anemic blood lacks vitality, anemic object models only superficially model the problem and consist of classes that implement little or no behavior.

**The pain of EJB development**

Another problem with EJBs is that development and testing are painfully tedious for the following reasons:

- **You must deal with annoyingly long edit-compile-debug cycles**—Because EJBs are server-side components, you must deploy them in the EJB container, which is a time-consuming operation that interrupts your train of thought. Quite often the time to redeploy a component crosses the 10-second threshold, at which point you might be tempted to do something else, like surf the Web or IM a friend. The impact on productivity is particularly frustrating when doing test-driven development, where it is desirable to run the tests frequently, every minute or two. Test-driven development and unit testing are common best practices for Java development made difficult by the infrastructure required when developing EJBs.

- **You face a lack of separation of concerns**—EJB often forces you to solve several difficult problems simultaneously—business logic design, database schema design, persistence mapping, etc.—rather than allowing you to work on one problem at a time. Not only is this mentally overwhelming but it also adds to the already long edit-compile-debug cycle. When you change a class, you might have to update the database schema before you can test your changes.

- **You must write a lot of code to implement an EJB**—You must write a home interface, a component interface, the bean class, and a deployment descriptor, which for an entity bean can be quite complex. In addition, you must write a number of boilerplate bean class methods that are never actually called but that are required by the interface the bean class implements. This code isn’t conceptually difficult, but it is busywork that you must endure.

- **You have to write data transfer objects**—A data transfer object (DTO) is a dumb data object that is returned by the EJB to its caller and contains the data the presentation tier will display to the user. It is often just a copy of the data
from one or more entity beans, which cannot be passed to the presentation tier because they are permanently attached to the database. Implementing the DTOs and the code that creates them is one of the most tedious aspects of implementing an EJB.

Developing EJBs can be a slow, mind-numbing process. While you can get used to it and find ways to occupy your time while waiting for components to deploy, it isn’t a good way to develop software. As I mentioned earlier, the nature of J2EE development with EJB precludes many of the best practices common in other types of Java development. Because the components must run in the application server in order to access the services it provides, an incremental development strategy that frequently executes the edit-compile-debug cycle is difficult. Eventually, many enterprise Java developers have become painfully aware of these limitations and have started to ask questions: Does the development I’m doing require all these services for which I’m paying such a high price? Is this the right tool for the job?

1.1.4 EJB 3 is a step in the right direction

The EJB standard isn’t frozen in amber. The designers of the specifications at Sun listen to developers and are modifying the EJB specification accordingly. The main goal of the newest EJB 3 standard is to simplify EJB development. It addresses some of the perceived problems and issues with the current specification:

- EJBs are POJOs, there is a lot less boilerplate code to write, and the code is less coupled to the application server environment.
- EJB 3 entity beans are intended to be the standard Java persistence mechanism and run in both J2EE and J2SE environments.
- EJB 3 supports the use of Java 5 annotations instead of difficult-to-write deployment descriptors to specify such things as transaction attributes, security attributes, and object/relational mapping.
- Entity beans support inheritance (finally!), making it possible to implement a true object model.
- EJB 3 also has reasonable defaults for much of the deployment information, so there is a lot less of it to write.
- EJB 3 entity beans can be used to return data to the presentation tier, which eliminates the need to write DTOs.

EJB 3 still has limitations. For example, it forces components into three categories—session beans, entity beans, and message-driven beans—even though in a typical
object model there are classes that do not fall into one of these three categories. As a result, many classes are unable to use the services provided by the EJB 3 container. Also, the June 2005 public draft of the specification still had only limited support for collection classes. In addition, there is no guarantee that the EJB 3 containers will provide fast and painless deployment of EJBs. As a result, EJB 3 still appears to be inferior to the lightweight technologies such as JDO, Hibernate, and Spring that I describe later in this chapter.

