Aspect-Oriented Programming

An Introduction to Aspect-Oriented Programming and AspectJ

Niklas Påhlsson

Department of Technology
University of Kalmar
S– 391 82 Kalmar SWEDEN

Topic Report for Software Engineering 2002-11-03

ABSTRACT

Separation of concerns is an important software engineering principle. It refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern (concept, goal, purpose, etc.).

The leading programming paradigm used today is object oriented programming (OOP). OOP is a technology that is well known and is reflected in the entire spectrum of current software methodologies and development tools. However, OOP has its limitations, some design decisions can not be illustrated with the object oriented model.

This report is an introduction to a programming paradigm called Aspect-Oriented Programming, a programming technique that makes it possible to express those programs that OOP fail to support. This report also includes an introduction to AspectJ, an AOP implementation in Java.

The reader of this report is expected to have at least basic understandings in object oriented programming.

KEYWORDS

Aspect-Oriented Programming, Separation of Concerns, AspectJ

CONTENTS

1. Introduction ... 4
2. Background – Separation of Concerns ... 4
    2.1. Cross-cutting concerns ... 4
        2.1.1. Implications ... 5
        2.1.2. A simple cross-cutting example ... 5
    2.2. Current solutions ... 6
3. AOP Fundamentals ... 6
    3.1. What is Aspect-Oriented Programming? ... 6
    3.2. Aspect Weaver ... 7
    3.3. Development stages ... 7
        3.3.1. Aspectual decomposition ... 7
        3.3.2. Concern implementation ... 7
        3.3.3. Aspectual recomposition ... 8
    3.4. AOP Language Implementations ... 8
4. AspectJ ... 8
    4.1. AOP implementation in AspectJ ... 8
        4.1.1. Aspects ... 8
        4.1.2. Joinpoints ... 8
        4.1.3. Pointcuts ... 9
        4.1.4. Advices ... 9
    4.2. HelloWorld, AOP version ... 9
5. Discussion ... 10
    5.1. Will AOP replace OOP or other programming paradigms? ... 10
    5.2. AOP benefits ... 10
    5.3. AOP disadvantages ... 10
6. Conclusions ... 11
7. References ... 12

1. INTRODUCTION

The leading programming paradigm used today is object oriented programming (OOP). OOP is a technology that is well known and is reflected in the entire spectrum of current software methodologies and development tools. However, OOP has its limitations, some design decisions can't be illustrated with the object oriented model.

Xerox PARC (Palo Alto Research Center) has worked on developing programming techniques that makes it possible to express those programs that OOP fail to support. This programming technique is called Aspect-Oriented Programming (AOP).

The term Aspect-Oriented Programming includes Multidimensional Separation of Concerns, Subject-Oriented Programming, Adaptive Programming and Composition Filters. In this report the term Aspect-Oriented Programming is used to describe the space of programmatic mechanisms for expressing crosscutting concerns since this is more commonly used.

The reader of this report is expected to have at least basic understandings in object oriented programming.

2. BACKGROUND – SEPARATION OF CONCERNS

Separation of concerns is an important software engineering principle. It refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern (concept, goal, purpose, etc.). Concerns can range from high-level notations like security and quality of services to low-level notations like buffering, caching and logging. They can also be functional, such as business logics or non-functional like synchronisation.

A typical system consists of several concerns. In the simplest form there are the core concerns, i.e. the natural components of the software. Except for these core concerns, there are system level concerns, like security, logging, authentication, persistence and so on; concerns that tend to affect several other concerns. For instance, if a logging feature is to be implemented in an application, it is likely that all the underlying modules will have code for logging, making the underlying modules less specialized and makes it very hard to predict what effects changes in the code for logging will have.

Even though the object oriented model offers some ability for separation of concerns, it still has difficulty localizing concerns which do not fit naturally into a single program module, or even several closely related program modules.

