

Enterprise AOP with Spring

SECOND EDITION

AspectJ in Action

Ramnivas Laddad

MEAP

MANNING



Unedited Draft

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.



**MEAP Edition
Manning Early Access Program**

Copyright 2008 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.
Please send your feedback to
ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

Table of Contents

PART 1 Understanding AOP and AspectJ

Chapter 1 Introduction to AOP

Chapter 2 Introducing AspectJ

Chapter 3 AspectJ join point model

Chapter 4 Dynamic crosscutting

Chapter 5 Static crosscutting

Chapter 6 Aspect

Chapter 7 The @AspectJ syntax

Chapter 8 AspectJ weaving models

Chapter 9 Spring AspectJ integration

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

PART 2 Applications of AspectJ

Chapter 10 Monitoring techniques

Chapter 11 Policy enforcement: Keeping design intact

Chapter 12 Design patterns

Chapter 13 Implementing concurrency control

Chapter 14 Transaction management

Chapter 15 Implementing security

Chapter 16 Implementing domain logic

Chapter 17 The next step

Appendices

Appendix A The AspectJ compiler

Appendix B Understanding Ant and Maven integration

Appendix C Understanding IDE integration

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

1

Introduction to AOP

This chapter covers

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

Reflect back on your last project and then another project a few years back. What's the difference? One word—complexity! Today's software systems are much more complex and all indications point to even faster growth in software complexity. Yet, human intelligence hasn't been able and is unlikely to match the complexity—blame the slow evolutionary process! So what can a software developer do?

If complexity is the problem, modularization is the solution. By breaking down the problem into more manageable pieces, we can have a better shot at implementing each piece. For example, when you face with complex software requirements, you are likely to break those down into multiple parts: business functionality, data access, presentation logic, and parts that cut across each of those. We call each of these functionalities as *concerns* of the system. In a

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

banking system, for example, you may have business functionality such as customer management, account management, and loan management. You may also have an implementation of data access and the web layer. We call these modules *core concerns*, since they form the core functionality of the system. Then there will be concerns such as security, logging, resource pooling, caching, performance monitoring, concurrency control, and transaction management that 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 implementing classes such as *LoanManagementService* as implementing business logic and *AccountRepository* as data access implementation. But what about all other functionalities? Wouldn't it be nice if you could implement modules that you can identify as *Security*, *Auditing*, or *PerformanceMonitor*? You can't with OOP alone. Instead, OOP will force you to spread the implementation of these functionalities across many modules. For example, you will see invocation of a transaction management API, along with associated exception handling, in each service class. Furthermore, classes such as *LoanManagementService* are now burdened with functionality other than loan management. In short, a typical OOP implementation creates an undesirable coupling between core and crosscutting concerns. Any addition of new crosscutting features and even certain modifications to the existing crosscutting features may force modifications to the relevant core modules. This is where Aspect-oriented Programming (AOP) comes to help.

AOP is a methodology that provides separation of crosscutting concerns by introducing a new unit of modularization—an *aspect*. With AOP you implement crosscutting concerns in aspects instead of fusing them in the core modules. It allows creating modules such as *Security*, *Auditing*, or *PerformanceMonitor*. Each such module focuses on a specific crosscutting functionality. This leads to creation of the core classes that are no longer burdened with crosscutting concerns, allowing each to evolve independently. An *aspect weaver* composes the final system by combining the core and crosscutting modules through a process called *weaving*. Thus AOP modularizes the crosscutting concerns in a clear-cut fashion, yielding an implementation that is easier to design, implement, and maintain.

In this opening chapter, we examine the fundamentals of AOP, the problems it addresses, and why *you* need to know about it.

1.1 Managing concerns

A *concern* is a specific requirement or consideration that a system must address in order to satisfy the overall system goals. A *software system* is the realization of a set of concerns. A concern can be classified into one of two categories:

- A core concern captures the central functionality of a module.
- A crosscutting concern captures requirements that cross multiple modules.

Figure 1.1 shows how these concerns often interact in a typical application.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

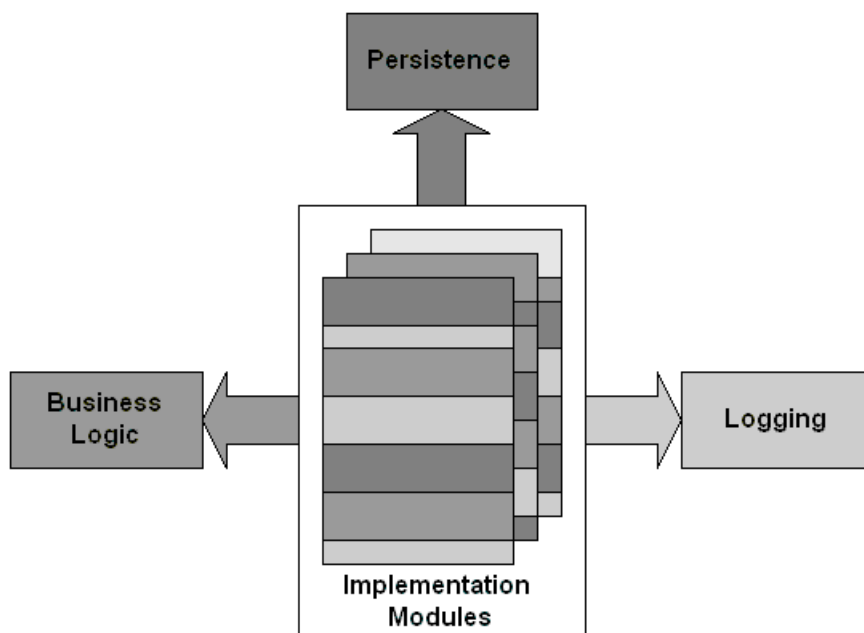


Figure 1.1: Viewing a system as a composition of multiple concerns. Each implementation module addresses some element from each of the concerns the system needs to address.

This figure shows how different modules in a system implement both core concerns and crosscutting concerns. This view portrays a system as a composition of multiple concerns tangled together by the current implementation techniques. This tangling makes it impossible to maintain the independence of the concerns. A pure object-oriented implementation of crosscutting concerns leads to two classic symptoms: code tangling and code scattering.

1.2 Symptoms of crosscutting concerns

We can broadly classify the symptoms of crosscutting concerns into two categories: code tangling and code scattering. If you see these symptoms in your system, it is most likely due to the conventional implementation of crosscutting concerns¹. Let's take a look at how you can recognize these symptoms.

¹ Note that code tangling and scattering may also stem from poor design and implementation (such as copy/pasted code). Obviously, you can fix such problems within the bounds of OOP. However, for crosscutting concern, in OOP the problem of crosscutting concerns is present even in well-designed systems.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

1.2.1 Code tangling

Code tangling is caused when a module is implemented to handle multiple concerns simultaneously. A developer often considers concerns such as business logic, performance, synchronization, logging, security, and so forth while implementing a module. This leads to the simultaneous presence of elements from each concern's implementation and results in code tangling. Figure 1.2 illustrates schematically the resulting code tangling in the module.

