AspectJ: An Approach to AOP

Carlos Pereira
Instituto de Engenharia Electronica e Telematica de Aveiro, Universidade de Aveiro
carlospereira221@gmail.com

Flavio Cruz
Center for Research in Advanced Computing Systems, Faculdade de Ciencias da Universidade do Porto
flavioc@dcc.fc.up.pt

ABSTRACT

Aspect Oriented Programming (AOP) is a new paradigm that tries to solve several problems of current programming paradigms by modularizing crosscutting concerns. In this paper we present a compiler named AspectJ that extends the Java language with a new abstraction unit called the aspect. We will present how crosscutting concerns can be declared, first by showing the new constructs that support the declaration of certain points of execution in the program, called the join points, and the advices, which associate a body of code to join points. AspectJ implements aspects by weaving advice code into standard code and then by compiling the result into JVM byte-code that can be executed by any implementation of the JVM.

Keywords

AOP, aspects, Java, OOP

1. INTRODUCTION

While recent software abstractions like classes or functions enable more modularization of the system functionality, several concerns crosscut multiple modules of a program and make the code less clean by obfuscating the important program logic. *Aspect Oriented Programming* (AOP) tries to solve this with the introduction of the *aspect* abstraction, where crosscutting concerns can be captured.

We present the AspectJ compiler, an implementation of AOP for the Java language that allows the declaration of aspects as a class-like abstraction. Each aspect defines the *join points* of the program and the corresponding *advice* declarations with the code that must be weaved into the program. AspectJ supports for either static or dynamic join points, and offers several ways of declaring *pointcuts*, which are groups of join points. In the next sections, we present the concepts behind AOP, followed by AspectJ itself and its *join point model*. Finally, we present the technical details behind AspectJ and then we outline some conclusions.

2. ASPECT ORIENTED PROGRAMMING

Aspect Oriented Programming (AOP) tries to tackle the problem of *crosscutting concerns* that is found in imperative and object oriented (OOP) paradigms [3]. These concerns defy well-known abstractions like classes, methods and functions and crosscut the natural modularity present in the implementation of complex systems by being dispersed across multiple classes or methods [2].

The objective of AOP is to provide new means of abstraction that can explicitly capture crosscutting concerns and separate them from the *core concerns*. These new abstractions are usually called *aspects* and they allow the programmer to easily develop, maintain and reuse crosscutting concerns. By placing secondary concerns into a single, identifiable place, aspects keep the main program logic cleaner and more easily understandable, without irrelevant details.

In AOP languages or language extensions, the aspect abstraction is defined by a *join point model*. This model is composed of several concepts: well-defined points in the execution of the program, called *join points*; groups of join points called *pointcuts*; and method-like constructs used to implement the crosscutting concerns, called the *advices*. An aspect is thus an unit of modular crosscutting implementation that is composed of pointcuts and the corresponding advices. By allowing the use of pointcuts to describe certain points in the code, we can associate advices, thus allowing the execution of secondary code at specific parts of the program (for example, before the execution of methods).

3. ASPECT J

AspectJ is based on the AOP paradigm, aiming to help the developer by adding constructs to Java that enable the modular implementation of crosscutting concerns. Additionally it aims for code modularity and reusability. In order to facilitate its adoption by programmers, it is defined upholding two main properties [3]:

- *Upward compatibility* - all legal Java programs must be legal AspectJ programs;
- *Platform compatibility* - all legal AspectJ programs must run on standard Java virtual machines.

The compiler that recognizes the new language extensions is called *ajc*. However, *ajc* is not in truth a new compiler technology, but an extension to the Java compiler that performs

several program transformations, with *aspect weaving*, and generates standard JVM byte-code.