Despite its limitations, it is extremely likely that EJB 3 will be widely used for the simple reason that it is part of the J2EE standard. It is also important to remember that EJB is an appropriate implementation technology for two types of applications:

- Applications that use distributed transactions initiated by remote clients
- Applications that are heavily message-oriented and need message-driven beans

But for many other applications superior alternatives exist that are considerably easier to use. The remainder of this book focuses on those alternatives: POJOs and lightweight technologies such as Spring, Hibernate, and JDO.

1.2 Developing with POJOs

Long before the EJB 3 specification was written, some developers disillusioned with EJB started to look for alternative frameworks. POJOs are an especially compelling alternative to EJBs. A POJO is simply a Java object that does not implement any special interfaces such as those defined by the EJB framework. The name was coined by Fowler, Rebeca Parsons, and Josh MacKenzie [Fowler POJO] to give regular Java objects an exciting-sounding name. Later in this section you will see how this simple idea has some surprisingly important benefits.

However, POJOs by themselves are insufficient. In an enterprise application you need services such as transaction management, security, and persistence, which were previously provided by the EJB container. The solution is to use the increasingly popular so-called “lightweight” frameworks that replace some “heavyweight” parts of the J2EE stack. They do not completely replace the J2EE stack but can be used in combination with some parts of it to provide important enterprise services.

The four lightweight frameworks that I describe in this book are Hibernate, JDO, iBATIS, and Spring. Except for JDO, which is a specification, they are open source projects, which have helped drive the adoption of POJOs and lightweight frameworks by the community. Hibernate and JDO are persistence frameworks, which map POJOs to a relational database. They are layered on top of JDBC and significantly
Developing with POJOs

increase developer productivity. iBATIS is also layered on top of JDBC, but it maps POJOs to SQL statements and is a very convenient way to execute SQL statements. The Spring framework has a wide range of features that make it easier to use than EJB, including the equivalent of container-managed transactions for POJOs.

An important feature of these technologies is that they are nonintrusive. Unlike EJ Bs, they provide transactions and persistence without requiring the application classes to implement any special interfaces. Even when your application’s classes are transactional or persistent, they are still POJOs, which means that you continue to experience the benefits of POJOs that I describe in this chapter.

Some excellent books are available that describe these frameworks in depth: *Hibernate in Action* [Bauer 2005], *Spring in Action* [Walls 2005], *iBATIS in Action* [Begin, forthcoming], and *Java Data Objects* [Russell 2003]. You do not need to read these books to understand and benefit from this book. But to apply what you learn here you do need to read them to learn the details.

In this section I will provide an overview of how to use POJOs and lightweight frameworks to redesign the money transfer service and make it easier to develop, test, and maintain. This new design is object-oriented POJO-based instead of a procedural EJB-based. It accesses the database using a persistence framework that is layered on top of JDBC instead of using JDBC directly. The business logic is encapsulated by a POJO façade instead of a session bean, and transactions are managed by the Spring framework instead of the EJB container. The business logic returns real business objects to the presentation tier instead of DTOs. The application is assembled by passing a component’s dependencies as setter or constructor arguments instead of the component using Java Naming and Directory Interface (JNDI) lookups. Because the design is object-oriented and uses these lightweight technologies, it is much more developer-friendly than the EJB version we saw earlier.

Table 1.1 summarizes the differences between the two designs.

<table>
<thead>
<tr>
<th>Organization</th>
<th>Classic EJB approach</th>
<th>POJO approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implementation</td>
<td>Procedural-style business logic</td>
<td>Object-oriented design</td>
</tr>
<tr>
<td>Database access</td>
<td>EJB-based</td>
<td>POJOs</td>
</tr>
<tr>
<td>Returning data to the presentation tier</td>
<td>JDBC/SQL or Entity beans</td>
<td>Persistence framework</td>
</tr>
<tr>
<td></td>
<td>DTOs</td>
<td>Business objects</td>
</tr>
</tbody>
</table>
Don’t worry if you are not familiar with all of these terms. In this section, I’ll examine each difference and explain and justify the POJO approach. You will see how to develop business logic using the POJO approach. I use the money transfer application from section 1.1.2 as an example.

