from The Rational Edge: Most introductions to Aspect-Oriented Programming are based on limited contexts for the technique’s adoption, which obscure the practical value of AOP. This article offers a practical set of examples for applying AOP techniques to software development projects.

Recently, I was asked to lead a discussion on aspect-oriented programming (AOP) for our Software Engineering Research Group (SERG). A few hours before the meeting, one of the students asked me, "So, what are aspects good for? And don't give me the logging example. That seems to be the only thing I see when I read anything about aspects."

His question caused me to stop and consider effective ways of applying AOP in some of the software systems I've worked on. It also made me realize that we need to think about how and when we should adopt new methods, especially when they require a new way of thinking. AOP, which I've talked about before in this column, seems to represent a new approach. I'd like to talk about some of the ways that I think AOP can be (and has been) used effectively. We'll also look at some recent advances in AOP that might help its adoption.

I will use aspect-oriented Java as the vehicle for examples in the discussion. There are, however, aspect-oriented implementations of several languages available today. These include AspectC++ and even AspectL, which is an aspect-oriented Lisp implementation.¹

A review of AOP concepts

If you are not familiar with AOP, there are many introductory articles on it, including my February 2004 article.² Many, if not most, introductions to AOP use logging as an example to illustrate the concept of aspects. (Logging is something that many people understand, and it is a good example of how AOP can be used.) Aspects are concerns that are crosscutting. That is, they are not easily encapsulated in a single class. Yet, if we strictly follow the object-oriented paradigm, we need to represent such concerns in a uniform, maintainable manner. This often involves delegating the crosscutting responsibility to a separate helper class and relying on every class that requires the functionality expressed by the concern to include calls to the delegate at the appropriate places. Ensuring that the developers consistently insert the login in the proper points in the code is difficult to enforce. Aspects provide a mechanism, albeit imperfect, for improving the situation.
There are a few concepts that you need to know in order to appreciate a discussion of AOP. The main concept is the join point. This is a concept that all programmers are familiar with, but we now have a name for it. A join point is "a well-defined point in the program flow." There are many types of join points, such as a method call or a method return, which can be either a normal return or a thrown exception.

With AOP, we need a way to identify join points in a program. AspectJ uses the pointcut to describe one or more join points. The pointcut is an expression that describes a set of join points. You can consider the pointcut to be a query of your code that returns a set of join points.

When you select a set of join points, you provide advice for them. Advice is executable code that runs when the join point is encountered during the program runs. Join points, pointcuts, and advice address the dynamic properties of your software. Advice alters the running characteristics of the program's code.

There is one concept that addresses the static nature of your system. This is the inter-type declaration. The inter-type declaration allows you to change the static structure of a program. You can add methods and variables, and change the inheritance hierarchy according to specific rules.

Just as the class is the unit of modularity for Java, the aspect is an additional unit of modularity for AspectJ. The aspect encapsulates the join points, pointcuts, inter-type declarations, and advice for a crosscutting concern. AOP is not a replacement for object-oriented analysis and design. It builds upon the OO paradigm by addressing situations where the OO approach does not adequately provide the most desirable solution.

AOP examples

Now let's look at examples of where AOP can be -- and is -- used. Some examples are found in production systems, and others are found in production and development situations. Let's start with a couple of examples for developers.

Execution tracing

I'm amazed at how many developers put some sort of print statements in their code to debug or trace a program's execution. I find that debuggers are good at giving this information. But we're not here to discuss the merits of getting to know your debugger. There are certainly valid reasons for wanting to produce some sort of textual trace of your program. The current set of AspectJ Development Tools (AJDT) in Eclipse has a nice example of an aspect that implements a program execution trace. The example is described in detail in the Eclipse help. You can find it in the AspectJ Programming Guide.

The example has a small Java application that has classes to represent two-dimensional figures, like circles and squares. It also has a main method that creates two circles and a square, and prints out their properties, like area, perimeter, and the distance between their center points. When you run the program, you get the output shown in Figure 1.
Figure 1: Input from the shape program

If we want to see the actual sequence of methods called, we have two choices. In the first approach, we could insert code at the beginning of each method that would print a message with the name of the method and the class, and the fact that the method was entered. We would have to do this for each method. In the second approach, we would create an aspect that will do the exact same thing. With this approach, we do not have to change any of the application's code.

