An Introduction to Aspect-Oriented Programming

Adolfo Gustavo Serra Seca Neto

December 13, 2003

Abstract

The objective of this report is to present Aspect-Oriented Programming (AOP). According to Gregor Kiczales [40], AOP is a "new evolution in the line of technology for separation of concerns—technology that allows design and code to be structured to reflect the way developers want to think about the system." First we will discuss the motivations for the development of AOP—the separation of *crosscutting* concerns and the alleged inability of current programming paradigms to support this separation. After that, two technologies for AOP, the AspectJ language and the JBoss AOP framework, will be presented and compared. Then we will briefly discuss some areas of research in AOP, such as refactoring to aspects, the impact of AOP on Software Engineering, the foundations of AOP and the relationship between AOP and Design Patterns [17]. Finally we will try to answer to the question: "Is AOP really necessary?".

1 Introduction

What is aspect-oriented programming? In the literature we can find several definitions:

- "AOP is a new evolution in the line of technology for separation of concerns—technology that allows design and code to be structured to reflect the way developers want to think about the system." [40]
- "AOP is a philosophy that is related to style of programming." [44]
- "AOP is a paradigm that extends the object-oriented paradigm by enabling you to write more maintanable code using units of software modularity called 'aspects'." [19]
- "AOP is one of the most promising solutions to the problem of creating clean, well-encapsulated objects without extraneous functionality. It is a paradigm that supports two fundamental goals:
  - Allow for the separation of concerns as appropriate for a host language.
  - Provide a mechanism for the description of concerns that crosscut other components." [33]

Although the history of AOP is recent, there have already appeared several tools that claim to be *aspect-oriented* [3]. But there are some questions that need to be answered if AOP is to be adopted by software practitioners:

- What is the motivation for the development of AOP?
- How does AOP solves some of the problems of Object-Oriented Programming (OOP)?
- Which are the technologies for AOP? What do they have in common? What are the differences amongst them?
- What are the new areas of research in AOP? What is the relationship between design patterns and AOP? What are the foundations of AOP?
- Is AOP really necessary?

1.1 Outline of the report

In Section 2, we will present and discuss the motivations for the development of AOP, as well as the benefits arising from adopting AOP. Then in Section 3, two technologies for AOP will be presented and compared: the AspectJ language and the JBoss AOP framework. Next, in Section 4, some areas of research related to AOP will be discussed and in Section 5 we will try to answer the question "Is AOP really necessary?". Finally, in Section 6 we will present our conclusions.

2 Motivation

The main motivation for the development of AOP was the alleged inability of OOP and other current programming paradigms to fully support the separation of concerns principle: [47]:

> The *separation of concerns* principle—decomposing a system into coherent, modular parts to localize changes to them—is a fundamental concept in software engineering. Over the past four decades, there have been some key developments that have transformed the way we think about concerns and their modularization during the development and evolution of software systems. These key developments include structured and procedural programming, object-oriented techniques, patterns and, more recently, aspect-oriented approaches. Aspect-oriented techniques do not advocate discarding existing separation of concerns mechanisms, e.g. object orientation. Instead the focus is to provide complementary mechanisms to support systematic identification, representation, modularization and composition of concerns that cut across an existing base separation and would otherwise be scattered across various modules.

According to AOP proponents, OOP is unable to properly represent crosscutting concerns. In order to further clarify this motivation, we present below definitions of concern, system-level concerns, core concern and crosscutting concerns [30]:

> A *concern* is a particular goal, concept or area of interest. In technology terms, a typical software system comprises several core and system-level concerns. For example, a credit card processing system's *core concern* would process payments, while its *system-level concerns* would handle logging, transaction integrity, authentication, security, performance, and so on. Many such concerns – known as *crosscutting concerns* – tend to affect multiple implementation modules. Using current programming methodologies, crosscutting concerns span over multiple modules, resulting in systems that are harder to design, understand, implement, and evolve.

Crosscutting structure results when one tries to represent crosscutting concerns with OOP. In order to explain what crosscutting structure means, Gregor Kiczales [40] has often used the `DisplayUpdating` aspect example [26]:

> (Crosscutting structure) means there are two or more structures (or decompositions) such that neither can fit neatly into the other. If we look at the classic AspectJ figure package example, shown in the diagram (see Figure 1) we see one structure, in black, that includes FigureElement, Point and Line; this structure talks about the state and drawing behavior of the figure elements. The second structure, in red, includes the DisplayUpdating aspect; it talks about how a change to the state of the figure elements should trigger a display refresh.
>
> One aspect of the setX, setY, setP1 and setP2 methods is described in the black figure element structure - how the setter methods change the state of figure elements. A different aspect of the same methods is described in the display updating structure - that they all refresh the display after being called.
>
> (I know this example seems trivial, and that it should actually keep track of exactly which displays each figure element is on, but in this simple form it really helps to see what crosscutting means.) In the figure, the DisplayUpdating aspect is a single unit. But that same behavior could not be localized as a single unit in the black structure. Trying to do so would force the code implementing the behavior to be scattered across all the methods that change display state. We say that the structure of the figure element state concern and the structure of the display refresh concern crosscut each other.
>
> Crosscutting structure differs from hierarchical structure. Crosscutting means carving the system up differently, not just removing details to get a more abstract view.
>
> Crosscutting is a deeper property than scattering. Scattering refers to the fact that in a given implementation, the code for a concern is spread out. Crosscutting refers to the inherent structure of the concerns, e.g. that the main graphical behavior and the display refresh behavior inherently crosscut each other. The goal of AOP is to enable the modular (not scattered) implementation of crosscutting concerns.