2.1. Cross-cutting concerns

The problems discussed above, are called cross-cutting concerns, as they cross-cut several other modules in the system. Crosscutting concerns are behaviours that span multiple, often unrelated, implementation modules. In additional, crosscutting concerns cannot be neatly separately from each other. Examples of crosscutting concerns are:

- Security (authorization and auditing)
- Logging and debugging
- Synchronization
- Persistence
- …

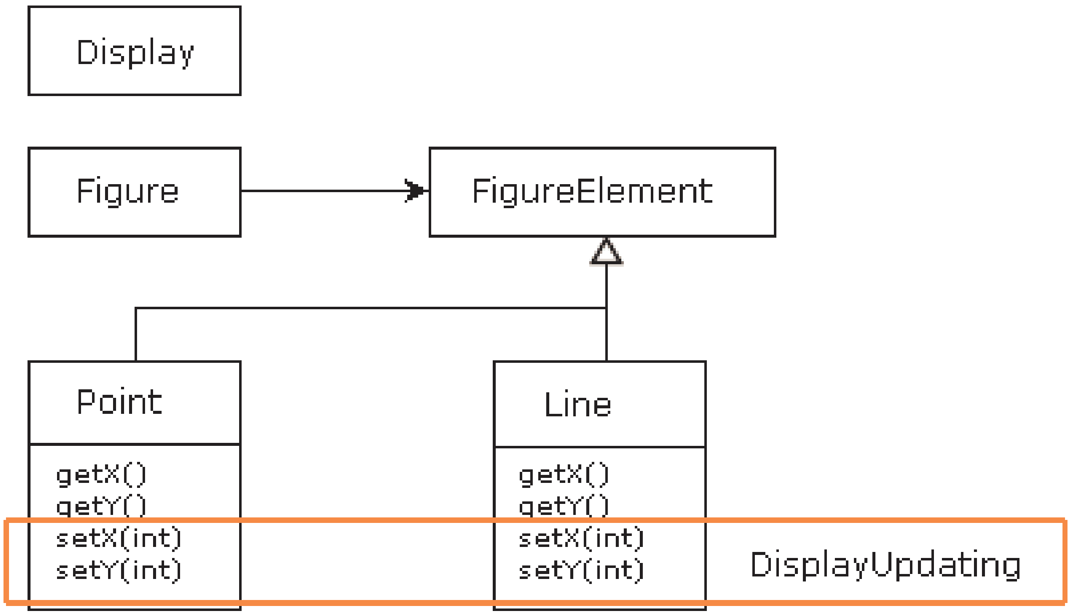
Why are the cross-cutting concerns a problem then? When modules in a system may interact simultaneously with several requirements, which mean that concerns are tightly intermixed, *code tangling* occurs. Since cross-cutting concerns spread over many modules, related implementations also spread over those modules. The concerns are poorly localized, and this is called *code scattering*.

2.1.1. Implications

Code tangling combined with code scattering affects software development in several ways, for example:

- Harder to reuse code – Since a module implements several concerns, it will be difficult to reuse the code.
- Lower productivity – Redundant implementation of multiple concerns shifts the developer's focus from the main concern to the peripheral concerns.
- Lower code quality – The two symptoms produces code with hidden problems.
- Limited evolution of the system – A limited view and constrained resources often produces designs that address only current concerns. Addressing future requirements often requires reworking the implementation. Since the implementation isn't modularized, that means touching many modules. Modifying each subsystem for such changes can lead to inconsistency. It also requires considerable testing effort to ensure that such implementation changes have not caused bugs.

2.1.2. A simple cross-cutting example



To illustrate the cross-cutting concern, here is a small example of this.

The simple UML diagram above shows a very simple figure editor. An abstract class, FigureElement, has two concrete classes, Point and Line, so far so good. Now, imagine that the screen manager should be notified whenever a FigureElement moves. This requires every method that moves a FigureElement to do the notification.

The red box in the figure is drawn around every method that must implement this concern, just as the Point and Line boxes are drawn around every method that implements those concerns. Notice that the box for DisplayUpdating fits neither inside nor around the other boxes in the figure, instead it cuts across the other boxes. This is what is called a cross-cutting concern.

