Aspect Oriented Programming – Comparing Programming Languages **Arquitectura de Sistemas de Software**

Grupo Nereus

Universidade do Porto

Faculdade de Engenharia

FEUP

Luís Miguel Lourenço (ei01007@fe.up.pt)
Nuno Cerqueira (ei01036@fe.up.pt)
Pedro Côrte-Real (ei01016@fe.up.pt)
Rui Barbosa (ei01018@fe.up.pt)

June 12, 2005

Abstract

This report gives an introduction to what Aspect Oriented Programming (**AOP**) is and is not and proceeds by showing how the concept is used in several programming languages. The chosen languages are **C++**, **Java** and **Ruby** and the comparison will try to show how the **AOP** concept is used differently as it is introduced in the programming concepts of the specific language.

Contents

List of Figures

List of Code Listings

Chapter 1

Introduction

In this report we describe what is Aspect Oriented programming, some of it's implications and how it complements other programming paradigms such as Procedural-Oriented Programming (**POP**) and Object-Oriented Programming(**OOP**). After introducing the concept in it's abstract and language-independent form we present concrete examples in a few programming languages and implement a small library using **AOP**. We have chosen some features for our library that make it possible to see how the choice of language and the way **AOP** is implemented on said language influence the way we can express ourselves in our programming.

We assume our readership has some background on programming language and object orientation theory. We also assume a passing knowledge of Java and C++. We will offer some explanation on Ruby and more esoteric language features, as needed.

1.1 What is AOP

If we look at the programming field from a distance we'll generally divide programming languages into two main camps: procedural programming (**POP**) and object-oriented programming (**OOP**). Like all programming techniques these were invented as ways to structure programming to achieve conceptual separation of components. This separation is obtained by dividing programs into sub-structures of some kind so that we can think and program each one as a reasonably separate and autonomous entity.

When using **POP** we divide our program into separate functions or procedures and compose the program flow by redirecting it from one to another in what is essentially a linear structure. With **OOP** a second dimension is added as we structure our code into encapsulated units linked to each other.

The control flow is now defined dynamically by the object structure and polymorphism. **AOP** tries to add a third dimension to this structure by describing so called aspects that cross-cut the structure of the program.

1.2 What is an Aspect

The aspect is the central structural concept of **AOP**, as the object is of **OOP**. An aspect is a concern of the program's domain model that cross-cuts it by having influence in distinct parts of the structure. With **OOP** such concerns are implemented by code that is distributed across the structure. **AOP** attempts to achieve a better code structure by aggregating such scattered code into a unified structure representing the whole aspect.

A simple example of an aspect is locking in threaded applications. If we have a program done with **POP** or **OOP** that was originally single-threaded and we want to extend it to a multi-threaded design we'll need to add explicit locking to several places throughout the code. With **AOP** the code would not have such explicit locking. Instead the programmer creates a new aspect where he defines which areas of the code need to be protected by which lock, and the locks themselves, all in contiguous code.

The places in the code which can be altered by an aspect are called the join points. Each aspect is defined by a set of point cuts and advices. Advices are pieces of code and point cuts select from the possible join points where to apply that code to. The programmer specifies a point cut, which selects a group from the possible join points, where to weave the advice. The aspect is code that affects a subset of the whole system's code. This subset does not need to follow the **OOP** or **POP** structures hence it's cross-cutting nature.

1.3 AOP is not Metaprogramming

A metaprogram is a program whose problem domain are other programs [8]. A simple example of such program is a compiler that works with a program in a certain programming language as input to produce an equivalent of that program as output. In this case the metaprogramming is the actual function of the code. However, more advanced uses of metaprogramming are those where the metaprogram is a way to generate or help generate the actual program for the original program domain. In these cases metaprogramming is not an end in itself but a tool to help program structuring.

There has been discussion on whether **AOP** is metaprogramming [2]. The pattern of join points and advices can be described as a program alteration

in the spirit of metaprogramming. However, in the same way we don't think as functions as assembly jump and return instructions we shouldn't think of **AOP** as a form of macro programming[1]. **AOP** is intended as a programming concept and not a specific instance of that concept. And just like **OOP** can be implemented in different ways, **AOP** does not need to be implemented as metaprogramming. If some **OOP** language was extended to support the **AOP** concepts, join points and advices would be integrated in the structure of the language and we wouldn't see the use of **AOP** as a transformation of the structure of the program but as part of it.

AOP is not explicit metaprogramming, it's a concept that allows us use direct semantics to program cross-cutting concerns [14]. Although **AOP** can be implemented as a meta-program, it's a higher-level concept. Metaprogramming is a generic technique that can be used for much more than implementing **AOP**. Inversely **AOP** is a higher-level technique designed for a specific problem. Looking at it as a form of meta-programming destroys it's semantic meaning the same way as looking at recursive programming as stack fiddling destroys the semantics of recursion.

This is an important distinction to make because of some common misconceptions. Programmers using relatively low level languages sometimes mistake **AOP** as being a way to add expressive power to a language. While that will certainly happen if **AOP** is added to such languages, it's important not to get hung up on that. That's just a byproduct of what **AOP** wants to achieve.

Similarly, programmers in higher level languages, with extensive reflection capabilities, sometimes look down to **AOP** as a less powerful mechanism than what they're used to. This is also a mistake. **AOP** is about expressing coding crosscutting concerns using directed semantics. It's about having a common, "easy to think about", structure to solve a common problem.

1.4 AOP Misconceptions

Because **AOP** can be modeled as a form of metaprogramming or just as a dynamic feature for a language, common misconceptions appear. Kiczales classifies these[15] in three big groups: Generalization, Utility and Mechanism errors.

Generalization errors are those that equate **AOP** to wider-reaching goals such as Separation of Concerns (**SOC**) or to specific implementations of the concept like AspectJ. This kind of error misses the point of **AOP** in a subtle way by not concentrating on the main message. Like **OOP**, **AOP** is a way to

[1] Macro programming as in Lisp macro programming.

structure programs and it is neither a specific technique or implementation, nor a wide-reaching concept like **SOC**. **AOP** helps us in achieving **SOC**, but they're different concepts.

Utility errors are made when it is thought that **AOP** should only be used for non-essential functions of the program or design, like logging. Although logging is a very good case for an **AOP** implementation (we will present one later on), there are several other uses for **AOP**. Kiczales finds three main kinds of aspects: General-purpose, Domain-specific and Application-specific. As **AOP** is used more in the design phase of software we will evolve from general-purpose aspects, that interact with the rest of the system without side-effects to aspects that concern the domain of the application to aspects that are specific of the application. These three kinds of aspects graduate **AOP** to a technique at the same level as **OOP** since it will be used in the main structure of an application, instead of it's currently more usual bolted on after the fact nature.

Equating **AOP** to a specific mechanism is another common misconception. Programmers of highly dynamic languages like Lisp tend to understand the **AOP** concept by equating it to specific features of their chosen language. Although this might help you understand what the program does when an aspect is used, it misses the point in the same way than saying that **OOP** is just about encapsulation. **AOP** is supported by specific techniques just like **OOP** is implemented in a specific way in C++ or Smalltalk.