It is widely accepted that two symptoms indicate a problematic implementation of crosscutting concerns using current methodologies [36]:

(Code) scattering: Scattering is the condition where a concern is implemented in several non-contiguous places in the program.

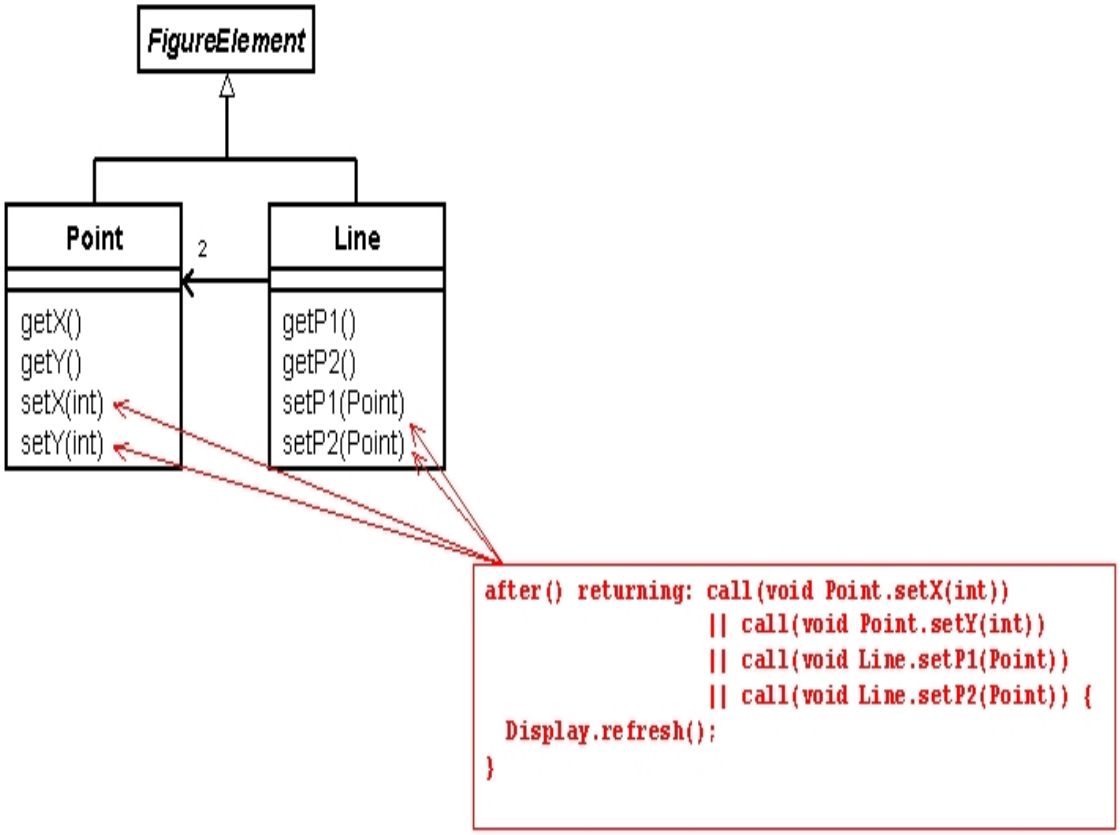

Figure 1: `DisplayUpdating` aspect example.

(Code) tangling: Tangling, the dual of scattering, occurs when several concerns overlap at a region in the program text. This hampers maintanability, as the programmer must mentally categorize statements in the program text by which concern they belong to.

2.1 The solution

Aspect-oriented programming has been proposed as a solution to the problem of representing crosscutting concerns. Some implementations of AOP have appeared and new ones continue to appear (see section 3). Although the term AOP was popularized by Kiczales *et al.* [29], techniques for the modularization of crosscutting concerns have been appearing for over a decade [47]. Actually, the term AOP serves to designate several different techniques with some features in common. However, there is not a standard implementation of AOP nor a common agreement on the *essential* characteristics of AOP (see subsection 4.2).

One important fact is that AOP is not meant to replace OOP. Actually, AOP and OOP are seen as complimentary paradigms and most implementations of AOP are heavily dependent on OOP features. For instance, in [7] the authors say: "We like to think of OOP as top-down software development, while AOP is left-right; they are completely orthogonal technologies that complement each other quite nicely." And although it is usually associated with OOP, AOP can be used to extend other programming paradigms as well.

There are several expected benefits of using AOP [47]:

- more readable and reusable code;
- a more natural mapping of system requirements to programming constructs;
- software that is more adaptable, maintainable and evolvable in the face of changing requirements.

