

Colorado Technical University



**Technical Report
Computer Science**

Aspect Oriented Programming

Reid R. Kaiser
reidkaiser@yahoo.com

**Technical Report Number
CTU-CS-2002-006**

Aspect-Oriented Programming

Reid R. Kaiser

reidkaiser@yahoo.com

Abstract

The paper provides an introduction to aspect oriented programming (AOP) and gives examples of its use.

1 Introduction

The goal of computer programming is to solve problems by describing the problem to a machine in a language that the machine can understand. Some problems are trivial and this translation process is very simple, requiring minimal technique. Other problems are not so simple and require a significant effort to separate the issues. This is the point where software development becomes difficult. How does one disassemble a problem into a number of separate problems or concerns effectively? One term that has been used to describe this process is *Separation of Concerns (SOC)*. The principle behind SOC is that any given problem includes many different concerns, which should be separated to cope with complexity and to achieve various quality factors[1]. Many approaches have been used throughout the years to separate concerns.

Separation of concerns has been attempted for many years through the methodology called structured design. The idea behind structured design is to break the problem apart by identifying the functions that the program needs to solve and coding them in a top-down, process oriented fashion. With structured design, concerns are separated through the use of subroutines and functions. Many successful programming projects have used this method and it is still widely used today. But it has its shortcomings. Maintainability has been cited as an issue with structured design [2]. Although poor maintainability is not limited to structured design, it is easy to see that some large structured programs are difficult to maintain. One of the reasons for this difficulty is that the concerns are sometimes difficult to localize through the use of functions and subroutines. Without having a localized concern, the modification of the concern may be diffi-

cult and require the modification of several subroutines and functions. It is sometimes difficult to determine the far-reaching effects when modifying a subroutine or function. Another criticism of structured design is that it does not follow the manner in which we typically think about systems. At least that is the argument that the object-oriented proponents believe. It may be true, however, that some projects do not lend themselves well to a structured design methodology and are easier to accomplish when approached in terms of objects. One example that comes to mind is the implementation of graphical user interfaces. They are filled with different graphical widgets that are easily thought of in terms of objects with properties, and are much more difficult to work with in terms of functions and subroutines. These shortcomings resulted in much research and one of the products of that research was object-oriented programming.

Object-orientation uses a different approach for the separation of concerns than structured design. Following the object-oriented programming paradigm, concerns are modeled as objects and classes [1]. This method is currently being used very effectively to successfully create complex systems. Object-oriented programming is based on the idea that one builds a software system by decomposing a problem into objects and then writing the code for those objects [3]. Some think that this is a more natural way to look at problems. It can make the localization of concerns easier.

As with all technologies, object-oriented programming has its shortcomings. Among these shortcomings is the ability to deal with behaviors that are global, or if not global, at least common across many of the parts of the system. That is where Aspect-oriented programming (AOP) comes into play. The goal of AOP is to make designs and code more modular, meaning that the concerns are localized, rather than scattered, and have well defined interfaces with the rest of the system [4]. This paper discusses the background of AOP, some of its concepts, some practical examples of how AOP is used, and finally the benefits and drawbacks of AOP.

2 AOP Background

As discussed earlier the idea of separation of concerns (SOC) has been around at least since the time of structured design. The Law of Demeter further supports this idea. In its sim-

plest form the Law of Demeter states that methods should only communicate with closely related objects. The motivation for this law is to ensure that software is as modular as possible [5]. However, to write code that follows the Law of Demeter in an ideal way, the methods, whose ad-hoc implementation are scattered across several classes, should and could be cleanly localized [6]. This means a single concern should not be implemented across different classes. Rather the concerns themselves ought to be separated into their own classes. As work was being done with SOC, research concerning Adaptive Programming (AP) was occurring which introduced the ideas of crosscutting behaviors. By combining the ideas of crosscutting behaviors, SOC, and high modularization of programs the term Aspect-Oriented Programming was coined to describe the technology that encompasses these ideas.

3 AOP Concepts

AOP introduces some new approaches for solving the problems of crosscutting concerns. This paper focuses on one implementation of AOP, which uses a tool called AspectJ. Many of the terms identified as AOP ideas leverage the tools provided by AspectJ to help the reader understand the concepts. AspectJ uses constructs called aspects, advices, pointcuts, and join points to address the issues of crosscutting concerns.

3.1 Crosscutting Concerns

Crosscutting concerns are those issues that appear in multiple places in the program. Issues that could not be cleanly localized or modularized into a single class often are implemented throughout the program. Since the implementation of these concerns “crosscut” the system, they are called crosscutting concerns. There are two primary ways in which crosscutting concerns are manifested in a system. They have been given the names **scattering** and **tangling**. Scattering is the term used to indicate that the code required to implement a single concern is spread across multiple classes, and tangling is the term used to indicate that code implementing multiple concerns is implemented in a single class. Both of these issues deal with concerns that are not cleanly localized.