The tracing example consists of several versions of tracing solutions using aspects. We will look at the final version. See the AspectJ Programming Guide for a discussion of the other versions.

The solution to a robust tracing mechanism consists of two files. The first is an abstract aspect. This is similar to an abstract class where some of the code is left to the programmer to implement in a derived class -- or, in this case, a derived aspect. This abstract aspect, called Trace, has several standard methods for printing messages about entering and exiting a method or constructor, and for formatting the output. If you were not using aspects, these methods would be in a helper class that you would use for outputting trace information. Trace also allows the programmer to set the type of tracing by setting a property called TRACELEVEL. There are three levels: no messages, messages that are not indented, and messages that are indented to show nested calls.

Trace defines three pointcuts; two of them are concrete and one abstract, as shown in Figure 2. The abstract pointcut, myClass, must be provided by any aspect that extends Trace. The purpose of the pointcut is to select the classes for the objects containing the join points that will be advised. This lets the developer decide which classes to include in the trace output.
Figure 2: Pointcuts in the trace aspect

The myConstructor pointcut selects join points at the beginning of any constructor for an object in the class selected by myClass. The join point is the actual constructor body. myMethod is similar to myConstructor, but it selects the execution of any method in a selected class. Note also that it omits the execution of the toString method since that is used in the advice.

The advice provided by the aspect is quite simple. There is advice that is injected before each of the join points and advice that executes after the join point. This is shown in Figure 3.

```java
before (Object obj): myConstructor(obj) {
    traceEntry("\n + thisJoinPointStaticPart.getSignature()", obj);
}

after (Object obj): myConstructor(obj) {
    traceExit("\n + thisJoinPointStaticPart.getSignature()", obj);
}

before (Object obj): myMethod(obj) {
    traceEntry("\n + thisJoinPointStaticPart.getSignature()", obj);
}

after (Object obj): myMethod(obj) {
    traceExit("\n + thisJoinPointStaticPart.getSignature()", obj);
}
```

Figure 3: Advice in the Trace aspect

In order to use the Trace aspect, you need to extend it and supply a concrete implementation for the abstract pointcut. Figure 4 shows the body of the example program's TraceMyClasses aspect. The pointcut says to select only objects that are instances of TwoDShape, Circle, or Square. The main method sets the TRACELEVEL, initializes the trace stream, and runs the example's main method.

```java
public aspect TraceMyClasses extends Trace {
    pointcut myClass(Object obj):
        this(obj) &
        (within [TwoDShape] || within [Circle] || within [Square]);

    /*
     * A main function for testing the trace aspect.
     */
    public static void main(String[] args) {
        Trace.TRACELEVEL = 2;
        Trace.initStream(System.err);
        ExampleMain.main(args);
    }
}
```

Figure 4: Concrete tracing aspect

Figure 5 shows part of the output from running the example. Notice that the output prints information about each object. This is part of each object's toString method. Since the myClasses pointcut publishes the object to the advice, the advice can easily add information about the object.
Figure 5: Example trace output

What are the benefits of the AOP approach to tracing over manually inserting trace code everywhere you need it? There are several.

- You have all of your source code pertaining to the tracing concern in one (two aspects) place.
- It is easy to insert and remove the tracing code. You simply remove the aspects from the build configuration.
- The trace code is everywhere you want it, even if you add new methods to the target classes. This eliminates human error. You also know that all trace code is removed, and you haven’t overlooked anything when you remove the aspect from your build configuration.
- You have a reusable aspect that can be applied and enhanced.

Design by Contract, or defensive programming

Bertrand Meyer introduced the concept of Design by Contract. This principle asserts that the designer of a class and the user of that class share assumptions about the class implementation. The contract includes invariants, preconditions, and post conditions. Design by Contract lets the class designer concentrate on the logic that implements the class functionality without worrying about the validity of arguments. This is, of course, if the contract states the preconditions for the arguments. Design by Contract avoids extra code and improves performance, as long as all clients of a class abide by the contract.

