Untangling

CHRIS RICHARDSON, CONSULTANT

A new breed of framework helps eliminate crosscutting concerns.

Separation of concerns is one of the oldest concepts in computer science. The term was coined by Dijkstra in 1974.¹ It is important because it simplifies software, making it easier to develop and maintain. Separation of concerns is commonly achieved by decomposing an application into components. There are, however, *crosscutting* concerns, which span (or cut across) multiple components. These kinds of concerns cannot be handled by traditional forms of modularization and can make the application more complex and difficult to maintain.

Examples of crosscutting concerns in enterprise Java applications include transaction management, security, persistence, and application assembly. Scattering the code that handles those concerns across multiple components is undesirable, however. This is because doing so would make each component more complex. Also, duplicating code in multiple modules can cause maintenance problems. Consequently, there has been a flurry of activity in recent years to develop frameworks and new forms of modularity that try to untangle these crosscutting concerns from the application's business logic.

In this article we look at the evolution of enterprise Java frameworks that tackle crosscutting concerns. We address how dissatisfaction with the first-generation frameworks, which were based on the EJB (Enterprise JavaBeans) programming model, prompted the development of dramatically better frameworks. These newer-generation frameworks are based on the POJO (Plain Old Java Object) programming model. Even though the focus of this article is Java, it contains many useful lessons for developers of frameworks and applications written in other languages.

Enterprise Java

Untangling Enterprise Java

DISILLUSIONMENT WITH ENTERPRISE JAVABEANS

EJB² is the standard Java framework for writing component-based, distributed business applications. It is a framework for building business logic components, known as *enterprise Java beans* (or EJBs, for short), and handles some of the most time-consuming aspects of writing enterprise applications. EJB provides services such as transaction management, authorization, persistence, and application assembly. In this section, we look at what worked and what didn't and how that led to the development of the next generation of the enterprise Java framework.

SEPARATION OF CONCERNS WITH THE EJB FRAMEWORK

Before the availability of EJB, Java developers were responsible for writing the transaction management, authorization, and persistence code themselves. In addition to being error-prone and time-consuming to write, this code was often intertwined with the business logic. In comparison, by using the EJB framework, making application components transactional, secure, and persistent requires only the declarative specification of these characteristics using separate metadata in XML configuration files, known as deployment descriptors.

As figure 1 shows, the EJB framework, which is also known as the EJB container, reads the deployment descriptor and implements the required behavior, often by intercepting calls to a component and executing extra code that handles the crosscutting concern. For example, EJB applications typically use declarative transactions, eliminating the need to write transaction management code entangled with each component's business logic. The EJB framework intercepts calls to an EJB and begins, commits, and rolls back transactions.

Similarly, the EJB framework simplifies the development of secure

applications by making it possible to secure components declaratively. An EJB's deployment descriptor specifies which users can access a component or a component method. The EJB framework intercepts calls to the EJB and verifies that the caller is authorized to access the component.

EJB also supports persistent components, called *entity beans*. The deployment descriptor for an entity bean describes how its attributes map to the database schema. The deployment descriptor maps simple values to columns and maps relationships to foreign keys and join tables. The EJB framework uses this mapping to generate SQL statements to query and update the database.

EJB partially succeeded in separating crosscutting concerns from the business logic. Responsibility for handling those concerns moved from the business logic components to the EJB framework. As it turned out, however, the first two versions of the framework—EJB 1.0 and EJB 2.0—did this in a fundamentally flawed way.

EJB ISSUES

The major flaw of the early specifications of EJB, which pervade the enterprise Java community today, is that they place severe demands on classes implementing these components. EJB 1.0 and 2.0 components must implement interfaces defined by the EJB framework and must

often call the EJB framework APIs. This tightly couples the components to the EJB framework, causing the following problems.

The first problem is that the separation of concerns is an illusion. Even though concerns such as security, transaction management, and persistence are separate from the code and configured in a deployment description, you cannot ignore them when developing business logic. For example, a persistent EJB component cannot be easily tested without the database. When testing the component's business logic, you are forced to think about database schema design. EJB prevents you from working on one concern at a time.