1.5 AOP as a Modularization Technique

POP and **OOP** give us a way to modularize the functional components of a design by separating them into self-contained structures of some sort. Crosscutting concerns don't lend themselves easily to this kind of composition. **AOP** gives us a way to group the code for these concerns together into a single structure. By having the crosscutting concern implemented in a grouped way we can replace it's implementation much more easily.

A micro-scale effect of this new modularization is the immediate effect on the quality of specific applications. Code is hard to write, and sometimes the best way to develop it is to prototype and throw away, learning something in the process. So far, if you wanted to throw away all of your locking code and start over, a major restructuring of your application would be needed. With **AOP** this can be achieved by replacing the old locking aspect with a new one.

A macro-scale effect could happen in the structure of programming frameworks. Frameworks are designed to be instantiated for a specific application

by composition and/or sub-classing, but there's limited control of the inner-workings of the framework itself, especially for crosscutting concerns such as locking, caching or transaction handling. For this reason big platform frameworks like **Java**'s **EJB** tend to grow by trying to be everything for everyone. **AOP** can help us simplify these frameworks giving more intimate and precise control to application programmers. Hopefully this can lead to a net reduction in the size of frameworks and an increase in their modularity. This change should reduce the risk of committing to the use of a specific framework in an application, since it will be possible to change its implementation of caching or locking if restrictions not foreseen by the framework developer were imposed by the application or the execution environment. More importantly, this can be done where it makes more sense, in terms of structuring and organisation: in the application part and not in the generic framework code.

Chapter 2

Aspects in Programming Languages

To get a feel of how aspects can be used and defined across different languages, we programmed a simple example using C++, Java and Ruby. Since it's possible to implement **AOP** in different ways across the languages and different choices are available, we had to choose some implementations. Java in particular has lots of different implementations, such as AspectJ, AspectWerkz and JBoss [12][11][13]. Naturally the language dictates the kinds of implementations that are feasible. It would be very hard to add **AOP** to C++ without using a preprocessor or altering the language.

The choice of these three different languages has to do with their "dynamism". C++ is a statically typed language, with very limited reflection capabilities (obtained by using the RTTI[1]). Ruby is a dynamically typed language with powerful reflection capabilities. It features capabilities such as open classes and meta classes. Java, while a descendant of the Algol family, and consequently of C/C++, has more powerful reflection mechanisms than C++ and features more runtime flexibility, such as class loadin. So, it sits in between Ruby and C++ as to its dynamic capabilities. Naturally this dynamism has a price in performance and memory consumption for the programs written in languages which have more of it. However, more importantly to our report, it changes the way we can express ourselves when we want to perform separation of concerns, particularly crosscutting concerns.

In each of the following sections a simple **AOP** example is shown in the three languages. The example shows how to use **AOP** to weave advice that is run before and after a method call. There are three methods in our test class, `TestClass`: `method1`, `method2` and `method3`. All methods take a

[1] TODO

string as argument and return another. In `method1` **AOP** is used to weave advice that runs before the method and prints the argument passed to it. In `method2` advice is weaved to run after the method, printing the value that was returned by the method. In `method3` advice is added around the method, changing it's arguments and return values.

2.1 C++

> Object oriented techniques, and C++ in particular, seem to be taking the software world by storm. (...) it seems that C++ itself is a major factor in this latest phase of the software revolution. (...) C++ has become popular because it makes it easier to design software and program at the same time. – Jack W. Reeves, 1992

C++ appeared as a natural successor of the popular C programming language, featuring some appealing capacities like templates and exceptions. But the most notorious change from C to C++ was obviously it's support for **OOP**, along with C's common **POP**. This might have been one of the factors that spur the acceptance of C++, since it allowed a smoother transition for C programmers.

Traditionally not being supported by some kind of interpreter or virtual machine, but instead relying on machine dependant binary code, and using a simple runtime environment, means there's less runtime overhead for programs written in C++. Among more recent languages such as Java and Ruby using a virtual machine is a popular tendency.

Of course there is a tradeoff between this efficiency and the flexibility offered. As an example, capabilities like reflection are very limited in C++. It is possible to check the type of an object in runtime using dynamic casts. One cannot, for instance, discover the methods that belong to a certain class at runtime.

AOP offers completely new possibilities to C++ programmers, impossible to do easily in any other simple way. Due to its static nature, the only way to implement aspects, besides changing the semantics of the language (and it's compiler) is using a weaver. A weaver is a program that adds a preprocessing step, that introduces all the code needed for the aspects into the code of the program that is being built.

The misuse of the mechanisms used to implement **AOP** in C++ for other purposes is a danger and should be avoided. This type of use subverts the entire idea of the use of aspects, as they should not be seen as a way to add flexibility to a language, but instead to solve a very specific kind of problem.

2.1.1 Available Tools

There are only a few implementations of **AOP** in C++, possibly due to the complexity of the weaver. One of the complexities stems from the fact that it needs to have an implementation of a complete parser of the language for example.

- AspectC++ is probably the easiest and most widely used tool for **AOP** in C++. It is largely inspired by AspectJ, so it's very similar to use. The program code is changed by a weaver on a preprocessing step, before compilation. AspectC++ is available under an open source license. There's an Eclipse plugin that makes the development with **AOP** easier in this IDE. Likewise, a company named Pure Systems (http://www.pure-systems.com) sells a similar plugin for Microsoft Visual Studio .NET and 7.0.

- XWeaver is "an extensible, customizable and minimally intrusive aspect weaver" which is being developed in ETH-Zurich. It works in a very similar way to AspectC++, but has a different aspect definition language as it's base: AspectX. AspectX aims to be a standard language for aspect definition, and a Java extension of XWeaver is being developed to exemplify its advantages. All the tools needed for the use of XWeaver are under the GPL license.

2.1.2 Code Example

AspectC++, like AspectJ, introduces aspects into C++ by adding some new keywords to the base language syntax, such as `aspect`, `pointcut`, `advice`, `before`, `execution`, ...

The `aspect` keyword defines a new C++ entity similar to the `struct` and `class` constructs. Several pointcuts can be defined within an aspect, using a specific expression matching syntax. Through this syntax one can define and restrict the member's methods, scope, name, return type, argument types where the aspect should be introduced. Along with this "regular expression" one can also define whether the aspects should occur during method call (before being loaded on the stack) or during method execution (after being loaded on the stack).

On the listing 2.1.2 is presented a very simple C++ example. The TestAspect is declared with the keyword `aspect` and defines one advice whose pointcut intercepts all TestClass's methods named "method1" with any number of arguments and any kind of return type. The interception should occur during the execution of the method, but before the method's code starts

running. In this simple example, the advice prints the original method's first parameter, which is assumed to be of string type.

```
aspect TestAspect
{
        advice execution("% TestClass::method1(...)") :
before()
        {
                string & msg = *((string *)tjp->arg(0));

                cout << "before method1: arg '" <<
                                        msg  << "'" << endl;
        }
}
```

Code Listing 2.1: Defining the C++ advice

The `tjp` pointer is the `this` pointer equivalent for aspects in AspectC++. Therefore, it is present in all advices and contains information such as the original method arguments and its types, that can be used on runtime to produce a type-compatible output.

2.2 Java

Java is less dynamic than Ruby, being a descendant of the Algol family of languages like C or C++. However it's virtual machine execution environment gives it more dynamic capabilities compared to C++, like run-time class loading. **AOP**'s first implementations were in Java and the technique seems to have a very active following in the Java community.

