



A look at aspect-oriented programming



[Gary Pollice](#)

Worcester Polytechnic Institute
27 0 2004

from The Rational Edge: Pollice provides an overview of AOP and offers some observations on its promise, what we need to do to realize that promise, and some issues and obstacles along the way.

In a perfect world, there would be no such thing as software rework. We would get the object model right the first time. The waterfall approach to development would work just fine. But alas, we don't live in a perfect world. That's why we continue to seek better ways to build our software systems.

As realists, we acknowledge that no one process, technique, language, or platform is good for all situations. We increase the number of tools in our repertoire so that when we need that special tool or technique, we are well-prepared to use it. The history of our industry is rife with examples of improvements in our approach to building software, from the introduction of high-level languages, structured programming, and the object-oriented approach, to the development of spiral and iterative methods, and so on. One of the latest entrants in this lineup is aspect-oriented programming (AOP), which represents one facet of aspect-oriented software development (AOSD). In this month's column, I'll provide an overview of AOP and offer some observations on its promise, what we need to do to realize that promise, and some issues and obstacles along the way. We'll learn a little bit about AOSD, too.

What is AOP?

Aspect-oriented programming is one of those ideas that seems new but has actually been around for quite a while. Many people contributed to the body of knowledge available today, and many can lay claim to being a founder. The person most commonly associated with AOP is Gregor Kiczales, currently at the University of British Columbia, where he works on software modularity research. From 1984 to 1999, Kiczales worked on AOP at the Xerox Palo Alto Research Center (PARC) and was a leader in developing implementations of it.

The best way to describe AOP is by example. Suppose you were responsible for maintaining a large software system that

Contents:

[What is AOP?](#)

[The logging example in AspectJ](#)

[What does this all mean to you?](#)

[Resources](#)

[Notes](#)

[About the author](#)

[Rate this article](#)

Subscriptions:

[dW newsletters](#)

[dW Subscription](#)

[\(CDs and downloads\)](#)



managed the payroll and personnel functions for your organization. What if management issued a new requirement that you create a log of all changes to an employee's data? This would include payroll changes, such as a change in the number of deductions, raises, overtime, and so on. It would also include any changes to the employee's title, personal data, and any other information associated with an employee. How would you go about meeting this requirement?

Most people, in the absence of a better way, would search through the code and insert calls to a logging method in appropriate places. If the system were well-designed and constructed, the code might require changes in only a few places. For most systems, though, insertions would be necessary in many places. If the system were object-oriented, we might build a logging class and then use an instance of it to handle the logging. We might need a hierarchy of classes to handle different files and databases. Attempting to fulfill the requirement in this way would not be a trivial job, and it would not be easy to ensure that we'd made all of the necessary insertions, either.

Even if we had a superior, object-oriented system implementation, we would encounter some problems. In their book, *Mastering AspectJ*,TM Joseph Gradecki and Nicholas Lesiecki note that OO systems often produce classes that are difficult to change, and code that can't be reused and is difficult to trace through (i.e., numerous inter-object messages make it difficult to follow a logical thread for the code).¹

Cross-cutting concerns

The type of logging required in our example is a *cross-cutting* concern. It cannot be isolated or encapsulated in one or two specific classes; the logging involves changes in many places across the system. Our desire is to keep the system maintainable, and therefore we want our logging approach to keep the code as clean and simple as possible. We would also like to avoid significant structural changes to the system's architecture. So how do we implement a cross-cutting concern such as logging? We could refactor all of the code by creating a logging class and performing the appropriate insertions. However, in a large system, this would be a time-consuming, error-prone job.

AOP is designed to handle cross-cutting concerns by providing a mechanism, the *aspect*, for expressing these concerns and automatically incorporating them into a system. AOP does not *replace* existing programming paradigms and languages; instead, it works *with* them to improve their expressiveness and utility. It enhances our ability to express the separation of concerns necessary for a well-designed, maintainable software system. Some concerns are appropriately expressed as encapsulated objects, or components. Others are best expressed as cross-cutting concerns.

