Enterprise AOP with Spring Applications

AspectJ IN ACTION

SECOND EDITION

SAMPLE CHAPTER

Ramnivas Laddad

FOREWORD BY ROD JOHNSON

AspectJ in Action
Second Edition

by Ramnivas Laddad

Chapter 1

Copyright 2010 Manning Publications

brief contents

PART 1 UNDERSTANDING AOP AND ASPECTJ 1

1 ■ Discovering AOP 3
2 ■ Introducing AspectJ 27
3 ■ Understanding the join point model 51
4 ■ Modifying behavior with dynamic crosscutting 87
5 ■ Modifying structure with static crosscutting 116
6 ■ Aspects: putting it all together 136
7 ■ Diving into the @AspectJ syntax 168
8 ■ AspectJ weaving models 199
9 ■ Integration with Spring 217

PART 2 APPLICATIONS OF ASPECTJ WITH SPRING 249

10 ■ Monitoring techniques 251
11 ■ Policy enforcement: keeping your design intact 288
12 ■ Learning design patterns 319
13 ■ Implementing concurrency control 344
14 ■ Managing transactions 373

15 ■ Securing applications 404
16 ■ Improving domain logic 431
17 ■ Taking the next step 459

appendix A ■ Setting up the example 469
appendix B ■ Using Ant with AspectJ 486
appendix C ■ Using Maven with AspectJ 491

Part 1

Understanding AOP and AspectJ

Part 1 of this book introduces aspect-oriented programming (AOP), the AspectJ language, and how it integrates with Spring. We'll discuss the need for a new programming methodology and the way this methodology is realized in AspectJ. Because AOP is a new methodology, we'll devote the first chapter to introducing it: why it is needed, and what its core concepts are. Chapter 2 shows the overall flavor of the AspectJ language through examples. The next four chapters will delve deeper into the AspectJ syntax. Together, these chapters should give you enough information to start writing simple code and see the benefits that AspectJ offers. Chapters 7 and 8 will explore the alternative syntax and weaving models. Given the mutual importance of Spring and AspectJ, this part of the book ends by devoting chapter 9 to Spring-AspectJ integration. Along the way, we'll examine many examples to reinforce the concepts learned.

You'll find the material in part 1 useful as a reference while reading the rest of the book. If you're new to AOP and AspectJ, we strongly recommend that you read this part first.

1 Discovering AOP

This chapter covers

- Understanding crosscutting concerns
- Modularizing crosscutting concerns using AOP
- Understanding AOP languages

Reflect back on your last project, and compare it with a project you worked on a few years back. What's the difference? One word: *complexity*. Today's software systems are complex, and all indications point to even faster growth in software complexity in the coming years. What can a software developer do to manage complexity?

If complexity is the problem, modularization is the solution. By breaking the problem into more manageable pieces, you have a better shot at implementing each piece. When you're faced with complex software requirements, you're likely to break those into multiple parts such as business functionality, data access, and presentation logic. We call each of these functionalities *concerns* of the system. In a banking system, you may be concerned with customer management, account management, and loan management. You may also have an implementation of data access and the web layer. We call these *core concerns* because they form the core functionality of the system. Other concerns, such as security, logging, resource pooling, caching, performance monitoring, concurrency control, and transaction

management, cut across—or *crosscut*—many other modules. We call these functionalities *crosscutting concerns*.

For core concerns, object-oriented programming (OOP), the dominant methodology employed today, does a good job. You can immediately see a class such as `LoanManagementService` implementing business logic and `AccountRepository` implementing data access. But what about crosscutting concerns? Wouldn't it be nice if you could implement a module that you identify as `Security`, `Auditing`, or `PerformanceMonitor`? You can't do that with OOP alone. Instead, OOP forces you to fuse the implementation of these functionalities in many modules. This is where aspect-oriented programming (AOP) helps.

AOP is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization—an *aspect*. Each aspect focuses on a specific crosscutting functionality. The core classes are no longer burdened with crosscutting concerns. An *aspect weaver* composes the final system by combining the core classes and crosscutting aspects through a process called *weaving*. Thus, AOP helps to create applications that are easier to design, implement, and maintain.

In this chapter, we'll examine the fundamentals of AOP, the problems it addresses, and why *you* need to know about it. In the rest of the book, we'll examine AspectJ, which is a specific implementation of AOP. Let's start by discussing how you manage various concerns without AOP, which will help you understand why you need AOP.

1.1 Life without AOP

How do you implement crosscutting concerns using OOP alone? Typically, you add the code needed for each crosscutting concern in each module, as shown in figure 1.1.

This figure shows how different modules in a system implement both core concerns and crosscutting concerns. Let's illustrate the same idea through a code snippet. Consider the skeleton implementation of a representative class that encapsulates some business logic in a conventional way, shown in listing 1.1. A system consists of many such classes.

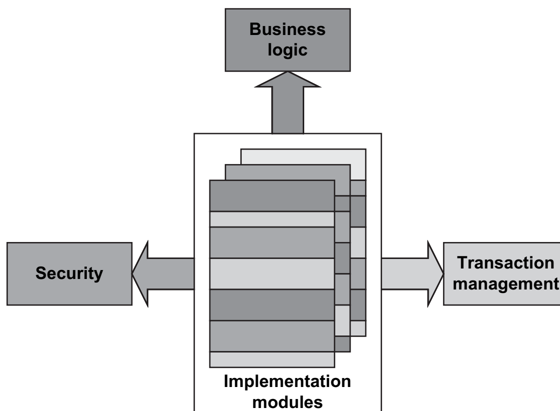

Figure 1.1 Viewing a system as a composition of multiple concerns. Each implementation module deals with some element from each of the concerns the system needs to address.

Listing 1.1 Business logic implementation along with crosscutting concerns

```
public class SomeBusinessClass extends OtherBusinessClass {
    ... Core data members
    ... Log stream
    ... Concurrency control lock

    ... Override methods in the base class

    public void someOperation1(<operation parameters>) {
        ... Ensure authorization

        ... Lock the object to ensure thread-safety

        ... Start transaction

        ... Log the start of operation

        ... Perform the core operation

        ... Log the completion of operation

        ... Commit or rollback transaction

        ... Unlock the object
    }

    ... More operations similar to above addressing multiple concerns

}
```


Although the details will vary, the listing shows a common problem many developers face: a conceptual separation exists between multiple concerns at design time, but implementation tangles them together. Such an implementation also breaks the Single Responsibility Principle (SRP)[1] by making the class responsible for implementing core and crosscutting concerns. If you need to change the invocation of the code related to crosscutting concerns, you must change each class that includes such an invocation. Doing so breaks the Open/Close principle[2]—open for extension, but closed for modifications. The overall consequence is a higher cost of implementing features and fixing bugs.

With conventional implementations, core and crosscutting concerns are *tangled* in each module. Furthermore, each crosscutting concern is *scattered* in many modules. The presence of code tangling and code scattering is a tell-tale sign of the conventional implementation of crosscutting concerns.[3] Let's examine them in detail.

1.1.1 Code tangling

Code tangling is caused when a module is implemented to handle multiple concerns simultaneously. Developers often consider concerns such as business logic, performance, synchronization, logging, security, and so forth when implementing a module.

---

[1] See http://www.objectmentor.com/resources/articles/srp.pdf for more details.

[2] See http://www.objectmentor.com/resources/articles/ocp.pdf for more details.