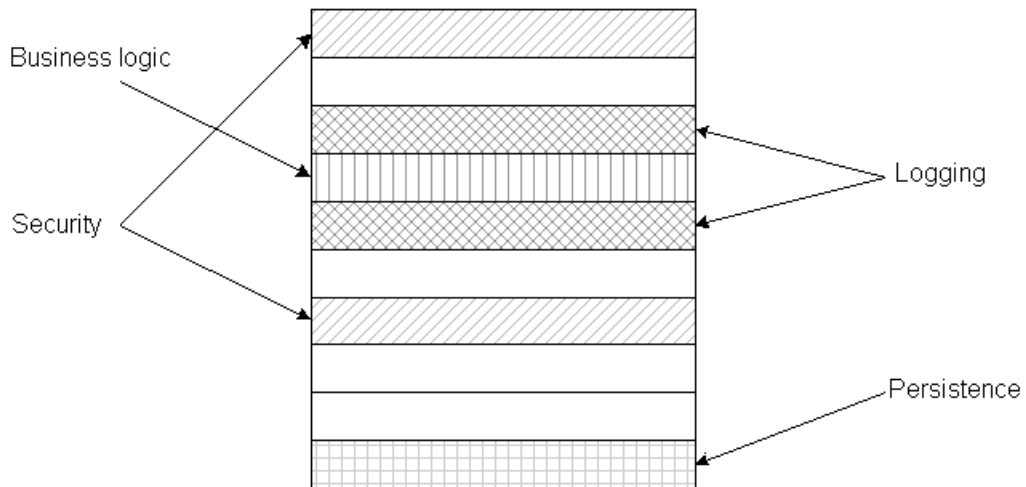


Figure 1.2: Code tangling caused by multiple simultaneous implementations of various concerns. The figure shows how one module manages a part of multiple concerns.

Let's illustrate the same idea through a code snippet. Consider the following skeleton implementation in listing 1.1 of a class that encapsulates some business logic in a conventional OOP way.

Listing 1.1: Business logic implementation along with crosscutting concerns

```
public class SomeBusinessClass extends OtherBusinessClass {
    ... Core data members
    ... Log stream #5
    ... Cache update status #3
    ... Concurrency control lock #2

    ... Override methods in the base class

    public void someOperation1(<operation parameters>) {
        ... Ensure authorization #1

        ... Lock the object to ensure thread-safety #2

        ... Ensure cache is up-to-date #3
    }
}
```

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

```

... Start transaction #4
... Log the start of operation #5
... Perform the core operation
... Log the completion of operation #5
... Commit or rollback transaction #4
... Unlock the object #2
}
... More operations similar to above addressing multiple concerns
}
#1 Security check
#2 Concurrency control
#3 Caching
#4 Transaction management
#5 Tracing

```

While the details will vary, the listing shows a common problem many developers face: there is a conceptual separation between multiple concerns at design time, but implementation tangles all those together. Such implementation also breaks the Single Responsibility Principle (SRP)² by making the class responsible for implementing core and crosscutting concern. It also forces changes to the class if you need to change invocation of the code related to crosscutting concern, you need to change the classes. This breaks the Open/Close principle³—open for extension, but closed for modifications.

Another way to look at code tangling is to use the notion of a multi-dimensional concern space. You can imagine that you are projecting requirements of the system onto an N-dimensional concern space, with each concern forming a dimension. The significance of this kind of system view is it shows us that each concern in this multidimensional space is mutually independent and therefore can evolve without affecting the rest. For example, changing the security requirement from one kind of authorization scheme to another should not affect the business logic. However, as we see in figure 1.3, a multi-dimensional concern space collapses into the code that implements the concerns is a continuous flow of calls and, in that sense, is one-dimensional. Such a mismatch results in an awkward mapping of the concerns to the implementation.

² See <http://www.objectmentor.com/resources/articles/srp.pdf> for more details.

³ See <http://www.objectmentor.com/resources/articles/ocp.pdf> for more details.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.
Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

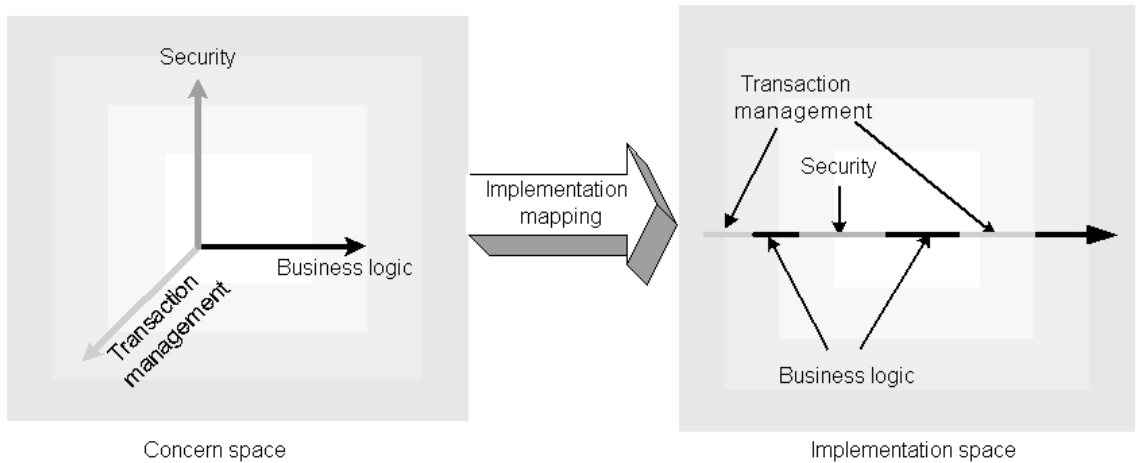


Figure 1.3: Mapping the N-dimensional concern space using a one-dimensional language. The orthogonality of concerns in the concern space is lost when it is mapped to the one-dimensional implementation space.

Since the implementation space is one-dimensional, its focus is usually the implementation of the core concern that takes the role of the dominant dimension. The implementation of crosscutting concerns then collapse onto the dominant dimension, causing code tangling. While we may naturally separate the individual requirements into mutually independent concerns during the design phase, OOP alone does not allow us to retain the separation in the implementation phase.

A real system would consist of many classes similar to the one in listing 1.1. All of those will address an identical set of crosscutting concerns. This leads to another symptom of code scattering.

