Improve modularity with aspect-oriented programming

AspectJ brings AOP to the Java language

Nicholas Lesiecki (ndlesiecki@apache.org)
Technical Team Lead
eBlox, Inc.

01 January 2002

Aspect-oriented programming (AOP) is a new programming technique that allows programmers to modularize crosscutting concerns (behavior that cuts across the typical divisions of responsibility, such as logging). AOP introduces aspects, which encapsulate behaviors that affect multiple classes into reusable modules. With the recent release of AspectJ by Xerox PARC, Java developers can now take advantage of the modularization AOP can provide. This article introduces AspectJ and illustrates the design benefits that result from its use.

Aspect-oriented programming (AOP) grew out of a recognition that typical programs often exhibit behavior that does not fit naturally into a single program module, or even several closely related program modules. Aspect pioneers termed this type of behavior crosscutting because it cut across the typical divisions of responsibility in a given programming model. In object-oriented programming, for instance, the natural unit of modularity is the class, and a crosscutting concern is a concern that spans multiple classes. Typical crosscutting concerns include logging, context-sensitive error handling, performance optimization, and design patterns.

If you have ever worked with code that addresses a crosscutting concern, you know the problems that result from a lack of modularity. Because the implementation of a crosscutting behavior is scattered, developers find the behavior difficult to reason about, implement, and change. Code for logging, for instance, intertwines with code whose primary responsibility is something else. Depending on the complexity and scope of the concern being addressed, the resulting tangle can be anywhere from minor to significant. Changing an application's logging policy could involve hundreds of edits -- an irritating, if doable, task. At the other end of the spectrum lies one of the founding cases of aspect-oriented programming. In an article entitled "Aspect-Oriented Programming," some of AspectJ's authors write about performance optimizations that drove a 768-line program into 35,213 lines. Rewritten with aspect-oriented techniques, the code shrank back to 1,039 lines while retaining most of the performance benefits.

AOP complements object-oriented programming by facilitating another type of modularity that pulls together the widespread implementation of a crosscutting concern into a single unit. These units
are termed aspects, hence the name aspect-oriented programming. By compartmentalizing aspect code, crosscutting concerns become easy to deal with. Aspects of a system can be changed, inserted or removed at compile time, and even reused.

**Learning by example**

To get a better feeling for aspect-oriented programming, let's look at the aspect-oriented extension to the Java programming language from Xerox PARC, AspectJ. In our example, we'll use AspectJ to do logging. The example is drawn from the open-source Cactus framework, which simplifies the testing of server-side Java components. The contributors to the framework decided to aid debugging by tracing all method calls within the framework. Version 1.2 of Cactus was written without AspectJ, and a typical method looked something like the one shown in Listing 1 below:

**Listing 1. Log calls manually inserted into every method**

```java
public void doGet(JspImplicitObjects theObjects) throws ServletException{
    logger.entry("doGet(...)");
    JspTestController controller = new JspTestController();
    controller.handleRequest(theObjects);
    logger.exit("doGet");
}
```

As part of the project's code conventions, every developer was asked to insert these lines into any method that he or she wrote. Developers were also asked to log each method's parameters. Without exhaustive code review on the part of the project's overseers, these sorts of conventions are difficult to enforce. In version 1.2, there were approximately 80 separate log calls spread across 15 classes. In version 1.3 of the framework, these 80 calls have been replaced with a single aspect that automatically logs both parameters and return values along with method entries and exits. Listing 2 contains a very simplified version of this aspect (for example, I have omitted the parameter and return value logging).

**Listing 2. Log calls automatically applied to every method**

```java
public aspect AutoLog{
    pointcut publicMethods() : execution(public * org.apache.cactus..*(..));
    pointcut logObjectCalls() :
        execution(* Logger.*(..));
    pointcut loggableCalls() : publicMethods() && ! logObjectCalls();
    before() : loggableCalls(){
        Logger.entry(thisJoinPoint.getSignature().toString());
    }
    after() : loggableCalls(){
        Logger.exit(thisJoinPoint.getSignature().toString());
    }
}
```
Let's break down the example and see what the aspect is doing. The first thing you'll notice is the aspect declaration. Aspects are similar to classes in their declaration, and they define Java types, just as classes do. In addition to its declaration, this aspect contains both pointcuts and advice.

**Pointcuts and join points**

To understand pointcuts, you'll need to know what a join point is. Join points represent well-defined points in a program's execution. Typical join points in AspectJ include method calls, access to class members, and the execution of exception handler blocks. Join points may contain other join points. For example, one method call may result in several other method calls before it returns. A pointcut, then, is a language construct that picks out a set of join points based on defined criteria. The first pointcut in the example, called `publicMethods`, selects all public method executions in the `org.apache.cactus` package. `execution` is a primitive pointcut (just as `int` is a primitive Java type). It selects the executions of any methods matching the signature defined within its parentheses. The signature is allowed to include wildcards; the one in the example contains several. A second pointcut called `logObjectCalls` selects all method executions in the `Logger` class. A third pointcut, `loggableCalls`, combines the two previous pointcuts using `&& !`, meaning that it selects all public methods in `org.apache.cactus` except those in the `Logger` class. (Logging the log methods would result in an infinite recursion.)