[3] Note that code tangling and scattering may also stem from poor design and implementation (such as copied/pasted code). Obviously, you can fix such problems within the bounds of OOP. But in OOP the problem of crosscutting concerns is present even in well-designed systems.

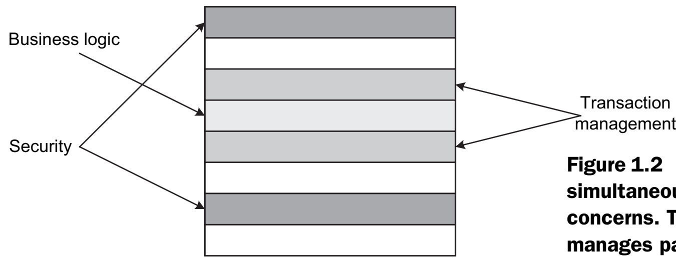

Figure 1.2 Code tangling caused by multiple simultaneous implementations of various concerns. This figure shows how one module manages parts of multiple concerns.

This leads to the simultaneous presence of elements from each concern's implementation and results in code tangling. Figure 1.2 illustrates code tangling in a module.

Another way to look at code tangling is to use the notion of a multidimensional concern space. Imagine that you're projecting the application requirements onto a multidimensional concern space, with each concern forming a dimension. Here, all the concerns are mutually independent and therefore can evolve without affecting the rest. For example, changing the security requirement from one kind of authorization scheme to another shouldn't affect the business logic. But as you see in figure 1.3, a multidimensional concern space collapses into a one-dimensional implementation space.

Because the implementation space is one-dimensional, its focus is usually the implementation of the core concern that takes the role of the dominant dimension; other concerns then tangle the core concern. Although you may naturally separate the individual requirements into mutually independent concerns during the design phase, OOP alone doesn't let you retain the separation in the implementation phase.

We've looked at the first symptom of crosscutting concerns when implemented using traditional techniques; now, let's move on to the next.

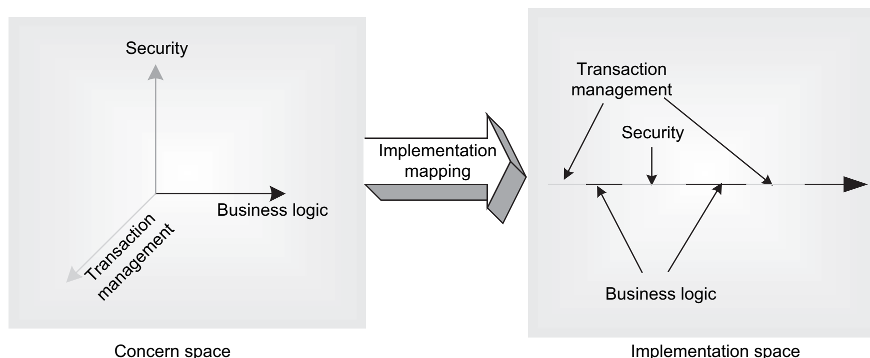

Figure 1.3 Mapping the *n*-dimensional concern space using a one-dimensional language. The orthogonality of concerns in the concern space is lost when it's mapped to the one-dimensional implementation space.

1.1.2 Code scattering

Code scattering is caused when a single functionality is implemented in multiple modules. Because crosscutting concerns, by definition, are spread over many modules, related implementations are also scattered over all those modules. For example, in a system using a database, performance concerns may affect all the modules accessing the database.

Figure 1.4 shows how a banking system implements security using conventional techniques. Even when using a well-designed security module that offers an abstract API and hides the details, each client—the accounting module, the ATM module, and the database module—still needs the code to invoke the security API to check permission. The code for checking permission is scattered across multiple modules, and there is no single place to identify the concern. The overall effect is an undesired tangling between the modules to be secured and the security module.

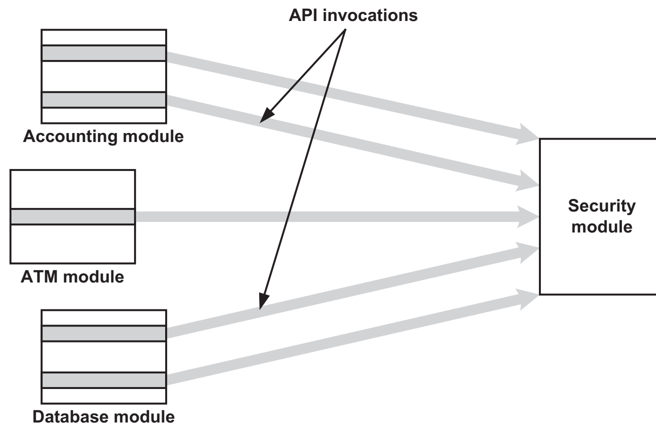

Figure 1.4 Implementation of a security concern using conventional techniques. The security module provides the API for authentication and authorization. But the client modules—accounting, ATM, and database—must embed the code to invoke the API to, say, check permission.

Code tangling and code scattering together impact software design and development in many ways: poor traceability, lower productivity, lower code reuse, poor quality, and difficult evolution. All of these problems lead us to search for better approaches to architecture, design, and implementation. Aspect-oriented programming is one viable solution. In the next section, we'll introduce you to AOP. Later in this chapter, we'll examine alternatives to AOP as well.

1.2 Modularizing with AOP

In OOP, the core concerns can be loosely coupled through interfaces, but there is no easy way to do the same for crosscutting concerns. This is because a concern is implemented in two parts: the server-side piece and the client-side piece. OOP modularizes the server part quite well in classes and interfaces. But when the concern is of a crosscutting nature, the client part (consisting of the requests to the server) is spread over all the clients.

NOTE We use the terms *server* and *client* here in the classic OOP sense to mean the objects that are providing a certain set of services and the objects using those services. Don't confuse them with networking servers and clients.

As an example, let's take another look at the typical implementation of a crosscutting concern in OOP, shown in figure 1.4. The security module provides its services through an interface. The use of an interface loosens the coupling between the clients and the implementations of the interface. Clients that use the security services through the interface are oblivious to the exact implementation they're using; any changes to the implementation don't require changes to the clients themselves. Likewise, replacing one security implementation with another is just a matter of instantiating the right kind of implementation. The result is that you can replace one security implementation with another with little or no change to the individual client modules. But this arrangement still requires that each client have the embedded code to call the API. Such calls must be included in all the modules requiring security and are tangled with their core logic.

Using AOP, none of the core modules contain calls to the security API. Figure 1.5 shows the AOP implementation of the same security functionality shown in figure 1.4. The security concern—implementation and invocations—now resides entirely inside the security module and the security aspect. For now, don't worry about the way in which AOP achieves this; we'll explain in the next section.

The fundamental change that AOP brings is the preservation of the mutual independence of the individual concerns. Implementations can be easily mapped back to the corresponding concerns, resulting in a system that is simpler to understand, easier to implement, and more adaptable to changes.