1.2.2 Code scattering

Code scattering is caused when a single functionality is implemented in multiple modules. Since 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 would implement 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 such as “check permission”. The code for checking permission is scattered across multiple modules and there is no one place to identify the concern. The overall effect is an undesired tangling between all the modules needing security and the security module itself.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

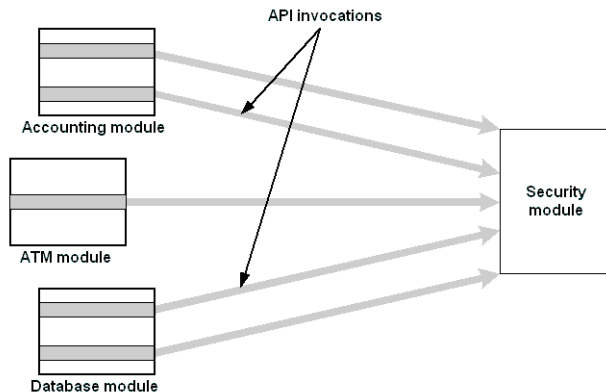


Figure 1.4: Implementation of a security concern using conventional techniques: The security module provides the API for authentication and authorization. However, the client modules—Accounting, ATM, and Database—each still need to embed the code to invoke the API to, say, check permission.

At the minimum code tangling and code scattering cause poor traceability and code duplication. However, there are many other implications as we study in the next section.

1.2.3 Implications of nonmodularity

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. While we will discuss each implication separately, they strongly affect one another. For example, poor traceability contributes to lower productivity and poor quality:

- *Poor traceability*—Simultaneous implementation of several concerns obscures the mapping of the concern to its implementation. This causes difficulty in tracing requirements to their implementation, and vice versa. For example, you would have to potentially examine all modules to trace the implementation of an authentication concern.
- *Lower productivity*—Simultaneous implementation of multiple concerns also shifts the focus from the main concern to the peripheral concerns. The lack of focus then leads to lower productivity as developers are sidetracked from their primary objective in order to handle the crosscutting concerns. Further, since different concern implementations may need different skill sets, either several people will have to collaborate on the implementation of a module or the developer implementing the module will need knowledge of each domain. The more concerns you implement together, the lower your probability of focusing on any one thing.
- *Lower code reuse*—If a module is implementing multiple concerns, other systems requiring similar functionality may not be able to readily use the module due to a

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

different set of concerns they might need to implement. Consider a service implementation. One project may need one form of secured access to the service, another may need a different form, and still another may need no security at all. The service implementation that directly includes a form of security implementation cannot be used in other project, even though the requirements of the service logic remain the same.

- *Poor quality*—Code tangling makes it more difficult to examine code and spot potential problems, and performing code reviews of such implementations is harder. For example, reviewing the code of a module that implements multiple concerns will require the participation of an expert in each of the concerns. Often not all of them are available at the same time, and the ones who are may not pay sufficient attention to the concerns that are outside their area of expertise.
- *Difficult evolution*—An incomplete perspective and limited resources often result in a design that addresses only current concerns. When future requirements arise, they often require reworking the implementation. Because implementation is not modularized, this may mean modifying many modules. Modifying each subsystem for such changes can lead to inconsistencies. It also requires spending considerable testing effort to ensure that this implementation change does not introduce regression bugs.

All of these problems lead to a search for better approaches to architecture, design, and implementation. Aspect-oriented programming offers one viable solution. In the next section, we introduce you to AOP. Later in this chapter, we will examine alternatives to AOP as well.

1.3 It's all about modularization

In OOP, the core modules can be loosely coupled through interfaces, but there is no easy way of doing the same for crosscutting concerns. This is because a concern is implemented in two parts: the server-side piece and the client-side piece. (The terms *server* and *client* are used here in the classic OOP sense to mean the objects that are providing a certain set of services and the objects using those services. They should not be confused with networking clients and servers.) OOP modularizes the server part quite well in classes and interfaces. However, when the concern is of a crosscutting nature, the client part, consisting of the requests to the server, is spread over all of the clients.

As an example, let's look at a typical implementation of a crosscutting concern in OOP: an authorization module that provides its services through an abstract interface. The use of an interface loosens the coupling between the clients and the implementations of the interface. Clients who use the authorization services through the interface are oblivious to the exact implementation they are using. Any changes to the implementation they are using will not require any changes to the clients themselves. Likewise, replacing one authorization implementation with another is just a matter of instantiating the right kind of implementation. The result is that one authorization implementation can be replaced with another with little or no change to the individual client modules. This arrangement, however, still requires that each

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

client have the embedded code to call the API. Such calls will need to be in all the modules requiring authorization and will be mixed in with their core logic.

Using AOP, none of the core modules will contain calls to check permission. Figure 1.5 shows the AOP implementation of the same security functionality shown in figure 1.4. The security concern (implementation and invocation) now resides inside the security module and the security aspect.

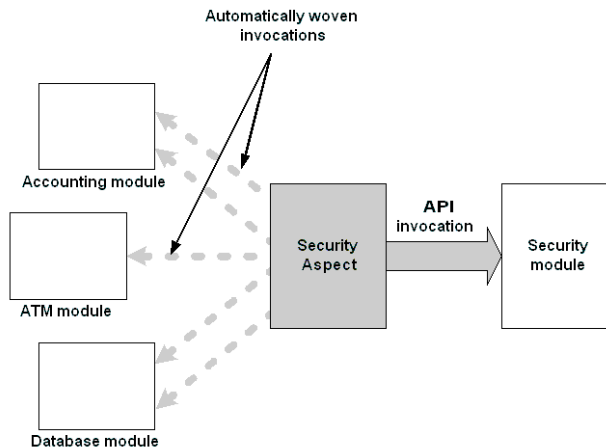


Figure 1.5: Implementation of a security concern using AOP techniques: The security aspect defines the interception points needing security and invokes the security API upon the execution of those points. The client modules no longer contain any security-related code.

The crosscutting security requirements are now mapped directly to just one module—the security aspect. With such modularization, any changes to the crosscutting security requirements affect only the security aspect, isolating the client modules completely. For now, don't worry about the way in which AOP achieves this. That will be explained in section 1.4.

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

1.4 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. As with any methodology, it can be implemented in various ways. For example, one realization of the OOP methodology specification consists of the Java language and tools such as the compiler. In a similar manner, each realization of AOP

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

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. This is commonly accomplished by a compiler or a runtime agent.