**Advice**

Now that the aspect has defined the points it should log, it uses advice to accomplish the actual logging. Advice is code that executes before, after, or around a join point. You define advice relative to a pointcut, saying something like "run this code after every method call I want to log." Hence the advice:

```
before() : loggableCalls(){
  Logger.entry(thisJoinPoint.getSignature().toString());
}
```

The advice uses the `Logger` class, whose entry and exit method look like this:

```
public static void entry(String message){
  System.out.println("entering method "+ message);
}
```

In the example, the `String` passed to the logger derives from `thisJoinPoint`, which is a special reflective object that allows access to the run-time context in which the join point executes. In the actual aspect used by Cactus, the advice uses this object to retrieve the method parameters passed into each logged method call. When the logging aspect is applied to the code, the results of a method call look something like this:

**Listing 3. Output of the AutoLog aspect**

```
entering method: void test.Logging.main(String[])
entering method: void test.Logging.foo()
exiting method: void test.Logging.foo()
exiting method: void test.Logging.main(String[])
```
Around advice

The Cactus example defines `before()` and `after()` advice. The third type of advice, `around()`, gives the aspect writer a chance to affect whether and when the join point is executed using the special `proceed()` syntax. The following (somewhat ornery) advice delivers the execution of the `say` method in the `Hello` class into the hands of chance:

```java
void around(): call(public void Hello.say()){
    if(Math.random() > .5){
        proceed();//go ahead with the method call
    }
    else{
        System.out.println("Fate is not on your side.");
    }
}
```

Developing with AspectJ

Now that you have better sense of what aspect code looks like, let's turn our attention briefly to the business of writing aspects. In other words, let's answer the question, "How can I get the above code to work?"

For aspects to affect regular class-based code, the aspects must be woven into the code they modify. To do this using AspectJ, you must compile your class and aspect code with the ajc compiler. ajc can operate either as a compiler or a precompiler, generating either valid .class or .java files, which you can then compile and run in any standard Java environment (with the addition of a small run-time JAR).

To compile with AspectJ, you will need to explicitly specify the source files (both aspect and class) that you want to include in a given compilation -- ajc does not simply search your classpath for relevant imports the way javac does. This makes sense because each class in a standard Java application acts as a somewhat isolated component. A class needs only the presence of classes that it directly references in order to operate correctly. Aspects represent aggregate behaviors that span multiple classes. Therefore, an AOP program needs to be compiled as a unit rather than one class at a time.

By specifying the files that are included in a given compilation, you can also plug and unplug various aspects of your system at compile time. For instance, by including or excluding the logging aspect described earlier from a compilation, the application builder can add or remove method tracing from the Cactus framework.

AspectJ is open source

Xerox makes AspectJ available under the Mozilla Public License. This is good news for open source enthusiasts. It's also good news for anyone wanting to adopt AspectJ in the near future: the product costs nothing, and you have the assurance of being able to examine the source code if you think you've found a critical bug. The open code base also means that AspectJ's source has received significant community review before hitting the market.

An important limitation to the current version of AspectJ is that its compiler can only weave aspects into code for which it has the source. In other words, you cannot use ajc to add advice to
precompiled classes. The AspectJ team regards this limitation as temporary, and the AspectJ Web site promises that a future version (officially 2.0) will allow bytecode modification.

**Tool support**

Several development tools are included as part of AspectJ’s release. This bodes well for AspectJ’s future, because it indicates a strong commitment on the part of the authors to making AspectJ friendly to developers. Tool support is particularly important for an aspect-oriented system because program modules can be affected by other modules of which they have no knowledge.

One of the most important tools released with AspectJ is a graphical structure browser that reveals at a glance how aspects interact with other system components. This structure browser is available as a plug-in for popular IDEs as well as a stand-alone tool. Figure 1 shows a view of the logging example discussed earlier.

**Figure 1. The graphical structure browser that ships with AspectJ reveals, among other things, which methods AutoLog advises**

![Graphical structure browser](image)

In addition to the structure browser and the core compiler, you can download an aspect-aware debugger, a javadoc tool, an Ant task, and an Emacs plug-in from the AspectJ Web site.

Let’s return to language features.

**Affecting class structure: Introduction**

Pointcuts and advice let you affect the dynamic execution of a program; *introduction* allows aspects to modify the static structure of a program. Using introduction, aspects can add new
methods and variables to a class, declare that a class implements an interface, or convert checked to unchecked exceptions.