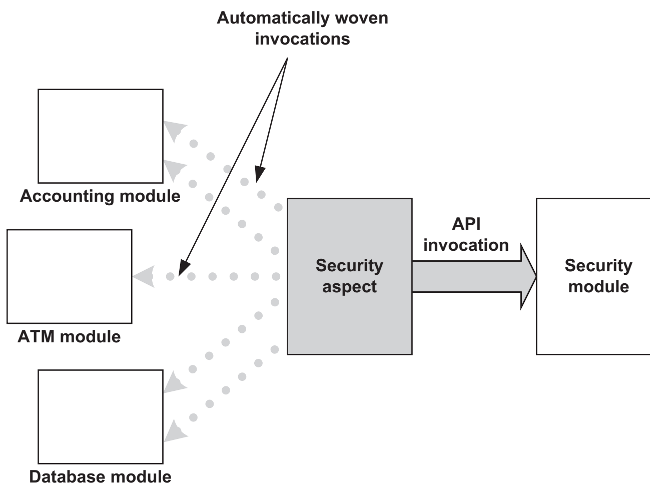

Figure 1.5 Implementing a security concern using AOP techniques: The security aspect defines the interception points needing security and invokes the security API upon the execution of those points. The client modules no longer contain any security-related code.

1.3 Anatomy of an AOP language

The AOP methodology is just that—a methodology. In order to be of any use in the real world, it must be implemented, or realized. Each realization of AOP involves specifying a language or a framework and associated tools. Like any other programming methodology, an AOP implementation consists of two parts:

- The *language specification* describes the language constructs and syntax to express implementation of the core and crosscutting concerns.
- The *language implementation* verifies the code's adherence to the language specification and translates the code into an executable form.

1.3.1 The AOP language specification

Any implementation of AOP must specify a language to implement the individual concerns and a language to implement the weaving rules. Note that an AOP system may offer a homogeneous language that doesn't distinguish between the two parts. This is likely to be the case in future AOP languages. Let's take a closer look at these two parts.

IMPLEMENTATION OF CONCERNS

As in other methodologies, the concerns of a system are implemented into modules that contain the data and behavior needed to provide their services. A module that implements the core part of the caching concern maintains a collection of cached objects, manages the validity of the cached objects, and ensures bounded memory consumption. To implement both the core and crosscutting concerns, we normally use standard languages such as C, C++, and Java.

WEAVING RULES SPECIFICATION

Weaving rules specify how to combine the implemented concerns in order to form the final system. After you implement the core part of the caching concern in a module (perhaps through a third-party class library), you must introduce caching into the system. The weaving rule in this case specifies the data that needs to be cached, the information that forms the key into the cache storage, and so forth. The system then uses these rules to obtain and update cache from the specified operations.

The power of AOP comes from the economical way of expressing the weaving rules. For instance, to modularize tracing concerns in listing 1.1, you can add a few lines of code to specify that all the public operations in the system should be logged. Here is a weaving specification for the tracing aspect:

- Rule 1: Create a logger object.
- Rule 2: Log the beginning of each public operation.
- Rule 3: Log the completion of each public operation.

This is much more succinct than modifying each public operation to add logging code. Because the tracing concern is modularized away from the class, it may focus only on the core concern, as follows:

```
public class SomeBusinessClass extends OtherBusinessClass {
    ... Core data members

    ... Override methods in the base class

    public void someOperation1(<operation parameters>) {
        ... Perform the core operation
    }

    ... More operations similar to above
}
```

Compare this class with the one in listing 1.1: all the code to perform tracing—the ancillary concerns from the class's point of view—have been removed. When you apply the same process to other crosscutting concerns, only the core business logic remains in the class. As you'll see in the next section, an AOP implementation combines the classes and aspects to produce a woven executable.

Weaving rules can be general or specific in the ways they interact with the core modules. In the previous logging example, the weaving rules don't need to mention any specific classes or methods in the system. On the other end of the spectrum, a weaving rule may specify that a business rule should be applied only to specific methods, such as the `credit()` and `debit()` operations in the `Account` class or the ones that carry the `@ReadOnly` annotation. The specificity of the weaving rules determines the level of coupling between the aspect and core logic.

The language used to specify weaving rules can be a natural extension of that language or something entirely different. For example, an AOP implementation using Java as the base language might introduce new extensions that blend well with the base language, or it could use a separate XML-based language to express weaving rules.

1.3.2 The AOP language implementation

The AOP language implementation performs two logical steps: It first combines the individual concerns using the weaving rules, and then it converts the resulting information into executable code. AOP implementation thus requires the use of a processor—*weaver*—to perform these steps.

An AOP system can implement the weaver in various ways. A simple approach uses source-to-source translation. Here, the weaver processes source code for individual classes and aspects to produce woven source code. The aspect compiler then feeds this woven code to the base language compiler to produce the final executable code. This was the implementation technique used in early implementations of AspectJ. The approach suffers from several drawbacks because the executable code can't be easily traced back to the original source code. For example, stack traces indicate line numbers in woven source code.

Another approach first compiles the source code using the base language compiler. Then, the resulting files are fed to the aspect compiler, which weaves those files. Figure 1.6 shows a schematic of a compiler-based AOP language implementation.

An AOP system may also be able to push the weaving process close to execution of the system. If the implementation of AOP is Java-based, a special class loader or a virtual

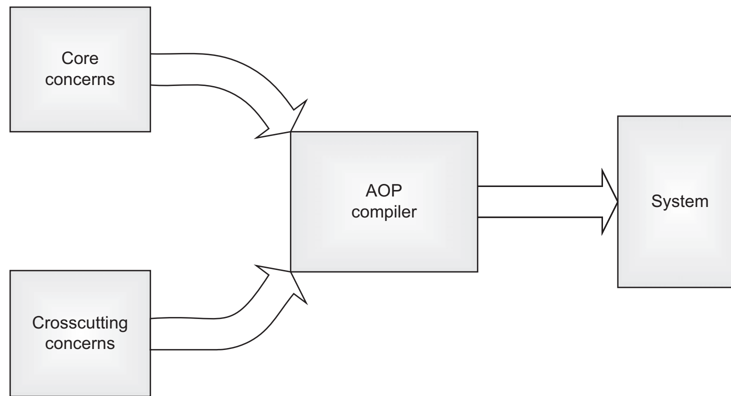

Figure 1.6 An AOP language implementation that provides a weaver in the form of a compiler. The compiler takes the implementation of the core and crosscutting concerns and weaves them together to form the final system.

machine (VM) agent can perform the weaving. Such an implementation first loads the byte code for the aspects, weaves them into the classes as they're being loaded, and supplies those woven versions of the classes to the underlying VM.

Yet another implementation could use automatically created proxies. In this case, each object that needs weaving is wrapped inside a proxy. Such an implementation typically works well in conjunction with another framework that controls the creation of objects. In this way, the framework can wrap each created object in a proxy.

So far, we've looked at the mechanics of an AOP system. Now, let's examine AOP's fundamental concepts.

1.4 Fundamental concepts in AOP

By now, it should be clear that AOP systems help in modularizing crosscutting concerns. But so do many other technologies, such as byte-code manipulation tools, direct use of the proxy design pattern, and meta-programming. How do you differentiate AOP from these options? To find out, we need to distill the core characteristics of AOP systems into a generic model. If a system fits that model, it's an AOP system.