AOP in Java can be misperceived as just a way to add useful dynamic features to the language and not for implementing aspects. However, the **AOP** are much more mature in Java than in other languages and the aspect languages defined make the fact that what you are programming is an aspect self-evident. The fact that programmers sometimes tend to use it as just a dynamic feature is parallel to the way beginning programmers treat the **OOP** in Java as just a way to split static functions in different files and not use objects at all, missing the concept entirely.

2.2.1 Available Tools

Java has a few mature implementations of **AOP**. Here we'll mention just a few popular ones:

- AspectJ was the first **AOP** tool and is the most popular one for Java. **AOP** is implemented as an extension to the language's syntax and a new compiler is implemented. Weaving the aspects with the rest of the code is done by a weaver implemented by the project and can happen at compile time and run-time.

- AspectWerkz is a different project from AspectJ although the two have announced that they will be merging in the future. This implementation takes a different approach from AspectJ in that it doesn't add any new syntax to Java. **AOP** is implemented using regular java and annotations. A weaver is implemented to weave the Java aspect code with the rest of the application's code. An advantage of this implementation is that it supports run-time weaving.

- JBOSS AOP is part of the JBOSS project and is another **AOP** implementation that uses pure Java with annotations. If differs from AspectWerkz in implementation details and seams to be geared for a more enterprise environment, integrating with the rest of the JBOSS products and supplying a set of standard aspects.

2.2.2 Code Example

The methods are defined in the form show in Listing 2.2.2. They receive a string as argument and return another one. Inside the method messages are printed showing what arguments are passed and what will be returned.

To catch and print the arguments passed to `method1` the following code is used in the aspect:

```
before(Object msg): args(msg) && call(public * TestClass.
method1(..))
{
    System.out.println("before " +
            "method1:" +
            " arg '" + msg + "'");

}
```

The first line defines the point cut to be before the call to `method1`. The method's arguments are passed to the code block below as `msg`. The output of the program with the aspect introduced is:

```
calling method1
before method1: arg 'hello'
method1 body with arg 'hello'
```

```
public class TestClass {

    public String method1(String msg) {
        System.out.println("method1 body with arg '" + msg +
"'");
        System.out.println("method1 will return '" + "bye'");
        return "bye";
    }

    (...)

    public static void main(String[] args) {
        TestClass testClass = new TestClass();

        System.out.println("calling method1");
        System.out.println("received from call '" +
                testClass.method1("hello")+"'\n\n");

    (...)
    }
}
```

Code Listing 2.2: Java Class with test methods

```
method1 body will return 'bye'
received from call 'bye'
```

Here we can see the code inside the original method being ran and before that our aspect code.

To catch and print the return value of `method2` the following code is used:

```
after() returning (Object msg) : call(public * TestClass.
method2(..))
{

    System.out.println("after " +

            "method2:" +

            " returned '" + msg + "'");

}
```

This code follows the same structure of the one before, definind the pointcut as after `method2` and printing the return value.

Finally to catch the arguments and the return value of `method2` and change them the code is:

```
Object around(Object msg) : args(msg) &&  call(public *
TestClass.method3(..))
{

    System.out.println("before " +

            "method3:" +

            " arg '" + msg + "'");



    Object ret = proceed(msg.toString().toUpperCase());



    System.out.println("after method3: returned '" + ret.
toString() + "'");
    return ret.toString().toUpperCase();

}
```

The code modifies what's passed and returned by the argument changing it to uppercase.

Here we saw examples of using AspectJ's extended Java language to program aspects. The language is an extension of Java as was seen in the code examples. New keywords are added, extending the semantic meaning of the language. To compile AspectJ code a new compiler is needed. This compiler accepts all pure Java code and produces compliant bytecode to be ran in the regular virtual machines. To develop aspect code in AspectJ IDE tools like the Eclipse plugin are very useful since they hide away much of this complexity.

2.3 Ruby

> The RubyLanguage also lets you do this - you can (for example) change the behavior of method invocation or attribute access on a particular object or class of objects, all at run time. For example, you could define your own tracing capability, point it at an object, and suddenly that object traces all accesses to its attributes, all with no change to that object's source. Once you have decent reflection and metaclasses, AOP is just part of the language.
>
> Dave Thomas, quote from C2 Wiki

Ruby is a fully object oriented "scripting"[2] language which features a high degree of dynamism. It has a wide array of features not found on most of the mainstream programming languages such as metaclasses, open classes and closures. Everything in Ruby is an object and the only way for objects to interact is to through message sending.

Particularly interesting from the standpoint of using AOP in the language are open classes and the powerful reflection mechanisms. Open classes refers to a mechanism that allows a class to be extended after it has been defined. Reflection allows one to ask a class about the methods it implements and change a class at runtime, among other thins. To give a feeling for how Ruby allows one to create simple AOP tools we will dive a bit into the language here, showing how these capabilities work.

Dave Thomas' quote

In the following code, the `>>` symbol is Ruby's interpreter prompt. Lines starting with `=>` have the return value of some method call. Lines that start with any prefix symbol are output from I/O methods. A class with a method `hello` can be defined in Ruby like this:

```
>> class MyClass
>>   def hello
>>     puts "hello"
>>   end
>> end
```

Notice that we end the definition of the class, the second `end` does that. If we instantiate an object of this class and try calling some of it's methods:

```
>> obj = MyClass.new
>> obj.hello
```

[2] There doesn't seem to be much consensus as to what a scripting language is, thus our use of quotation marks.

```
hello
>> obj.goodbye
NoMethodError: undefined method `goodbye' for #<MyClass:0
x478b8>
```

Calling `hello` successfully outputs "hello", however we can see that the method `goodbye` is not defined. We can reopen reopen the class and add a method, as seen in the following code:

```
>> class MyClass
>>   def goodbye
>>     puts "goodbye"
>>     end
>>   end
```

This adds a method, `goodbye`, to the class. If we now call `goodbye` on the object we instantiated the result is:

```
>> obj.goodbye
goodbye
```

Changing the class affects all the objects that we instantiate from it and all the objects already instantiated. This would not suffice for AOP, as we need to change the code of the methods already in the class, not add new ones. Making code run before and/or after a particular method can be done by changing the method's name and adding a new one that calls it. Printing "before" and "after" between before and after the `hello` method is easily done like this:

```
>> class MyClass
>>   alias_method :old_hello, :hello
>>   def hello
>>     puts "before"
>>     old_hello
>>     puts "after
>>   end
>> end
>> obj.goodbye
before
hello
after
```

The downside of using this technique is possible names collisions. While conventions can stop this from happening it can be a problem using libraries from different sources. Ruby 2.0 (the next version) will add pre and post hooks on methods [6] to allow this to be done in a cleaner way.