1.2.1 Using an object-oriented design

Rather than structuring the money transfer example around methods such as transfer() and its helper methods, the code should be structured around an object model, which is a collection of classes that typically corresponds to real-world concepts. For example, in the money transfer application, the object model consists of classes such as Account, OverdraftPolicy, and BankingTransaction. In addition, there is a TransferService that coordinates the transfer of money from one account to another. Figure 1.2 shows the design.

![Figure 1.2 An object model for the money transfer application](image-url)
Developing with POJOs

An Account maintains its balance and has an OverdraftPolicy, which determines what happens when the account is about to become overdrawn. OverdraftPolicy is an example of a Strategy pattern [Gang of Four] and there are two implementations of OverdraftPolicy: one for each type of real-world policy. Better yet, an OverdraftPolicy could encapsulate a rules engine and thereby enable the business rules for overdrafts to be changed dynamically. TransferTransaction, which is a subclass of BankingTransaction, records the transfer of money between two accounts.

Using an object-oriented design has a number of benefits. First, the design is easier to understand and maintain. Instead of consisting of one big class that does everything, it consists of a number of small classes that each have a small number of responsibilities. In addition, classes such as Account, BankingTransaction, and OverdraftPolicy closely mirror the real world, which makes their role in the design easier to understand.

Second, our object-oriented design is easier to test: each class can and should be tested independently. For example, we could write unit tests for Account and for each implementations of OverdraftPolicy. In comparison, an EJB can only be tested by calling its public methods, for example, transfer(), which is a lot more difficult. You can only test the complex functionality exposed by the public methods rather than test the simpler pieces of the design.

Finally, the object-oriented design in figure 1.2 is easier to extend because it can use well-known design patterns, such as the Strategy pattern and the Template Method pattern [Gang of Four]. Adding a new type of overdraft policy simply requires defining a new subclass of OverdraftPolicy. By contrast, extending an EJB-style procedural design usually requires changing the core code, and rewriting or chaining procedure calls together.

As you can see, our object-oriented design has some important benefits. But it is essential to know when it is not a good choice. Later in this book I describe how to decide between procedural and object-oriented approaches.

1.2.2 Using POJOs

Once you break free of the constraints imposed by the EJB 2 programming model, implementing the object model shown in figure 1.2 is easy. Java provides all of the necessary features, including fine-grained objects, relationships, inheritance, and recursion. It is straightforward to implement expressive object models like this one using POJOs and thus benefit from improved maintainability and testability. Java is an object-oriented language, so it is foolish not to use its capabilities.

As a bonus, POJOs have these other important benefits:
CHAPTER 1
Developing with POJOs: faster and easier

- Easier development—There is less cognitive load because rather than being forced to think about everything—business logic, persistence, transactions etc.—at once you can instead focus on one thing at a time. You can first design and implement the business logic and then, once that is working, you can deal with persistence and transactions.

- Faster development—You can develop and test your business logic outside of the application server and without a database. You do not have to package your code and deploy it in the application. Also, you do not have to keep the database schema constantly in sync with the object model or spend time waiting for slow-running database tests to finish. Tests can run in a few seconds and development can happen at the speed of thought—or at least as fast as you can type!

- Improved portability—You are not tied to a particular implementation technology. The cost of switching to the next generation of Java technology is minimized because you have to rewrite only a small amount of code, if any.

I was genuinely surprised by how POJOs changed how I went about development because I’d become so accustomed to the cumbersome EJB approach. As with the TiVo box I described earlier, I had to use them before I appreciated their true value. But now I couldn’t imagine reverting to the old way of working. Of course, you still need to handle persistence and transactions, which is where lightweight frameworks come in.

1.2.3 Persisting POJOs

When the time comes to persist the POJOs that implement the business logic, there are some powerful object/relational mapping frameworks to choose from. The main ones are JDO, which is a standard from Sun, and Hibernate, which is an extremely popular open source framework. In addition, the specification for EJB 3 entity beans appears to be potentially quite powerful.