Take for example a figure editor. This figure editor represents the images on the screen with different objects. You might have an object that represents a line, a point, a circle, etc. and each of these objects implement a number of methods. Every time that a change is made to one of the figures on the screen (like moving it from one point to the next), something must occur (like a method call) that tells the screen to repaint. The issue of repainting the screen would be considered a crosscutting concern since it is a concern with several different objects.

In typical object-oriented programming a crosscutting concern (such as screen repainting) is accomplished by calling the appropriate screen method whenever one of the several methods of the objects on the screen is called. One of the issues when working in this type of an environment involves the ability to maintain the code. Whenever the screen repaint method changes either its name or its parameters, the change would need to be propagated to the calling methods. Therefore all of the objects that call on those methods must be changed as well. Those calls may not be easy to locate. The idea behind AOP is to take those types of concerns that are scattered throughout the program and bring them together into a single structure called an aspect. By allowing a crosscutting concern to be handled within a single aspect, the implementation of the concern can be localized. With the concern localized we no longer have the same problems that arise by having the issues spread around the application (such as the propagation of changed method calls).

3.2 Aspects

Crosscutting concerns are common occurrences in many programming environments and the problems with them are obvious. To deal with these problems AOP offers aspects, mechanisms beyond subroutines and inheritance for localizing the expression of a crosscutting concern [3]. Rather than implementing the issue, such as screen repainting, at different points throughout the program, aspects provide the ability to tackle the concern within one construct. Not only is the action defined within the aspect but also the calls to the action are defined within the aspect. The calls within the aspect are not invoked explicitly as a typical method, but rather defined to be

triggered by certain actions. The mechanisms that aspects use to implement the actions are called **advices**.

3.3 *Advices*

Advices are mechanisms that allow certain actions or methods to execute before or after the other methods within the program execute. The advices are, in a sense, triggered by the actions of objects throughout the system. A single advice can be triggered by many different calls to several methods of multiple classes. This allows certain actions to take place in a similar fashion throughout the application. Advices can be configured to execute *before*, *after*, or *around* a method. As one would expect, *before* advices are executed before the call, *after* advices after the call, and *around* advices are executed before and after the call. The point at which an advice can be implemented is defined as a **join point**.

3.4 *Join Points*

To implement the aspects, AOP uses a concept called join points. Join points are certain well-defined points in the execution flow of the program [7]. The typical place to implement a join point is at a method. Join points define the place in the program where the aspects are woven in. Since crosscutting concerns, by definition, are issues that are spread apart throughout the program these join points must be grouped together to define the related concern, which AOP does with **pointcuts**.

3.5 *Pointcuts*

To identify the crosscutting concern, AOP uses a concept called **pointcuts**. A pointcut is a group of related join points and defines the appropriate concern. With the join points grouped together into a pointcut, the same action can be defined to affect all join points within the pointcut. Advices are generally defined on entire pointcuts.

4 Examples

Currently a couple of languages exist that implement AOP. The most commonly used AOP language is an extension of Java called AspectJ. This section describes some ways in which AspectJ can be used to implement some realistic applications of AOP.

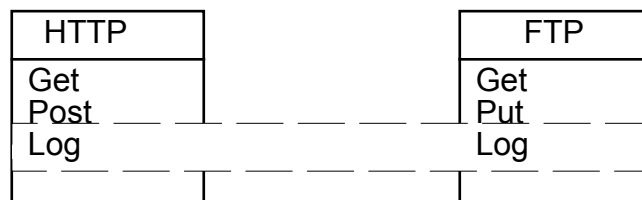
4.1 Logging

One of the most obvious implementations of an aspect is to provide logging. Certain types of applications may require several different points throughout the program where the results of actions must be written to a log file. Normally, to implement a logging function requires the coder to write statements throughout the program that perform the logging functions. Wouldn't it be convenient to have a single class that handles all of the logging capabilities? That is what an aspect attempts to do.

Look, for example, at an Internet server. Internet servers provide different mechanisms to service requests that come into the server. One of the more common requests that the server services is web traffic. The two most common methods used on the web are the Get method and the Post method. They both handle the requests in a slightly different manner. The Internet server may also implement an FTP server. This FTP server also implements a couple of methods called Get and Put. Both of these servers could implement many other methods but for simplicity only the previously mentioned methods will be considered. Most web and FTP servers provide a mechanism for logging the requests that come into the server. With each item that is logged, the method of the request, the parameters that are sent with the request, and the success or failure of the request are stored.