Also important to implement **AOP** tools might me knowing if some method exists in a class. Ruby's reflection mechanisms allows us to do this easily:

```
>> MyClass.method_defined?(:hello)
=> true
>> MyClass.method_defined?(:no_such_method)
=> false
```

AspectR, which we talk about in the next section, uses these techniques, along with metaclasses to implement an **AOP** library. There are however other ways to use **AOP** in Ruby. Dynamic proxies, which are a way to interpose a method call before another one can also be done easily in Ruby. A simple dynamic proxy can be created using Ruby's reflection. Every time a method that doesn't exist on a particular object is called, `method_missing` method will be called on that object, if it's defined. The following example uses MyClass and illustrates this:

```
>> class DynamicProxy
>>   def initialize(object)
>>     @obj = object
>>   end
>>   def method_missing(method, *args)
>>     puts "calling #{method}"
>>     @obj.send(method, *args)
>>   end
>> end
>> proxy = DynamicProxy.new(obj)
>> proxy.hello
calling hello
hello
```

We intercept the call to the method, modifying the arguments to it and it's return value can also be done.

The downside of using techniques like this to implement AOP are it's effects on performance. Ruby isn't, at this point in time, suited for performance critical applications and we will worry about that particular aspect here.

2.3.1 Available Tools

AOP doesn't seem to have made big inroads into the Ruby community. The tool we used, AspectR, is the most complete AOP tool for Ruby. There are some proposals for AOP in Ruby [4] and [5] but we haven't found much more interest on it than that. Our guess is that this is partly influenced by the

language dynamism and culture and by it's areas of use. Other possibility is TODO misonception.

AspectR implements some of AspectJ's functionality, some of it is missing though. TODO: Mudar de síto para conclusões What might be interesting to point out is that it's implemented on 300 lines of code. This shows that having a flexible language makes it that much easier to include new abstraction into it.

2.3.2 Code Example

AspectR is a little bit different from AspectJ. The advice to weave into the code is created as methods inside a class inheriting from `AspectR::Aspect`. Each method in this class has four parameters. The first one is the name of the method being advised, the second is the object receiving the method call, the third is the return/exit status and the fourth argument is an array of the arguments passed to the advised method. The return/exit status is `true` if the method exited with an exception and an array with the return values if it exited normally.

AspectR doesn't support around advice like AspectJ. TODO: which we used in section... We can still modify the arguments and return values if they're not immutable because the variables passed by AspectR reference the original objects.

In our simple example the advice to be weaved can be seen on listing 2.3.2. The `log_pre` and `log_post` methods simply log the entry and exit into a method. The other two, `log_modify_pre` and `log_modify_post` change the argument and return value, respectively.

Defining the pointcuts is accomplished by calling a method inherited from `AspectR::Aspect` – `wrap`, after instantiating the class that cointains the advice. The method should be called with the class to weave, the methods with the pre and after advice and the method to be advised. Instead of specifying a single method it's also possible to use a regular expression to match a set of methods. The code on listing 2.3.2 does this. We instantiate our `TestLogger` class and repeatedly call `wrap`, passing our `TestClass` as the class to wrap, and the various methods to be advised.

2.4 Comparison

As can be seen, for simple cases, it's relatively easy to integrate **AOP** into the language. All the three tools we used can easily be used for the simple cases we tried out. AspectJ and AspectC++ have similar syntax and semantics.

```
class TestLogger < AspectR::Aspect
  def log_pre(method, object, exitstatus, *args)
    puts "#{object.class}##{method}: args = #{args.inspect}"
  end

  def log_post(method, object, exitstatus, *args)
    puts "#{object.class}##{method} exited return args: #{
exitstatus[0].inspect}"
  end

  def log_modify_pre(method, object, exitstatus, *args)
    puts "#{object.class}##{method}: args = #{args.inspect}"
    args[0].upcase!
  end

  def log_modify_post(method, object, exitstatus, *args)
    exitstatus[0].upcase!
  end
end
```

Code Listing 2.3: Defining the advice

```
logger = TestLogger.new
logger.wrap(TestClass, :log_pre, nil, 'method1')
logger.wrap(TestClass, nil, :log_post, 'method2')
logger.wrap(TestClass, :log_modify_pre, :log_modify_post, '
method3')
```

Code Listing 2.4: Defining the pointcuts

AspectR is a little different, as it doesn't introduce any new syntax into the language and implements **AOP** using it's standard OO programming. This is possible due to Ruby's dynamic nature.

AspectC++ uses preprocessing to weave the advice, this means a pass through the code is required before passing it to the standard C++ toolchain. AspectJ is an extension to the Java language, adding new keywords to it. There's an AspectJ compiler that directly generates Java bytecode. Having the language running on top of a virtual machine makes it easier to do this. AspectJ adds little runtime overhead as a result of using **AOP**. AspectR does all it's work at runtime and doesn't modify the language in any way. In Ruby there isn't any difference between compile time and run time so this makes sense.

AspectJ extends the Java language, introducing new keywords into it and having a compiler which understands them. This approach has the advantage of allowing the **AOP** system to easily be optimized without affecting the

programs written with it. The virtual machine could even be aware of aspects and perform low level optimizations for them.

AspectR's approach clearly adds runtime overhead but might offer greater flexibility. One advantage of having the language include **AOP** facilities is that it might make it easier to have an IDE show the crosscutting code while one looking at a method which is affected by it. Since AspectR allows the separation of the definition of advice from the specification of the point cuts one might define those cuts next to the code which is affected by them. This does have the issue of scattering the point cuts through out the code. How much of a problem this is, or if it's a problem at all probably depends on the circumstances. A problem we faced is that AspectR doesn't support the "around" point cut of AspectJ. While it was possible to circumvent this in our simple example, in other situations it wouldn't be. It would however be easy to extend AspectR to support it, as it a small (300 lines of code) library. This compactness is the result of the powerful abstractions Ruby has.

There is debate as if **AOP** as implemented by AspectJ is flexible enough to effectively deal with all of the cross cutting concerns that crop up or if having the point cuts specified using the base language reflection mechanisms is advantageous [10]. Having the point cuts expressed in the base programming language might be easier to use, as no new syntax or semantics have to be learned, and provide more flexibility.

Chapter 3

Implementing a Logging Library

To get a feel for how more complex crosscutting concerns could be implemented across different languages and particularly how the dynamism of the language affects this we conceived a simple logging library to implement in C++ and Ruby. While logging is an often used example for **AOP**, which can result in the "Can you give me some example of **AOP** that isn't logging?" that Kiczales talked about **??**, we should point out that we chose logging just because it's simple to apply it to a small example. We want to concentrate not on the logging, but on how and when it can be configured. Some of the things we want to see is how we can intercept the argument of a method being called, how we can specify the methods to which our crosscutting code should be applied, how we can intercept object creation and destruction and how we can vary the code that's being crosscutted at runtime.

We will not explain the libraries we have created, both in C++ and Ruby, in detail here. That would be tedious and require too much space. We concentrate on how we can implement the use cases we talk about on the following section, using both Ruby and C++ and their respective **AOP** implementations.

3.1 Library Use Cases

Here's a list of the use cases our library should implement:

1. Specify methods, using regular expressions, as logging points. A list of classes is associated with the methods. Both entry and exit from the methods should be logged. At entry the arguments passed to the

method should also be logged. At exit the return value should be logged or the type of the exception it raised, if occurred.

2. Specify classes for which object creation and destruction is to be logged.

3. Have this configuration in an external file that can be changed at runtime and have it's changes immediately reflected.

These are just general, abstract use cases. Making the two versions exactly equals wasn't our objective, just to define the scope of the library. Neither was our objective to create robust libraries, but to explore and get a feel for how these use cases can be implemented.