The logging example in AspectJ

Let's continue looking at the logging problem and see how it might be solved using AspectJ, an AOP implementation that extends the Java language. AspectJ is perhaps the best known and most widely used AOP implementation. Even if you are not familiar with Java, you should be able to understand the major ideas I will present below.

Figure 1 depicts one way we might structure our modification of the system. The Finance system has an interface and several methods for updating an employee's financial data. The names of the methods all begin with the word `update` (e.g., `updateFederalTaxInfo`), and each financial update takes an `Employee` object as an argument. Employees' personnel information is also updated through an `Employee` object, using the methods shown in Figure 1.

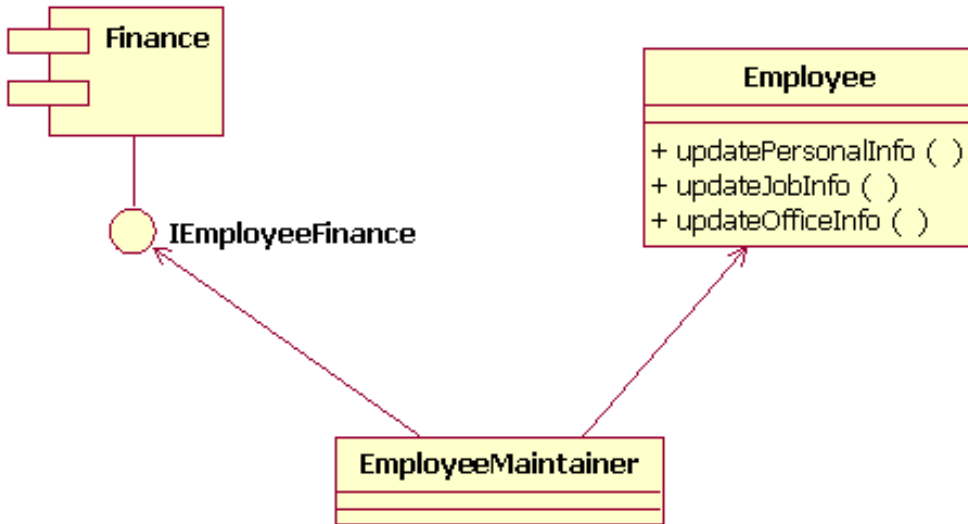


Figure 1: Classes involved in updating employee information

We can describe, in prose, what we need to do. Every time we call any of the updating functions and the update succeeds, we write a logging message. For simplicity, we will say that we print a logging message to the standard output. In the real system we would write to a log file. We would take three steps to implement our solution in AspectJ:

1. Identify places in the code where we want to insert the logging code. This is called *defining join points* in AspectJ.
2. Write the logging code.
3. Compile the new code and integrate it into the system.

I will describe each of these steps in the next three sections.

Define the join points

A *join point* is a well-defined point in the code at which our concerns crosscut the application. Typically, there are many join points for each concern. If there were just one or two, we could manually change the code with little effort.

In AspectJ, we define join points by grouping them into *pointcuts*. (The AspectJ syntax is rich, and we will not attempt to describe it completely here.) To start, we define two pointcuts, one each for grouping the join points in the `Employee` class and the `IEmployeeFinance` component. The following code defines the two pointcuts:

```

pointcut employeeUpdates(Employee e):
    call(public void Employee.update*Info()) && target(e);

pointcut employeeFinanceUpdates(Employee e) :
    call (public void update*Info(Employee)) && args(e);
  
```

The first pointcut, named `employeeUpdates`, describes all join points where we call an `Employee` object's method that begins with the string `update` and ends with the string `Info`, and has no arguments. It also specifically identifies the method as defined in the `Employee` class through the `target` designator. The second pointcut, `employeeFinanceUpdates`, describes all join points where there is a call to any method that begins with `update`, ends with `Info`, and has one argument of type `Employee`. Together, these two pointcuts define all of the join points with which we are concerned. If we add more updating methods to the `Employee` class or the `IEmployeeFinance` component, calls to them will automatically be included in the pointcuts, as long as we maintain the same naming conventions. This means that we do not have to deliberately include logging code every time we add an updating method.