Transparent persistence with JDO and Hibernate

JDO and Hibernate provide transparent persistence, which means that the classes are unaware that they are persistent. The application just needs to call the persistence framework APIs to save, query, and delete persistent objects. The persistence framework automatically generates the SQL statements that access the database using an object/relational mapping, which is defined by XML documents or Java 5 annotations. The object/relational mapping specifies how classes map to tables, fields map to columns, and relationships map to either foreign keys
or join tables. JDO and Hibernate can also run outside of the application server, which means that you can test your persistent business logic without deploying it in a server. You can, for example, simply run tests from within your integrated development environment (IDE).

**Encapsulating the calls to the persistence framework**

Even though Hibernate and JDO provide transparent persistence, some parts of an application must call the JDO and Hibernate APIs to save, query, and delete persistent objects. For example, `TransferService` must call the persistence framework to retrieve the accounts and create a `BankingTransaction`. One approach is for `TransferService` to call the persistence framework APIs directly. Unfortunately, this would couple `TransferService` directly to the persistence framework and the database, which makes development and testing more difficult.

A better approach is to encapsulate the Hibernate or JDO code behind an interface, as shown in figure 1.3. The persistence framework, which in this example is...
CHAPTER 1

Developing with POJOs: faster and easier

Hibernate, is encapsulated by the repository classes. Each repository consists of an interface and a Hibernate implementation class and is responsible for one type of object. The JDO implementation would be similar.

In this example, repositories call the Hibernate APIs to access the database. AccountRepository finds accounts and BankingTransactionRepository creates BankingTransactions. The TransferService is written in terms of the AccountRepository and BankingTransactionRepository interfaces, which decouples it from the persistence framework and the database. By the intelligent use of interfaces, you can avoid coupling your domain logic to a particular persistence framework. This will enable you to test the domain model without the database, which simplifies and accelerates testing. It also enables you to use a different persistence framework if your needs change. For example, changing this application from Hibernate to JDO or even EJB 3 is simply a matter of changing the concrete classes that access the persistence framework. It’s a generally accepted observation that loosely coupled applications are easier to maintain and test, and you will see examples of how to do this throughout this book.

1.2.4 Eliminating DTOs

Another way to improve a J2EE application is to eliminate the DTOs, also known as value objects. A DTO is a simple object consisting of only fields (i.e., no behavior) and is used to return data from the business tier to the presentation tier. An EJB application uses DTOs because EJB 2 entity beans cannot be efficiently accessed by the presentation tier. Each call to an entity bean might be a remote call and/or a separate database transaction. As a result, they must only be accessed by the session façade, which copies data from them into DTOs. The trouble with using DTOs, however, is that they and the code that creates them are extremely tedious to develop and can sometimes be a significant portion of a J2EE application. Hibernate, JDO, and EJB 3 objects do not have this limitation and can be accessed directly by the presentation tier. As a result, we eliminate many or all of the DTOs in an application.

Returning domain objects to the presentation tier

There are a couple of ways to return Hibernate, JDO, and EJB 3 objects to the presentation tier. One option is for the business tier to return objects that are still persistent. This can be simpler to implement but requires the presentation tier to manage database connections, which is sometimes neither desirable nor possible. I will describe this option in more detail in chapter 8.
Another approach, which is described in detail in chapter 7, is for the business tier to return detached objects. A detached object is a previously persistent object that is no longer connected to the database. Instead of copying values from a persistent object into a DTO, the business tier detaches the object and returns it to the persistent tier. This approach eliminates the need for DTOs while keeping all database accesses in the business tier.

Different persistence frameworks handle detached objects in different ways. In Hibernate and EJB 3, objects are automatically detached but the application must ensure that all of the objects required by the presentation tier are loaded, which can sometimes require extra calls to the persistence framework. In JDO 2.0 an application must explicitly detach the required objects by calling a JDO API.