It is also assumed that there will be an improvement in the comprehension and maintanability of complex programs by localizing behaviors (*concerns*) that would otherwise be scattered and tangled [36]. But there is not a common agreement that this will happen by adopting AOP (see subsection 5).

3 AOP Technologies

Nowadays, there are many implementations of AOP and many more are appearing. The AOSD Technology website [3] provides links to a collection of AOSD tools, methods and research projects. The page is divided into two sections:

Tools for Practicioners is a smaller list that includes only those tools which are actively supported and are being used in a variety of commercial projects. At the time of this writing, there were only six tools listed: AspectC++, AspectJ, AspectWerkz, JAC, JBoss AOP and Nanning.

Research Projects is a broad list of research projects developing AOSD technologies, techniques, methods, foundations etc.

Here we will be mainly interested in AspectJ (see subsection 3.1) and JBoss AOP[1] (see subsection 3.2). AspectJ is a seamless aspect-oriented extension to Java that enables the modular implementation of a wide range of crosscutting concerns. AspectJ gave origin to AspectC++ that, as the name suggests, is very much alike AspectJ. In fact, the AspectC++ project intends to extend the AspectJ approach to C/C++. Both tools are based on the linguistic approach to AOP (see subsection 4.2).

JBoss AOP is the Java AOP architecture used for JBoss application server. And according to [37], AspectWerkz and Nanning are very similar to JBoss AOP. They all use the so-called non-linguistic approach to AOP: no construct is added to the Java language. Instead, Java based frameworks and XML files are used to implement AOP concepts. In [37], the same example (a program that uses a Mixin class to output "Hello World" and creates trace calls to the `helloWorld` method on the Mixin) is implemented in the three systems in order to show how similar they are. The main difference between these systems lies in their implementation: AspectWerkz and JBoss AOP use ByteCode manipulation (byte code modification at runtime), while Nanning uses dynamic proxies. According to [37], from this example one can understand how these frameworks implement each of the four main principles of AOP:

Interception (Advice): The tracing is done using an Interceptor (also known as Advice).

[1] Although this seems to be an unimportant detail, these two are the only ones in the list whose current version numbers are 1 or more, indicating that they are possibly the most mature amongst them. AspectJ is in its 1.1.4 version while JBoss AOP's version is described as "jboss-aop-DR1", where the DR1 means Developers Release number 1.

Introduction: The output is done using and Introduction (also known as Mixin).

Inspection: The interceptor works out what method is being called.

Modularization: The "object" that is run is composed of a number of "things" - the Mixin and the Interceptor, each of which is a standalone module.

JAC (Java Aspect Components), the last implementation of AOP listed in [3], is a framework to build aspect-oriented distributed applications in Java [43]. The JAC project consists in developing an aspect-oriented middleware layer [52]. Its approach is "widely inspired from the AOP guidelines [29] and the AspectJ [27] programming concepts. In fact, JAC can be regarded as a research and implementation effort to apply an AspectJ-like model to dynamic distributed programming" [43].

3.1 AspectJ

AspectJ is "a seamless aspect-oriented extension to Java that enables the modular implementation of a wide range of crosscutting concerns" [4]. It is an AOP language based on the linguistic approach (see section 4.2): it is defined by a set of language constructs: join points, pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, while inter-type declarations statically affect the class hierarchy of a program. Here we will briefly present these constructs; a more complete coverage can be found in [51].

3.1.1 Join points

A *join point* is a well-defined point in the flow of a program. In object-oriented programs, there are several kinds of "things that happen" that are determined by the language. These are called the join points of the language. Join points consist of things like method calls, method executions, object instantiations, constructor executions, field references and handler executions. For instance, a call to a method `setX(int)` on an instance of `Point` class is a join point.

3.1.2 Pointcuts

Pointcuts are a means of referring to collections of join points and certain values at those join points [28]. Pointcuts are used in the definition of advice. A *pointcut designator*, or simply *pointcut*, selects particular join points by filtering out a subset of all join points, based on defined criteria. For example, the pointcut

```
pointcut setter(): target(Point) &&
               (call(void setX(int)) ||
               (call(void setY(int)));
```

picks out each call to `setX(int)` or `setY(int)` when called on an instance of `Point`.

3.1.3 Advice

Advice are method-like constructs used to define additional behavior at join points [28]. Advice are defined by associating actions to pointcuts. An *advice* in AspectJ is used to define additional code that should be executed at join points. There are three basic types of advice in AspectJ:

- The *before* advice runs just before the join points picked out by the pointcut;
- The *after* advice runs just after each join point picked out by the pointcut, regardless of whether it returns normally or throws an exception
- The *around* advice traps the execution of the join point; it runs instead of the join point. The original action associated with the join point can be invoked through the special `proceed` call.

For example, the advice

```
after(): setter() {
    System.out.println("was set");
}
```

prints a message immediately after calls to either `setX(int)` or `setY(int)` on an instance of `Point`.

