Aspects, Concerns and Java

James W. Cooper

I keep hearing about Aspect-oriented programming and every time I do, some sort of screen goes up in my mind and I find I am missing the point. I think this is because the originators of this valuable technique have chosen terms (Cross-cutting, Concerns, Aspects) that have such powerful common meanings that they drown out the new meanings they are imbuing the words with. For example, I can't hear "cross-cut" without thinking of the "Wells Fargo Wagon" number in the *Music Man*. "Montgomery Ward sent me a bathtub and a cross cut saw!"

Recently I heard a great talk by Erik Hilsdale from PARC (the organization formerly known as Xerox PARC) and the point of Aspect-oriented programming finally became much clearer to me. The essence of Aspect-oriented programming (AOP, if you will) is that in any OO program, even if it well-designed, there will be some method calls that appear in several places throughout the program. If you change the call to that method, or change the error handling of that method, you may need to make changes in all of the occurrences of this call. If the change is subtle, this may not be caught by the compiler, and could lead to annoying run-time errors that might be difficult to track down.

For a simple-minded example, consider a graphics program which can draw lines, squares and circles. You probably would organize such a program to use line objects, square objects and circle objects, each of which might have a draw method. In addition, you might have a screen update method that each of these objects would have to call, to refresh the actual display. Thus, even though the program is well structured in the classical OO sense, it would be nice if there was a view of the program where all of these separate calls could be treated as a single spot in the view. Then you could change and polish that code by making the change in a single place.

While our drawing example is necessarily quite simple and has simpler solutions, larger complex programs frequently have code that is not and cannot be modularized. Calls to some methods are often scattered throughout any otherwise well-designed program. Specifically calls to things such as program loggers must of necessity be scattered throughout your program. But in general, if you modularize some aspects of your program, it may not be possible to modularize others at the same time.

When code is scattered in different fragments throughout a program, it is hard to see its structure and hard to get a good view of the apparent tangling of the code. It's hard to change such code efficiently, and hard to find all the cases that have to be changed.

It would also be nice if there was a language where you could carry out this sort of viewing and changing efficiently The AspectJ language is a language for just this purpose. Furthermore, AspectJ is itself written in Java and therefore is available wherever Java runs. You can download AspectJ from their web site, www.AspectJ.org and can read any number of articles about the language that are provided there. In addition to the compiler code, there are plugins for Eclipse, JBuilder and emacs. The language was developed at PARC under a DARPA grant, and while the project at PARC is ending soon, the AspectJ organization will persist.

In AspectJ, each of the spots you would like to consolidate is called a *concern* and the process of developing this consolidation is called *crosscutting*. The specific spots you can consolidate or join are called *join points*. They can be

- method calls or constructor calls – happen before the object is called
- method or constructor execution
- Field get or set
- Exception handler execution
- Class initialization – when static initializers are called
- Object initialization – when dynamic initializers are called.

Pointcuts and Aspects

In AspectJ, you define *join points* by describing what sort of code you want to connect to, using a *pointcut* statement. Suppose we wanted to define all the spots where the setP1 or the setP2 methods of the Line object are called. We simply specify these calls and give them the name *move* in a *pointcut* statement:

```
pointcut move();        //names the spot
      call (void Line.setP1(Point))  //where this call occurs
            ||                       // OR
      call (void Line.setP2(Point)); //where this call occurs
```

Now we've defined the place where these moves occur. What can we do with them?

AspectJ allows you to specify *advice* or, in other words, code that should be executed before, after, around, after returning and after throwing exceptions. This advice is part of a special class called an *aspect*. An *aspect* is a class than can crosscut other classes: In the aspect below, we describe what should happen when any of the move pointcuts are called:

```
aspect MoveTracking {

      pointcut move() :
        call (void Line.setP1(Point)) ||
        call (void Line.setP2(Point)) ;

//code runs after each call
      after() returning move() {
            screen.update();
      }
}
```

The point of this is that your program can evolve more easily, with changes now possible in one single place.

When I first saw this code I found it really hard to grasp for some reason, because of the odd syntax and the whole idea of classes that cut across a program. But now that it has begun to sink in, I think this is a really powerful approach to the persistent problem of evolving programs with calls to lots of methods throughout. No matter how cleverly you construct your object hierarchy, you always end up with some calls that get scattered throughout your code.

Multi-Class Aspects

You can also specify aspects across a number of classes. In this example, the pointcut move includes calls to different methods in the Point, Line and FigureElement classes.

```
aspect DisplayUpdating {

  pointcut move():
    call(void FigureElement.moveBy(int, int)) ||
    call(void Line.setP1(Point))              ||
    call(void Line.setP2(Point))              ||
    call(void Point.setX(int))                ||
    call(void Point.setY(int));

  after() returning: move() {
    Display.update();
  }
}
```

The Killer Example – Logging

One of the most powerful examples Hilsdale presented in his talk was the case of calls to a logging program. We discussed logging in the January 2003 issue, and showed how to use both the log4j package and the built-in logging in Java 1.4.

Logging by definition requires that you place calls throughout your code, making notes to a log file of events and problems as they occur. Let's look at a simple error logging example:

```
aspect SimpleErrorLogging {

      Log log = new Log();    //some logger

      pointcut publicEntries():
        receptions(public * com.yourcom.printers.*.* (..));

      after() throwing (Error e): publicEntries() {
            log.write("stuff");
      }
}
```

In a nutshell, this slightly puzzling code says that if *any* public method of any type with any signature in any class in the com.yourcomm.printers package throws an error it is logged here. This means that all logging from all errors takes place in one location, and that we can make any changes to that logging in that single place. As the authors note in their articles, this technique is more explicit, more modular and more concise than placing logging in each try – catch block. Further, you are spared the difficulty of carrying an instance of your Logger class around the program.

But Is it Java?

This code looks really weird at first if you are accustomed to writing Java code, but it is real Java because the AspectJ system converts these statements to real Java before it is compiled and executed. Versions of AspectJ, with slightly different features, have been written both as pre-compilers and as a full compiler. So while it is not fair to say that AspectJ is exactly Java, it is true that all Java programs are AspectJ programs.

What Else Can I Do With It?

You can use AspectJ and Aspect-oriented programming in a number of powerful ways. For example, you can enforce code changes by creating an aspect that describes a code pattern you want to replace, in case you missed one, and then issue an error message if that pattern occurs somewhere. Aspects are class-like, and can exist as abstract base classes with several concrete implementations.

Summary

Aspect-orient programming is not some thimblerigger's confidence trick, and you don't need 76 Trombones to horn your way into it. It is an elegant and powerful approach to adding more structure to large programs, to prevent overpowering complexity from taking hold.

References

You can read extensively about Aspect-oriented programming and AspectJ at the web site, www.AspectJ.org. In particular look at

1. "Getting Started with AspectJ"
2. "Keynote: Aspect-Oriented Programming."
3. "An Overview of AspectJ"

I have taken most of the examples from those papers.