An example of introduction
Let’s say that you had an object that represented a cache of persistent data. To gauge the “freshness” of the data, you might decide to add a timestamp field to the object so that you could easily detect whether the object and the backing store were out of sync. However, since the object represents business data, it makes sense to separate this mechanical detail from the object. With AspectJ, you could use the syntax shown in Listing 4 to add timestamping to an existing class:

Listing 4. Adding variables and methods to an existing class
```
public aspect Timestamp {
    private long ValueObject.timestamp;
    public long ValueObject.getTimestamp(){
        return timestamp;
    }
    public void ValueObject.timestamp(){
        //"this" refers to ValueObject class not Timestamp aspect
        this.timestamp = System.currentTimeMillis();
    }
}
```

You declare introduced methods and member variables almost exactly as you would regular class members, except that you must qualify which class you declare them on (hence ValueObject.timestamp).

Mix-in style inheritance
AspectJ allows you to add members to interfaces as well as to classes, allowing for mix-in style inheritance à la C++. If you wanted to generalize the aspect in Listing 4 so that you could reuse the timestamp code for a variety of objects, you could define an interface called TimestampedObject, and use introduction to add the same members and variables to the interface instead of to a concrete class, as shown in Listing 5.

Listing 5. Adding behavior to an interface
```
public interface TimestampedObject {
    long getTimestamp();
    void timestamp();
} //and
public aspect Timestamp {
    private long TimestampedObject.timestamp;
    public long TimestampedObject.getTimestamp(){
        return timestamp;
    }
    public void TimestampedObject.timestamp(){
        this.timestamp = System.currentTimeMillis();
    }
}
```
Now you can use the `declare parents` syntax to cause `ValueObject` to implement your new interface. `declare parents`, like other AspectJ type expressions, can apply to multiple types at once:

```
declare parents: ValueObject || BigValueObject implements TimestampedObject;
```

Now that you've defined the operations a `TimestampedObject` supports, you can use pointcuts and advice to automatically update the timestamp when the correct circumstances arise. We'll cover that in the next section because it illustrates how to access context in advice.

### Other AspectJ features

Using pointcuts, you can easily define situations in which a timestamped object should have its stamp updated. The additions to `Timestamp`, shown in Listing 6, stamp the object after any call to a setter method:

#### Listing 6. Accessing context in advice

```java
pointcut objectChanged(TimestampedObject object) :
    execution(public void TimestampedObject+.set*(..)) &&
    this(object);
/*TimestampedObject+ means any subclass of TimestampedObject*/
after(TimestampedObject object) : objectChanged(object){
    object.timestamp();
}
```

Notice that the pointcut defines arguments that the `after()` advice uses -- in this case, the `TimestampedObject` -- which has a setter called on it. The `this()` pointcut identifies all join points where the currently executing object is of the type defined within the parentheses. Several other kinds of values can be bound into advice arguments, including method arguments, exceptions thrown by a method, and the target of a method call.

### Custom compilation errors

I found custom compilation errors one of the coolest features of AspectJ. Suppose that you wanted to isolate a subsystem, so that client code would have to go through an intermediary to access worker objects (this situation occurs in the Facade design pattern). With the `declare error` or `declare warning` syntax, you can customize the ajc compiler's response to the potential occurrence of a join point, as shown in Listing 7.

#### Listing 7. Defining a custom error

```java
public aspect FacadeEnforcement {
    pointcut notThruFacade() : within(Client) && call(public * Worker.*(..));

declare error : notThruFacade():
    "Clients may not use Worker objects directly."
}
```
The within pointcut is similar to this(), except that ajc detects it entirely at compile time (most pointcuts can discriminate based on run-time information).

Error handling

I recognize the worth of checked exceptions in the Java language. However, I have frequently wished for a workaround, some sort of simple “turn this exception into a run-time exception” command. Many times, a method I'm writing has no meaningful response for an exception—and, chances are, neither does any of the method's potential users. I don't want to discard the exception, but I don't want to track its presence up through all of its callers either. There are artful ways to use try/catch blocks to accomplish this end, but none so elegant as declare soft in AspectJ. The class shown in Listing 8 attempts to do some SQL work.