3.1 Installation

The installation of AspectJ can be done by downloading the AspectJ binary at `http://eclipse.org/aspectj/downloads.php`. The AspectJ Development Tools which provides Eclipse platform based tool support for AOSD with AspectJ, is also available at `http://eclipse.org/ajdt/`. AJDT provides the user with a lot of functionalities, including code highlighting, indication of sections on the code where advices will be introduced, among others. Despite its usefulness, the utilization of AJDT is not on the scope of this paper.

The installation of AspectJ includes the *ajc* compiler, browser, documentation tool and Ant tasks. Most of these are included in the ASPECTJTOOLS.JAR. The runtime classes required to use AspectJ however, are included in the ASPECTJRT.JAR. To install these two libraries, it is necessary to execute the downloaded ASPECTJ.JAR using java:

```
java -jar aspectj.jar
```

A wizard will be launched, where the user will have to specify the installation directory and the location of his JVM. Then he must only follow the instructions on screen and complete the installation.

3.2 Using the Compiler

The *ajc* tool is the AspectJ language compiler. To demonstrate how to build an AspectJ program let's assume that ajc is installed and that we have a normal java class - MainClass.java - and an aspect class operating on that class - An Aspect.aj. The simplest way to compile this code is done by invoking the compiler and passing the classes as arguments:

```
ajc MainClass.java AnAspect.aj
```

Note that in *ajc* all classes must be passed as arguments, since the compiler is not capable of finding the classes in the current folder [1]. The general invocation format for ajc is as follows:

```
ajc [options] [file... |@file... |-arglist file...]
```

ajc will compile files listed in its command line or files found in the list file. The list filename can be specified either by "@" symbol or -ARGLIST command option. A particularly useful option of *ajc* is -SHOWWEAVEINFO. This allows the developer to analyze where the aspect code affects the source code.

Once built, the project will run on any JVM. However, when running programs, the JVM must know the location of the library ASPECTJRT.JAR.

ajc possesses a lot of options [1] on how to link an aspect into existing Java code. The previous method was based on *source compilation*. Another method is to use *binary weaving*. An example follows:

```
ajc -inpath Application.jar NewAspect.aj -outjar ApplicationWithAspect.jar
```

This method allows the developer to use an already compiled application and add the new aspects into a new application without recompiling previous code. This way, source code is not required in order to add aspects to an application. This is only possible due to AspectJ's bytecode transformation properties instead of source transformation.

3.3 Join Point Model

In AOP, the join point model provides the framework that makes it possible for the execution of a program's aspect and non-aspect code to be properly coordinated [3]. As such, in AspectJ, crosscutting elements are defined by join points which represent well-defined points in the execution of the program. In order to do this, the AspectJ compiler can find places in the source code where some predefined operation is executed. AspectJ detects and operates on the following kinds of join points [4]:

- method call and method execution;
- constructor call and constructor execution;
- initializer execution and static initializer execution;
- object preinitialization and object initialization;
- field reference and field assignment;
- exception handler execution.

In order to detect these join points, AspectJ defines in its grammar a syntactic construct called *pointcut designator*.

3.3.1 Pointcut designator

A pointcut designator can be composed of several join points in addition to values inherent to the execution context of those join points [3]. In AspectJ, it is possible to implement two types of pointcuts: *primitive pointcuts* and *user-defined pointcuts*.

Primitive pointcuts are used to identify the previously declared join points supported by AspectJ. Some relevant examples are: `execute(method)` - to identify whenever a given method is executed; `instanceof(object)` - matches if the object type is an instance of `myobject`; `cflow(pts)` - matches all joint points that are strictly within the dynamic extent of the join points matched by `pts`; or `initializations(type)` - match all classes or object that possess the same specified type. Primitive pointcuts describe either *dynamic join points*, which require runtime checking to be executed (examples: `instanceof` or `cflow`), or *static join points*, that do not require runtime checking at all.