**Using a façade to retrieve and detach domain objects**

An important design decision is determining which class will be responsible for calling the persistence framework to retrieve and detach the objects required by the presentation tier. For example, the money transfer business logic must retrieve the recent transactions and detach them along with the account objects. You could make this the responsibility of the `TransferService`, but doing so would make it more complicated and couple it to the needs of the presentation tier. Moreover, because the business tier must sometimes call the persistent framework to ensure that the domain objects can be returned to the presentation tier, making the `TransferService` call the detachment logic would mix together pure business logic with infrastructure details.

Unless the service is very simple and contains little or no business logic, a better option is to retrieve and detach the required objects in a separate class—`TransferFacadeImpl`. As figure 1.4 shows, `TransferFacadeImpl` implements the `TransferFacade` interface, which specifies the methods that can be called by the business logic’s client and plays a role similar to that of an EJB component interface. It returns a `TransferResult` that contains the domain objects.

Like the EJB we saw earlier, `TransferFacade` defines a `transfer()` method that returns a `TransferResult`. It calls `TransferService` and `TransactionRepository`, and creates `TransferResult`. As you can see, `TransferResult` is the only DTO in this example. The rest of the objects returned to the presentation tier are domain objects. Later in chapter 7, we look at a more elaborate example of a façade.

**1.2.5 Making POJOs transactional**

Let’s review what we have done so far. We replaced a procedural design with an object-oriented design, replaced entity beans with POJOs plus a persistence framework (either Hibernate or JDO), and eliminated DTOs. Because of these changes,
we have a design that is easier to understand, maintain, and extend. In addition, the edit-compile-debug cycle is extremely short. We now have an application where most of the code is sufficiently modular that you can write unit tests. We haven’t yet discussed how to eliminate the TransferService EJB. Even though it is a simple class that calls the object model classes, development slows down considerably any time we have to change it because of the deployment requirement. Let’s see what we can do about that.

Although session beans support distributed applications, the main reason they are used in many applications is because they provide container-managed transactions. The EJB container automatically starts a transaction when a business method is invoked and commits the transaction when the method returns. It rolls back the transaction if a RuntimeException is thrown. Container-managed transactions are extremely useful. They free you from writing error-prone code to manually manage transactions. Consequently, if you want to replace session beans with POJOs, you should use an equally convenient mechanism to manage transactions. This naturally takes us to the Spring framework.
Managing transactions with Spring

There are several lightweight mechanisms for making POJOs transactional. One very popular framework that provides this capability is Spring. Spring is a powerful J2EE application framework that makes it significantly easier to develop enterprise Java applications. It provides a large number of features, and I’m only going to provide a brief overview of a few of them in this chapter. For more information see Spring in Action [Walls 2005].

The Spring framework provides an extremely easy-to-use mechanism for making POJOs transactional that works in a similar way to container-managed transactions. Spring will automatically begin a transaction when a POJO method is invoked and commit the transaction when the method returns. It can also rollback a transaction if an error occurs. Spring can manage transactions using the application server’s implementation of the Java Transaction API (JTA) if the application accesses multiple resources such as a database and JMS. Alternatively, Spring can manage transactions using the persistence framework or JDBC transaction management APIs, which are simpler and easier to use because they do not require an application server.

When using the Spring framework, we can make a POJO transactional by defining it as a Spring bean, which is simply an object that is instantiated and managed by Spring. Defining a Spring bean requires only a few lines of XML. The XML is similar to a deployment descriptor and configures Spring’s lightweight container, which is a sophisticated factory for constructing objects. Each entry in the XML file defines the configuration of a Spring bean, which includes its name, its POJO implementation class, and a description of how to instantiate and initialize it. An application obtains a bean by calling the Spring bean factory with the name and expected type of the bean:

```java
BeanFactory beanFactory = ...
TransferFacade tf = (TransferFacade)
    beanFactory.getBean("TransferFacade", TransferFacade.class);
```

This code fragment calls the `BeanFactory.getBean()` method with `TransferFacade` as the name of the bean and `TransferFacade` as the expected class. The bean factory will throw an exception if a bean with that name does not exist or is of a different type.