The tight coupling of the business logic to the framework also causes annoyingly long edit-compile-debug cycles. Deploying EJB components in the EJB container is a time-consuming operation that often 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 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 EJB components.

This flaw with the EJB specification is made even worse because business logic is not portable between framework versions. Despite being a standard, EJB has rapidly evolved in incompatible ways in its brief history. There were significant and incompatible changes between EJB 1.0 and EJB 2.0, and between EJB 2.0 and EJB 3.0. To take full advantage of the new and improved features of each release of the specification, you must rewrite your components. This can be

quite challenging if you are responsible for maintaining an EJB application with a lifetime of more than a couple of years.

These problems motivated the enterprise Java community to find better ways of untangling crosscutting concerns. Much of the innovation came from the developers of open source Java frameworks such as Spring and Hibernate that supported a radically different programming model based on POJOs.

PROGRAMMING WITH POJOs

Today, the consensus in the enterprise Java community is to build business logic components using POJOs. This approach is simpler yet more powerful than the old-style EJB approach. A POJO is a Java object that does not implement any special interfaces such as those defined by the EJB framework or call any framework APIs. The name was coined by Martin Fowler and others³ to give regular Java objects an exciting-sounding name and encourage developers to use them. This simple idea has some surprisingly important benefits, including significantly better separation of concerns.

How Noninvasive Frameworks Handle Transactions, Security, and Persistence

Untangling Enterprise Java

NONINVASIVE FRAMEWORKS

POJOs by themselves are, of course, insufficient. You still need the services that were previously provided by the EJB framework such as transactions and security. The solution is to use the so-called “noninvasive” frameworks, which provide those services for POJOs. Popular examples of such frameworks include Spring,⁴ which provides declarative transaction management; Hibernate⁵ and JDO (Java Data Objects),⁶ which provide persistence; and Acegi security,⁷ which is an extension to Spring that provides authentication and authorization for POJOs. In addition, the latest version of the EJB specification—EJB 3.0—is POJO-based.

As with EJB 2.0, using these frameworks involves writing metadata that describes how your components should behave. Unlike EJB 2.0, however, these frameworks provide transaction management, security, persistence, and application assembly without requiring the application classes that need those services to implement framework interfaces or call framework APIs. They impose only what are, in practice, at most minimal constraints on the application classes. These frameworks are typically called directly only by the small number of application components that create, find, and delete persistent objects.

As figure 2 shows, your application consists of POJOs that are independent of the frameworks that make them transactional, secure, or persistent.

The transaction management and security frameworks intercept calls to the POJO components, check that the caller is authorized, and manage transactions. The persistence framework is responsible for storing the state of persistence components in the database.

Another important difference between these frameworks and EJB 2.0 is that the XML metadata is typically more user-friendly and more concise. In addition, some frameworks allow you to write metadata in the form of Java 5 annotations instead of XML. An annotation is a Java language construct that provides declarative information about a program element. Annotations are growing in popularity primarily because they are even more succinct than XML by virtue of being embedded within the source code next to the class, field, or method they describe. They also avoid the problem of fragile linkages between the source code and the XML metadata.

Annotations can be, however, a double-edged sword. Because annotations are part of the source code, your

application can end up being tightly coupled to the framework. Later, I will show examples of the metadata supported by these frameworks.

BENEFITS OF POJOs AND NONINVASIVE FRAMEWORKS

The concept of POJOs is extremely simple, but using them can dramatically improve development. POJOs and noninvasive frameworks have the following benefits:

- **Improved separation of concerns.** Instead of being forced to think about everything at once—business logic, persistence, transactions, etc.—you can focus on one thing at a time. You can first design and implement the business logic and then, once that is working, deal with persistence and transactions. This reduces the cognitive load and makes development significantly easier.
- **Faster development.** You can test your business logic without worrying about persistence or transaction management. You do not have to deploy your components to test them. Nor do you 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 framework and the cost of switching to the next generation of Java framework is minimized. Instead of making widespread changes to the code, you have only to change the metadata and rewrite a small amount of code. Sometimes, you can even upgrade to a newer version of a framework without any code or metadata changes.