To implement a crosscutting concern, an AOP system may include many of the following concepts:

- *Identifiable points in the execution of the system*—The system exposes points during the execution of the system. These may include execution of methods, creation of objects, or throwing of exceptions. Such identifiable points in the system are called *join points*. Note that join points are present in all systems—even those that don't use AOP—because they're points during execution of a system. AOP merely identifies and categorizes these points.
- *A construct for selecting join points*—Implementing a crosscutting concern requires selecting a specific set of join points. For example, the tracing aspect

discussed earlier needs to select only the public methods in the system. The *pointcut* construct selects any join point that satisfies the criteria. This is similar to an SQL query selecting rows in database (we'll compare AOP with databases in section 1.5.2). A pointcut may use another pointcut to form a complex selection. Pointcuts also collect context at the selected points. For example, a pointcut may collect method arguments as context. The concept of join points and the pointcut construct together form an AOP system's *join point model*. We'll study AspectJ's join point model in chapter 3.

- *A construct to alter program behavior*—After a pointcut selects join points, you must augment those join points with additional or alternative behavior. The *advice* construct in AOP provides a facility to do so. An advice adds behavior before, after, or around the selected join points. Before advice executes before the join point, whereas after advice executes after it. Around advice surrounds the join point execution and may execute it zero or more times. Advice is a form of *dynamic crosscutting* because it affects the execution of the system. We'll study AspectJ's dynamic crosscutting implementation in chapter 4.
- *Constructs to alter static structure of the system*—Sometimes, to implement crosscutting functionality effectively, you must alter the static structure of the system. For example, when implementing tracing, you may need to introduce the logger field into each traced class; *inter-type declaration* constructs make such modifications possible. In some situations, you may need to detect certain conditions, typically the existence of particular join points, before the execution of the system; *weave-time declaration* constructs allow such possibilities. Collectively, all these mechanisms are referred to as *static crosscutting*, given their effect on the static structure, as opposed to dynamic behavior changes to the execution of the system. We'll study AspectJ's static crosscutting support in chapter 5.
- *A module to express all crosscutting constructs*—Because the end goal of AOP is to have a module that embeds crosscutting logic, you need a place to express that logic. The *aspect* construct provides such a place. An aspect contains pointcuts, advice, and static crosscutting constructs. It may be related to other aspects in a similar way to how a class relates to other classes. Aspects become a part of the system and use the system (for example, classes in it) to get their work done. We'll examine AspectJ's implementation of aspect in chapter 6.

Figure 1.7 shows all these players and their relationships to each other in an AOP system.

Each AOP system may implement a subset of the model. For example, Spring AOP (discussed in chapter 9) doesn't implement weave-time declarations due to its emphasis on its runtime nature. On the other hand, the join point model is so central to AOP that every AOP system must support it—everything else revolves around the join-point model.

When you encounter a solution that modularizes crosscutting concerns, try to map it onto the generic AOP model. If you can, then that solution is indeed an AOP system. Otherwise, it's an alternative approach for solving the problem of crosscutting concerns.

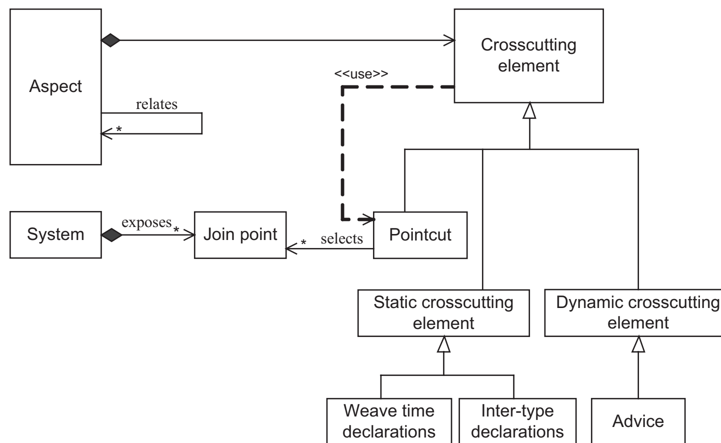

Figure 1.7 Generic model of an AOP system. Note that not every system implements each part of the model.

1.5 AOP by analogy

When you're learning a new technology, it sometimes helps to compare it with existing technologies. In this section, we'll attempt to help you understand AOP by comparing it with Cascading Style Sheets (CSS), database programming, and event-oriented systems. The purpose of this section is to help those familiar with at least one of these technologies to understand AOP by analogy.

1.5.1 Cascading Style Sheets (CSS)

CSS is a widely supported mechanism to separate content from presentation in HTML pages. Without CSS, formatting information is fused with content (causing tangling), and similar content elements have presentational information spread into multiple places (causing scattering). CSS helps the situation by letting the main document focus on content by separating the formatting information into a document called a *stylesheet*.

A core concept in CSS is a *selector* that selects document elements matching a certain specification. For example, the `body p` selector can select paragraphs inside the body element. You can then associate presentational information with a selector and, for example, set the background color of such elements to blue using the `body p {background: blue;}` element.

AOP acts on classes in the same way that CSS acts on documents. AOP lets you separate crosscutting logic from the main-line logic. AOP's pointcuts have the same selection role as CSS selectors. Whereas CSS selectors select structural elements in a document, pointcuts select program elements. Similarly, the blocks describing the formatting information are analogous to AOP advice in functionality.

Often, the selection mechanism requires more information than merely using the inherent characteristics of a structure such as `body p`. It's common practice to supplement content elements with additional metadata through the `class` attribute. For

example, you can mark an HTML paragraph element as `menu` by using the tag `<p class="menu">`. Then, in the stylesheet, you can select such an element by using the `p.menu` selector and apply appropriate presentation characteristics. In AOP, practitioners face the same problem—selection through a pointcut often requires information beyond merely relying on inherent characteristics of the program elements such as class and method names. The use of Java annotations plays a role similar to the class attribute in HTML documents. You can, for example, mark a method as `@Transactional` and utilize it in a pointcut expression.

There are similarities from the adoption perspective as well. Through WYSIWYG HTML editors, it's easy to create a good-looking HTML. That apparent simplicity led to many initial web documents embedded with formatting information. But when we realized that it's difficult to create a consistent look when every element's formatting is specified independently, developers started to look favorably at CSS. AOP has encountered a similar trend. There is a level of comfort in embedding the implementation of crosscutting functionality inside classes; you can see exactly what's going to happen. But you soon realize that creating a consistent implementation is nearly impossible when similar code is scattered in many places. In addition, using CSS requires a level of expertise and understanding of the semantics associated with the elements in a document. Using AOP requires similar understanding of the semantic separation between core and crosscutting elements.

CSS works at the structure level, and database triggers offer similar separation at the programming level. Let's see how that technique compares with AOP.

1.5.2 Database systems

Database systems offer "dynamic crosscutting" targeted toward data access, whereas AOP offers a similar mechanism toward general programming. It offers two good analogies to AOP concepts: SQL with pointcuts and triggers with advice.

SQL AND POINTCUTS

A join point is like a row in a database, whereas a pointcut is like an SQL query. An SQL query selects rows according to a specified criterion such as "rows in accounts table, where the balance is greater than 50". It provides access to the content of the selected rows. Similarly, a pointcut is a query over program execution that selects join points according to a specified criterion such as "method execution in the `Account` class, where the method name starts with 'set'". It also provides access to the join point context (objects available at the join point, such as method arguments).