As well as being a highly configurable way to instantiate objects, a Spring bean factory can be configured to return a proxy instead of the original object. A proxy, which is also known as an interceptor, is an object that masquerades as the original object. It executes arbitrary code before and after invoking the original
object. In an enterprise Java application, interceptors can be used for a number of purposes, including security, database connection management, and transaction management.

In this example application, we can configure the Spring bean factory to wrap TransferFacade with a proxy that manages transactions. To do that, we must define several beans, including those shown in figure 1.5. This diagram shows the TransferFacade bean, along with PlatformTransactionManager, TransactionInterceptor, and BeanNameAutoProxyCreator, the Spring classes that make TransferFacade transactional.

The BeanNameAutoProxyCreator bean wraps TransferFacade with a TransactionInterceptor, which manages transactions using the PlatformTransactionManager. The PlatformTransactionManager in this example is implemented by the HibernateTransactionManager class, which uses the Hibernate Transaction interface to begin, commit, and roll back transactions. Listing 1.1 shows an excerpt from the XML configuration file that defines these beans.

### Listing 1.1 Configuring Spring transaction management

```xml
<beans>
  <bean id="TransferFacade" class="com.example.TransferFacadeImpl"/>
  ...
</beans>

<bean id="PlatformTransactionManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  ...
</bean>

<bean id="TransactionInterceptor" class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="PlatformTransactionManager"/>
  ...
</bean>
```

**Figure 1.5** The Spring bean definitions required to make TransferFacade transactional
Let’s take a closer look at this listing:

1. This defines a bean called TransferFacade, which is implemented by the TransferFacadeImpl class.

2. This defines a bean called PlatformTransactionManager, which is implemented by the HibernateTransactionManager class that manages transactions using the Hibernate API.

3. This defines a bean called TransactionInterceptor, which is implemented by the TransactionInterceptor class that makes an object transactional. TransactionInterceptor intercepts calls to the object and calls a PlatformTransactionManager to begin, commit, and roll back transactions. It has a transactionManager property, which specifies which PlatformTransactionManager to use, and a transactionAttributeSource property, which specifies which methods to make transactional. In this example, all method calls are configured to be transactional.

4. This defines a bean called BeanNameAutoProxyCreator, which wraps TransferFacade with TransactionInterceptor. It has an interceptorNames property, which specifies the list of interceptors to apply, and a beanNames property, which specifies the beans to wrap with interceptors.

These bean definitions arrange for the bean factory to wrap TransferFacade with TransactionInterceptor. When the presentation tier invokes what it thinks is TransferFacade, TransactionInterceptor is invoked instead. The sequence of events is shown in figure 1.6.
Let’s look at the sequence of events:

1. The presentation tier calls TransferFacade but the call is routed to TransactionInterceptor.
2. TransactionInterceptor begins a transaction by calling PlatformTransactionManager, which begins a transaction using either the JTA provided by the application server or the transaction management API provided by the persistence framework.
3. TransactionInterceptor invokes the real TransferFacadeImpl.
4. The call to TransferFacadeImpl returns.
5. TransactionInterceptor commits the transaction by calling PlatformTransactionManager.
6. The call to TransactionInterceptor returns.

In step 5 TransactionInterceptor could also roll back the transaction if the TransferMoney service threw an exception. By default, TransactionInterceptor emulates EJBs and rolls back a transaction if a RuntimeException is thrown. However, you can write rollback rules that specify which exceptions should cause a transaction to be rolled back. Using rollback rules simplifies the application and decouples it from the transaction management APIs by eliminating code that programmatically rolls back transactions. This is one example of how the Spring framework is more flexible than an EJB container.

Another benefit of using Spring is that you can test your transactional POJOs without deploying them in the application server. Because code that uses JDO or Hibernate can also be tested within your IDE, you can often do a lot of development
Developing with POJOs

without ever starting up an application server. In fact, I often find that the only time I need to use one is when developing code that uses a service such as JMS that is provided by the application server. Even when working on the presentation tier I am able to use a simpler web container such as Jetty. This is yet another example of how lightweight frameworks make your life as a developer easier.