2.2. Current solutions

Since the cross-cutting concerns are not new phenomenas, a few techniques have emerged to modularize their implementations. Today's methods of a general type of separation of concerns, includes among others, design patterns, application frameworks, application servers, mix-in classes and domain specific solutions. The main disadvantage of these solutions is that the core concern must be transformed to fit the solution selected for the problem. This report will not go any further into these methods. Instead, it will focus on a relatively new programming paradigm called Aspect-Oriented Programming.

3. AOP FUNDAMENTALS

3.1. What is Aspect-Oriented Programming?

Aspect-Oriented Programming (AOP) is a new programming paradigm developed at Xerox PARC. AOP strives to help the developer to separate concerns to overcome the problems with cross-cutting concerns described earlier, and it provides language mechanisms that explicitly capture crosscutting structure. This makes it possible to program crosscutting concerns in a modular way, and get the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has greater potential for reuse. This is done by improving code modularization using aspects. What OOP has done for object encapsulation and inheritance, AOP does for crosscutting concerns.

An aspect is, by definition, modular units that cross-cut the structure of other units. An aspect is similar to a class by having a type, it can extend classes and other aspects, it can be abstract or concrete and have fields, methods, and types as members. It encapsulates behaviours that affect multiple classes into reusable modules.

AOP languages use five main elements to modularize crosscutting concerns:

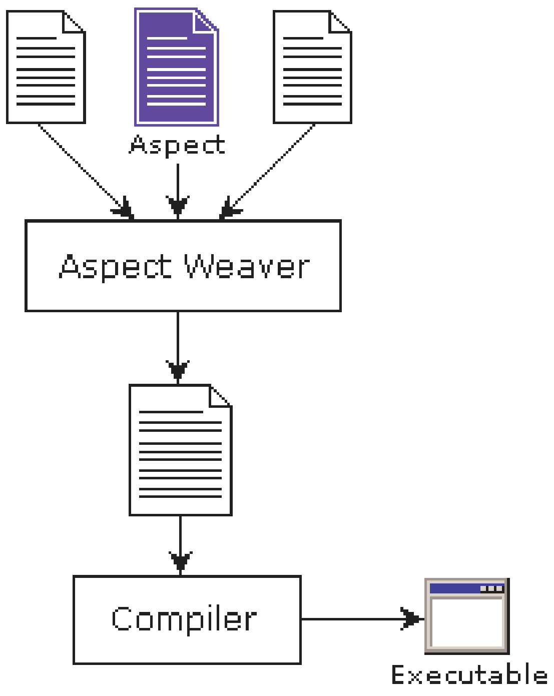
- Joinpoints
- A means of identifying joinpoints.
- A means of specifying behaviour at joinpoints.
- Encapsulated units combining joinpoints specifications and behaviour enhancements.
- A method of attachment of units to program.

The joinpoints are well-defined points in the execution of a program like method calls, field access, conditional checks, loop beginnings, assignments and object constructions. More about how the AOP language modularizes crosscutting concerns in section 4 where an implementation of AOP in Java, AspectJ, is introduced.

An important notice is that AOP does not make crosscutting concerns to anything else than a crosscutting concern. AOP will turn a tangled and scattered implementation of a crosscutting concern into a modularized implementation of a crosscutting concern. When AOP is applied, the concern will still be crosscutting, but the structure will be clearer.

3.2. Aspect Weaver

Compiling a program developed using AOP is a little bit different than usual compiling. AOP lets codes and aspects to be woven together by an aspect weaver before the program is compiled into an executable. Aspect weavers work by generating a joinpoint representation of the component program, and then executing (or compiling) the aspect programs with respect to it.



3.3. Development stages

Aspect-Oriented Programming involves three different development steps. To explain each step a simple credit card processing module example is used.

3.3.1. Aspectual decomposition

In the first step, the requirements are decomposed to identify crosscutting and common concerns. Here, the core concerns are separated from the crosscutting system level concerns.

In our simple credit card processing module, we could identify these three concerns: core credit card processing, logging and authentication.

3.3.2. Concern implementation

In the second step of the development, each concern is implemented separately.

In the credit card processing module, we would implement the core credit card processing unit, logging unit and authentication unit separately.

3.3.3. Aspectual recomposition

In the last step, an aspect integrator specifies recomposition rules by creating modularization units (aspects). The recomposition process is also called weaving or integrating.