TRIGGERS AND ADVICE

Database programming often uses triggers to respond to changes made in data. For example, you can use a trigger to audit changes in certain tables. The following snippet calls the `logInventoryIncrease()` procedure when inventory increases:

```
CREATE OR REPLACE TRIGGER inventory_increase_trigger
    AFTER UPDATE OF count ON inventory
    FOR EACH ROW
        WHEN (new.count > old.count)
        CALL logInventoryIncrease(:new.itemID, :old.count, :new.count);
```

The static condition, such as the name of the table and the modified column, as well as the dynamic condition, such as the difference in the column value, are analogous to AOP's pointcut concept. Both describe a selection criterion to "trigger" certain actions. The stored procedure specified in the trigger is analogous to AOP's advice.

Database triggers and AOP's advice both modify the normal program execution to carry additional or alternative actions. But there are some obvious differences. Database triggers are useful only for database operations. AOP has a more general approach that can be used for many other purposes. But note that AOP doesn't necessarily obviate the need for database triggers, for reasons such as performance and bringing uniformity to multiple applications accessing the same tables.

Similar to database triggers, event-oriented programming includes the notion of responding to events.

1.5.3 Event-oriented programming

Event-oriented programming is essentially the observer design pattern (we'll discuss it as an alternative to AOP in section 1.7.3). Each interested code site notifies the observers by firing events, and the observers respond by taking appropriate action, which may be crosscutting in nature.

In AOP, the program is woven with logic to fire virtual events and to respond to the events with an action that corresponds to the crosscutting concern it's implementing. But note this important difference: Unlike in event-based programming, there is no explicit code for the creation and firing of events in the subject classes. Executing part of the program constitutes the virtual-event generation. Also, event systems tend to be more coarse-grained than an AOP solution implements.

Note that you can effectively combine event-oriented programming with AOP. Essentially, you can modularize the crosscutting concern of firing events into an aspect. With such an implementation, you avoid tangling the core code with the event-firing logic.

Now that you have a good understanding of AOP, let's turn our attention to a bit of history and the current status of AOP implementations.

1.6 Implementations of AOP

Much of the early work that led to AOP today was done in research institutions. Cristina Lopes and Gregor Kiczales of the Palo Alto Research Center (PARC), a subsidiary of Xerox Corporation, were among the early contributors to AOP. Gregor coined the term *AOP* in 1996 and started AspectJ, the first implementation of AOP.

But AOP is a methodology with many possible implementations. Each implementation takes a slightly different view on the target use case and programming constructs. Let's see who the dominant players are and how they size up against each other.

1.6.1 AspectJ

AspectJ is the original and still the best implementation of AOP. After a few initial releases, Xerox transferred the AspectJ project to the open source community at eclipse.org. In its early implementations, AspectJ extended Java through additional

keywords to support AOP concepts, similar to the way C++ extended C to support OOP concepts. As an implementation, it provided a special compiler.

Until a few years back, AspectJ had a close cousin: AspectWerkz. This AOP system followed the core AspectJ model, except that it used metadata expressed through Javadoc annotations, Java 5 annotations, or XML elements in place of additional keywords. In AspectJ version 5, AspectWerkz merged with AspectJ, offering developers a choice of technologies including a new @AspectJ (pure Java 5 annotation-based) syntax. We'll study that syntax in chapter 7.

AspectJ's primary tool support is an Eclipse plug-in, AspectJ Development Tools (AJDT). One of AJDT's most important features is a tool for visualization of crosscutting, which is helpful for debugging a pointcut specification. Although you write classes and aspects separately, you can visualize the combined effect even before the code is deployed.

Scala and AspectJ

Scala is a new language that compiles source code to the standard Java byte code. Scala maps program elements to byte code in a manner similar to that of Java. This lets you use Scala with AspectJ. You can see a working example at http://blog.objectmentor.com/articles/2008/09/27/traits-vs-aspects-in-scala. Note that, unlike Scala, other JVM languages such as JRuby and Groovy use mapping that heavily relies on reflection. Therefore, AspectJ may not be used as readily with those. Instead, you need specialized languages that work with them, as you'll see in section 1.6.3.

The AspectJ language has an alternative implementation called the AspectBench compiler (abc; http://aspectbench.org). The focus of this project is to provide a flexible implementation to support experimenting with new AspectJ language features and optimization ideas.

1.6.2 Spring AOP

Spring is the most popular lightweight framework for enterprise applications. To satisfy the needs of enterprise applications, it includes an AOP system based on interceptors and the proxy design pattern. Earlier implementations of Spring AOP (prior to Spring 2.0) offered a somewhat complex programming model. The new programming model, based on AspectJ, offers a much better programming experience and enables Spring users to write custom aspects without difficulty.

Like AspectJ, Spring AOP, through the Spring IDE (an Eclipse plug-in), provides support for visualizing crosscutting in the IDE. We'll examine how Spring uses AspectJ in detail in chapter 9 and in examples throughout the book.

Spring.NET is the .NET counterpart of the Spring Framework. It includes AOP support that is similar to Spring AOP.

1.6.3 Other implementations of AOP

Many other implementations of AOP in Java are available. JBoss (http://www.jboss.org/jbossaop), an open source application server, offers an AOP solution that includes a pointcut language similar to that of AspectJ. In addition, the AOP Alliance API is implemented in frameworks such as Guice (http://code.google.com/p/google-guice) and Seasar (http://www.seasar.org). (Spring used to offer a programming model based on the AOP Alliance API, but that model has been designated a transitional technology status due to the availability of the AspectJ-based model.)

AspectJ has been an inspiration for AOP implementations for other languages such as Aquarium for Ruby (http://aquarium.rubyforge.org), Aspect-Oriented C (http://research.msrg.utoronto.ca/ACC), and AspectC++ (http://www.aspectc.org). Groovy, like Ruby, makes it possible to implement an AOP-like functionality through its metaobject protocol (MOP) facility (see http://www.infoq.com/articles/aop-with-groovy for an explanation of this approach). But as with Ruby, efforts are underway to introduce an AspectJ-like syntax to provide a domain-specific language (DSL) to simplify writing aspects (see http://svn.codehaus.org/grails-plugins/grails-aop for the code of the yet-to-be-released grails-aop project).

AOP has generated quite a bit of interest in the .NET world. Due to the use of byte code representations and the possibility of using proxies, .NET offers choices similar to those available in the Java world. In addition to Spring.NET, prominent AOP solutions in .NET include PostSharp (http://www.postsharp.org) and Aspect# (http://www.castleproject.org/aspectsharp). LOOM.NET (http://www.dcl.hpi.uni-potsdam.de/research/loom) is a research project that's exploring static and dynamic weaving in .NET.

1.7 Alternatives to AOP

The problem AOP addresses isn't new. The concerns of auditing, transaction management, security, and so on emerged as soon as we started implementing nontrivial software systems. Consequently, many competitive technologies deal with the same problem: frameworks, code generation, design patterns, and dynamic languages. Let's look at those alternatives.

NOTE Although I'll compare these techniques as alternatives to AOP and (not surprisingly) show how AOP outshines each of them when it comes to dealing with crosscutting concerns, I don't mean that these techniques are useless. Each of these approaches is appropriate for a set of problems. AOP works well alongside these techniques and, in some cases, can enhance their implementation.

1.7.1 Frameworks

Frameworks such as servlets and Enterprise JavaBeans (EJB) offer specific solutions to a focused set of problems. For example, the servlet specification offers filters to deal with requests made using the HTTP protocol. Given that each framework deals with a

specific problem, it may also provide some solutions for dealing with the associated crosscutting concerns. For example, with the servlet framework, you may use filters to implement concerns such as security.