1.4.1 The AOP language specification

Any implementation of AOP must specify a language to implement the individual concerns and a language to implement the rules for weaving the concern implementations together. While I talk in terms of two separate languages, an AOP system may not 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. For example, a module that implements the core part of the caching concern will maintain a collection of cached objects, manage the validity of the cached objects, and ensure 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 integrate the implemented concerns in order to form the final system. For example, once you implement the core part of the caching concern in a module (perhaps through a third-party class library), you need to 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 concern in for listing 1.1, you can specify to log all the public operations in the system in just a few lines of code. For example, here is 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 actually modifying each public operation to add logging code. Since the tracing concern is modularized away from the class, it may focus only on the core concern as follows.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

```

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’ point of view—have been removed and only the core business logic remains. As we will see in the next section, an AOP implementation will combine the classes and aspects to produce a woven executable.

Weaving rules can be very general or very specific in the ways they interact with the core modules. For example, in the previous logging example, the weaving rules did not 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 that is to be applied to only a specific methods, such as the credit and debit operations in an `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 for specifying weaving rules could 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 core Java language, or it could use a separate XML-based language to express weaving rules.

1.4.2 The AOP implementation

The AOP 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.

The weaver can be implemented in various ways. A simple way is through source-to-source translation. Here, the weaver processes source code for individual classes and aspects and produces woven source code. A regular language compiler may then process the formed code. The aspect compiler then feeds this converted code to the base language compiler to produce the final executable code. Using this approach, a Java-based AOP implementation converts individual source input files into woven Java source code and then lets the Java compiler convert it into the byte code (in fact, this implementation technique was used in early implementations of AspectJ). This simple approach suffers from several drawbacks as the executable code cannot be easily traced back to the source files written. For example, stack traces would indicate line numbers in woven source modules.

Another approach could be that the source code would first be compiled into class files using the base language compiler. The class files would then be fed to the aspect compiler, which would weave the aspects into the class files to produce woven class files. Figure 1.6 shows a schematic of a compiler-based AOP language implementation.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

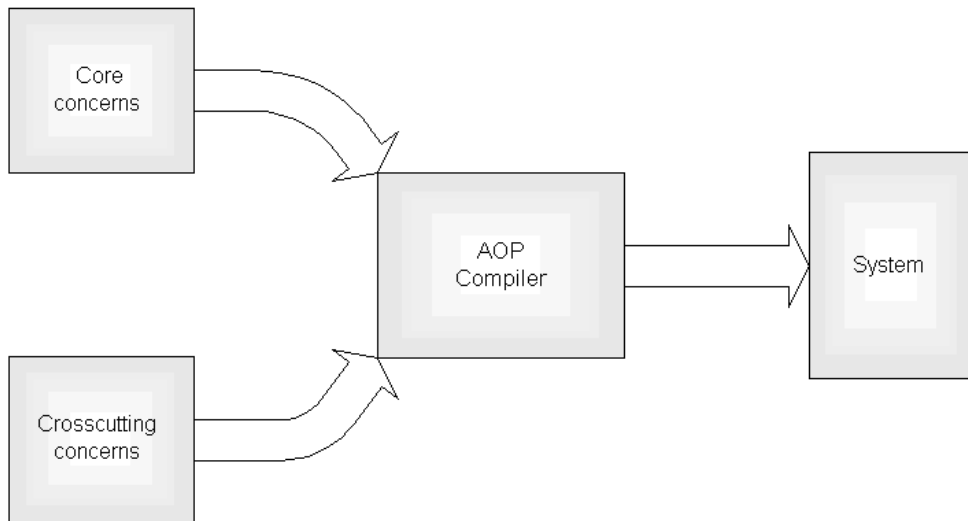


Figure 1.6 An AOP language implementation that provides a weaver in the form of a compiler. The compiler takes the implementation of the core and crosscutting concerns and weaves them together to form the final system.

It may also be possible to push the weaving process close to execution of the system. For example, if the implementation of AOP is Java-based, a special class loader or a VM agent could perform the weaving. Such an implementation will first load the byte code for the aspects, weave them into the classes as they are being loaded, and supply those woven versions of the classes to the underlying virtual machine (VM).

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

So far, we have looked at the mechanics of an AOP system. Now we take a look at the fundamental concepts behind AOP.

1.5 Generic model for AOP systems

By now, it should be clear that AOP systems help in modularizing crosscutting concerns. However, so are many other solutions, such as direct byte code manipulation tools, for example, CGLIB or ASM, direct use of the proxy design pattern, or even meta-programming. How do you recognize AOP from all these other options? To find out, we need to distill core characteristics of AOP systems into a generic model. If a system fits that model, it is an AOP system; otherwise, it is not.

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

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

1. Identifiable points in the execution of the system: Such points may include execution of methods, creation of objects, or throwing of an exception. 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, since join points are simply points during execution of a system. AOP merely gives identifies and categorizes these points.
2. A construct for selecting join points: Implementing a crosscutting concern will require selecting a specific set of join points. For example, the tracing aspect discussed earlier needs to select only the public methods in the system. In AOP, the **pointcut** construct selects any join point that satisfies the criteria. This is similar to an SQL query selecting rows in database (we will compare AOP with databases in section 1.7.2). Pointcuts also collect context at the selected points. For example, a pointcut may collect method argument as context. The concept of join points and construct of pointcuts together form an AOP system's **join point model**. We will study AspectJ's join point model in chapter 3.
3. A construct to alter program behavior: Once a pointcut selects join points, we need to augment those join points with additional or alternative behavior. For example, when implementing tracing, you need to log entry into the public methods. The **advice** construct in AOP provides a facility to do so. An advice adds behavior before, after, or around the selected join points. Around advice surrounds the join point execution and may execute it zero or more times. Advice is a form of **dynamic crosscutting** since it affects the execution of the system. We will study AspectJ's dynamic crosscutting implementation in chapter 4.
4. A construct to alter static structure of the system: Sometimes, to implement dynamic crosscutting effectively, you need to alter the static structure of the system. For example, when implementing tracing, you may need to introduce the logger field into each traced class. The **inter-type declaration** constructs make such modifications possible. Furthermore, in some situations, you may need to detect certain conditions, typically the existence of particular join points, before an execution of the system. The **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 will study AspectJ's static crosscutting support in chapter 5.
5. A module to express all crosscutting constructs: Since 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 will contain pointcuts, advice, and static crosscutting constructs. An aspect may be related to other aspects in similar way to a class relates to others. Furthermore, aspects are a part of the system and aspects use the system (for example classes in it) to get its work done. We will examine AspectJ's implementation of aspect in chapter 6.

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

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.
 Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

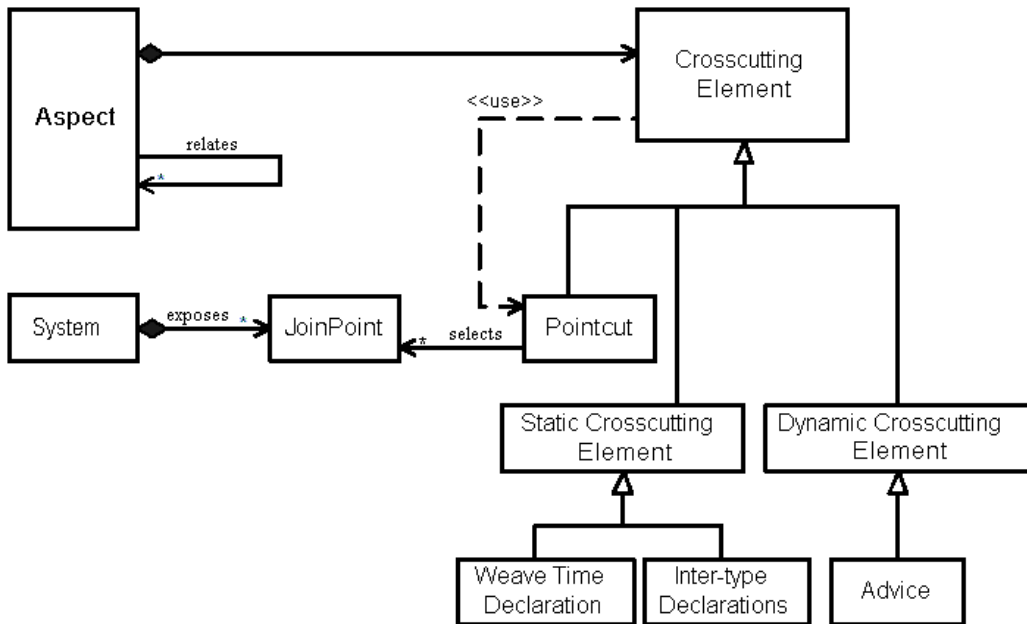


Figure 1.7: Generic model of AOP systems. Note that not every system will implement each part of the model.

Each AOP system may choose a subset of the model. For example, Spring AOP (discussed in chapter 9) does not implement weave-time declarations due to its emphasis on 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 is an alternative approach for solving the problem of crosscutting concerns. In the next section, we look at some commonly known alternatives to AOP.