Write the logging code

The code that implements logging is similar to any method in Java, but it is placed in a new type, called an *aspect*. The aspect is

the mechanism we use to encapsulate code related to a specific concern. The implementation of the aspect for logging employee changes might look like the following:

```
public aspect EmployeeChangeLogger {
    pointcut employeeUpdates(Employee e) : call(
        public void Employee.update*Info()
        && target(e);

    pointcut employeeFinanceUpdates(Employee e) : call(
        public void update*Info(Employee))
        && args(e);

    after(Employee e) returning : employeeUpdates(e)
        || employeeFinanceUpdates(e) {
        System.out.println("\t>Employee : " + e.getName() +
            " has had a change ");
        System.out.println("\t>Changed by " +
            thisJoinPoint.getSignature());
    }
}
```

First, notice that the aspect structure is similar to a class in Java. The aspect is typically placed in its own file, just like a Java class. Although the usual method is to include the previously defined pointcuts here in the aspect code, as we have done, we could instead include them closer to code containing the join points.

Following the pointcuts, we have a section of code that is similar to a method in regular Java code. This is called *advice* in AspectJ. There are three different types of advice: before, after, and around. They execute before the join point, after the join point, and instead of the join point, respectively. There are many variations you can use to customize your advice. In our example, we choose to perform the logging immediately after the updating method in the join point returns. Notice also that we combined the two pointcuts by naming each of them immediately after the colon in the advice header and connecting them with a logical “or.” We were able to do this easily because both pointcuts have an Employee parameter.

Two statements in the advice print out the fact that the employee’s information was changed, along with the name of the employee. This is easy to arrange, since the affected employee object is passed in as an argument to the advice. The second statement identifies the exact join point where the advice is executed and makes use of the AspectJ `JoinPoint` class. Whenever advice executes, it has an associated join point referenced by `thisJoinPoint`.

Compile and test

Now that we have written the code for logging, we need to compile it and integrate it into the existing system. For our example, we have implemented two classes: `Employee` and `EmployeeFinance`. We also have a simple test class with a main method, as shown below:

```
public static void main(String[] args) {
    Employee e = new Employee("Chris Smith");
    // Do something to change some of the employee's
    // information here.
    e.updateJobInfo();
    e.updateOfficeInfo();
    EmployeeFinance.updateFederalTaxInfo(e);
    EmployeeFinance.updateStateTaxInfo(e);
}
```

This code runs just fine without any AOP implementation. For our example, the bodies of all the update methods just contain a print statement. When we run this example we get the following output:

```
Updating job information
Updating office information
Updating federal tax information
Updating state tax information
```

In order to incorporate our aspect into the system, we add the aspect source code to the project and build with the AspectJ compiler, `ajc`. The compiler takes each aspect and creates class files that contain the advice code. Then, calls to the appropriate methods in these class files are *woven* into the original application code. With the current release of AspectJ, this weaving takes place at the Java bytecode level, so there are no intermediate source files you can look at to see the final code. However, you can decompile the Java bytecode if you are curious.

I use the Eclipse environment for development, and the AspectJ plug-in takes care of invoking the compiler with the proper arguments. Once we have compiled the project with the AspectJ compiler, we get the following output:

```
Updating job information
  >Employee : Chris Smith has had a change
  >Changed by void employee.Employee.updateJobInfo()
Updating office information
  >Employee : Chris Smith has had a change
  >Changed by void employee.Employee.updateOfficeInfo()
Updating federal tax information
  >Employee : Chris Smith has had a change
  >Changed by void employee.EmployeeFinance.updateFederalTaxInfo(Employee)
Updating state tax information
  >Employee : Chris Smith has had a change
  >Changed by void employee.EmployeeFinance.updateStateTaxInfo(Employee)
```

Now we know which employee had information changed, and where the changes were made. The logging could certainly be much more elaborate, but the basic technique would not change.