3.1.4 Inter-type declarations

Static crosscutting affects the static type signature of the program. In AspectJ, *inter-type declarations* are the way to do static crosscutting. Aspects can declare members (fields, methods, and constructors) that are owned by other types: these are called inter-type members. For instance, with

```
boolean Server.disabled = false;
```

we can declare that each `Server` has a `boolean` field named `disabled`, initialized to `false`. And

```
public int Point.getX() { return this.x; }
```

declares that each `Point` has an `int` method named `getX` with no arguments that returns whatever `this.x`. Aspects can also declare that other types implement new interfaces or extend a new class.

3.1.5 Aspects

Kiczales defined an *aspect* as 'a well modularized implementation of a crosscutting concern'. Aspects are defined by aspect declarations, which have a form similar to that of class declarations. Aspects can implement interfaces and extend other aspects. However, aspects have no constructors and cannot be instantiated. Aspect declarations may include pointcut declarations, advice declarations, inter-type declarations as well as other kinds of declarations permitted in class declarations. In the example in subsection 3.3, the code for a complete AspectJ aspect will be presented.

3.2 JBoss AOP framework

JBoss is an application server that provides enterprise-class security, transaction support, resource management, load balancing, and clustering. Recently, a framework for AOP was incorporated into JBoss. The JBoss AOP framework is (as its name suggests) a framework for programming with aspects.

JBoss uses a non-linguistic approach to AOP. Therefore, differently from AspectJ, no new keywords are introduced to the Java language (for example, there is no aspect construct). Classes and interfaces from the framework in conjunction with XML files are used to implement crosscutting concerns. Below we present some definitions that will be useful for understanding the example in subsection 3.3.

3.2.1 Interceptors

In JBoss AOP, advices are implemented using *interceptors* [7]. The programmer can define interceptors that intercept method invocations, constructor invocations, and field access. An advice is logic that is triggered by a certain event. It is behavior that can be inserted between a caller and a callee, a method invoker and the actual method.

3.2.2 Introductions

Introductions are the way to add methods or fields to an existing class. They are equivalent to inter-type declarations in AspectJ, although the syntax for its use in Java programs can be a bit different. They allow one to change the interfaces an existing class currently implements and to introduce a mix-in class that implements that new interface. With introductions it is possible to bring multiple inheritance to plain Java classes.

3.2.3 Metadata

Metadata is additional information that can be attached to a class either statically or at runtime. In JBoss AOP it is possible to attach metadata dynamically to a given instance of an object.

3.2.4 Pointcuts

Pointcuts tell the AOP framework which interceptors to bind to which classes, what metadata to apply to which classes, or what classes to which an introduction will be introduced. Pointcuts define how various AOP features are applied to the classes in JBoss AOP applications.

3.2.5 XML files

In JBoss AOP, XML files are used for the attachment of interceptors, introductions and metadata to Java classes as well as for the definition of pointcuts. More details about the syntax of JBoss AOP XML files can be found in [6].

3.3 Example

Here we will present a simple example to show how JBoss AOP and AspectJ give support to AOP features. It is a program that traces calls to the methods and the constructor of a class. The same base class will be used in both implementations: it is a plain Java class called `AOPTest` – an extended version of the `POJO`[2] class shown in [7]. This class has only one constructor and two methods: `helloWorld` and `byeWorld`. The static method `main` is used only to create an instance of `AOPTest` and call its methods. Following is the code for this class:

```
public class AOPTest
{
   public AOPTest() {}

   public void helloWorld() { System.out.println("Hello World!"); }
   public void byeWorld() { System.out.println("Bye, bye, World!"); }

   public static void main(String[] args)
   {
      AOPTest at = new AOPTest();
      at.helloWorld();
      at.byeWorld();
   }
}
```

3.3.1 JBoss AOP implementation

In order to trace calls to the constructors and methods of the class above, in JBoss AOP one has to write an interceptor class (`TracingInterceptor`), a class that implements the Interceptor interface below[3]:

```
public interface Interceptor
{
   public String getName();
   public InvocationResponse invoke(Invocation invocation) throws Throwable;
}
```

So the `TracingInterceptor` class is implemented as follows:

```
public class TracingInterceptor implements Interceptor
{
   public String getName() { return "TracingInterceptor"; }

   public InvocationResponse invoke(Invocation invocation) throws Throwable
   {
      String message = null;
      if (invocation.getType() == InvocationType.METHOD)
      {
         MethodInvocation mi = (MethodInvocation)invocation;
         message = "method: " + mi.method.getName();
      }
      else if (invocation.getType() == InvocationType.CONSTRUCTOR)
      {
         ConstructorInvocation ci = (ConstructorInvocation)invocation;
```

[2]POJO is an acronym for *Plain Old Java Object.*

[3]All interceptors in JBoss AOP must implement the org.jboss.aop.Interceptor interface.

```
         message = "constructor: " + ci.constructor.getName();
      }
      else
      {
         // Do nothing for fields.  Just too verbose
         return invocation.invokeNext();
      }
      System.out.println("<<Tracing>> Entering " + message);
      InvocationResponse rsp = invocation.invokeNext();
      System.out.println("<<Tracing>> Leaving " + message);
      return rsp;
   }
}
```