One thing to keep in mind is that the key ideas behind POJOs and noninvasive frameworks also apply to other programming languages. Business logic written in

languages other than Java can benefit from being independent of the infrastructure frameworks that provide the necessary services. For example, some developers in the .NET community talk about Plain Old .NET Objects (PONOs) and use the .NET version of Hibernate called NHibernate. The Spring framework also comes in a .NET version.

EJB 3.0 IS A STEP IN THE RIGHT DIRECTION

Open source developers have been responsible for most of the recent innovation in the enterprise Java community. The EJB standard isn't frozen in amber, however. The designers of the specifications at Sun listen to developers and are modifying the EJB specification accordingly.

The new EJB 3.0 standard embraces the POJO programming model. EJB 3.0 components no longer implement EJB interfaces and rarely need to call EJB APIs. As a result, EJB components are less tightly coupled to the EJB 3.0 framework and development is significantly easier than with EJB 2.0. Open source frameworks such as Spring and Hibernate, however, have a richer set of features.

Now that we have looked at the basic POJO programming concepts, it's time to look at some of the details. The rest of this article shows how the noninvasive frameworks handle three important concerns: transaction management, persistence, and application assembly.

TRANSACTION MANAGEMENT WITH POJOs

Transaction management is an important crosscutting concern in enterprise applications. Transactions are essential for ensuring the integrity of data stored in enterprise information systems. The classic example is transferring money from one bank account to another. The credit and debit must be executed atomically within a transaction to prevent money from disappearing.

Each business logic component could manage transactions programmatically by calling transaction management APIs to begin, commit, or roll back a transaction. It is generally far better, however, to separate transaction management concerns from the business logic by using declarative transaction management. This approach is less error-prone, simplifies the code considerably, and makes it easier to maintain.

A POJO application can use either the Spring framework or EJB 3.0 for transaction management. When using one of these frameworks, you configure the transactional behavior by writing metadata in the form of either XML or Java 5 annotations. The framework will then automatically execute the method within a transaction.

Consider the money transfer example. You could

implement this behavior using code that looks something like this:

```
public interface MoneyTransferService {

    BankingTransaction transfer(String fromAccountId,
        String toAccountId, double amount);

}

public class MoneyTransferServiceImpl implements
    MoneyTransferService {
    ...
}
```

The `MoneyTransferService` interface defines a `transfer()` method, which is implemented by the `MoneyTransferServiceImpl` class. The `transfer()` method is responsible for performing the money transfer and needs to be executed within a transaction. In figure 3, listing 1 shows how to

A Transaction in Spring and EJB 3.0

Listing 1: Spring XML example

```
<aop:config>
  <aop:advisor
    pointcut="execution(*
    *.MoneyTransferService.*(..))"
    advice-ref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="*" />
  </tx:attributes>
</tx:advice>
```

Listing 2: EJB 3.0 annotation example

```
@Local
public interface MoneyTransferService {
    ...
}

@Stateless
public class
MoneyTransferServiceImpl
implements
MoneyTransferService {
    ...
}
```

FIG 3

Untangling Enterprise Java

make this method transactional using the Spring framework, and listing 2 shows how to do the same thing using EJB 3.0.

The Spring version uses XML metadata to specify that each method defined by `MoneyTransferService` executes within a transaction. The `<aop:advisor>` element tells Spring how to intercept method calls and handle crosscutting concerns. This element has a `pointcut` attribute, which specifies when the transaction management logic should run, and an `advice-ref` attribute, which specifies what to do at those points. The `advice-ref` attribute references `"txAdvice"`, which is defined by the `<tx:advice>` element. This element configures transaction management, and in this example the `"**"` wildcard specifies that all methods should be transactional.

Under the covers, the Spring framework implements transaction management using a general-purpose AOP (aspect-oriented programming) mechanism.⁸ AOP enables the modular implementation of crosscutting concerns, which impact many parts of the application, without scattering the code related to that concern through other modules. In this particular case, the XML metadata specifies when to manage transactions, but Spring applications typically use AOP to handle a variety of other crosscutting concerns, including security, logging, and caching.

