

# Report on An Overview of AspectJ

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Grisworld

Notes by Tai Hu, CMSC631 Fall 2002

## 1 Introduction

AspectJ is an aspect-oriented extension to JAVA program language. It enables a different way to further and cleaner modularize all concerns of interest in a complex systems comparing with the object-oriented mechanism. The AOP approach has a number of benefits. First, it improves performance because the operations are more succinct. Second, it allows programmer to spend less time rewriting the same code. Overall, AOP enables better encapsulation of distinct procedures and promotes future interoperation. The current design and implementation of AspectJ is upward compatible, platform compatible, tool compatible and programmer compatible. All these compatibilities let AspectJ could work with the current JAVA platform without any modification to the JVM. Also it could be seamlessly integrated into existing JAVA tools, such as JBuilder, Ant Builder, and Eclipse, etc.

## 2 Terminologies

Some important terms should be understood before we get into the detail of the paper.

**Join Point** – well-defined points in the execution of the program, such as, method call (a point where method is called), method execution (a point where method is invoked) and method reception join points (a point where a method received a call, but this method is not executed yet).

**Pointcuts** – a means of referring to collections of join points and certain values at those join points.

**Advice** – method-like constructs used to define additional behavior at join points

**Aspect** – units of modular crosscutting implementation composed of pointcuts, advice, and ordinary JAVA member declarations. For instance, exception handling in JAVA programming language should be explicitly handled in each try/catch blocks. However, as a whole, exception handling is just one aspect (or interesting concern). With AOP, we could centralize it at one place in order to cease developer from rewriting the same code.

## 3 How It Works

Here I will use the tracing example from AspectJ release 1.1b to illustrate how AspectJ works.

### 3.1 An Overview of Tracing Example

This example developed a tracing aspect for some existing shape objects. From this example, we could clearly see that how we could extract crosscutting concerns from the exist system and implement an aspect as a plug-in which could plug in or pull out without any effects on the current system. The core application only includes four java files which are TwoDShape.java, Circle.java, Square.java and ExampleMain.java. TwoDShape.java is the abstract super class for both Circle.java and Square.java. ExampleMain.java is the main driver for the application. All these codes are pure standard JAVA programs which could be compiled and run by javac and java commands respectively.

### 3.2 Implementing Tracing Aspect Plug-in

First of all, we should define a Trace class which will take responsible to output the trace message into the designated output stream. The member functions of this class are as follows:

```
public class Trace {
    public static int TRACELEVEL = 0;
    public static void initStream(PrintStream s) {...}
    public static void traceEntry(String str) {...}
    public static void traceExit(String str) {...}
}
```

As we could see here, if we don't have AspectJ, we have to call traceEntry() and traceExit() methods every where in the existing JAVA code in order to trace the program and see the messages. But with AspectJ we could concentrate the goal in one place and seamlessly plug into the existing code to achieve the exactly same tracing functionality. A tracing aspect could be defined as follows:

```
aspect TraceMyClasses {
    pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before (): myConstructor() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }

    before (): myMethod() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
}
```

```
    after(): myMethod() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}
```

From the above code, we could see that there are three pointcuts designators and four advices. All these pointcuts designators are user defined which represent a set of join points we interested. The myClass pointcuts designator is using within primitive designator to indicate that we only concerned the points where all the codes are inside of TwoDShape, Circle and Square classes. The rest of two pointcuts further narrow down our concerns. myConstructor includes all the points at class's constructors. And myMethod includes all the points at class's member functions. Here || and && are logical operations which have the same definition as those in JAVA or C++ programming language. .. and \* are wildcards supported by AspectJ. Then four advices specify when, where and what message should be printed to the output stream. Here before means the point right before the execution of the method body and after means the point right after execution of the method body. So when the execution of the program hits these join points, corresponding traceEntry and traceExit methods will be called and a message will be sent to the designated output stream.

### **3.3 How to Compile**

AspectJ release contains a compiler called ajc which is an extension to the javac compiler. It will transform the source program into a form in which there is an explicit corresponding method for each dynamic join point that might have advice at runtime. Then it will compile the code into standard JAVA byte code. So that it could be invoked in the standard JVM without any modifications. I will suggest Eclipse 2.1 with AJDT plug-in for AspectJ development. You could easily create an AspectJ project, invoke ajc compiler and run your application without leaving this IDE. Also you could get very nice tools for your AspectJ program, such as Aspect Visualizer, etc.

## **4 Conclusion**

Aspect-oriented programming enables the clean modularization of crosscutting concerns such as: error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, multi-object protocols. But as my own opinion, for a real large and complex system, it is still very hard to extract aspect out. AspectJ may be more appropriate for adding debugging support or any plug-ins into existing code without a modification to the old code.