Notice that there is no construct here that is not from the Java language. After defining the interceptor, we have to attach the interceptor to the `AOPTest` class. To do this we need to define a pointcut. For JBoss AOP, pointcuts are defined within an XML file. The one below attachs `TracingInterceptor` to `AOPTest`:

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
   <interceptor-pointcut class="AOPTest">
     <interceptors>
       <interceptor class="TracingInterceptor"/>
     </interceptors>
   </interceptor-pointcut>
</aop>
```

Then, we can compile the two classes above with a standard Java compiler, not forgetting to import the classes from the JBoss AOP framework that are used in `TracingInterceptor`. After that, we have to execute the `AOPTest` class with JBoss AOP class loader. The default Java class loader is not used, because JBoss AOP does bytecode manipulation on aspected classes as they are loaded into the virtual machine. It is the JBoss AOP class loader that dynamically weaves the bytecode for the `AOPTest` class with the bytecode for the `TracingInterceptor` class by looking at the XML file[4]. The following results are written on the screen, where we can see clearly how the code from `TracingInterceptor` has intercepted the base class code:

```
<<Tracing>> Entering method: main
<<Tracing>> Entering constructor: AOPTest
<<Tracing>> Leaving constructor: AOPTest
<<Tracing>> Entering method: helloWorld
Hello World!
<<Tracing>> Leaving method: helloWorld
<<Tracing>> Entering method: byeWorld
Bye, bye, World!
<<Tracing>> Leaving method: byeWorld
<<Tracing>> Leaving method: main
```

3.3.2 AspectJ implementation

In order to trace exactly the same calls advised by the JBoss AOP implementation, we have to define the `Aspect_AOPTest` aspect below:

```
public aspect Aspect_AOPTest {

      pointcut AOPTestOperation() : execution(* AOPTest.* (..));

      pointcut AOPTestConstructor() : initialization(AOPTest.new (..));
```

[4]Details on how this class loader is used can be found in [7]. And a more complete coverage of JBoss AOP features is given in [6].

```
      before() : AOPTestOperation() {
         System.out.println(
            "<<Tracing>> Entering method: "
               + thisJoinPoint.getSignature().toString());
      }

      after() : AOPTestOperation() {
         System.out.println(
            "<<Tracing>> Leaving method: "
               + thisJoinPoint.getSignature().toString());
      }

      before() : AOPTestConstructor() {
         System.out.println(
            "<<Tracing>> Entering constructor: "
               + thisJoinPoint.getSignature().toString());
      }

      after() : AOPTestConstructor() {
         System.out.println(
            "<<Tracing>> Leaving constructor: "
               + thisJoinPoint.getSignature().toString());
      }
}
```

The aspect above defines two pointcuts: the first one for `AOPTest` operations (`helloWorld()`, `byeWorld()` and `main`) and the second for the `AOPTest` constructor. After that, four advice[5] are defined for printing the tracing messages: two `before` and two `after` advices.

This aspect and the `AOPTest` class must be compiled by an AspectJ compiler. It is the compiler that weaves the bytecode for the base class with the instructions introduced by the aspect. Then, the resulting bytecode for the base class can be run with a standard Java interpreter. The result is almost the same as the one we have obtained with JBoss AOP:

```
<<Tracing>> Entering method: void AOPTest.main(String[])
<<Tracing>> Entering constructor: AOPTest()
<<Tracing>> Leaving constructor: AOPTest()
<<Tracing>> Entering method: void AOPTest.helloWorld()
Hello World!
<<Tracing>> Leaving method: void AOPTest.helloWorld()
<<Tracing>> Entering method: void AOPTest.byeWorld()
Bye, bye, World!
<<Tracing>> Leaving method: void AOPTest.byeWorld()
<<Tracing>> Leaving method: void AOPTest.main(String[])
```

3.3.3 Comparing AspectJ and JBoss

In my opinion, AspectJ seems to be the most promising approach to adding support for aspects to Java. At least, it is regarded as the most mature approach at the time of this writing. It is clear that the main differences between AspectJ and JBoss AOP are related to the fact that the first one adopts the linguistic approach and the second the non-linguistic one. Therefore, when programming in AspectJ one needs to learn new constructs and use a new compiler which also works as a weaver. On the other hand, in JBoss AOP one has to learn a framework and instantiate its classes and interfaces when necessary. Besides that, one has to learn how to write the needed XML files and the JBoss AOP must have total control over the Java class loader to work.

Nothwithstanding, basically the same things can be made with both tools (with only small differences). The "JBoss AOP vs. AspectJ" page [1] shows source code demonstrating how both systems can implement before, after

[5] In this example, two advice would be enough if we had used the `around` advice, but the code would become less clear.

and around advices, as well the related concepts of Introductions (in JBoss AOP) and Inter-type Declarations (in AspectJ). In summary, in both AOP technologies one can have an application with one or more Java classes written by an oblivious programmer[6] and then write functionality that crosscuts the existing application in many ways.

4 Areas of Research

The development of AOP has given rise to several areas of research. For instance, an emerging area of study is on the use of aspects for refactoring [45]; it is possible to refactor plain OO systems to aspect-oriented (AO) systems as well as to refactor existing AO systems [2]. Besides that, the apperance of new programming constructs usually leads to new developments in Software Engineering. There are several works trying develop systematic means for the identification, modularisation, representation and composition of crosscutting concerns (the aspects) throughout the software development life cycle; aspect-orientation has become a paradigm for all stages of this lifecycle, such as requirements engineering, architecture design and detailed design. For instance, there are several works trying to extend UML [10] to better represent AOP concepts.

In the following subsections we will discuss two of these research areas which we regard as very important for developers:

- the efforts in understanding the relationship between design patterns and AOP and
- the studies on the foundations of AOP.

4.1 Design Patterns and AOP

Software design patterns [17, 8, 49] offer flexible solutions to common software development problems. The research on design patterns and AOP has three main subjects:

- Using design patterns to explain and understand AOP;
- Implementation of known design patterns in AOP languages;
- New patterns for Aspect Oriented Software Design.

4.1.1 Using design patterns to explain and understand AOP

Since their introduction in [17], patterns have been used, among other things, to explain software design models. Therefore, they naturally have been used to explain issues regarding implementations of AOP [50, 6]. For instance, the Interceptor pattern [49] is used in JBoss AOP. And according to [50], part of the functionality of AOP can be implemented by design patterns, such as the separation of code with specific functions. For instance, we can use the Adapter pattern [17] in order to add behaviour to a method of a class. However, the implementation of the Adapter pattern presented there leads to a duplication of code, while the AOP implementation does not.

4.1.2 Implementation of known design patterns in AOP languages

The implementation of known design patterns in AOP languages can be used to verify the suitability of these languages to software development. A number of GoF patterns (the design patterns presented in [17]) involve crosscutting structures in the relationship between roles in the pattern and classes in each instance of the pattern. The invasive nature of pattern code, and its scattering and tangling with other code also creates documentation problems. Therefore they are natural candidates for implementaion with AOP.

A work by Jan Hahnemann and Gregor Kiczales [20] has put together a compilation of GoF Design Patterns in Java and AspectJ. According to the authors, their results "indicate that using AspectJ improves the implementation of many GoF patterns. In some cases this is reflected in a new solution structure with fewer or different participants; in other cases, only the implementation of the classes in the originally proposed solution changes (for example, by moving pattern code from the participants into an aspect)."

They also state that patterns with crosscutting structure between roles and participant classes see the most improvement. The improvement comes primarily from modularizing the implementation of the pattern. This is directly reflected in the implementation being textually localized. Localizing patterm implementation provides

[6] An oblivious programmer is one that does not necessarily know anything about AOP and its constructs.

inherent code comprehensibility benefits – the existence of a single named unit of pattern code makes the presence and structure of the pattern more explicit. In addition, it provides an anchor for improved documentation of the code.

4.1.3 New patterns for Aspect Oriented Software Design

Besides implementing well-known design patterns, several researchers are trying to find common patterns in AOP that do not appear on object-oriented (OO) ones, and on defining refactorings to aspect-oriented code [46]. This area is promising because as time goes by and experience is gained with implementing AOP systems, several patterns specific to AOP will naturally emerge.

4.2 Foundations of AOP