Back to the credit card processing module, we would specify each operation's start and completion to be logged and that each operation must be authenticated before it proceeds with the business logic.

3.4. AOP Language Implementations

One might wonder what tools are available that enables aspect-oriented software development. The most documented and used tool for AOP is AspectJ, an AOP implementation in Java. Later on, in this report, AspectJ will be introduced.

There are many other implementations of AOP in other languages, in different shapes, for example, Squeak/Smalltalk, C, C++, C#, Perl, well, the most common languages are supported. All these implementations have something in common. They all are under development and/or only very early versions are available and this makes it hard to use AOP in "serious" projects.

4. ASPECTJ

AspectJ is a free aspect-oriented extension to Java. With just a few new constructs, AspectJ provides support for modular implementation of crosscutting concerns. In AspectJ's dynamic joinpoint model, joinpoints are well-defined points in the execution of the program. Since AspectJ's language construction extends the Java language, so every Java program is also a valid AspectJ program.

4.1. AOP implementation in AspectJ

The weaver in AspectJ is a compiler, and apart from the compiler, AspectJ also includes tools for debugging and documenting the code. The AspectJ compiler produces standard class files that follow Java bytecode specification. The bytecode can then be interpreted in any compliant Java Virtual Machine (JVM).

4.1.1. Aspects

In AspectJ, an aspect is declared by a keyword, "aspect", and is defined in terms of joinpoints, pointcuts and advices.

4.1.2. Joinpoints

The central concept in AOP, as in AspectJ, is the joinpoints. Joinpoints are well defined points in a program's execution. AspectJ limits the available joinpoints to:

- Constructor call
- Object and class initialization execution
- Method call
- Read/write access to a field
- Exception handler execution

Joinpoints are predefined in AspectJ. When writing an aspect, you will need to specify what joinpoint you want to take action on when defining pointcuts.

4.1.3. Pointcuts

Pointcuts, or pointcut designators, are program constructs to designate joinpoints and collect specific context at those points. The criteria can be explicit function names or function names specified by wildcards.

4.1.4. Advices

Pointcuts are used in the definition of advice. An advice is code that runs upon meeting certain conditions. In AspectJ there are three different advices; before advice, after advice and around advice.

4.2. HelloWorld, AOP version

HelloWorld may not be the best example to show the strength in AspectJ and AOP, but since it is a standard practise to start with that and any program that might show some of the benefits of AOP would take too much place in the report, here is a standard HelloWorld program.

First, we have a simple class containing two methods for printing messages:

```
// HelloWorld.java
public class HelloWorld
{
    public static void sayHello()
    {
        System.out.println("Hello World");
    }

    public static void sayHelloToPerson(String name)
    {
        System.out.println("Hello " + name);
    }
}
```

The aspect in this example will add greeting and gratitude manners. Before the program prints a message, it should print “Good day!”, and after the message, it should print “See you soon!”. So, this is how the implementation looks:

```
// MannersAspect.java
public aspect MannersAspect
{
    pointcut callSayMessage() : call(public static void HelloWorld.say*(..));
    before() : callSayMessage()
    {
        System.out.println("Good day!");
    }
    after() : callSayMessage()
    {
        System.out.println("See you soon!");
    }
}
```

In MannersAspect, a pointcut called callSayMessage is defined. This captures all calls to public static methods with names that start with say and that have any type of parameters (or none). In our example, it would capture sayHello and sayHelloToPerson. The pointcut has two advices, before and after reaching the callSayMessage printing “Good day!” and “See you soon!”.

What about the around advice then? How does it work? The around advice is called instead of the code that the pointcut refers to. In an around advice, a proceed call can be made to call the real code. This can be used as a verifier, to determine if a method should be called or not.

If you want to learn more about AspectJ, I recommend http://www.aspectj.org/, the official homepage for AspectJ. Here you can download AspectJ. There is also support for some of the most common development environments for the Java platform, for example JBuilder, Emacs, Forte and Eclipse. You will also find lots of papers, examples and tutorials there.