3.2 C++

All the implementations of **AOP** for C++ use preprocessing to weave the advice into the code. This makes it impossible to change the advice at runtime. The only plausible approach to log method calling is to weave a test into every method of the program. To weave log code into every single method the following regular expression was defined:

```
execution("% ...::%(...)") && !execution("% LoggerLib::...::%(...)")
```

This regular expression should catch all method calls in all classes in all namespaces except for the LoggerLib namespace, otherwise a recursive advice would take place. AspectC++ handles these recursive advice calls in a rather opaque way. It doesn't output any sort of warning and the compiled application seems to exit as soon as any adviced method is called.

Now that the library can insert its aspect in every method, all it needs is to filter out which methods should be logged and which shouldn't. This is achieved by matching the method's signature, supplied by the JoinPoint pointer (see subsection 2.1.2), with each of the regular expressions defined by the user in the configuration file. The configuration file is constantly checked for modifications, in order to detect updates int the user regular expressions list.

Constructors and destructors aren't considered normal methods by AspectC++. The special keywords `construction` and `destruction` are used instead of `execution` or `call` to introduce aspects for constructors and destructors. In resemblance to their method counterparts, these two keywords take a regular expression as argument to define which classes should be monitored. Even though these two special facilities, the solution applied for

catching all method calls could not be repeated here. Regular expressions for constructors and destructors seem to work rather differently. When trying to capture all construction and destruction calls with a generic regular expression, all constructors and destructors seem to be affected, even those defined in the global and `std` namespaces. This is rather problematic since most of these constructors and destructors are part of already compiled libraries and AspectC++ cannot perform weaving for them hence compile errors are output.

Since the logging mechanism should occur before and after each of the selected methods, `around` keyword was used in our advice, as shown in listing 3.2. Using `around` one can control the execution of the original original method by calling `tjp->proceed`, and manipulate its arguments and return value. Due to this direct meddling with stack values such as arguments and return values, close control should be kept over the advice's code to prevent the programmer from shooting himself in the foot. An example is trying to alter a class's object returned by value when not calling the original method. This leads to some unexpected behavior or even segmentation faults, since one would be trying to access memory that hasn't yet been properly initialized.

```
advice  execution("% ...::%(...)") &&
                          !execution("% LoggerLib::...::%(...)"
) : around()
{
        // ... advice code before the original method

        tjp->proceed();

        // ... advice code after the original method
}
```

Code Listing 3.1: Logging C++ advice

3.3 Ruby

As we said in section 2.3, AspectR alters the code at runtime. This makes it easy to add advice to code at runtime. This easily allows us to changing the methods which are logged while a program that uses the library is running. AspectR has two ways to add advice to methods. One which can be undone, the `wrap` method we used on section **??**. The other one, us-

ing the `wrap_with_code` method, can't. Since we can "unadvise" a method implementing the third use case is straightforward.

In Ruby every operation is achieved by sending messages, this includes instantiating a new object. The way this works is that the `new` class method calls `alloc` and then `initialize`. The initialize method corresponds to Java's and C++'s constructors. It's responsibility is to initialize the new object's state according to any passed arguments and calling the parent's class initialize, if desired. It's up to whomever creates a class to define this method, `new` should generally not be redefined. It is impossible to redefine `alloc` in a normal program. This means we can intercept object creation just like we intercept any other method, by weaving our advice into `initialize`. Object destruction works slightly differently. In Ruby it's possible to define a finalizer by calling a class method in the class `ObjectSpace`, the method `define_finalizer`. This method takes as parameters the object[1] for which to create a finalizer and a closure[2] which is run after the object is garbage collected. Removing a finalizer for an object is done by calling `remove_finalizer`.

Since the configuration is stored in an external text file that means we only have strings with the class names. AspectR needs an object to weave the advice into the code. This doesn't pose a problem since we can use Ruby's reflection to get the class object. A class object is referenced by a global constant in Ruby, to get it we just need to ask `ObjectSpace` for it:

```
klass = ObjectSpace.const_get(klass)
```

Since the definition of the advice is done using a regular class, which has to be instantiated for weaving, we can can have this class accept parameters. These parameters can then be used by the advice. We use this to allow the way the logging is done to be customizable.

AspectR has no direct support for object construction and destruction. As we've seen, because of the way Ruby works this is no problem for object construction. However, for object destruction it is. There's no method we can call on an object to define a finalizer for it. This is an explicit Ruby design decision. To intercept object destruction we have to fall back on doing some metaprogramming 1.3. The way to do it is to ask `ObjectSpace` for every instance of a specific class and add a finalizer which performs the logging:

```
finalizer = lambda {@writer.object_destruction(Time.now,
class_name)}
ObjectSpace.each_object(klass) do |obj|
```

[1] The object can be a class. In Ruby everything's an object.

[2] A closure is a block of code that's saved and can have references to it's lexical scope. Those references are maintained for the block, so when it runs they're valid.

```
  ObjectSpace.define_finalizer(obj, finalizer)
end
```

We create a block that will run the finalization code and then define it as a finalizer for every object instance of a specific class.

3.4 Comparison

This use of aspects isn't well suited for C++. Programming it in Ruby was much simpler to implement and allowed for a more natural use of **AOP**. Unexpected problems cropped up while doing the C++ version. Some arose from the inherent C++ low level which obliges a sharp eye on the source code and its memory manipulation, and some other problems were due to a more complicated way of creating an aspect and defininig pointcuts and advices in the AspectC++ syntax.

Chapter 4

Conclusion

Throughout this report we've seen what the **AOP** concept is about in it's abstract form and how it is implemented in several programming languages. With the implementation of our library we explored what is like to use aspect programming to add features to a program without having to change it's structure or add a lot of code in several places. To conclude this report we'll do a few considerations on the various aspects of **AOP**, from the concept, to it's implementation and use.

4.1 AOP as a Concept

The **AOP** concept is still in it's early days. It's too soon to tell if it will catch on as much as **OOP** has. Just like it happened with **OOP** it's possible that **AOP** will explode in popularity when a programming language with the concept built-in catches on.

We've come to the conclusion **AOP** is a good technique for program structuring and that it complements **OOP** in a very effective and non-intrusive way. It's good to note that **AOP** is just a concept not a specific implementation. Because of this the way it's implemented in different languages depends on the existing structure and semantic. To make sense in your favorite programming language **AOP** must be properly integrated with the rest of the semantic meaning and keywords of the language.

4.2 AOP Tools

As we tried **AOP** in a few languages we found that it's implementations were, generally, not very mature. In the two less dynamic languages we tried **AOP** proved to be very much a bolted-on concept. A lot of new keywords

were added to the language, changing is execution structure and semantic meaning.

Assuming the concept catches on, the natural evolution of **AOP** will probably be to become a part of programming languages, becoming better integrated in the general structure of the language itself.

4.3 Programming with AOP

After programming with **AOP** we've concluded that it really is a good way to separate cross-cutting concerns from the rest of the code. The technique started being used for general aspects applicable to every application, like logging, tracing or locking. As time goes on **AOP** will be used for more specific aspects, first relative to a certain domain and then as another structuring technique for concrete applications.