What does this all mean to you?

Aspect-oriented technology has many potential benefits. It provides a way of specifying and encapsulating cross-cutting concerns in a system. This might, in turn, allow us to do a better job of maintaining systems as they evolve — and we do know that they *will* evolve. AOP would let us add new features, in the form of concerns, to existing systems in an organized manner. The improvements in expressiveness and structure might allow us to keep systems running longer, and to incrementally improve them without incurring the expense of a complete rewrite.

AOP may also be a great addition to quality professionals' toolboxes. Using an AOP language, we might be able to test application code automatically without disturbing the code. This would eliminate a possible source of error.

We are in the early stages of understanding the full potential of AOSD. It seems clear that the technology offers enough benefits to warrant further exploration and experimentation. How far are we from everyday usage of AOP languages for application development? It depends on whom you ask.

Now that we've seen some benefits, let's look at some of the risks concerning AOSD as well as what is needed to bring it into the software development mainstream. The next sections discuss these issues.

Quality and risks

Having looked at AOSD from a quality viewpoint and done a little exploring with AspectJ, I've seen potential risks along with

the benefits. I will discuss three issues that illustrate some of the challenges we may face with respect to quality as AOSD becomes more popular.

The first issue is how we will need to modify our process to accommodate AOP. One of the most effective techniques for detecting software defects is through code inspections and reviews. During a review, a team of programmers critiques code to determine if it meets its requirements. With object-oriented programs, we can review classes, or sets of related classes, and reason about them. We can look at the code and determine if it handles unexpected events properly, has logical flaws, and so on. In an OO system, each class completely encapsulates the data and behavior of a specific concept.

With AOP, however, we can no longer reason about a class just by looking at the code for it. We do not know whether the code might be either augmented by advice from some aspect or completely replaced by such advice. To be able to reason about an application's code, we must be able to look at the code from each class as well as the code for any aspects that might affect the class's behavior. However, it is possible that the aspects have not been written yet. If that is so, then how much can we truly understand about the behavior of the application class's code when we consider it in isolation?

In fact, the way to think about correctness in AOP code is the inverse of how we consider it for object-oriented programming (OOP) code. With OOP, we go from inside out: We consider a class, make assumptions about its context, and then reason about its correctness, both in isolation, and in terms of how it will interact with other classes. With AOP, we need to look from the outside in, and determine the effects of each aspect on every possible join point. Determining how to reason correctly about AOP, and developing the appropriate techniques and tools to help us, is a rich research area.

A second, more practical issue regarding the potential widespread adoption of AOP is the development of tools and techniques for testing, especially unit testing. Because code may be changed by an aspect, unit tests that run perfectly on a class may behave quite differently when the class is integrated into an AOP system. The following example illustrates this point.

As you probably know, a *stack* is a data structure that is used to add and remove items in a last-in, first-out manner. If you push the numbers 2, 4, and 6 onto a stack and then pop the stack twice, you will get 6 and 4, in that order. Writing a unit test for a Stack class is straightforward. We can read the requirements specification and the code and do a very good job of ensuring that the implementation is correct. One of the first things I did when I began to look at AspectJ was to implement a stack and see what I could do with aspects to change the behavior. I implemented a simple change: Increment each item by 1. This is very easy to do. Now, my unit test — which pushes 2, 4, and 6 onto the stack and pops the top two elements off to verify that they are 6 and 4 — fails. The code that the unit test is testing did not change, yet the behavior changed. Instead of 6 and 4, the results are now 7 and 5.