The role of AOP in the Spring framework

The technology underlying Spring’s transaction management mechanism is known as Aspect-Oriented Programming (AOP). AOP is a declarative mechanism for changing the behavior of an application without requiring any modification to the application itself. You write rules that specify new code to be executed when methods are called and, in some cases, fields are accessed or objects instantiated. In this example the BeanNameAutoProxyCreator arranged for the TransactionInterceptor to be executed whenever the TransferFacade was called without any code changes. AOP is not limited to transaction management, and in this book you will see examples of interceptors that implement security, manage database connections, and automatically retry transactions.

I’m using the Spring AOP implementation in this book for the simple reason that it provides the AOP interceptors for managing transactions, JDO, and Hibernate connections. It is important to remember that the techniques described in this book will work equally as well with other lightweight containers such as PicoContainer, and other AOP mechanisms like AspectJ [Laddad 2003]. However, as of this writing, Spring provides the best implementation of the features required by enterprise applications such as the Food to Go application, which is the example application used throughout the rest of this book.

The Spring framework is one example of a growing number of technologies that are compelling alternatives to EJBs. Using Spring AOP provides the same benefits of using EJB session beans but also allows you to use POJOs for your problem domain. An EJB container provides a large number of services, including transaction management. But is it worth compromising the design of the application to take advantage of these services—especially if you can implement them using a technology such as Spring in an à la carte fashion?

1.2.6 Configuring applications with Spring

Most applications consist of multiple components that need to access one another. A traditional J2EE application uses JNDI as the mechanism that one component uses to access another. For example, the presentation tier uses a JNDI lookup to obtain a reference to a session bean home interface. Similarly, an EJB uses JNDI to
access the resources that it needs, such as a JDBC DataSource. The trouble with JNDI is that it couples application code to the application server, which makes development and testing more difficult. The Spring framework provides POJOs with a much easier-to-use mechanism called dependency injection, which decouples application components from one another and from the application server.

Dependency injection is another powerful feature of Spring’s bean factory. Spring beans can be configured to depend on other beans, and when Spring instantiates a bean, it will pass to it any required beans, instantiating them if necessary. Two main types of dependency injection are used with Spring: constructor injection and setter injection. With constructor injection, the container passes the required objects to a component’s constructor; with setter injection, the container passes the required objects by calling setters.

Dependency injection was used earlier to wire together the Spring beans—TransactionInterceptor, PlatformTransactionManager, and BeanNameAutoProxyCreator—that provide transaction management. It can also be used to wire together application components. In the money transfer example, we can configure the TransferFacade bean to depend on TransferService and TransferService to depend on HibernateAccountRepository and HibernateBankingTransactionRepository:

```xml
<beans>

...<bean id="TransferFacade" class="TransferFacadeImpl">
  <constructor-arg ref="TransferService"/>
</bean>

<bean id="TransferService" class="TransferServiceImpl">
  <constructor-arg ref="AccountRepository"/>
  <constructor-arg ref="BankingTransactionRepository"/>
</bean>

...
</beans>
```

The first bean definition specifies that TransferFacadeImpl’s constructor take a TransferService as a parameter. The second bean definition specifies that TransferServiceImpl’s constructor be passed AccountRepository and BankingTransactionRepository. When Spring instantiates TransferFacade, it will also instantiate TransferService, HibernateAccountRepository, and HibernateBankingTransactionRepository. See the online source code, which can be downloaded from
http://www.manning.com/crichardson, for the definition of the Hibernate-
AccountRepository, HibernateBankingTransactionRepository, and Hibernate-
ObjectDetacher, along with the configuration of the Hibernate SessionFactory
and the JDBC DataSource.