When you build libraries for wide usage, you may not be able to make assumptions about the validity of the arguments that are passed to your methods. You need to check the arguments before proceeding with the logic of each method. This is an example of defensive programming. You assume that anything that can go wrong possibly will, and you handle it gracefully.
Let's say that you're going to take the simple shapes program and make it publicly available. You want to ensure that all coordinates are in the first Euclidean quadrant -- that is, the x and y coordinates are non-negative. This is a valid constraint to make if the points are going to be represented in a window's coordinates as well, since most window systems start with the upper left point as (0, 0) and increase the x-coordinate to the right and the y-coordinate as you move down. For your internal needs, you want to use Design by Contract since you have control over the developers in your organization that will use the classes. When you publish it to outside clients, you want to check the arguments and throw an exception if the arguments are invalid. Aspects provide an elegant way to implement just what you require.

We will build an aspect to check all of the arguments in the public methods. The first thing we will do is construct the pointcuts. We will use the myClass pointcut from the previous example and add pointcuts to select constructors that need argument checking, and the distance method to ensure that it's not called with a null value. Figure 6 shows the set of pointcuts we need. Notice that the second pointcut specifies that the target of the pointcut is an instance of TwoDShape. This means that only calls to the distance method in such an object are selected by this pointcut.

```java
pointcut myClass(Object obj):
    this(obj) &&
    (within(TwoDShape) || within(Circle) || within(Square));

/**
 * Because of the way that the Circle and Square classes are implemented,
 * we only need to check the constructors that take three doubles.
 */
pointcut myConstructor(Object obj, double a1, double a2, double a3):
    myClass(obj) && execution(new(double, double, double))
    && args(a1, a2, a3);

pointcut distance(TwoDShape obj, TwoDShape shape):
    target(obj) && call(double distance(TwoDShape))
    && args(shape);
```

**Figure 6: Pointcuts for argument checking**

Finally, we need to add the appropriate advice. For simplicity, we want to print a message when an invalid argument is encountered and then change the real values to zero, in the case of the constructor, and ignore the call to distance when a null value is passed. Figure 7 shows these two advice items.
When we attempt to execute the following statements:

```java
Circle c3 = new Circle(3.0,2.0,-2.0);
c1.distance(null>);
```

in our program, we get the following output:

```
Negative argument encountered at: execution(tracing.Circle(double, double, double)) All arguments changed to 0.0
Null value given to distance at: call(double tracing.Circle.distance(TwoDShape))
```

We could do a lot more with the error messages by showing the exact line number and source file name, but this example shows the primary technique.

In a large project where you have many classes and expose several interfaces, you might organize your code with a separate directory for the aspects that implement argument checking. I can imagine several ways of organizing the aspects so they are easily identifiable and maintainable. When you build the system for internal use, you use an internal build configuration, and when you build it for external use, you use a configuration that includes the aspects. The Eclipse AJDT makes the creation of new build configurations simple.

**Aspects and design patterns**

Design patterns have become *de rigueur* for good programming. AOP may give us a way to improve upon existing patterns and discover new ones. In fact, the injection of code for crosscutting concerns is a pattern of sorts. Currently, some researchers are evaluating the implementation of design patterns using an AOP approach. Jan Hannemann at the University of British Columbia has been investigating the topic as part of his Ph.D. research. Nicholas Lesiecki has also written about aspects and design patterns for IBM developerWorks. Check out his article for a more detailed discussion than I provide here.
Let's look at a very simple example of how one might implement a standard design pattern, the Adapter pattern, in AspectJ.

Figure 8 shows a Unified Modeling Language (UML) diagram for the Adapter pattern. In this pattern, the client needs a service and makes a request for it. There may be many providers for the service, and each of them may have a different name for the service or some other non-standard requirements that the service requester must abide by. Good object-oriented design suggests that we encapsulate the request for the service in a target interface, program to the interface, and build an adapter, as necessary, to act as a mediator between the client and the service (the Adaptee in the diagram).

**Figure 8: Adapter pattern**

The approach to adapters seems quite reasonable and rational. But what if you have a legacy system that was not designed with patterns like adapters? The original designers did not realize the possibility of future change. Also, calls to the service may be spread throughout the application. How do you implement a new, improved service? Without aspects, you will probably refactor the system, locate every call to the service, design an adapter, and implement the Adapter pattern. Depending upon the number of places where the old service was called, this can be quite a daunting task. Refactoring for code improvement is a worthy goal; however, we don't always have that luxury. We have to settle for evolving the code as best we can under the time constraints.

In this case, we can build an AOP version of the Adapter pattern that will solve the immediate problem and provide a step towards a more organized, better-designed system.