Similarly, the EJB framework addresses crosscutting concerns such as transaction management and security. The EJB3 specification even provides limited support for interceptors—which, to an extent, matches the goals of AOP. But as you'll see later, it falls short of being a complete solution.

You can use AOP along with an underlying framework. In such an arrangement, the core framework deals with the target problem and lets aspects deal with crosscutting concerns. For example, the core Spring Framework deals with dependency injection for configuration and enterprise service abstraction to isolate beans from the underlying infrastructure details, while employing AOP to deal with crosscutting concerns such as transaction management and security.

A framework's approach to crosscutting concerns often, but not always, boils down to either employing code generation or implementing appropriate design patterns. Let's examine the two in more details.

1.7.2 Code generation

Code-generation techniques shift some responsibility of writing code from the programmer to the machine. (Of course, programmers do have to write code for the generators.) These techniques represent powerful ways to deal with a wide range of problems and often are helpful in raising the level of abstraction. You can modify the original code to add functionality such as observer notifications or produce additional artifacts such as proxy classes. In the process, code generation takes care of one of the drawbacks of using design patterns directly: manual modifications in many places.

A variation of code generation works at the compiled code level through byte-code manipulation tools. Instead of producing source code that needs to be compiled into machine code, the code generator directly produces machine code. For Java, the difference between source code–level generation and byte-code generation is small, given how directly source code maps to byte code. Many Java technologies, such as Hibernate and JRuby, use byte-code manipulation techniques as the basis for their implementation.

In Java 5, the annotation language feature lets you attach additional information to the program elements. Code-generation techniques can take advantage of those annotations to produce additional artifacts such as Java code or XML configuration files. Java 5 even provides an Annotation Processing Tool (APT), to simplify the process. But APT forces you to understand low-level details such as the syntax tree, and that makes it difficult to use unless you acquire specific skills. It's no surprise that few non-framework programmers use APT. AOP, on the other hand, can provide simpler solutions to process annotations, as you'll see in rest of the book.

Many systems, most notably AspectJ, use byte-code manipulation as the underlying technique in implementing AOP. The difference is how it employs the technique as part of the overall AOP model. First, it provides a much simpler programming

model, making it easier for you to create modularized crosscutting implementations without knowing low-level details such as the abstract syntax tree. It essentially provides a DSL targeted at crosscutting concerns. Users are isolated from byte-code manipulation mechanisms, which aren't for the faint of heart. Furthermore, by limiting power, AOP nudges you toward writing better code. In short, although code generation is capable of doing anything AOP can do (and a lot more), AOP brings a level of discipline that is essential for good software engineering when it comes to dealing with crosscutting concerns.

1.7.3 Design patterns

Design patterns also provide solutions to crosscutting concerns. In this section, we'll take a comparative look at some of the design patterns—observer, chain of responsibility, decorator, proxy, and interceptor—that help with crosscutting concerns. You'll see that quite a few similarities exist between design patterns and AOP—you can view a few design patterns as a "poor man's" AOP implementation.

OBSERVER PATTERN

The well-known observer design pattern decouples the subject (an object of interest) from observers (objects that need to respond to changes in the subject). When a subject changes its state, it notifies all observers of the change by calling a method such as `notify<ChangeType>()`, passing it an event object that encapsulates the information about the change. The notification method iterates over all the observers and calls a method on each observer (in message-oriented systems, these details change a bit, but the overall scheme remains the same). The called method in the observer includes the logic appropriate to respond to the event.

AOP's advice may superficially look like an event responder, but there are some important differences. First, you won't see invocations such as `notify<ChangeType>()` in the subject class; thus, advice decouples the observer pattern logic from the subject. As a result, you can produce notifications that weren't planned in advance, making your system extensible over what it was originally designed to be and bringing it new life in some situations. Second, the context collected by pointcuts (equivalent to information carried in an event object) is much more flexible and powerful in AOP. Pointcuts can collect just the right amount of context needed for advice logic. With a typical event model, you end up passing everything that you might possibly need.

CHAIN OF RESPONSIBILITY

The chain of responsibility (COR) pattern, shown in figure 1.8, puts a chain of processing objects in front of a target object. Before or after invoking the target object, the objects in the chain may perform additional work.

Successfully applying the COR pattern has two prerequisites. First, you should have only one or a small number of target methods, whose processing needs to be augmented. Second, the associated framework should already support the pattern. For example, the filter implementation in the servlet framework implements the COR pattern. It works well there because both prerequisites are met: it targets only one method—`doService()`—and the filter-management code is implemented as a part of

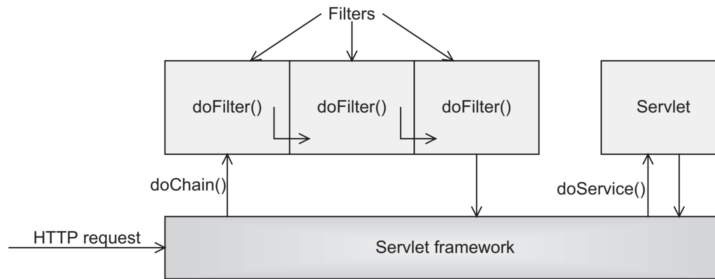

Figure 1.8 Chain of responsibility as implemented by the servlet framework. The filter chain allows applying additional logic such as coarse-grained security around the doService() method.

the framework. In this setup, some coarse-grained crosscutting concerns—ones that deal at the HTTP request level—may be modularized into servlet filters. But for any concern that needs to go beyond the `doService()` method, filters offer no solution.

AOP works in similar ways, except it doesn't have either of the prerequisites. Instead, each aspect deals with the problem head on by advising appropriate code.

DECORATOR AND PROXY

The decorator and proxy[4] design patterns use a wrapper object that can perform some work before, after, or around invocation of the wrapped object or its representation, as shown in figure 1.9. This additional work can be crosscutting in nature. For example, each method may perform a security check before the wrapped object's method is invoked.

Java offers dynamic proxies that reduce the code required for creating wrapper types and routing each method. Using this feature, you dynamically create a proxy for a given set of interfaces, supplying an invocation handler. The proxy implements all the specified interfaces and invokes the invocation handler when any interface method is called. You'll see a full example of dynamic proxies in chapter 9, section 9.2.1.

Implementing crosscutting concerns using dynamic proxies requires control over the creation of each object so that it may be wrapped in a dynamic proxy. Because

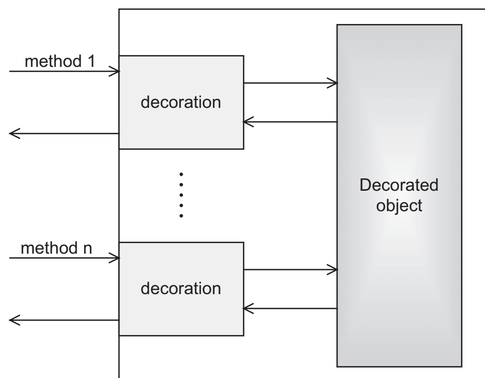

Figure 1.9 The decorator design pattern. The original object is wrapped in the decorator that presents the same interface as the decorated object. Each method passes through the decoration, which can implement functionality such as security and transaction management.