Dependency injection is an extremely easy way to configure an application.
Instead of using an object containing code to look up its dependencies, they are
automatically passed in by the bean factory. It doesn’t have to call any application
server APIs. In chapter 7, I’ll show how dependency injection is a useful way of decou-
pling components from one another and the application server environment.

1.2.7 Deploying a POJO application

As I mentioned earlier, one of the great things about POJOs and lightweight
frameworks is that you can do a lot of development without going near an applica-
tion server. Eventually, however, you do need to deploy the application. An applica-
tion that uses Spring for transaction management and Hibernate or JDO for
persistence can often be deployed in a simple web container-only server such as
Jetty, Tomcat, or WebLogic Express, as shown in figure 1.7.

The application is simply packaged as a web archive file (WAR) and deployed in
the server’s web container. It would use either a JDBC connection pool provided by
the application server or an open source implementation such as DBCP [DBCP]. If
the application needed to be clustered for scalability and reliability, then it would
use the clustering feature of the web container.

An application only needs to be deployed in a full-blown application server
(e.g., WebLogic Server or JBoss) if it requires those parts of the J2EE stack such as
JTA or JMS that are not provided by the web container or some third-party soft-
ware. You might also want to deploy your application in a particular server if you
wanted to use a vendor-specific feature. For example, some application servers
have sophisticated security and management capabilities. Only some applications
have these requirements, and if you break the dependency on EJBs by using POJOs
and lightweight technologies, you can often deploy an application in a simpler
and, in some cases, cheaper server.

1.2.8 POJO design summary

Let’s review the design of the money transfer service that uses a POJO object model,
Spring for transaction management and dependency injection, and Hibernate for
persistence. The design, which is shown in figure 1.8, has more components than
the EJB-based design described earlier in section 1.1.2. However, this more modular
design is easier to understand, test, and extend than the original version. Each class
has a small number of well-defined and easy-to-understand responsibilities. The use
of interfaces for the repositories simplifies testing by allowing the real implemen-
tations of the repositories to be replaced with stubs. OverdraftPolicy enables the
design to be extended to support new types of overdrafts without requiring modi-
fications to existing code.

The core of the business logic consists of object model described earlier in sec-
tion 1.2.1 and includes classes such as Account and OverdraftPolicy. The
AccountRepository and BankingTransactionRepository classes encapsulate the
Hibernate APIs. AccountRepository defines a method for retrieving accounts, and
BankingTransactionRepository provides a method for creating transactions.
TransferService is a simple service that looks up the accounts by calling Account-
Repository and calls credit() and debit() on them. It also creates a Banking-
Transaction to record the transfer.

TransferFacade is a simple wrapper around TransferService that retrieves the
data required by the presentation tier. This functionality could be implemented
by TransferService, but implementing it in a separate class keeps Transfer-
Service focused on transferring money and away from the presentation tier and
the details of detaching objects. TransferFacade is wrapped with a Spring Trans-
actionInterceptor that manages transactions.
I have omitted some of the details, but I hope you can see what you can accomplish with POJOs and lightweight frameworks such as Spring. By using Spring, we have the functionality we formerly needed from the EJB container. By using POJOs, we also have a design and structure of code that is impossible if we use the heavyweight J2EE application server and all its services. Using the lighter weight tools allows us to improve the structure, maintainability, and testability of our code.
1.3 **Summary**

Building enterprise Java applications with a simple technology—POJOs—in conjunction with lightweight frameworks such as Spring, Hibernate, and JDO has some surprising benefits. You have the freedom to develop expressive object models rather than being forced down a procedural path. You get the benefits of EJB, such as declarative transaction management and security, but in a much more developer-friendly form. You can work on your core business logic without being distracted by enterprise “issues” such as transaction management and persistence. You can develop and test your code without being slowed down by deployment. As a bonus, because the lightweight frameworks are noninvasive you can readily take advantage of new and improved ones that will inevitably be developed.

In the next chapter we look at the design decisions you need to make when using them to develop an enterprise application.
Here 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 elsewhere. He has a computer science degree from the University of Cambridge in England and lives in Oakland, CA.