1.6 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, there are many competitive technologies to deal with the same problem: frameworks, code generation, design patterns, and dynamic languages. Let's take a look at those alternatives. Note that, while I compare these techniques as alternative to AOP and (not surprisingly) show how AOP outshines each of them when it comes to dealing with crosscutting concerns, I do not mean that these techniques are useless. Each of these techniques is

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

appropriate for a set of problems. In fact, AOP can work alongside these techniques quite well. Furthermore, in some cases, AOP can enhance their implementation.

1.6.1 Frameworks

Frameworks such as servlets and EJB offer specific solutions to a focused set of problems. For example, the servlet specification offers a framework 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 common crosscutting concerns in that problem space. For example, the servlet framework provides basic support for intercepting HTTP requests.

Similarly, the EJB framework addresses a wide set of crosscutting concerns such as transaction management and security. In the EJB3 specification, it even provides limited support for interceptors, which, to an extent, match the goals of AOP. However, as we will see later, it falls short of being a complete solution.

Note that you may 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.

Framework approaches to crosscutting concerns often, but not always, boil down to either employing some form of code generation or implementing appropriate design patterns. Let's examine the two in more details.

1.6.2 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 themselves. These techniques represent powerful ways to deal with a wide range of problems and often are helpful in raising the level of abstraction. They can modularize crosscutting concerns by modifying the original code such as adding observer notifications or producing additional artifacts, such as automatic proxy classes, thus taking care of one of the drawbacks of a direct use of design patterns—manual modifications in many places.

A variation of code-generation works at the compiled code level. 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 rather small, given how directly source code maps to byte-code.

In Java 5, the annotation language feature allows attaching 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 a tool, the Annotation Processing Tool (APT), to simplify the process. However, APT forces understanding low-level details such as the syntax tree and that makes it difficult for a programmer to use it unless he acquires specific skills. It is no surprise that you hardly see

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

many non-framework programmers using APT. AOP, on the other hand, can provide simpler solutions to process annotations, as we will 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 a part of the overall AOP model. First, it provides a much simpler programming model making it easier for a developer to create modularized crosscutting implementations without knowing low-level details such as the abstract syntax tree. It essentially provides a domain-specific language (DSL) targeted towards crosscutting concerns. The user is isolated from byte-code manipulation mechanisms as well, which is not for the faint of heart. Furthermore, by limiting power, it nudges developers towards writing better code. In short, while code-generation is capable of doing anything AspectJ can do (and a whole lot more), AspectJ brings a level of discipline that is essential to good software engineering when it comes to dealing with crosscutting concerns.

1.6.3 Design patterns

Design patterns provide well-understood solutions to recurring problems. Some of the recurring problems are due to crosscutting concerns. This should be a big surprise, since while AOP is only about 10 years old, the crosscutting problem existed for as long as we have been creating software systems. In this section, we take a comparative view of some of the design patterns—observer, chain of responsibility, decorator and proxy, as well as interceptor—that help with crosscutting concerns. You will see that there are quite a few similarities and in fact, you can view a few design patterns as a “poor man’s” AOP implementation.

OBSERVER

A classic technique in UI programming is to respond to events such as mouse move and key pressed. For example, you may show a dialog box when a key is pressed. Even on server-side enterprise applications, message-oriented architectures involve sending and responding to messages. The observer design pattern forms the basis for the event programming to decouple the event source (the subject) from the event responder (the observer). 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 change. The notification method iterates over all the observers and calls a method on each (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. The key point of the observer pattern is that the subject knows nothing about which particular objects are observing it. The pattern also supports an arbitrary number of observers.

AOP’s advice may superficially look like an event responder. However, there are some important differences. First, there is no explicit firing of “events” in AOP. In other words, you won’t see invocations such as `notify<ChangeType>()` thus decoupling of all observer pattern logic from the subject class. Second, the context collected by pointcuts (equivalent to information carried by an “event” object) is a lot more flexible and powerful in AOP. Pointcuts

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

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, as shown in figure 1.8, puts a chain of processing objects in front of a target object. Before or after invoking the command object, processing objects may perform additional work or interrupt the chain.

Successful use of the COR pattern has two prerequisites: there is only one (or a small number) of target methods and the associated framework already supports 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 the framework itself. In this setup, some coarse grained crosscutting concerns—ones that deal at HTTP request level—may be modularized into servlet filters. However, anything that needs to go beyond the `doService()` method, filter offers no solution.

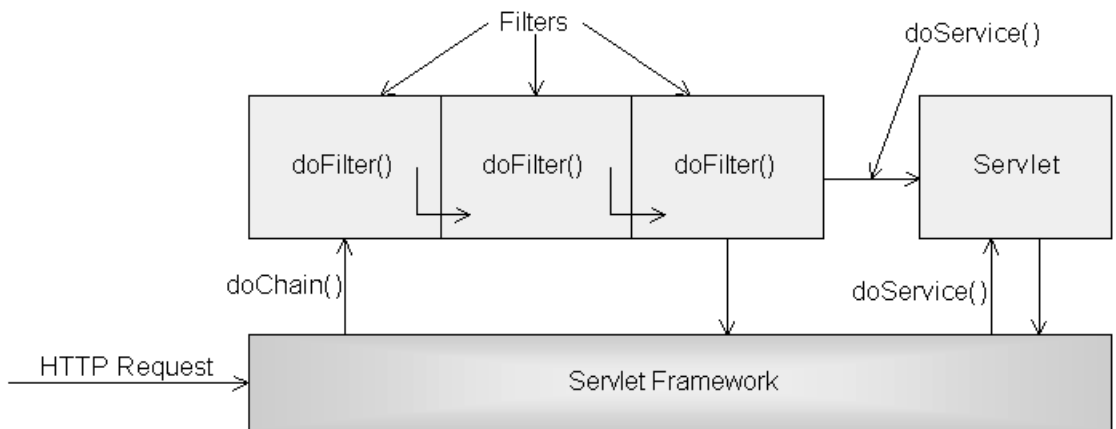


Figure 1.8: Chain of responsibility as implemented by the servlet framework. The filter chain allows applying additional logic such as coarse-grained security around the `doService()` method.

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

DECORATOR AND PROXY

The decorator and proxy⁴ design pattern use a wrapper object that can perform some work before, after, or around invocation of the wrapped object or its representation. This additional

⁴ I lump these two patterns together, as the distinction between the two isn't significant from the AOP perspective. A decorator holds onto a real object that it decorates, whereas proxy may

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

work can be crosscutting in nature. For example, each method may perform a security check before the wrapped object's method is invoked.

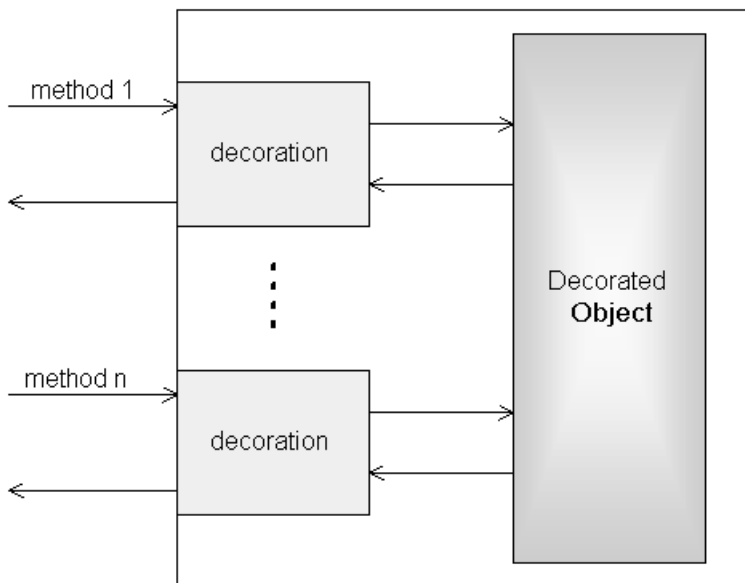


Figure 1.9: The decorator design pattern. The original object is wrapped in the decorator that presents the same interface as the decorated object. Each method passes through the decoration, which can implement functionality such as security and transaction management.

A direct use of decorator and proxy design patterns for crosscutting concerns implementation require substantial effort. However, these patterns may be used as the underlying implementation technique as a part of an AOP system. The Spring Framework, as we will see in chapter 9, makes use of the proxy design pattern internally to avoid exposing the pattern to the users. This isn't unlike the byte-code manipulation technique—cumbersome as a programming technique to deal with crosscutting concerns, but a perfectly fine underlying technology to implement AOP systems.

Another design pattern, interceptor, is often used along with the proxy design pattern. Let's compare how it stands up against AOP for crosscutting concerns.

INTERCEPTOR