[4] I lump these two patterns together because the distinction between them isn't significant from the AOP perspective. A decorator holds onto a real object that it decorates, whereas a proxy may not necessarily hold any real object but simulates holding one. In a way, the proxy design pattern is a generalization of the decorator design pattern.

proxies invoke the invocation handler for every method, proxies must include logic to apply crosscutting concern selectively. Furthermore, dynamic proxies intercept method invocations only: they can't crosscut object creation, field access, exception handling, and so on.

Direct use of the decorator and proxy design patterns to implement crosscutting concerns requires substantial effort. But you can use these patterns as the underlying implementation technique as a part of an AOP system. The Spring Framework, as you'll see in chapter 9, uses the proxy design pattern internally to avoid exposing it to users. This isn't unlike the byte-code manipulation technique—cumbersome as a programming technique to deal with crosscutting concerns, but a fine underlying technology to implement AOP systems.

Another design pattern, interceptor, is often used along with the proxy design pattern. Let's see how it compares to AOP for crosscutting concerns.

INTERCEPTOR

An interceptor performs additional logic by intercepting method invocations on an object. The interceptor pattern lets you express crosscutting logic in an interceptor object. Used with the proxy or decorator design pattern, this pattern offers a reasonable solution for a wide range of crosscutting problems. For example, Java supports the creation of dynamic proxies, which you can configure with an interceptor. Implementing the interceptor pattern generically and successfully requires a fair amount of machinery; thus it's best to leave such an implementation to a framework.

Let's consider the newest implementation of the interceptor pattern in EJB3. The earlier versions of the EJB framework offered a solution for a specific set of crosscutting concerns: primarily transaction management and role-based security. EJB3 offers a way to modularize user-specific crosscutting concerns through the interceptor approach:

```
public class TracingInterceptor {
    private Logger logger = ...

    @AroundInvoke
    public Object trace(InvocationContext context) throws Exception {
        logger.log("Entering " + context.getMethod().getName()
                      + " in " + context.getBean().getClass().getName());
        return context.proceed();
    }
}
```

Then, you can apply the interceptor to target classes and methods, as shown in the following code snippet:

```
@Stateless
@Interceptors({TracingInterceptor.class})
public class InventoryManagementBean {
      ...
}
```

You can target specific methods by marking each method with the `@Interceptors` annotation. On the other extreme, you can declare an interceptor as a default interceptor

that applies to all beans except those that opt out. EJB3's implementation has a few limitations: for example, an interceptor may be applied only to EJBs and not to ordinary types in the system, which may pose restrictions on certain usages. The programming model is also complex and type-unsafe: the intercepted context (intercepted bean, method name, method arguments) is accessed through the `InvocationContext` object, whose method returns `Object` and may require casting before using the context.

But the real problem with the EJB interceptor design (and many similar interceptor implementations) is the missing key abstraction of pointcuts. Classes and methods need to declare that they must be intercepted, reducing the interceptor to a more macro-like facility. As a result, although the logic equivalent to AOP's advice is modularized, the pointcut equivalent logic is spread in all intercepted types. And due to the generic nature of the join point context, the interceptor method may need complex logic to pluck arguments from the context.

Note that the Spring Framework, in versions prior to 2.0, used a mechanism similar to `InvocationContext` and thus suffered from programming complexities similar to EJB3 interceptors. But Spring's AOP always used a pointcut notion to avoid the problem of spreading selection logic in multiple places. The AspectJ integration introduced in Spring 2.0 removes the need for `InvocationContext`-like logic and raises the pointcut implementation to a new level, as you'll see in chapter 9.

1.7.4 Dynamic languages

Dynamic languages have recently gained popularity, and rightly so. A dynamic language, when combined with a framework that takes advantage of the underlying language, can provide powerful solutions for a set of problems. For example, Ruby combined with Rails, or Groovy combined with Grails, provides simpler solutions for certain kinds of web applications. Most dynamic languages offer a meta-programming facility that lets you modify the structure and behavior of a program during its execution. The meta-programming facility may modularize crosscutting concerns. For example, you can modify an existing method's implementation to wrap it with code that performs the crosscutting functionality before or after dispatching the original method.

Although meta-programming is a fine tool for dealing with crosscutting concerns, you must keep in mind a few considerations:

- You need to use a dynamic language that supports meta-programming. The static versus dynamic languages war hasn't concluded, nor will it conclude any time soon, so you'll have to make a considered choice.
- Meta-programming may be too powerful a tool for you; a more disciplined approach may be appropriate.
- Tooling to support crosscutting implementation is difficult to imagine with general-purpose meta-programming facilities offered by dynamic languages.

This is a reason why Dean Wampler, a long-time AOP expert, started the Aquarium project (http://aquarium.rubyforge.org) to bring AOP to Ruby. It shows that although AOP is popular in statically typed languages, it also has a role in dynamically

typed languages. Interestingly, as seen from this project, it's relatively easy to build AOP capabilities on top of core meta-programming support provided by the underlying language. By providing an aspect-focused DSL to express pointcuts, Aquarium provides a solution to modularize the pointcut portion of AOP in Ruby.

It's instructive to note that the father of AOP, Gregor Kiczales, who wrote *The Art of the Metaobject Protocol* (MIT Press, 1991), thought that AOP was better suited for crosscutting concerns instead of a direct application of meta-programming.

In a way, statically typed languages use AOP to gain meta-programming support. In contrast, dynamic languages benefit from AOP as a disciplined application of meta-programming.

1.8 Costs and benefits of AOP

Nothing comes free! Software engineering, like any engineering discipline, is all about optimizing costs and benefits. AOP isn't free either. Critics of AOP often talk about how difficult it is to understand. And indeed, AOP takes time, patience, and practice to master. But the main reason behind the difficulty is the newness of the methodology. When was the last time a brand-new programming methodology was accepted without its share of adaptation resistance? AOP demands that you think about system design and implementation in a new way.

When you use AOP, you get a lot of benefits, but you must also understand some costs in order to make informed decisions. In this section, we'll discuss first the costs and then the benefits of AOP.

1.8.1 Costs of AOP

Some of the costs associated with AOP are the usual suspects associated with any new technology:

- Making an investment in learning AOP
- Hiring skilled programmers
- Following an adoption path to ensure that you don't risk the project by overextending yourself
- Modifying the build and other development processes
- Dealing with the availability of tools

Well-understood mitigation techniques are available for some of these issues:

- Making a proper investment in learning the technology (and you already took a step by reading this book)
- Doing due diligence in checking skill availability (which is becoming increasingly easy due to Spring's popularity)
- Following a gradual adoption path, which we'll show in part 2 of the book

Tools for AOP aren't as mature as they are for Java (although they're more mature than for most other languages that run inside the Java VM). Fortunately, a lot of effort is currently under way to improve the tooling around AOP, so this isn't a serious impediment to adopting it.

But one cost, still common to most new technologies, deserves more in-depth treatment: the cost of abstraction. Abstraction lets you hide inessential details and thus reduce the complexity of the underlying system. Good abstraction leads to the creation of well-isolated modules. Because each module represents a much smaller subsystem, abstraction offers a way to contain complexity at a level you can cope with. Modularity is a divide-and-conquer approach to managing complexity. But the abstraction introduced by AOP isn't without costs.

NEED FOR GREATER SKILLS

Creating the right level of abstraction is a highly skilled job (many correct abstractions are possible in a given system). A thorough understanding of the costs and benefits is a hallmark of good software engineering. Merely understanding the implementation mechanisms won't yield useful abstractions.

