

Excerpt
POJOs in Action
by
Chris Richardson
from Manning Publication

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.

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`

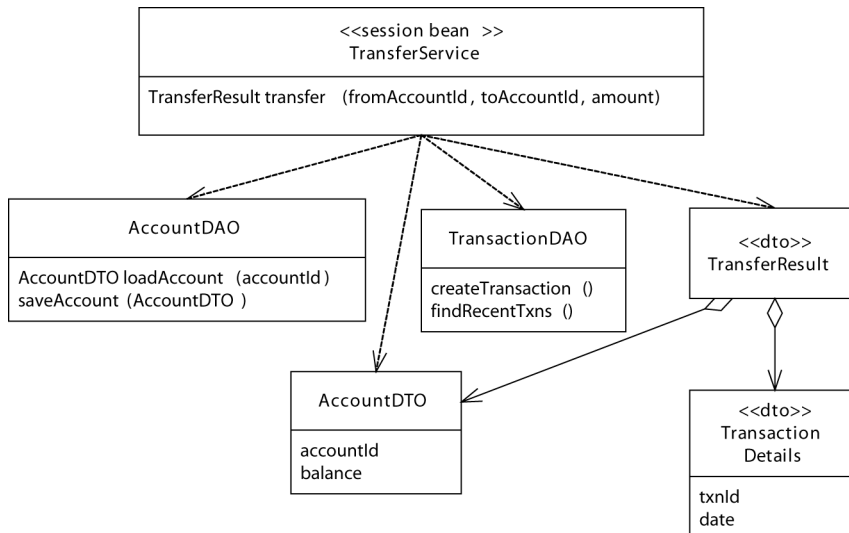


Figure 1.1 The money transfer service implemented using a typical EJB-based design

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, heavy-weight 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.