Bibliography

[1] Programming Ruby, 2nd Edition, Dave Thomas, The Pragmatic Programmers

[2] [aosd-discuss] AOP and Meta-Programming, Various authors, AOSD.net mailing list archive 7

[3] Dynamic Proxies != AOP? It doesn't hold!!, Val

[4] Cut-based AOP, Thomas Sawyer & Peter Vanbroekhoven 20

[5] Event Oriented AOP, Thomas Sawyer & Peter Vanbroekhoven 20

[6] Rite 19

[7] Java Dynamic Proxies: One Step from Aspect-oriented Programming, DevX.com, Lara D'Abreo

[8] C2.com: Meta Programming, Various Authors, C2.com wiki page 7

[9] Aspects And Dynamic Languages, Various Authors, C2.com wiki page

[10] Aspect Weaving: Using the Base Language's Introspective Facilities to Determine Join Points, Stefan Hanenberg, Robert Hirschfeld, Rainer Unland, Katsuya Kawamura 23

[11] AspectWerkz 11

[12] AspectJ 11

[13] JBoss AOP 11

[14] It's Not Metaprogramming, Gregor Kiczales, Software Development Online, November 2004 8

[15] Common Misconceptions, Gregor Kiczales, Software Development Online, March 2004 8

[16] Double Decker, Gregor Kiczales, Software Development Online, January 2005

Appendix A

Example Code

A.1 C++

A.1.1 TestAspect.ah

```
1  #ifndef TESTASPECT_AH
2  #define TESTASPECT_AH
3
4  #include <iostream>
5  #include <string>
6  #include <algorithm>
7  #include <cctype>
8
9  #include "TestClass.h"
10
11 using namespace std;
12
13 aspect TestAspect
14 {
15
16         advice execution("% TestClass::method1(...)") :
   before()
17         {
18                 string & msg = *((string *)tjp->arg(0));
19
20                 cout << "before " <<
21                         "method1:" <<
22                         " arg '" << msg  << "'" << endl;
23         }
24
25         advice execution("% TestClass::method2(...)") : after
   ()
26         {
27                 string & msg = *((string *)tjp->result());
```

```
28
29                  cout << "after " <<
30                          "method2:" <<
31                          " returned '" << msg << "'" << endl;
32          }
33
34          advice execution("% TestClass::method3(...)") :
    around()
35          {
36                  string & arg = *((string *)tjp->arg(0));
37                  cout << "before " <<
38                          "method3:" <<
39                          " arg '" << arg  << "'" << endl;
40
41                  arg = "HELLO";
42
43                  tjp->proceed();
44
45                  string & retVal = *((string *)tjp->result());
46                  cout << "after " <<
47                          "method3:" <<
48                          " returned '" << retVal << "'" <<
    endl;
49
50                  retVal = "BYE";
51          }
52
53  };
54
55  #endif
```

A.1.2 TestClass.cpp

```
1   #include "TestClass.h"
2
3
4   string TestClass::method1(string msg) {
5           cout << "method1 body with arg '" << msg << "'" <<
    endl;
6           cout << "method1 will return 'bye'" << endl;
7           return "bye";
8   }
9
10  string TestClass::method2(string msg){
11          cout << "method2 body with arg '" << msg << "'" <<
    endl;
12          cout << "method2 will return 'bye'" << endl;
13          return "bye";
14  }
```

```
15
16 string TestClass::method3(string msg){
17         cout << "method3 body with arg '" << msg << "'" <<
   endl;
18         cout << "method3 will return 'bye'" << endl;
19         return "bye";
20 }
21
22 int main()
23 {
24         TestClass testClass;
25         string msg("hello");
26
27         cout << "calling method1" << endl;
28         cout << "received from call '" << testClass.method1(
   msg) << "'\n\n" << endl;
29
30         cout << "calling method2" << endl;
31         cout <<  "received from call '" << testClass.method2(
   msg) << "'\n\n" << endl;
32
33         cout << "calling method3" << endl;
34         cout <<  "received from call '" << testClass.method3(
   msg) << "'\n\n" << endl;
35
36         return 0;
37 }
```

A.1.3 TestClass.h

```
1  #ifndef TESTCLASS_H
2  #define TESTCLASS_H
3
4  #include <iostream>
5  #include <string>
6
7
8  using namespace std;
9
10 class TestClass
11 {
12
13         public:
14             string method1(string msg);
15             string method2(string msg);
16             string method3(string msg);
17 };
18
19 #endif
```

A.2 Java

A.2.1 TestAspect.aj

```
1  /*
2   * Created on 2/Mai/2005
3   *
4   * TODO To change the template for this generated file go to
5   * Window - Preferences - Java - Code Style - Code Templates
6   */
7
8
9  /**
10  * @author Rui
11  */
12 public aspect TestAspect
13 {
14
15     public TestAspect()
16     {
17     }
18
19
20     before(Object msg): args(msg) && call(public * TestClass.
   method1(..))
21     {
22         System.out.println("before " +
23                 "method1:" +
24                 " arg '" + msg + "'");
25
26     }
27
28     after() returning (Object msg) : call(public * TestClass.
   method2(..))
29     {
30         System.out.println("after " +
31                 "method2:" +
32                 " returned '" + msg + "'");
33     }
34
35     Object around(Object msg) : args(msg) &&  call(public *
   TestClass.method3(..))
36     {
37         System.out.println("before " +
38                 "method3:" +
39                 " arg '" + msg + "'");
40
41         Object ret = proceed(msg.toString().toUpperCase());
42
```

```
43          System.out.println("after method3: returned '" + ret.
    toString() + "'");
44          return ret.toString().toUpperCase();
45      }
46
47  }
```

A.2.2 TestClass.java

```
1   /*
2    * Created on 2/Mai/2005
3    *
4    * TODO To change the template for this generated file go to
5    * Window - Preferences - Java - Code Style - Code Templates
6    */
7
8
9   /**
10   * @author Rui
11   */
12  public class TestClass {
13
14      public String method1(String msg) {
15          System.out.println("method1 body with arg '" + msg +
    "'");
16          System.out.println("method1 will return '" + "bye'");
17          return "bye";
18      }
19
20      public String method2(String msg) {
21          System.out.println("method2 body with arg '" + msg +
    "'");
22          System.out.println("method2 will return '" + "bye'");
23          return "bye";
24      }
25
26      public String method3(String msg) {
27          System.out.println("method3 body with arg '" + msg +
    "'");
28          System.out.println("method3 will return '" + "bye'");
29          return "bye";
30      }
31
32      public static void main(String[] args) {
33          TestClass testClass = new TestClass();
34
35          System.out.println("calling method1");
36          System.out.println("received from call '" +
37                  testClass.method1("hello")+"'\n\n");
```

```
38
39          System.out.println("calling method2");
40          System.out.println("received from call '" +
41                  testClass.method2("hello")+"'\n\n");
42
43          System.out.println("calling method3");
44          System.out.println("received from call '" +
45                  testClass.method3("hello")+"'\n\n");
46      }
47  }
```