5. DISCUSSION

5.1. Will AOP replace OOP or other programming paradigms?

One question that arises early is the one above; will AOP replace the methods we know today? AOP is a very new programming paradigm, just outcome from the academic world. Sure, it can and probably will add new standard in programming, but it will not replace anything we use today. When OOP became the new standard in the programming world, procedural/functional programming did not disappear. They are a part of OOP, and so will OOP be in AOP, it will constitute the base of AOP, just as procedural is to OOP.

AOP complements object-oriented programming by facilitating another type of modularity that pulls together the prevalent implementation of a crosscutting concern into a single unit. When programming in AOP you use procedures, function, objects and aspects, each when most suitable. AOP is not the final answer to separation of concerns, there will be new developments in this area in the future, but it is likely that AOP will be a part of these solutions, just as OOP is a part of AOP.

5.2. AOP benefits

Since AOP strives to help the developer to separate concerns to overcome the problems with cross-cutting concerns, the benefits of AOP are the same benefits that come out of the ability to modularize implementations of crosscutting concerns.

AOP helps overcoming the problems caused by code tangling and code scattering. As we saw earlier, code scattering and tangling leads to some implications, like lower productivity, hard to reuse code and evolving the system.

AOP addresses each concern separately with minimal coupling, which resulting in a modularized implementation even in the presence of other crosscutting concerns. Such an implementation produces a system with less duplicated code. The modularized implementation also results in a system that is easier to understand and maintain. As the aspect modules can be unaware of other crosscutting concerns, it is also easy to add newer functionality by creating new aspects. The modularized implementation of crosscutting concern also makes code more encapsulated and this make code more reusable.

In the discussions above, AOP is only used in development concerns, for example logging. The system's behaviour does not change when adding or removing these aspects. There is however some other concerns AOP can be used to solve. For instance, an aspect can verify that an objects state is correct in method calls, and that parameters to methods are within the range of acceptable values. When the system works, is tested and going to be released, the aspect can be removed without affecting the system.

5.3. AOP disadvantages

As AOP is a new technology, it is not very well tested and documented. AOP is today only theory; nothing is tested in the "real world" with large scaled projects, so there are no guarantees that the theory works practical.

It is a limited amount of developing tools for AOP today. AspectJ is the leading AOP implementation, and there are others, but all in early versions. This makes it hard to estimate risks using AOP.

6. CONCLUSIONS

Aspect-Oriented Programming introduces a new way of handling crosscutting concerns, a problem hard to solve in ordinary object-oriented programming. As this is becoming a bigger and bigger problem as system grows and become more and more complicated, it is not unlikely that AOP will get its breakthrough in the near future. The problems with AOP today are the lack of development tools and documentation/results of other projects.

It is hard to predict what AOP will become in the future, but one thing is clear, AOP will be part of future programming paradigms.

7. REFERENCES

AOP (2002) http://www.bluefish.se/aop/ (2002-10-19)

Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse (2002) http://msdn.microsoft.com/msdnmag/issues/02/03/AOP/default.aspx (2002-10-19)

Aspect-Oriented Software Development (2002) http://aosd.net (2002-10-19)

AspectJ (2002) http://www.aspectj.org (2002-10-19)

Background: Aspect-Oriented Programming (2000) http://www.cs.man.ac.uk/cnc/mscprojects/aspect/node1.html (2002-10-19)

Hofmeister C., Nord R., Soni D. (2000) Applied Software Architecture, Addison-Wesley, USA

Hürsch W., Lopes C. (1995) Separation of Concerns, Technical report by the College of Computer Science, Northeastern University, USA

I want my AOP! (2002) http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html (2002-10-19)

Java Technology (2002) http://www-106.ibm.com/developerworks/java/library/j-aspectj/index.html?dwzone=java (2002-10-19)

Kiczales et al. (1997) Aspect-Oriented Programming, in *European Conference on Object Oriented Programming (ECOOP)*, Finland

Kiczales et al. (2001) An Overview of AspectJ, in *European Conference on Object Oriented Programming (ECOOP)*, Budapest, Hungary

Lamping (1999) The role of the base in aspect oriented programming, in *European Conference on Object Oriented Programming (ECOOP)*, Lisbon, Portugal