The EJB 3.0 example uses the `@Local` and `@Stateless` annotations to make `MoneyTransferService` transactional. By default, each of the methods defined by `MoneyTransferService` will be transactional. An EJB 3.0 application can use additional annotations to specify the transactional attributes of individual methods.

I'm glossing over the details but the key thing to notice is that although the two sets of metadata in the two examples are very different, they have one common feature: There are no calls to any transaction management APIs. It's not shown in this example, but you can even write metadata that specifies which exceptions should cause the transaction to be rolled back. Transactions are handled entirely by the framework, which reads and processes the metadata.

PERSISTING POJOs

Database access is another key crosscutting concern since enterprise Java applications must almost always access a database. For example, the money transfer service must read and update the account database table. The busi-

ness logic could access the database by simply executing SQL statements. This approach can be time-consuming and error-prone, however. It can also be difficult to write SQL that is portable across databases. Because of these drawbacks, a better approach for many applications is to separate persistence from the business logic by using transparently persistent objects.

When using persistent objects, the developer writes metadata that specifies how the object model maps to the database. They describe how classes map to tables, simple values map to columns, and relationships between objects map to either foreign keys or join tables. The persistence framework (a.k.a. object/relational mapping framework) uses the mapping metadata to generate the SQL statements to load and store the persistent objects.

When used appropriately, this approach can significantly reduce the amount of database access code that needs to be written. The application manipulates objects, and all database accesses are done behind the scenes by the persistence framework. The persistence framework tracks the changes made by the application to the objects and automatically writes any modified objects back to the database.

For example, a bank account could be represented by the following class:

```
public class Account {

    private double balance;

    private String accountId;

    public Account(String accountId,
        double initialBalance) {
        this.accountId = accountId;
        this.balance = initialBalance;
    }

    public double getBalance() { return balance; }

    public String getAccountId() { return accountId; }

    public void debit(double amount) { balance -= amount; }

    public void credit(double amount) { balance += amount; }
}
```

This class has a balance and account ID field; and various methods include `debit()`, which debits the account, `credit()`, which credits the account, and `getBalance()`, which returns the current balance.

One way to persist an account class is to map the class to an `ACCOUNT` table and to map each field to a column of that table. In figure 4, listing 1 shows how this is done using EJB 3.0, and listing 2 shows how to persist the class using Hibernate 3.

The EJB 3.0 example uses annotations to define how the class and its fields map to the `ACCOUNT` table and how Hibernate 3 uses XML. Both of these frameworks use this metadata to generate SQL to load, store, and delete instances of this class. The `Account` class is unaware that it is persistent.

The only parts of the application that are aware of the persistence frameworks are those components that call the persistence framework to create, find, and delete persistent objects. Fortunately, these components typically make up only a small part of the application. Moreover, they are encapsulated using interfaces. For example, the banking example has an `AccountDAO` that defines methods for creating and finding accounts. Here is its interface:

```
public interface AccountDAO {  
  
    Account findAccount(String accountId);  
  
    Account createAccount(String accountId,  
                           double initialBalance);  
}
```

This interface hides the persistence framework, which decouples those components that use `AccountDAO` from the persistence framework.

ASSEMBLING AN APPLICATION

Application assembly is another crosscutting concern. Real-world applications are almost always composed of multiple interdependent components. At runtime, a component must be able to obtain references to other components. For example, in a banking application a component that needs to transfer money between accounts would need to have a reference to `MoneyTransferService`, mentioned earlier. One common approach is to use the Service Locator pattern,⁹ which is typically provided by the infrastructure framework. For example, enterprise Java applications can use JNDI (Java Naming and Directory Interface),¹⁰ which enables a component to look up an interface to another component by name. The

Service Locator pattern promotes loose coupling between components, but it has the drawback of making each and every component responsible for looking up its own dependencies. This also couples each component to the infrastructure framework.

For many applications, a better approach is to use dependency injection,¹¹ which separates the lookup of dependencies from the application components. With this approach, the dependencies of a component are specified declaratively using metadata. When instantiating a component, the framework looks up the component's dependencies (recursively instantiating them if necessary) and passes them to the component as either constructor parameters or setter method parameters. This eliminates any dependency lookup code from the application components.