The interceptor pattern allows expressing crosscutting logic into an interceptor object. By attaching the interceptor to specific program elements, you can have the interceptor logic invoked around the intercepted elements. This pattern when used along with the proxy or

not necessarily hold any real object, but simulates holding one. In a way, proxy is a generalization of decorator design pattern.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

decorator design pattern offers a reasonable solution for a wide range of crosscutting problems. For example, Java supports creation of dynamic proxies, which can be configured with an interceptor. Implementing the interceptor pattern generically and successfully requires a fair amount of machinery and thus is best left 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: transaction management and role-based security, primarily. 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 such method with the `@Interceptors` annotation. On the other extreme, you can declare an interceptor as a default interceptor, which applies to all beans, except those that opt-out. EJB3's implementation has a few limitations, such as 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 a bit complex and type-unsafe as access to the intercepted context (intercepted bean, method name, method arguments) are accessed through the `InvocationContext` object whose method returns `Object` and may require casting before using.

However, the real problem with the EJB interceptor design (and many other similar interceptor implementations) is the missing key abstraction of pointcuts. Instead of declaring classes and methods that need to be intercepted, classes and methods need to declare that they need to be intercepted, reducing it to a more macro-like usage. As a result, while the logic equivalent to AOP's advice is modularized, the pointcut equivalent logic is spread in all intercepted types. Further, due to generic nature of join point context, the interceptor method may need complex logic to pluck arguments from the correct place.

Note that the Spring framework, in versions prior to 2.0, used a mechanism similar to `InvocationContext` thus suffering from programming complexities similar to EJB3 interceptors. However, Spring's AOP always used a notion of pointcut to avoid the problem of spreading selection logic in multiple places. Furthermore, the AspectJ integration introduced in

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

Spring 2.0 removes the need for `InvocationContext`-like logic and raises the pointcut implementation to a new level, as we will see in chapter 9.

1.6.4 Dynamic languages

Dynamic languages have recently gained popularity and rightly so. Especially when combined with frameworks based on the languages, they can provide powerful solutions for a set of problems. For example, Ruby when combined with Rails or Groovy when combined with Grails provide simpler solutions for certain kinds of web applications. Each dynamic language offers a meta-programming facility that allows modifying the structure and behavior of a program during its execution. The meta-programming facility may modularize the advice portion of a crosscutting concern. For example, you may modify an existing method's implementation to wrap it with code that performs the crosscutting functionality before or after dispatching the original method.

While meta-programming is a fine tool for dealing with crosscutting concerns, there are a few considerations. First, such usage requires switching to a dynamic language. The static vs. dynamic languages "war" hasn't been concluded yet, nor it will be concluded anytime soon. So you will have to make a considered choice here. Second, meta-programming may be too powerful a tool for many programmers and a more disciplined approach might be more appropriate. Third, due to its constrained expression power, it is probably easier to create AOP tooling that helps in understanding the crosscutting structure. Such tooling 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 a project to bring AOP to Ruby through the Aquarium project (<http://aquarium.rubyforge.org>). It shows that, while AOP is popular in statically-typed languages, it has a role to play even in dynamically-typed languages. Interestingly, as seen from this project, it is relatively easy to build AOP capabilities on the 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 is instructive to note that the father of AOP, Gregor Kiczales, wrote the "The Art of the Metaobject Protocol (The MIT Press, 1991)" book. He clearly knows meta-programming! He still thought that AOP is better suited for crosscutting concerns and a direct application of meta-programming.

In a way, statically typed languages use AOP on the way to more comprehensive meta-programming support in order to break free from the limitations posed by the language, whereas dynamic languages benefit from AOP as a "disciplined" application of meta-programming.

1.7 AOP by analogy

It sometimes helps in understanding a new technology if we compare it with existing technologies. In this section, we attempt to understand AOP by comparing it with Cascading Style Sheet (CSS), database programming, and event-oriented systems. The purpose of this

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

section is to help those familiar with at least one of these technologies to understand AOP by analogy. Therefore, we will not consider all the details of these other technologies.

1.7.1 Cascading Style Sheet (CSS)

CSS is a widely supported mechanism to separate content from presentation in HTML pages. Without CSS, you would have to fuse content with formatting (tangling) and similar elements will have presentational information spread into multiple places (scattering). CSS helps the situation by letting the main document focus on content with formatting separated into a separate document called a stylesheet. A core concept in CSS is a *selector* that selects document elements matching a certain specification. For example, you can select paragraphs inside the body element by using the `body p` selector. You can associate presentational information with a selector for example, you can set background color of such elements to blue using `body p {background: blue;}`.

AOP acts on classes in the same way that CSS acts on documents. AOP let's you separate crosscutting logic from the main-line logic. AOP's pointcuts have the same selection role as CSS selectors. While 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 the structure such as `body p`. Therefore, it is a common practice to supplement content element with additional metadata through the `class` attribute. For example, you may mark a 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 find the same problem—selection through a pointcut often requires information beyond merely relying on inherent characteristics of program elements such as class and method names. Hence, the use of Java 5 annotations plays a similar role to the class attribute in HTML documents. You may, for example, mark a method as `@Transactional`, and utilize that in a pointcut expression.

There are similarities from the adoption perspective as well. Through WISIWYG HTML editors, it is easy to create a good-looking HTML. That apparent simplicity led to many initial web documents embedded with formatting information. However, upon realizing that it is difficult to create a consistent look as every element has its formatting specified independently, many developers started to look at CSS favorably. AOP has seen 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. However, you soon realize that creating a consistent implementation is nearly impossible when similar code is scattered in many places. Furthermore, the use of CSS requires a level of expertise and understanding of semantics separated from presentation elements. The use of AOP requires similar understanding of semantic separation between core and crosscutting elements.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

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

1.7.2 Database systems

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

SQL AND POINTCUT

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 the 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 may use a trigger to audit changes in certain tables. The following snippet calls the `logInventoryIncrease()` procedure upon increase in the count column of the inventory table.