According to [39], a common agreement on the *essential* characteristics of AOP is still missing. Such an agreement would yield a *definition* of the essential characteristics of AOP which could help identify whether an existing or new programming approach is AOP or not. For instance, the definition should draw the line between an AOP environment and a meta-object protocol or an OOP environment. Two other goals for work on foundations of AOP are *(ii)* developing a classification scheme for AOP systems and *(iii)* developing a common terminology for AOP. The only attempt to define AOP independently from a concrete approach is by Filman et al., who stated that "AOP is quantification and obliviousness" [13, 12]:

> The distinguishing characteristic of Aspect-Oriented Programming (AOP) systems is that they allow programming by making quantified programmatic assertions over programs written by programmers oblivious to such assertions.

However, according to [39], there are some issues in AOP that are essential but are not addressed by this definition, such as, for instance, issues of abstraction, structuring and modularisation for the assertion part of the programming model. There is much work to be done in this area.

One basic yet important distinction is the one between linguistic and non-linguistic approaches to AOP. When new AOP languages are created or current languages are extended with new constructs for AOP, such as in AspectJ and AspectC++, we say that we have the linguistic approach. On the other hand, in non-linguistic approaches no new language constructs are needed: "non-linguistic approaches allow programming in an AOP style, e.g., by offering a restricted meta-object protocol or supporting definition of programming units which act as aspects, for instance in a framework" [39]. JBoss AOP, AspectWerkz and Nanning are representatives of this approach.

5 Is AOP really necessary?

As with any other new technology, there are advantages and disadvantages in using AOP. The advantages proclaimed by AOP supporters were stated in subsection 2.1. But there are also some disadvantages appointed by adversaries of AOP. One of the main disadvantages is the need to get used to different language constructs (in the linguistic approach) or learn frameworks (in the non-linguistic approach). Another disadvantage is that it demands a new compiler (in AspectJ) or class loaders (in JBoss AOP), as well as development tools. Some other weaknesses specific to AspectJ are discussed in [50].

Nikita Ivanov, in an Internet forum [25], wrote that "AOP is far from being widely accepted by the software engineering community." There he raised some issues, among which I highlight a few:

1. AspectJ is a whole new language (i.e. couple of dozens of new keywords on top of Java) and it is a way too much to pay for a new programming technique. OOP-language based AOP framework should give better value.
2. Besides logging and security check examples, are there any real-life applications of AOP large enough for good statistical analyses?
3. Free-defined, unrestricted pointcuts promote spaghetti-like coding and ad-hoc design.
4. AOP may introduce excessively tight coupling between aspect and advised code. Advised code cannot be freely changed if there is a mutating aspect attached to it.
5. Since aspects are defined in a different language (or XML), the logic gets split into two semantically different places further complicating coupling.

And Ralph Johnson, in response to an email messsage, said (cited in [16]):

> My reservation of Aspect-Oriented Programming is that we already have fairly good techniques for separating aspects of programs, and we don't use them. I don't think the real problem will be solved by making better techniques for separating aspects. We don't know what should be the aspects that need separating, and we don't know when it is worth separating them and when it is not.

It is clear that there is a price to pay when one adopts AOP. In my opinion, for the reasons presented in section 2, AOP is necessary at least for some specific projects and deserves more experiments in order to verify if it is really useful for software development in general. My own research project with my advisor Prof. Marcelo Finger is on the application of AOP to the modularization of Tableaux proof strategies in automated theorem provers. We think that the ability to handle crosscutting concerns will be essential for a better modularization of strategies.

6 Conclusion

In this report we have seen that the main motivation for the development of AOP was the alleged inability of OOP and other current programming paradigms to fully support the separation of concerns principle. According to its proponents, AOP solves some problems of OOP by allowing an adequate representation of crosscutting concerns.

Two technologies for AOP, AspectJ and JBoss AOP, have been presented and compared. In my opinion, AspectJ seems to be the most mature approach to adding support for aspects to Java. The main differences between AspectJ and JBoss AOP are related to the fact that the first one adopts the linguistic approach to AOP and the second the non-linguistic one. In spite of that, basically the same things can be made with both tools; one can have an application with one or more Java classes written by an oblivious programmer, and then write functionality that crosscuts the existing application in many ways. The comparison between the two technologies was illustrated by a tracing example.

The development of AOP has given rise to several areas of research. We have stated that the development of AOP made it possible to think of refactoring plain OO systems to AO systems as well as refactoring existing AO systems. Besides affecting coding, aspect-orientation has become a paradigm for all stages of the software development lifecycle, such as requirements engineering, architecture design and detailed design, leading to the need of extending existing modelling formalisms such as UML. But even more important for developers are the efforts in understanding the relationship between design patterns and AOP as well as the studies on the foundations of AOP.

Finally, we concluded that AOP is necessary at least for some specific projects and deserves more experiments in order to verify if it is really useful for software development in general.

References

[1] asato (asato@ncfreak.com). JBoss AOP vs. AspectJ, June 2003. http://www.ncfreak.com/asato/doc/jboss-aop-vs-aspectj.html.