In the context of AOP, you must understand how to fit the new unit of modularity—aspects—into the system. For that, you must apply decomposition techniques to separate core concerns from crosscutting concerns. All this requires experience that is often best gained by applying AOP in a gradual manner. The second part of the book provides details of this strategy.

On the flip side, crosscutting logic is separated from business logic. This enables you to use developers who understand only the business logic and not the intricacies of the crosscutting functionality.

COMPLEX PROGRAM FLOW

Abstraction, by its nature, hides the details. In software systems, higher levels of abstraction always mean that less information is available at the code level. Looking at a code segment doesn't tell you the whole story that will unfold during system execution. For example, in OOP, due to polymorphic methods, you can't tell the exact method that will be executed at runtime, because the choice of method is based on the type of the object, not the static type of the declared variable. This makes analyzing program flow a complex task. Even in procedural languages such as C, if you use function pointers, the program flow isn't static and requires some effort to be understood.

AOP abstracts away program flow even further. You may not know (except through good tooling support) that a crosscutting action will take place in a certain part of the code. Many programmers new to AOP get stuck until they realize that this separation of concerns is the whole point. If you insist on understanding the exact program flow, it's a sign that you need to reflect a little longer on the core ideas of AOP. But just as OOP requires a few years of practice before you understand the underlying core ideas, most developers get AOP eventually.

1.8.2 Benefits of AOP

Now that you know the costs, let's look at the benefits.

SIMPLIFIED DESIGN

The architect of a system is often faced with underdesign/overdesign issues. If you underdesign, you may have to make massive changes later in the development cycle. If you overdesign, the implementation may be burdened with code of questionable

usefulness. With AOP, you can delay making design decisions for future requirements because you can implement those as separate aspects. You can focus on the current requirements of the system.

AOP works in harmony with one of the most popular trends of agile programming by supporting the practice of "You aren't gonna need it" (YAGNI). Implementing a feature just because you may need it in the future often results in wasted effort because you won't actually need it. With AOP, you can practice YAGNI; and if you do need a particular kind of functionality later, you can implement it without having to make system-wide modifications. Even for the feature that you need, agile programming promotes developing them progressively. AOP helps you add features incrementally through the introduction of aspects, often without modifying the rest of the code.

CLEANER IMPLEMENTATION

AOP allows a module to take responsibility only for its core concern, thus following the SRP; a module is no longer liable for other crosscutting concerns. For example, a module implementing business logic is no longer responsible for the security functionality. This results in cleaner assignments of responsibilities, reduced code clutter, and less duplication. It also improves the traceability of requirements to their implementation, and vice versa.

Reduced code tangling makes it simpler to test code, spot potential problems, and perform code reviews. Reviewing the code of a module that implements only one concern requires the participation of an expert in the functionality implemented by that module. Such a simplified process leads to higher-quality code.

Reduced code scattering avoids the cost of modifying many modules to implement a crosscutting concern. Thus, AOP makes it cheaper to implement a crosscutting feature. By letting you focus on the core concern of a module and make the most of your expertise, AOP also reduces the cost of the core concerns.

The end effect is a cheaper overall feature implementation, better time-to-market, and easier system evolution.

BETTER CODE REUSE

The key to greater code reuse is a more loosely coupled implementation. If a module is implementing multiple concerns, other systems requiring similar functionality may not be able to use the module if they implement a different set of crosscutting concerns. With AOP, because you can implement each crosscutting module as an aspect, core modules aren't aware of crosscutting functionality. By modifying the aspects, you can change the system configuration. For example, a service layer may be secured in a project with one security scheme, in another project with another scheme, or in still another project with no security at all by including or excluding appropriate aspects. Without AOP, a service layer tied with a specific security implementation may not be reused in another project.

1.9 Summary

The most fundamental principle in software engineering is that the separation of concerns leads to a system that is simpler to understand and easier to maintain. Various

methodologies and frameworks support this principle in some form. For instance, with OOP, by separating interfaces from their implementation, you can modularize the core concerns well. But for crosscutting concerns, OOP forces the core modules to embed the crosscutting concern's logic. Although the crosscutting concerns are independent of each other, using OOP leads to an implementation that no longer preserves independence in the implementation.

Aspect-oriented programming changes this by modularizing crosscutting concerns in a generic and methodical fashion. With AOP, crosscutting concerns are modularized by encapsulating them in a new unit called an aspect. Core concerns no longer embed the crosscutting concern's logic, and all the associated complexity of the crosscutting concerns is isolated into the aspects. AOP marks the beginning of a new way of dealing with a software system by viewing it as a composition of mutually independent concerns. By building on top of existing programming methodologies, AOP preserves the investment in knowledge gained over the last few decades.

In the last few years, AOP has become a practical technology. It has been deployed in many organizations, big and small, to add powerful features that we might have otherwise shied away from or implemented in a laborious manner.

In the next eight chapters, we'll study a specific implementation of AOP for Java, AspectJ, as well as its integration with Spring. Those chapters and the rest of the book will provide examples that use this technology to solve real problems. In chapter 2, you'll see how AspectJ implements AOP for Java.

JAVA/SOFTWARE ENGINEERING

"This book teaches you how to think in aspects. It is essential reading for both beginners who know nothing about AOP and experts who think they know it all."
— Andrew Eisenberg, AspectJ Development Tools Project Committer

AspectJ IN ACTION

Ramnivas Laddad FOREWORD BY ROD JOHNSON

To allow the creation of truly modular software, OOP has evolved into aspect-oriented programming. AspectJ is a mature AOP implementation for Java, now integrated with Spring.

AspectJ in Action, Second Edition is a fully updated, major revision of Ramnivas Laddad's best-selling first edition. It's a hands-on guide for Java developers.

After introducing the core principles of AOP, it shows you how to create reusable solutions using AspectJ 6 and Spring 3. You'll master key features including annotation-based syntax, load-time weaver, annotation-based crosscutting, and Spring-AspectJ integration. Building on familiar technologies such as JDBC, Hibernate, JPA, Spring Security, Spring MVC, and Swing, you'll apply AOP to common problems encountered in enterprise applications.

This book requires no previous experience in AOP and AspectJ, but it assumes you're familiar with OOP, Java, and the basics of Spring.

What's Inside

- Totally revised Second Edition
- When and how to apply AOP
- Master patterns and best practices
- Code you can reuse in real-world applications

An expert in enterprise Java, **Ramnivas Laddad** is well known in the area of AOP and Spring. He is a committer on the Spring Framework project.

For online access to the author and a free ebook for owners of this book, go to manning.com/AspectJinActionSecondEdition

"Clear, concisely worded, well-organized ... a pleasure to read."
—From the Foreword by Rod Johnson, Creator of the Spring Framework

"Ramnivas showcases how to get the best out of AspectJ and Spring."
—Andy Clement
AspectJ Project Lead

"One of the best Java books in years."
—Andrew Rhine
Software Engineer, eSecLending

"By far the best reference for Spring AOP and AspectJ."
—Paul Benedict, Software Engineer, Argus Health Systems

"Ramnivas expertly demystifies the awesome power of aspect-oriented programming."
—Craig Walls
author of *Spring in Action*

ISBN 13: 978-1-933988-05-4
ISBN 10: 1-933988-05-3

MANNING $49.99 / Can $62.99 [INCLUDING eBOOK]