A.3 Ruby

A.3.1 test_class.rb

```
1   #!/usr/bin/env ruby
2
3   require 'test_logger.rb'
4   require 'aspectr'
5   include AspectR
6
7   class TestClass
8     def method1(arg)
9           puts "method1 body with arg '#{arg}'"
10          ret = "bye"
11          puts "method1 will return '#{ret}'"
12          return ret
13    end
14
15    def method2(arg)
16      puts "method2 body with arg '#{arg}'"
17      ret = "bye"
18      puts "method2 will return '#{ret}'"
19      return ret
20    end
21
22    def method3(arg)
23      puts "method3 body with arg '#{arg}'"
24      ret = "bye"
25      puts "method3 will return '#{ret}'"
26      return ret
27    end
28  end
29
30  def call_methods
31    obj = TestClass.new
32    puts "calling method1"
```

```
33    ret = obj.method1("hello")
34    puts "received from call '#{ret}'"
35    puts
36    puts "calling method2"
37    ret = obj.method2("hello")
38    puts "received from call '#{ret}'"
39    puts
40    puts "calling method3"
41    ret = obj.method3("hello")
42    puts "received from call '#{ret}'"
43  end
44
45  if __FILE__ == $0
46    # Before aspect is injected into code.
47    call_methods
48
49    # After aspect is injected into code. Pre, post and wrap of
50    # the methods.
51    logger = TestLogger.new
52    logger.wrap(TestClass, :log_pre, nil, 'method1')
53    logger.wrap(TestClass, nil, :log_post, 'method2')
54    logger.wrap(TestClass, :log_modify_pre, :log_modify_post, '
    method3')
55    print "\n" + "-" * 10 + "\n\n"
56    call_methods
57  end
```

A.3.2 test_logger.rb

```
1   #!/usr/bin/env ruby
2
3   require 'aspectr'
4
5   class TestLogger < AspectR::Aspect
6     def log_pre(method, object, exitstatus, *args)
7       puts "before #{object.class}##{method}: arg '#{args.
    inspect}'"
8     end
9
10    def log_post(method, object, exitstatus, *args)
11      puts "after #{object.class}##{method}: returned: '#{
    exitstatus[0].inspect}'"
12    end
13
14    def log_modify_pre(method, object, exitstatus, *args)
15      puts "#{object.class}##{method}: arg = '#{args.inspect}'"
16      args[0].upcase!
17    end
18
```

```
19    def log_modify_post(method, object, exitstatus, *args)
20      exitstatus[0].upcase!
21    end
22  end
```

Appendix B

Logging Library Code

B.1 C++

B.1.1 main.cpp

```
1  #include <iostream>
2  #include <string>
3  #include <iostream>
4  #include <unistd.h>
5
6  #include "RegularExpression.h"
7  #include "Cake.h"
8  #include "Airplane.h"
9  #include "Door.h"
10
11 int main(void)
12 {
13         unsigned int usleepTimeUSec = 200000;
14
15         Door * door = NULL;
16         Airplane * airplane = NULL;
17         Cake * cake = NULL;
18
19         while(true)
20         {
21                 std::cout << "
   ---------------------------------------" << std::endl;
22
23                 // Constructor call block
24                 door = new Door();
25                 usleep(usleepTimeUSec);
26                 airplane = new Airplane();
27                 usleep(usleepTimeUSec);
28                 cake = new Cake();
```

```
29                  usleep(usleepTimeUSec);
30
31
32                  // Class methods
33                  airplane->fly();
34                  usleep(usleepTimeUSec);
35                  airplane->land(10, 20);
36                  usleep(usleepTimeUSec);
37
38                  cake->cut("big");
39                  usleep(usleepTimeUSec);
40                  cake->eat();
41                  usleep(usleepTimeUSec);
42
43                  door->lock("my_key");
44                  usleep(usleepTimeUSec);
45                  door->open();
46                  usleep(usleepTimeUSec);
47
48
49                  // Desctructor call block
50                  delete cake;
51                  usleep(usleepTimeUSec);
52                  delete airplane;
53                  usleep(usleepTimeUSec);
54                  delete door;
55                  usleep(usleepTimeUSec);
56          }
57
58          return 0;
59  }
60
61
```

B.1.2 Logger.cpp

```
1   #include "Logger.h"
2
3   using namespace LoggerLib;
4
5   const std::string Logger::CONF_FILENAME = "logger.conf";
6
7   Logger::Logger() :
8   lastTimeRead(0)
9   {
10          clearRegexVector();
11  }
12
```

```
13 Logger::~Logger()
14 {
15         clearRegexVector();
16 }
17
18 void Logger::update()
19 {
20         if (hasConfFileChanged())
21         {
22                 updateRegexesList();
23         }
24 }
25
26 bool Logger::hasConfFileChanged()
27 {
28         if ( 0 == stat(CONF_FILENAME.c_str(), &fileStat))
29         {
30                 time_t current = fileStat.st_mtime;
31                 bool retVal = (current > lastTimeRead);
32                 lastTimeRead = current;
33                 return retVal;
34         }
35         return false;
36 }
37
38 void Logger::updateRegexesList()
39 {
40         std::ifstream file(CONF_FILENAME.c_str());
41         std::string input;
42         std::string currentRegex;
43
44         clearRegexVector();
45
46         while(!file.eof())
47         {
48                 file >> input;
49                 regexVector.push_back(new RegularExpression(
   input));
50         }
51
52         file.close();
53 }
54
55 void Logger::clearRegexVector()
56 {
57         for (int x = 0 ; x < regexVector.size() ; x++)
58         {
59                 delete regexVector[x];
60         }
```

```
61          regexVector.clear();
62  }
63
64  bool Logger::matches(std::string text)
65  {
66          for (int x = 0 ; x < regexVector.size() ; x++)
67          {
68                  if (regexVector[x]->validate(text))
69                  {
70                          return true;
71                  }
72          }
73          return false;
74  }
75
76
77
```

B.1.3 Logger.h

```
1   #ifndef LOGGER_H
2   #define LOGGER_H
3
4   #include <iostream>
5   #include <vector>
6   #include <fstream>
7   #include <string>
8
9   #include <sys/types.h>
10  #include <sys/stat.h>
11  #include <unistd.h>
12
13  #include "RegularExpression.h"
14
15  typedef std::vector<LoggerLib::RegularExpression *>
    RegexVector;
16
17  namespace LoggerLib
18  {
19          class Logger{
20          public:
21                  Logger();
22                  virtual ~Logger();
23
24                  void update();
25                  bool matches(std::string text);
26
27                  static const std::string CONF_FILENAME;
28
```

```
29          protected:
30                  bool hasConfFileChanged();
31                  void updateRegexesList();
32                  void clearRegexVector();
33
34          private:
35                  RegexVector regexVector;
36                  struct stat fileStat;
37                  time_t lastTimeRead;
38          };
39  }
40
41  #endif // LOGGER_H
```