[2] asato (asato@ncfreak.com). Refactoring in AspectJ, September 2003. http://www.ncfreak.com/asato/doc/refactoring-in-aspectj.html.

[3] Aspect-Oriented Software Association. AOSD Technology, December 2003. http://www.aosd.net/technology/index.php.

[4] Aspect-Oriented Software Association. aosd.net, December 2003. http://www.aosd.net.

[5] Kent Beck and Ralph Johnson. Patterns generate architectures. *Lecture Notes in Computer Science*, 821:139–149, 1994.

[6] Bill Burke. JBoss Aspect Oriented Programming, 2003. http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop.

[7] Bill Burke and Adrian Brock. Aspect Oriented Programming and JBoss, May 2003. http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html.

[8] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John-Wiley and Sons, 1996.

[9] Carlos J. P. de Lucena. Separation of concerns and multi-agent systems group, 2003. http://www.teccomm.les.inf.puc-rio.br/SoCagents/.

[10] José Eduardo Zindel Deboni. *Modelagem orientada a objetos com a UML*. Futura, 2003.

[11] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44, 2001.

[12] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA)*, 2000. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf.

[13] Robert E. Filman. What is Aspect-Oriented Programming, Revisited. In *Workshop on Advanced Separation of Concerns, 15th European Conference on Object-Oriented Programming*, June 2001.

[14] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Object Technology Series. Addison-Wesley, Reading, Massachusetts, 1997.

[15] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Adisson-Wesley Longman, 1999.

[16] Martin Fowler. Who Needs an Architect? *IEEE Software*, (1):2–4, August 2003.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Adisson-Wesley, 1994.

[18] Joseph D. Gradecki and Nicholas Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, 2003.

[19] William Grosso. Aspect-Oriented Programming and AspectJ. *Dr.Dobbs Journal*, August 2002.

[20] Jan Hahnemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, ans Applications (OOPSLA)*, pages 161–173, November 2002. http://www.cs.ubc.ca/~jan/AOPDS/.

[21] Stefan Hanenberg and Rainer Unland. A proposal for classifying tangled code, 2002. http://citeseer.nj.nec.com/hanenberg02proposal.html.

[22] Richard Hightower and Nicholas Lesiecki. *Java Tools for eXtreme Programming - Mastering Open Source Tools including Ant, JUnit and Cactus*. Wiley, 2002.

[23] Object Technology International Inc. Eclipse Platform Technical Overview, February 2003.

[24] Wes Isberg and the AspectJ team. Get Test-Inoculated! *Software Development*, May 2002.

[25] Nikita Ivanov. AOP: revolutionary, evolutionary or rudimentary?, December 2003. http://www.theserverside.com/home/thread.jsp?thread_id=19499&article_count=81.

[26] Gregor Kiczales. Interview with Gregor Kiczales, July 2003. http://www.theserverside.com/events/videos/GregorKiczalesText/interview.jsp.

[27] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44:59–65, 2001.

[28] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[29] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[30] Ramnivas Laddad. I want my AOP!, Part 1. *Java World*, January 2002.

[31] Ramnivas Laddad. I want my AOP!, Part 2. *Java World*, March 2002.

[32] Ramnivas Laddad. I want my AOP!, Part 3. *Java World*, April 2002.

[33] Ramnivas Laddad. *AspectJ in Action*. Manning, 2003.

[34] Ken Wing Kuen Lee. An Introduction to Aspect-Oriented Programming, August 2002. Reading Assignment. COMP 610E 2002 Spring Software Development of E-Business Applications. The Hong Kong University of Science and Tcehnology.

[35] Nicholas Lesiecki. Test flexibly with AspectJ and mock objects. *IBM's Developer Works*, May 2002.

[36] Karl Liebebrherr, David H. Lorenz, and Johan Ovlinger. aspectual Collaborations: Combining Modules and Aspects. *The Computer Journal*, 542–565, 2003.

[37] Nick Lothian. A Journey through three Aspect Oriented Frameworks, June 2003. http://www.mackmo.com/nick/blog/java/~permalink=aop-framework.html.

[38] Katharina Mehner and Awais Rashid. GEMA: A Generic Model for AOP. In *Belgian and Dutch Workshop on AOP, Twente, The Netherlands*, 2003. http://www.comp.lancs.ac.uk/computing/oop/Publications.php.

[39] Katharina Mehner and Awais Rashid. Towards a Generic Model for AOP (GEMA). Technical Report CSEG/1/03, Computing Department, Lancaster University, 2003. http://www.comp.lancs.ac.uk/computing/oop/Publications.php.

[40] Tzilla Elrad (Moderator), Mehmet Aksit, Gregor Kiczales, Karl Liebherr, and Harold Ossher. Discussing Aspects of AOP. *Communications of the ACM*, 44:33–38, October 2001.

[41] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten. Does Aspect-Oriented Programming Work? *Communications of the ACM*, 44:75–77, 2001.

[42] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[43] Renaud Pawlak, Laurence Duchien, Gerard Florin, Fabrice Legond-Aubry, Lionel Seinturier, and Laurent Martelli. JAC: An Aspect-Based Distributed Dynamic Framework, December 2002. http://jac.objectweb.org/docs/JAC.pdf.

[44] Renaud Pawlak and the JAC development team. JAC - A Framework for Aspect-Oriented Programming in Java, December 2003. http://jac.objectweb.org/.

[45] Carlos Perez. Refactoring to Aspects, May 2003. http://www.artima.com/weblogs/viewpost.jsp?thread=4842.

[46] Eduardo Kessler Piveta and Luiz Carlos Zancanella. Observer Pattern using Aspect-Oriented Programming. In *Third Latin American Conference on Pattern Languages of Programming*, 2003.

[47] Awais Rashid and Lynne Blair. Editorial: Aspect-oriented Programming and Separation of Crosscutting Concerns. *The Computer Journal*, 46(5):527–528, 2003.

[48] Charles Richter. *Designing Flexible Object-Oriented Systems with UML*. Macmillan Technical Publishing, 1999.

[49] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[50] Sérgio Soares and Paulo Borba. AspectJ - Programação orientada a aspectos em Java. *Tutorial no SBLP 2002, 6o. Simpósio Brasileiro de Linguagens de Programação. 5 a 7 de Junho, PUC-Rio, Rio de Janeiro, Brasil*, pages 39–55, 2002.

[51] AspectJ Team. The AspectJ Programming Guide, December 2003. http://dev.eclipse.org/viewcvs/indextech.cgi/ checkout /aspectj-home/doc/progguide/index.html.

[52] JAC Team. The JAC Project, December 2003. http://jac.objectweb.org/.

[53] Michael J. Yuan and Norman Richards. Lightweight Aspect-Oriented Programming - Puuting the Interceptor pattern to work. *Dr.Dobbs Journal*, August 2003.