A simple client that uses a service is shown in Figure 9. The service will return the distance from one point to the current Client object.
Figure 9: Simple client

The interface for the service describes a single method:

double useService(Client clt, double x, double y);

The new, improved service provider has a differently named method, with different parameters. Its interface is:

double useNewService(double x0, double x1, double y0, double y1);

We want to build an adapter that will sit between calls to the old service, get the proper arguments for the new service, invoke the new service, and return the results to the client -- without having to change the client. Our aspect to do this is shown in Figure 10.
public aspect AdapterAspect
{
    pointcut adaptService(Client cli, double x1, double y1):
        this(cli) &&
        call (double IService.useService(*, double, double)) &&
        args(*, x1, y1);

    double around(Client cli, double x1, double y1):
        adaptService(cli, x1, y1)
    {
        double x0 = cli.getX();
        double y0 = cli.getY();
        return NewService.useNewService(x0, x1, y0, y1);
    }
}

Figure 10: Adapter aspect to invoke the new service

Notice how simple the aspect is. We declare a pointcut to identify every call to the original service's useService method. Then we use the around advice to replace the call with a call to the new service, after extracting the required information from the calling client.

There are benefits and disadvantages to this approach. First, we are able to make the change across all of the code in the application with this one simple aspect. We also have encapsulated the concern for calling the service in one place. Now, if the new service is ever replaced by a newer service, we just need to change this aspect's code. These are certainly benefits to a more robust system.

The main disadvantage is that the old client is still in the system, even though it isn't used. We most likely will go back and refactor the system to use a more standard Adapter pattern if we have the time (which we never seem to have). At the very least, we can modify the original service's useService method with code that returns a dummy value like 0.0, since it will never be called.

Industrial-strength uses

So far, we have looked at fairly limited, yet useful examples of applying AOP. We might ask how well the technique scales up. There is one notable example that I will point out here and describe briefly. A more detailed look will require another article in this column.

Some of you may be familiar with the Spring Framework. The Spring Framework is an application framework that supports developing enterprise applications. It provides a layered J2EE framework for building complex enterprise applications. One of the fundamental technologies used in Spring is AOP. Spring has developed its own aspect-oriented implementation that allows crosscutting logic to be applied to code at run time, rather than through a compiler as with AspectJ. However, it provides integration features that let you easily incorporate AspectJ with the Spring Framework.

Spring is currently being used by many organizations to build better enterprise applications that are well designed and require less development time. In my estimation, it is an excellent example of how we can apply aspects to real problems.
Where do we go from here?

AOP will become, in my opinion, part of the software developer's toolkit in the future. I don't know if we will get to the point where we build systems where we consider AOP the primary design mechanism, but I do think we will find ways to enhance our OO designs with aspects. A couple of things need to occur to help speed the process along.

First, we need some standard AOP implementations that are stable. This process has already begun. AspectJ version 5 is a merger of the two most popular Java AOP languages, AspectJ and AspectWerkz. Standard implementations in other languages like C++ will also help.

Second, we need to develop metrics that enable us to reason about the effectiveness of applying AOP to particular systems. For example, when we re-implement design patterns using AOP, are they any better than the standard OO patterns? Are there cases where they are better and where they are not? What are the measures that we must make in order to get the information and develop these metrics? The days of the naive approach -- where we believe that if a system is object-oriented (or pick your technology), then it's good -- are over. We need to make design and implementation decisions based upon empirical evidence.  

Third, we need to continue to develop tools that support AOP. I'm happy to say that the new AJDT for Eclipse is a great set of tools. These tools continue to improve the support that we need in order to effectively and efficiently use aspects. The tools must also support a new process for dealing with aspects.

Aspects are here to stay. They're still a ways off from becoming a part of mainstream applications, but they're getting closer every day. I recommend you take a look at them and get ahead of the curve. 

Notes

1 Information on AspectC++ can be found at http://www.aspectc.org/ and AspectL at http://common-lisp.net/project/closer/aspectl.html


3 The definitions I'm using are taken from the excellent AspectJ Programming Guide: http://www.eclipse.org/aspectj/doc/released/progguide/index.html


5 The Gang of Four is how many refer to Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides who wrote Design Patterns. This is the seminal book in the design pattern corpus.

7 See http://www.springframework.org/


9 If any of the readers are using AOP and are willing to talk to me about their experiences, and possibly provide input for a research study I'm working on to develop AOP metrics, please contact me.