The Spring framework uses XML metadata to specify how to inject the dependencies into a component. Similarly, when using EJB 3.0, you annotate an EJB's fields or setters to specify the dependency to be passed in. For

Persistence in EJB 3.0 and Hibernate 3

Listing 1: EJB 3.0 persistence

```
@Entity  
@Table(name="ACCOUNT")  
class Account {  
    ...  
    @Id  
    @Column(name="ACCOUNT_ID")  
    private int id;  
  
    @Column(name="BALANCE")  
    private double balance;  
    ...  
}
```

Listing 2: Hibernate 3 persistence

```
<class name="Account"  
    table="ACCOUNT"  
    <id name="id"  
        column="ACCOUNT_ID">  
    <generator  
        class="native" />  
    </id>  
  
    <property name="balance"  
        column="BALANCE"/>  
    ...  
</class>
```

FIG 4

Untangling Enterprise Java

example, here is an EJB 3.0 service that accesses *MoneyTransferService*:

```
class SomeServiceImpl {
    @EJB
    private MoneyTransferService transferService;
    ...
}
```

The `@EJB` annotation tells the EJB container to initialize the `transferService` field with a reference to *MoneyTransferService*.

Using dependency injection has two main benefits. It simplifies the application components by eliminating the lookup logic. A component simply uses the dependencies that are passed to it without knowing how they were obtained. Also, because the components no longer call the Service Locator, dependency injection helps decouple components from the infrastructure framework. Dependency injection is a key enabling mechanism for the POJO programming model.

SUMMARY

The enterprise Java community has developed a variety of frameworks for separating the business logic from crosscutting concerns such as transaction management, security, persistence, and application assembly. Early attempts were only partially successful. The EJB 2.0 standard was excessively complex and failed to provide true separation of concerns. Today, however, enterprise Java component technologies have improved dramatically. The EJB 2.0 standard has been replaced by simpler, yet more powerful frameworks that support the POJO programming model.

POJOs are objects that do not implement special APIs or call infrastructure frameworks. Crosscutting concerns are handled by noninvasive frameworks such as Spring, Hibernate, and EJB 3.0. These frameworks provide services without requiring the POJOs to implement particular interfaces or call the framework. For example, a POJO can be transactional without calling transaction management APIs, and a POJO can be persistent without calling the persistence framework APIs.

Using POJOs with noninvasive frameworks has many important benefits. Development is easier because you can focus on one concern at a time. Development is faster because you can test components without the infrastruc-

ture framework or the database. Using POJOs also enables your application to take advantage of new and improved enterprise Java frameworks. You can really improve your application by untangling crosscutting concerns. Q

ACKNOWLEDGMENTS

I would like to thank the anonymous reviewers and the following individuals for their helpful comments on drafts of this article: Azad Bolour, Jonas Bonér, Adrian Colyer, and David Vydra.

REFERENCES

1. Dijkstra, E.W. 1982. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, 60-66. Springer-Verlag.
2. JSR 153: Enterprise JavaBeans 2.1; <http://www.jcp.org/en/jsr/detail?id=153>; JSR 220: Enterprise JavaBeans 3.0; <http://www.jcp.org/en/jsr/detail?id=220>.
3. Fowler, M. <http://www.martinfowler.com/bliki/POJO.html>.
4. Spring framework; <http://www.springframework.org>.
5. Hibernate Object/Relational Mapping framework; <http://www.hibernate.org/>.
6. JSR 243: Java Data Objects 2.0—An Extension to the JDO Specification.
7. Acegi Security System for Spring; <http://acegisecurity.sourceforge.net>.
8. Laddad, R. 2003. *AspectJ in Action*. Greenwich, CT: Manning.
9. Fowler, M. <http://www.martinfowler.com/articles/injection.html>.
10. JNDI; <http://java.sun.com/products/jndi/>.
11. See Reference 9.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

CHRIS RICHARDSON (cer@acm.org) is a developer and architect with more than 20 years of experience. His consulting company specializes in helping enterprise Java developers become more productive and successful. He 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, California. Web site and blog: www.chrisrichardson.net.

© 2006 ACM 1542-7730/06/0600 \$5.00