This is a trivial example, and one that probably would not occur in a real-life situation. But it shows that a malicious programmer can cause a lot of harm quite easily. Even if we ignore malicious programmers, many defects occur because there are unwanted side effects of changes we make. It might be very difficult to ensure that an aspect implemented for very valid reasons does not have unwanted effects on existing program functionality.

The third issue is the testing process itself. Once we have a set of tools and techniques, how do we modify our testing process to use these effectively and support our overall development goals? Although this issue may not be a major one, I believe it will need to be addressed before we can really do a good job of testing software built with aspects.

Other barriers to AOSD adoption

Quality issues may be the biggest deterrents to adopting AOSD methods, but they are not the only ones. AOSD is a new paradigm. As did other paradigms when they were new (e.g., object-oriented software development), this one will take time to be adopted widely because it involves a learning curve. First we must learn basic techniques and mechanics, then advanced techniques, and then how to best apply the techniques and when they are most appropriate.

Tools will play a major part in industry adoption of AOP. In addition to compilers and editors, we need tools to help us reason about systems, identify potential cross-cutting concerns, and help us test in the presence of aspects. As we find methods to help us represent the systems better, such as representing aspects in UML, our tools must evolve to support these methods.

Although I discussed AspectJ in this article, it is neither the only AOP implementation nor the only one for Java. Furthermore, other paradigms similar to AOP are also emerging. The multi-dimensional separation of concerns paradigm, for example, has been under development at IBM Research, and an implementation is available in the HyperJ tool (<http://www.alphaworks.ibm.com/tech/hyperj>). Using any new paradigm is risky until standard implementations are established for the languages you work with. AspectJ, for instance, is a still-evolving AOP implementation. The danger is that you may develop software that incorporates cross-cutting concerns, and your implementation approach will either cease to be supported or will change significantly.

Moving toward a better way to build software

Clearly, we have a way to go in developing tools and processes to make AOP viable for everyday use. However, I do not believe that any of the issues I have discussed in this article represent insurmountable problems. I *do* believe that when we have developed a proven set of tools and processes for AOP, we will have a better way to build software than we do today.

As Barry Boehm said about agile processes,² we must approach AOP with care. Whether we are early adopters or we wait for this technology to become more mainstream, we need to ensure that our software investment will provide acceptable returns today and in the future. That's just good business sense.

Resources

If you are interested in AOSD and AOP, here are some places to start your investigation.

Communications of the ACM, October 2001, has many articles on AOSD.

"Improve Modularity with Aspect-oriented Programming," by Nicholas Lesiecki provides a good overview: <http://www-106.ibm.com/developerworks/java/library/j-aspectj/>.

"Test Flexibility with AspectJ and Mock Objects," by Nicholas Lesiecki shows how AOP can be used to help with unit testing: <http://www-106.ibm.com/developerworks/java/library/j-aspectj2/?open&l=007,t=gr>

Mastering AspectJ, by Joseph D. Gradecki and Nicholas Lesiecki, is a book published by John Wiley, 2003.

For a more detailed list of resources, visit my Web page at WPI: <http://www.cs.wpi.edu/~gpollice/Interests/AOSD.html>

Notes

¹ *Mastering AspectJ*,TM by Joseph D. Gradecki and Nicholas Lesiecki, John Wiley, 2003, p. 8.

² "Get Ready for Agile Methods, with Care", Barry Boehm, *IEEE Computer*, January, 2002.

About the author



Gary Pollice is a Professor of Practice at Worcester Polytechnic Institute, in Worcester, MA. He teaches software engineering, design, testing, and other computer science courses, and also directs student projects. Before entering the academic world, he spent more than thirty-five years developing various kinds of software, from business applications to compilers and tools. His last industry job was with IBM Rational Software, where he was known as "the RUP Curmudgeon" and was also a member of the original Rational Suite team. He is the primary author of *Software Development for Small Teams: a RUP-Centric Approach*, published by Addison-Wesley in 2004. He holds a B.A. in mathematics and M.S. in computer science.