```
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. However, 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. That said, AOP doesn't necessarily obviate the need for database trigger for reasons such as performance as well as bringing uniformity across multiple applications accessing the same tables.

Like database triggers, event-oriented programming includes the notion of responding to events that we compare with AOP next.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

1.7.3 Event-oriented programming

Event-oriented programming is essentially the observer design pattern that we discussed earlier in section 1.6.3. Each interesting code site notifies the observers by firing events and the observers respond by taking appropriate action, which may be crosscutting in nature.

One way to look at AOP is that the program is woven with logic to “fire” *virtual* events and “respond” to the events with an action that corresponds to the crosscutting concern it is implementing. Note, however, an important difference: Unlike event-based programming, there is no explicit creation and firing of events and you won’t see any code related to them. The mere execution of a part of the program constitutes the virtual event generation. Also, event systems tend to be more coarse grained than what an AOP solution may implement.

Note that you can combine event-oriented programming with AOP quite effectively. 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.

1.8 The lay of the land

For years now, many theorists have agreed that the best way to create manageable systems is to identify and separate the system concerns. This general topic is referred to as “separation of concerns” (SOC). In a 1972 paper, David Parnas proposed that the best way to achieve SOC is through modularization—a process of creating modules that hide their decisions from each other. In the ensuing years, researchers have been studying various ways to manage concerns. OOP provided a powerful way to separate core concerns. However, it provides no good solution when it came to crosscutting concerns. This is where AOP comes into the play.

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. However, AOP is a methodology with multiple 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.

1.8.1 AspectJ

Gregor Kiczales led the team at Xerox that created AspectJ, one of the first practical implementations of AOP, in the late 1990s. Later, Xerox transferred the AspectJ project to the open source community at eclipse.org. After this transfer, many AspectJ developers were employees of IBM, lead by Adrian Colyer. Later, Adrian left IBM to join SpringSource (where I work, as well). Recently, Andy Clement, a lead AspectJ developer, also joined SpringSource. Currently AspectJ is a SpringSource portfolio project and enjoys its sponsorship.

Until a few years back, AspectJ had a close cousin—AspectWerkz. This AOP system developed by Jonas Boner and Alexandre Vasseur followed the core AspectJ model except that

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

it used metadata expressed through Javadoc annotations, Java 5 annotations, or XML elements, in place of language extensions. In AspectJ5, AspectWerkz merged with AspectJ offering developers a choice of technologies including a new `@AspectJ` (pure Java 5 annotation-based) syntax. We will study that syntax in chapter 8.

AspectJ's primary tool support is an Eclipse plugin, AJDT. One of AJDT's most important features is a tool for visualization of crosscutting, which is helpful for debugging pointcut specifications. While developers write classes and aspects separately, they can visualize their combined effect even before the code is deployed.

The AspectJ language has an alternative implementation called 'abc'—the AspectBench compiler. The focus of this project is to provide a flexible implementation to support experimenting with new AspectJ language and optimization ideas.

1.8.2 Spring AOP

Spring is the most popular lightweight framework for enterprise applications. Spring, to satisfy the needs of enterprise applications, includes an AOP system based on interceptors and the proxy design pattern. Earlier implementation 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. This enables Spring users to write their custom aspects without much difficulty. We will examine how Spring uses AspectJ in detail in Chapter 9 and through examples of the combination throughout the book.

Like AspectJ, Spring AOP, through the Spring IDE (an Eclipse plugin) provides support for visualizing crosscutting inside the IDE.

There are many other implementations of AOP in Java such as dynaAOP, Nanning Aspects, AOPSYS, and (a part of) qi4j framework. JBoss, an open source application server, also offers an AOP solution that includes a pointcut language similar to that of AspectJ. There is also the AOP Alliance API implemented in frameworks such as Guice and Seasar. Incidentally, Spring has been offering a programming model based on the AOP Alliance API. However, that model is now relegated as a transitional technology due to the availability of the AspectJ-based model. There are also AOP systems for other languages, many inspired by AspectJ: Aquarium for Ruby, AspectC# for C#, and AspectC++ for C++.