User-defined pointcuts define new pointcut designators using combinations of primitive or user-defined pointcuts with several special operators that can be applied to pointcut descriptors, such as `||` (logical or), `!` (logical not) and `&&` (logical and). The following example shows the declaration of a new pointcut called *printHello* that uses the primitive pointcut *execute*:

```
pointcut printHello() : execute (* print(..)) || execute(* write(..));
```

Finally, it is also possible to access data that is in the execution context of the join points. In user-defined pointcut declarations values can be expressed by a combination of positional and name matching and then can be used and manipulated in advices.

3.3.2 Advice

An advice defines the code that will run at certain join points when picked out by pointcuts. In other words, it contains the business logic that will be executed when a certain pointcut is triggered [4]. AspectJ allows programmers to state their advices as a method-like mechanism embedding all code inside [3]. The three basic kinds of advices are [4]:

- *after* - the goal of the after advice is to execute its body after the corresponding join point;
- *before* - the advice's body is executed before the join point picked by its pointcut is reached;
- *around* - the around advice enables the developer to replace the join point with an arbitrary code. The statement `proceed()` can be used to indicate that the replaced join point code can be executed.

Using them, the programmer may define when will the code run according to the corresponding pointcut. Following is an example of an advice using the *after* statement:

```
after() returning() : printHello() {
     printf(" World");
}
```

In this example, the defined pointcut is *printHello*. The advice states that the *printf* method should be executed when the pointcut is detected, in this case, immediately after the previous declared instructions in the *printHello* pointcut returns.

3.3.3 Aspect

Aspects represent the basis for AOP as the principal items for crosscutting implementations on which all the pointcuts and their corresponding advices are declared.

Aspects can be implemented similarly to Java classes, possessing properties like class extension, interface implementation or inner declarations. Unlike classes, aspects cannot be instantiated by hand but are automatically instantiated [4]. It is possible however to create abstract aspects and later extend these in other aspects.

Following is an example of an aspect using the previous declared pointcut and advice:

```
public aspect AnAspect {
    pointcut printHello() : execution (* print(..));

    after() returning() : printHello() {
         printf("World");
    }

    before() : printHello() {
         printf("This is an ");
    }
}
```

As an example to describe this aspect, suppose that the declared *print* method in the base code outputs the sentence *"Hello"*. The included advices state that whenever the *print* method is executed in the code, then the sentence *"This is an"* will be printed right before and the sentence *"World"* right after. In sum, an execution of the print method would output the sentence *"This is an Hello World"*.

4. TECHNICAL DETAILS

The main purpose of AspectJ is to insert advice code at the declared pointcuts and ensure that both core and advice code runs well together, in a process called *aspect weaving*. As AspectJ is at its core only a specification, it does not force any implementation method. However, the implemented tool itself tries to do most of the work during the compile-time, by inserting the advice code directly into the pointcuts before compilation. This has the advantage of allowing the compiler to catch errors earlier and increasing the efficiency of the final program.

Code not affected by aspects is compiled as traditional code, while code where advices apply is transformed to insert static points corresponding to the dynamic join points. For pointcut descriptors like `cflow` or `instanceof`, the weaver must insert code at the static points for runtime checking.

Before and after advices are compiled as standard methods and are called at the static points in the program. Around advices are compiled into multiple methods, one for each static point in the code. While this increases the code size, we trade that for boosted runtime efficiency of the code since it is easier to access state through the call-stack and to implement the `proceed` statement without costly runtime mechanisms.

5. CONCLUSIONS

In this paper, we presented the AspectJ compiler, an AOP extension and compiler to the Java language. We presented the most important aspects of the join point model found in AspectJ and how it can be used to define aspects. Finally, we gave some information in how the weaving of aspects is done in order to generate standard JVM byte-code.

6. REFERENCES

[1] E. Foundation. ajc, the aspectj compiler/weaver. `http://www.eclipse.org/aspectj/doc/released/devguide/ajc-ref.html`.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[3] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*. SpringerVerlag, 1997.

[4] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams, Indianapolis, IN, USA, 2002.