In this example two classes are used to implement the Internet server, one, called HTTP, to handle the web requests and one, called FTP, to handle the FTP requests. The class HTTP supports the methods get and post, and the class FTP supports the methods get and put. In the traditional environment the logging requirement can be met through a couple of options. One mechanism is to embed the code that performs the logging within each of the methods that re-

quire logging. This is not very desirable since it requires redundant code to be implemented within several different classes and methods. It also does not follow the principles of modularization. Another more desirable mechanism is to create another class for the logging function. This class would have a method that would write an event to the log file. This method is certainly better than the one previously discussed but it has its problems as well. It requires the class that implements the logging to be instantiated and the method called every time one of the methods of the HTTP or FTP classes are executed. A method similar to this uses a global function. A global function is one that can be called every time the logging capability is required. One issue with these last two ideas is the potential for change within the event logging method or function. If a change that modifies the parameters or name of the method or function is implemented, the calling methods need to be modified. For a non-trivial system, the calling methods might not be easily found.



Both Log methods perform the same function

Figure 1: HTTP and FTP Classes.

One other alternative to implement logging is by using an aspect. The mechanisms used in this example follow those that are provided within the AspectJ programming language. For the example, the name of the aspect will be LogEvent. The aspect needs to have a pointcut identified. An aspect can have multiple pointcuts but for this first example only one is needed. As discussed earlier, a pointcut is a group of join points that have the same crosscutting concern. Since the crosscutting concern is the logging feature the pointcut will be called log with the join points of HTTP.get, HTTP.post, FTP.get, and FTP.put. Most environments that implement logging features write the data to the log after the event has taken place, so an *after* advice will be used. The aspect that implements the information previously described will look similar to the following.

```

aspect LogEvent{
    pointcut log():
        call(void HTTP.get()) ||
        call(void HTTP.post()) ||
        call(void FTP.get())||
        call(void FTP.put());

    after() : log(){
        <Code that writes to the log>
    }
}

```

Figure 2: Logging aspect declaration.

The manner in which the aspect works is that if any of the methods identified in the pointcut are executed, the code identified in the *after* advice is executed. No explicit calls to the functions within the aspect take place. The code within the *after* advice is executed after the method completes its execution. The advice identifies the type of advice it is (*before*, *around*, or *after*) as well as the pointcut that it is implemented for. The only advice identified in Figure 2 is an *after* advice on the pointcut log. Multiple advices and multiple pointcuts can be defined in the same aspect.

4.2 Data Validation

Another example for a potential use of AOP is that of data validation. In many cases a program receives data input from a large number of sources. The source could be a form that a user enters data into or another system. If the values that are received from the data source contain a date it would be important to verify that the date is a valid date. Many other data validations can be accomplished. It would be advantageous if they could all be captured in an aspect.

```

aspect ValidateData{
    before(form f, chr datevalue, int age){
        call(void f.insert(datevalue, age)){
            CheckDate(f, datevalue)
            CheckAge(f, age)
        }
    }
    boolean CheckDate{
        <Code to implement Check Date>
    }
    boolean CheckAge{
        <Code to implement age check>
    }
}

```

Figure 3: Data validation aspect.

In Figure 3, functions are included to validate two kinds of data. One function (*CheckDate*) validates a date and the other (*CheckAge*) validates an age. Those functions can be used by any advice in the aspect. In the example a form exists that inserts a date and an age into the system. The insertion of the data takes place when the form executes its *insert* method with the date value and the age as parameters. In the example system it is important that the date is the proper format and the age is within a certain age range before the data is inserted into the system. Therefore, since the data must be validated before it is inserted into the system a *before* advice is defined. The *before* advice is defined such that it is triggered by a call to the *insert* method of the form class. When the *insert* method of the form is called, the functions *CheckDate* and *CheckAge* are executed. Upon successful completion, the *insert* method of the form can be completed.

4.3 Development Code

In most development environments code is written into the system to perform certain debugging functions. These functions are intended to provide insight into the system to allow easier debugging and verification of the code while developing. In a similar fashion, code is written to facilitate testing by providing checkpoints and variable values. This code is not intended to be delivered with the system and requires significant effort to remove. This is another interesting manner in which to use an aspect. An aspect can be created to localize all of the debugging functions. With all of the debugging code in the aspect, it is relatively easy to remove the aspect once development is complete and recompile the program without it. Then the debugging code, along with its performance overhead, is not delivered with the production code. The same thing can be done with the logging aspect described previously. If it is determined that logging requires too many resources, the aspect can be removed from the program and no other code needs to be modified.