Listing 8. A class with checked exceptions

```java
public class SqlAccess {
    private Connection conn;
    private Statement stmt;

    public void doUpdate(){
        conn = DriverManager.getConnection("url for testing purposes");
        stmt = conn.createStatement();
        stmt.execute("UPDATE ACCOUNTS SET BALANCE = 0");
    }

    public static void main(String[] args)throws Exception{
        new SqlAccess().doUpdate();
    }
}
```

If I don't use AspectJ or declare the exception in each method signature, I would have to insert try/catch blocks to deal with the checked SQLException (thrown from nearly every method in the JDBC API). With AspectJ, I can use the following inner aspect to automatically rethrow it as an org.aspectj.lang.SoftException:

Listing 9. Softening exceptions

```java
private static aspect exceptionHandling{
    declare soft : SQLException : within(SqlAccess);

    pointcut methodCall(SqlAccess accessor) : this(accessor) 
        && call(* * SqlAccess.*(..));

    after(SqlAccess accessor) : methodCall (accessor){
        System.out.println("Closing connections.");
        if(accessor.stmt != null){
            accessor.stmt.close();
        }
        if(accessor.conn != null){
            accessor.conn.close();
        }
    }
}
```
The pointcut and advice close the connection and statement after each method in the SQLAccess class, regardless of whether it throws an exception or returns normally. It’s probably overkill to use an error handling aspect for one method, but if I were to add any other methods that used the connection and statement, the error handling policy would also apply to them. This automatic application of aspects to new code is one of AOP’s key strengths: authors of new code do not have to be aware of crosscutting behaviors in order to participate in them.

**Conclusion**

Is AspectJ worth using? Grady Booch describes aspect-oriented programming as one of three movements that collectively mark the beginning of a fundamental shift in the way software is designed and written. (See his "Through the Looking Glass" in the Resources section.) I agree with him. AOP addresses a problem space that object-oriented and other procedural languages have never been able to deal with. Within a few weeks of my introduction to AspectJ, I’ve seen it provide elegant, reusable solutions to problems I thought were fundamental limitations of programming. It’s fair to say that AOP is the most powerful abstraction I’ve learned of since I began using objects.

Of course, AspectJ does have a learning curve. As with any language or language extension, it has subtleties that you need to grasp before you can leverage its full power. However, the learning curve is not too steep -- after reading through the developer’s guide and working through a few examples, I found myself ready to compose useful aspects. AspectJ feels natural, as if it fills in a gap in your programming knowledge rather than extending it in a new direction. Some of the AspectJ tools are a bit rough around the edges, but I haven’t encountered any major problems.

Given the power of AspectJ to modularize the un-modularizable, I think it's worth using immediately. If your project or your company aren't ready to use AspectJ in production, you can easily apply AspectJ to development concerns such as debugging and contract enforcement. Do yourself a favor and check out this language extension.
Resources

- You can download AspectJ and its associated tools from http://www.eclipse.org/aspectj/. The site also hosts an FAQ, mailing lists, excellent documentation, and links to other resources on AOP. It's a good place to begin further research.
- Grady Booch's "Through the Looking Glass" (Software Development, July 2001) discusses the future of software engineering and predicts the rise of multifaceted software, software that can be composed in multiple ways at once. The article cites AOP as one of the first multifaceted movements within programming.
- Another multifaceted approach, coming from the IBM Research team, is Hyperspaces. Hyperspaces goes beyond the encapsulation of crosscutting concerns and attempts to manage multi-dimensional separation of concerns. Hyper/J -- Java support for Hyperspaces -- offers on-demand remodularization of a system. Heady stuff.
- IBM Research is also focusing on subject-oriented programming, which offers subjects -- collections of classes or class fragments that have a unique "view" of a system. Subjects are composed into applications with composition rules.
- asod.net serves as a central source of information on AOP in general. The site offers links to other AOP initiatives, aspect implementations in other languages, aspect-like source modification, and articles on aspect theory.
- One of the founding papers on AOP, "Aspect Oriented Programming" (PDF), offers an early look at the development of a new way of programming. The example cited in the beginning of the article, regarding performance optimizations that required 35,000 lines of code without aspects and 1,000 lines with, was drawn from this paper.
- JavaWorld offers a short series introducing AOP, "I want my AOP."
- Eric Allen's Diagnosing Java Code column covers many "bug patterns" that result from inconsistent handling of crosscutting concerns. Allen discusses ways to minimize these with OO techniques. "Bug patterns: An introduction" covers a common type of bug emerging from repeated code -- which often can't be avoided in OOP. He mentions AspectJ specifically in "The Null Flag bug pattern" in connection with exceptions.
- The logging example in this article was drawn from Jakarta's Cactus project.
- Find other Java resources on the developerWorks Java technology zone.
About the author

Nicholas Lesiecki entered the world of Java programming during the dot-com boom and has since risen to prominence in the XP and Java communities with the publication of *Java Tools for Extreme Programming* -- a how-to manual for leveraging open source build and testing tools in agile processes such as XP. He currently leads development of eBlox, Inc.’s flagship online catalog system, storeBlox. In addition to making frequent speaking appearances at the Tucson JUG, he maintains active commiter status on Jakarta's Cactus project (a server-side unit-testing framework). Contact Nick at ndlesiecki@apache.org.

© Copyright IBM Corporation 2002

Trademark
(www.ibm.com/developerworks/ibm/trademarks/)