1.9 Cost and benefit of AOP

Nothing comes free! Software engineering, like any engineering discipline, is all about optimizing cost and benefits. AOP isn't free either. Critics of AOP often talk about the difficulty of understanding it. And indeed AOP takes some time, patience, and practice to master. However, the main reason behind the difficulty is simply the newness of the methodology. After all, when was the last time a brand-new programming methodology was accepted without its share of adaptation resistance? AOP demands thinking about the system design and implementation in a new way. When you use AOP, while you get a lot of benefits, there are some costs that you must understand in order to make informed decisions. In this section, we first discuss the costs and then the benefits of AOP.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

1.9.1 Cost of AOP

Some of the costs associated with AOP are the usual suspects associated with any new technology: investment in learning it, ability to hire skilled programmers, and adoption path to ensure that you don't risk the project by overextending yourself, modifications of build and other development processes, and the availability of tools. There are well-understood mitigation techniques for each of these issues: 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), and following a gradual adoption path that we show in the second part of the book. However, there is one cost, while still common to most new technologies, which deserves more in-depth treatment: the cost of abstraction.

1.9.2 Cost of abstraction

Abstraction is a mechanism to allow focusing on what is essential, while hiding inessential details. Abstraction reduces apparent complexity of the underlying problem by creating well-isolated modules. Since each module represents a much smaller subsystem, it offers a way to contain complexity to a level where developer can cope with it with ease. Modularity is a "divide and conquer" approach to managing complexity.

Yet, abstraction introduced by AOP isn't without costs. Let's understand what these costs entail.

- **Higher skills requirement:** Creating a right level of abstraction is a highly skilled job (there are many right abstractions possible in a given system). A thorough understanding of the cost and benefit is a hallmark of a good software engineer. Mere understanding of implementation mechanisms won't yield useful abstractions. In the context of AOP, you must understand how to fit in the new unit of modularity—aspects—into the system. For that you will need to 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.
- **Complex program flow:** Abstraction, by its very nature, hides the details. In software systems, higher levels of abstraction always mean less information is available at the code level alone. Merely looking at a code segment isn't going to tell the whole story that will unfold during system execution. For example, in OOP, due to polymorphic methods, you cannot tell the exact method that will be executed at runtime as the choice of method will be based on the actual type of the object, not just 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 is not static and requires some effort to be understood.

AOP abstracts away program flow even further. You may not know (except through good tooling support, and even then not completely) that a crosscutting action will be taking place at certain part of the code. Many programmers new to AOP get stuck at

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

this point, until they realize that this separation of concerns is the whole point. If one insists on understanding exact program flow, that is a sign that he or she needs to reflect on the core ideas of AOP a bit longer. However, just as OOP required a few years of practice to understand the underlying core ideas, most developers get AOP eventually.

1.9.3 Benefits of AOP

Now that we know the cost, let's look at the benefits, which far outweigh the perceived costs. Among these benefits are:

- *Cleaner responsibilities of the individual module*—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 accessing a database is no longer responsible for pooling database connections as well. This results in cleaner assignments of responsibilities, leading to improved traceability.
- *Higher modularization*—AOP provides a mechanism to address each concern separately with minimal coupling. This results in modularized implementation even in the presence of crosscutting concerns. Such implementation results in a system with much less duplicated code. Because the implementation of each concern is separate, it also helps avoid code clutter. A modularized implementation results in an easier-to-understand and easier-to-maintain system.
- *Easier system evolution*—AOP modularizes the individual aspects and makes core modules oblivious to the aspects. Adding new functionality is now a matter of including a new aspect and requires no change to the core modules. Furthermore, when we add a new core module to the system, the existing aspects crosscut it, helping to create a coherent evolution. The overall effect is a faster response to new requirements.
- *Late binding of design decisions*—Architect of a system is often faced with underdesign/overdesign issues. If they underdesign, they may have massive changes later in the development cycle. If, on the other hand, they overdesign, the implementation may be burdened with code of questionable usefulness. With AOP, the architect can delay making design decisions for future requirements because it is possible to implement those as separate aspects. Architects can now focus on the current core requirements of the system. New requirements of a crosscutting nature can be handled by creating new aspects. Further, AOP works in harmony with one of the most popular trends of Extreme Programming (XP) by supporting the practice of “You aren't gonna need it” (YAGNI). This is a result of the observation that implementing a feature just because you may need it in the future often results in wasted effort because you won't actually need it. Now with AOP, you can practice YAGNI, and if you do need a particular kind of functionality later, you can implement it without system wide modifications.
- *More code reuse*—The key to greater code reuse is a more loosely coupled

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

implementation. Because AOP implements each aspect as a separate module, each module is more loosely coupled than equivalent conventional implementations. In particular, core modules aren't aware of each other—only the weaving rule specification modules are aware of any coupling. By simply changing the weaving specification instead of multiple core modules, you can change the system configuration. For example, a database module can be used with a different logging implementation without changes to either of the modules.

- *Improved time-to-market*—Late binding of design decisions allows a much faster design cycle. Cleaner separation of responsibilities allows better matching of the module to the developer's skills, leading to improved productivity. More code reuse leads to reduced development time. Easier evolution allows a quicker response to new requirements. All of these lead to systems that are faster to develop and deploy.
- *Reduced costs of feature implementation*—By avoiding the cost of modifying many modules to implement a crosscutting concern, AOP make it cheaper to implement the crosscutting feature. By allowing each implementer to focus more on the concern of the module and make the most of his or her expertise, the cost of the core requirement's implementation is also reduced. The end effect is a cheaper overall feature implementation.

1.10 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 exist to support this principle in some form. For instance, with OOP, by separating interfaces from their implementation, you can modularize the core concerns well. However, for crosscutting concerns, OOP forces the core modules to embed the crosscutting concern's logic. While the crosscutting concerns themselves are independent of each other, the use of OOP leads to an implementation that no longer preserves the 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 new ways of dealing with a software system by viewing it as a composition of mutually independent concerns. By building on the top of existing programming methodologies, it preserves the investment in knowledge gained over the last few decades.

Over the last few of years, AspectJ has become a very practical technology. It has been deployed in many organizations—small and big—to add powerful feature that one would have otherwise shied away from or implemented in a laborious manner. The success of AspectJ is in a big way due to its connection with Spring. By combining a lightweight container with a

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to ajja@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.

general-purpose AOP solution, the Spring framework offers an attractive package for enterprise software development.

In the next 8 chapters, we will study a specific implementation of AOP for Java, AspectJ, as well as its integration with Spring. The rest of the book will show specific examples that use this technology to solve real problems. If you are not yet convinced of the power of AOP, those examples will most definitely convince you.

Thanks for participating in AspectJ in Action 2nd edition Manning Early Access Program.

Please send your feedback to

ajia@ramnivas.com or at <http://www.manning-sandbox.com/forum.jspa?forumID=413>.