5 Benefits

There are several benefits to implementing the crosscutting concern as an aspect. Many of the benefits are realized because of the localization of the concern. With the concern localized it is easier to make changes to it. Rather than being scattered throughout the code, the concern is implemented in one location and only needs to be modified in the single location. In addition, the changes do not need to be propagated to the method calls. If a parameter or name change is implemented within the aspect, it is not necessary to search through the code for the methods that will be affected by the change. All of the affected methods are available in the pointcut definitions of the aspect.

One potential benefit is the savings in lines of code. One example cited an AOP re-implementation of the memory management feature of a program required a total of 4559 lines of code versus the 35213 lines of the previous “tangled” version [8]. This savings seems quite significant and may speak more about the inefficiencies of their previous implementation than

their AOP re-implementation. Nevertheless, it is a little premature to say that this kind of savings is available across the board but the potential is certainly there and more in depth studies should be performed.

Another benefit of the AOP approach to programming is the ability to remove portions of the program that are no longer necessary. As was pointed out earlier, debug code could be removed prior to the system being deployed. With the debug code localized into one aspect the process of removing it becomes easy and less prone to errors than the alternative.

6 Drawbacks

AOP solves some of the problems that the other programming paradigms do not address but there are some drawbacks to this technology. The most obvious drawback is common to all new technologies, and that is the learning curve. AOP builds on some of the OOP principles but the concept of an aspect is new and requires some additional up front work.

Another drawback to AOP is that there is no standard way to depict a crosscutting aspect in UML, although some are attempting to solve this problem [9]. Without a method for describing a crosscutting concern it may be difficult for the developers to understand when to use an aspect versus when to use a typical class. Without a standard notation available, an in-house notation would need to be developed to ensure that the need for an aspect is propagated to the developers.

The verdict is still out on the performance implications of using aspects. Some of the reports indicate a significant performance enhancement while others indicate none or worse. Undoubtedly, there might be some additional overhead when implementing aspects. As new technologies are added to existing tools (like AspectJ to Java) there are bound to be additional complexities that would lead to decreased performance. However, there may be cases where the existence of an aspect, along with its benefits, makes the coding of the problem more efficient. In cases where there are significant improvements in coding efficiencies, we could see a performance improvement that offsets the performance degradation caused by the complexities intro-

duced by aspects. More research is needed to identify the circumstances that lead to improved or worsened performance.

7 Conclusion

AOP addresses a problem that structured design and object-oriented programming have been unable to solve effectively. The issue of crosscutting concerns has been a problem for some time, and AOP provides a solution to it. But does AOP solve the problem effectively enough to change the way an organization does business? That depends a lot on the organization and the significance of the problems that they have with crosscutting concerns. Will it become the next hot programming paradigm? It is impossible to determine what the next “big thing” is going to be but AOP has the potential to make an impact. However, more research is needed to come to an accurate assessment of AOP. First a detailed performance cost-benefit analysis would be helpful. In many projects performance is a driving factor and it would be good to understand the performance implications of using AOP. Second, an understanding of the impact to developer efficiency with this technology is important. The trivial examples that were identified in this paper show that it is not too complicated to think in terms of aspects but more complex implementations may prove differently. Finally, it is important to come up with a methodology for identifying and depicting crosscutting concerns. AOP solves some important programming problems but the verdict is still out on the impact it will make in the programming community.

8 References

1. Aksit, M., *TRESE Aspects and advanced separation of concerns homepage*. 2000.
2. Winograd, T., *Beyond Programming Languages*. CACM, 1979. **22**(77): p. 391-401.
3. Elrad, T., R.E. Filman, and A. Bader, *Aspect-Oriented Programming*. CACM, 2001. **44**(10): p. 29-32.
4. Elrad, T., et al., *Discussing Aspects of AOP*. CACM, 2001. **44**(10): p. 33-38.
5. Lieberherr, K., I. Holland, and A. Riel. *Object-Oriented Programming: An Objective Sense of Style*. Proc, *OOPSLA*. 1988: ACM.

6. Lieberherr, K.J., *Connection Between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP)*.
7. Kiczales, G., et al., *Getting Started With AspectJ*. Communications Of The ACM, 2001. **44**(10): p. 59-65.
8. Kiczales, G., et al. *Aspect-Oriented Programming*. in *European Conference on Object-Oriented Programming (ECOOP)*. 1997. Finland: Springer-Verlag.
9. Stein, D., S. Hanenberg, and R. Unland. *A UML-based Aspect-Oriented Design Notation For AspectJ*. in *1st International Conference on Aspect-Oriented Software Development*. 2002. Enschede, The Netherlands: ACM Press.