B.1.4 LoggerAspect.ah

```
1   #ifndef TESTASPECT_AH
2   #define TESTASPECT_AH
3
4   #include <iostream>
5   #include <string>
6
7   #include "Logger.h"
8
9
10  aspect LoggerAspect
11  {
12
13          LoggerLib::Logger logger;
14
15
16
17          advice  execution("% ...::%(...)") &&
18                          !execution("% LoggerLib::...::%(...)"
    ) : around()
19          {
20                  logger.update();
21                  bool matches = logger.matches(JoinPoint::
    signature());
22
23                  if (matches)
24                  {
25                          std::cout << "running before " <<
26                                  JoinPoint::signature() << "
    args: ";
27                          for (int i = 0 ; i < JoinPoint::args
    () ; i++)
28                          {
```

```
29                                      //::printvalue(tjp->arg(i),
    JoinPoint::argtype(i));
30                              }
31                              std::cout << std::endl;
32                      }
33
34
35                      tjp->proceed();
36
37                      if (matches)
38                      {
39                              std::cout << "running after " <<
40                                      JoinPoint::signature() << "
    returning: ";
41                              //::printvalue(tjp->result(),
    JoinPoint::resulttype());
42                              std::cout << std::endl;
43                      }
44              }
45
46      };
47
48      #endif
```

B.1.5 RegularExpression.cpp

```
1   #include "RegularExpression.h"
2
3   using namespace LoggerLib;
4
5   RegularExpression::RegularExpression(std::string & pattern) :
6   preg(NULL)
7   {
8           setPattern(pattern);
9   }
10
11  RegularExpression::~RegularExpression()
12  {
13          regfree(preg);
14          preg = NULL;
15  }
16
17  bool RegularExpression::validate(std::string & text)
18  {
19          return  0 == regexec(
20                                                  preg,
21                                                  text.c_str(),
22                                                  0,
23                                                  NULL,
```

```
24                                              0) ;
25  }
26
27  int  RegularExpression::setPattern(std::string & pattern)
28  {
29          if (preg != NULL)
30          {
31                  regfree(preg);
32                  preg = NULL;
33          }
34          preg = new regex_t();
35          return regcomp(preg, pattern.c_str(), REG_EXTENDED);
36  }
37
```

B.1.6 RegularExpression.h

```
1   #ifndef REGULAREXPRESSION_H
2   #define REGULAREXPRESSION_H
3
4   #include <stdlib.h>
5   #include <regex.h>
6   #include <string>
7
8   namespace LoggerLib
9   {
10          class RegularExpression
11          {
12                  public:
13                          RegularExpression(std::string &
    pattern);
14                          bool validate(std::string & text);
15                          int setPattern(std::string & pattern)
    ;
16                          virtual ~RegularExpression();
17
18                  private:
19                          regex_t * preg;
20          };
21  }
22
23  #endif // REGULAREXPRESSION_H
```

B.2 Ruby

B.2.1 dynalog.rb

```
1  require 'aspectr'
2  require 'yaml'
3
4  # A simple method entry/exit logger  which is configured
   using a YAML file.
5  # Each entry in the file contains one or more class names and
    a regular
6  # expression. The regular expression is used to match method
   names in the
7  # class/classes. The file can be changed while the program is
    running, this
8  # will immediatly be reflected on the logging performed.
9  class Dynalog
10   def initialize(log_file, writer)
11     @file   = log_file
12     @writer = writer
13     @aspect = Dynalog::Aspect.new(writer)
14   end
15
16   def start
17     ConfWatcher.new(self, @file)
18   end
19
20   # Array with the configuration in use. Configuration is an
   Array of Hashes.
21   # Each Hash contains the keys 'classes' and 'regex'.
22   def conf
23     @conf.clone
24   end
25
26   # Use a new configuration. Calls to methods in the old one
   are no longer
27   # logged.
28   def new_conf(conf)
29     if @old_conf
30       unwrap_methods(@old_conf)
31       run_remove_finalizers(@old_conf['destruction'])
32     end
33     wrap_methods(conf)
34     run_add_finalizers(conf['destruction'])
35     @old_conf = conf
36   end
37
38   private
39   # Stops logging on the given _configuration_.
```

```
40    def unwrap_methods(conf)
41      run_wrap_methods(@aspect.method('unwrap'), conf['methods'
    ])
42    end
43
44    # Starts logging on the given _configuration_.
45    def wrap_methods(conf)
46      run_wrap_methods(@aspect.method('wrap'), conf['methods'])
47    end
48
49    # Wrap or unwrap methods with logging. method is the method
     from AspectR to
50    # call.
51    def run_wrap_methods(method, conf_method)
52      conf_method.each do |hash|
53        regex   = hash['regex']
54        classes = hash['classes']
55        classes.each do |klass|
56          klass = ObjectSpace.const_get(klass)
57          method.call(klass, :log_pre, :log_post, /#{regex}/)
58        end
59      end
60    end
61
62    def run_remove_finalizers(conf_destruction)
63      conf_destruction.each do |klass|
64        klass = ObjectSpace.const_get(klass)
65        ObjectSpace.each_object(klass) do |obj|
66          ObjectSpace.undefine_finalizer(obj)
67        end
68      end
69    end
70
71    def run_add_finalizers(conf_destruction)
72      conf_destruction.each do |klass|
73        klass = ObjectSpace.const_get(klass)
74        class_name = klass.to_s
75        finalizer = lambda {@writer.object_destruction(Time.now
    , class_name)}
76        ObjectSpace.each_object(klass) do |obj|
77          ObjectSpace.define_finalizer(obj, finalizer)
78        end
79      end
80    end
81  end
82
83  # Watch a _configuration_ file. Detects changes to it  and
    passes new
84  # _configuration_ to logger.
```

```
85  class Dynalog::ConfWatcher
86    def initialize(logger, file)
87      @logger = logger
88      @file = file
89      @mtime = mtime_file
90      logger.new_conf(read_file)
91      start_thread
92    end
93
94    def mtime_file
95      File.mtime(@file)
96    end
97
98    def read_file
99      conf = YAML.load(File.open(@file))
100   end
101
102   private
103   # Start the watcher thread. The logger will be called
    whenever the
104   # configuration file changes.
105   def start_thread
106     Thread.new do
107       loop do
108         sleep 0.5
109         new_mtime = mtime_file
110         if new_mtime > @mtime
111           @logger.new_conf(read_file)
112         end
113         @mtime = new_mtime
114       end
115     end
116   end
117 end
118
119 # Aspect with code for logging. Will call a writer object's
    methods with
120 # information for it to log.
121 class Dynalog::Aspect < AspectR::Aspect
122   def initialize(writer)
123     @writer = writer
124     super
125   end
126
127   def log_pre(method, object, exitstatus, *args)
128     @writer.method_entry(Time.now, object.class, method, args
    )
129   end
130
```

```
131   def log_post(method, object, exitstatus, *args)
132     if args == true
133       args = $!
134     else
135       args = args[0]
136     end
137     @writer.method_exit(Time.now, object.class, method, args)
138   end
139 end
140
141 class Dynalog::RegularWriter
142   def initialize(io)
143     @io  = io
144   end
145
146   def object_destruction(timestamp, klass)
147     @io << "#{timestamp}: destroyed #{klass}\n"
148   end
149
150   def method_entry(timestamp, klass, method, args)
151     if args.empty?
152       args = ""
153     else
154       args = args.inspect
155     end
156     @io << "#{timestamp}: enter #{klass}.#{method}(#{args})\n
    "
157   end
158
159   def method_exit(timestamp, klass, method, ret)
160     if ret.kind_of?(Exception)
161       @io << "#{timestamp}: exit #{klass}.#{method} exception
     #{ret}\n"
162     else
163       @io << "#{timestamp}: exit #{klass}.#{method} with #{
    ret.inspect}\n"
164     end
165   end
166 end
```