Aspect Oriented Programming

2002-11-12

By Gustav Evertsson, pt99gev@student.bth.se

Contents

CONTENTS 1
INTRODUCTION 2
THE PROBLEM 2
THE ASPECTJ LANGUAGE 3
ASPECTS 3
JOIN POINTS 3
ADVICES 4
DESIGN IMPROVEMENTS 5
DEVELOPMENT 5
Logging 5
Tracing 5
Profiling 5
Errors and Warnings during Compiling 5
PERFORMANCE 5
EXCEPTION HANDLING 6
STATIC CROSSCUTTING 6
CONCLUSION 7
REFERENCES 8

Introduction

This paper is written to give an overview of what Aspect Oriented Programming (AOP) is and what it can be used for. This paper is done as a part of the course Advanced Software Engineering (pad004) at Blekinge Institute of Technology.

The Problem

AOP is the solution, but the solution to what problem? Object-Oriented (OO) divide the world in different Objects and components and that can be a problem when it comes to functionality that cross cut the object world. It can be hard to modularize this into classes. Example of such functionality can be synchronization, performance optimization, exception handling etc.

Another problem with OO is that you need to decide all the interfaces before any implementation can start. This is because it is hard to change an interface afterward because it may change a lot of classes. AOP solves this so you can modify the static structure of them afterward without changing any code within the classes.

The AspectJ Language

The language described here is AspectJ. That is an implementation of AOP in Java which is developed by Xerox Parc and version 1.0 was released in November 2001. It works as a precompiler and you can see the generated code which can help a lot during development before you getting used to the new syntax. It also gives the advantage that the end users don't need to install anything special to run the programs except the virtual machine.

AOP is constructed from OO and only extends the benefits that you already have without taking anything away. Your aspects can have the same functionality as you classes with instance and static methods, attributes and so on. The aspects can add functionality to the classes and/or change it without changing the code in them in any way.

Where the aspect is used much depend one what it is used for. If it is temporary, part of the system or the system can function without it but better with it.

One common implementation is that functionality appears in multiple methods. It may also be temporary used only during the development. Example of this is logging and error handling. The solution is to place it just on one place and use that in the entire system. This makes the code cleaner and less coding is needed.

Example of functionality that can be separated into an aspect is synchronization and performance optimization. This is functionality that is often used in classes but not a part of the normal business logic. It is also aspects that can be reused in many classes and also be reused in different projects.

Aspects

Aspects are for AOP what classes are for OO. It gathers all the functionality inside of it. It can extend other aspects or classes in the same way as with classes.

```
Aspect ExampleAspect {
}
```

This example doesn't do much; just declare a new aspect with the name "ExampleAspect".

Join Points

It is with join points you decide where the aspects is executed. AspectJ includes 11 different join points. From a normal method call to more advanced join points such as "within" and "target". The join points can be combined as a boolean expression. Many of the different join points take the name of the class and/or method as a argument and AspectJ accept "*" and ".." as wildcards their.

```
Aspect ExampleAspect {
   pointcut p() : call(int ExampleClass.*(..));
}
```

A join point has been added to the previous example. This join point is fired every time a method in the "ExampleClass" is called that takes zero or more arguments and returns an integer.

Advices

Advices are the executable part of the Aspects. It defines what code to run when a join point is fired. It exist three basic kinds of advices; before, after and around. They all works as a method call with some limitations. The before and after advices can't return anything and around must return the declared type. Before and after advices are executed as it sounds before or after for example methods calls and around replace the method calls.

```
Aspect ExampleAspect {
   pointcut p() : call(int ExampleClass.*(..));

   before() : p() {
      System.out.println(“Hello World!”);
   }
}
```

The previous added pointcut is now handled by the new advice.

Design improvements

The main design improvement you get with AOP is better modularization. Redundant code can be placed in an aspect instead of copied to all classes that need it. You can concentrate on putting the business logic in the classes and the rest can be handled with Aspect. This makes the code easier to read and inspect. But the aspect can also very easy make the code harder to follow because if you don't know about the pointcuts will you don't know what's happening by just follow the logic path of the code. This can specially be a problem if the design is changed later during the development and functionality is added with aspects.

Development

On big advantage with AOP is that you can put the debugging code outside the normal code and can easily turn it on when you need it and off when you compile the final version. In normal case is it hard to handle this type of code. You don't want it in the final version but you need it for maintenance so it may not be the best idea to just delete it either.

Logging

Put a point cut on all method calls except the output method. Then make an advice that implement before() and output the method name and all the parameters.

Tracing

Extend the logging functionality so it has an after() advice too. And add a stack that push when before() is called and pop when after() is called. With this small implementation can all method calls be tracked and faults can be found without changing the code in any ways.

Profiling

If performance is a problem must the bottlenecks be found and this can be a time consuming task in a normal project. But with some help from AOP can this be done with a few lines of code. You only have to change the tracing example so it has a timer and outputs the difference from the before() to after() calls.

Errors and Warnings during Compiling

The AspectJ compiler has a function to find warnings and errors during compiling with the help of pointcuts. The only limitation is that it does not accept pointcuts that needs runtime information. This can be used to check that code standards is followed and classes that needs to be executed in a special way is that.

```
declare warning: call(ExampleClass.new(..))
"Constructor Called!";
```

This example outputs a warning if the constructor of the "ExampleClass" is called.

Performance

Aspects can also be used to improve the performance of the system. This type of functionality in normally used within many classes and it can therefore be hard to find a good place to put them. Instead can you put them in an aspect and use that in the entire system. Example of this type of functionality is polling, caching, buffering etc.

Exception handling

The normal case is that you want to terminate the current function and output an error message to the user. Because this is handled in almost the same way in the entire program can it be a good idea to put this in an aspect. The handler() pointcut picks an execution of an exception handler.

Static Crosscutting

Aspects can in many ways drastically change how the original class works. Aspects can also change the static structure by adding fields and methods and even change the class hierarchies. This can be very powerful tool and you can change the original class beyond recognition.

```
declare parents: ExampleClass extends ExampleParent;
```

This example changes the "ExampleClass" so it extends "ExampleParent". All methods must be implemented if new abstract methods are added; they can already be implemented or added by aspects.

Conclusion

AOP is a very powerful tool that I think will become much bigger in the future. But for now is too much of an academic project and under development to be really useful for a bigger audience. More research and standards is needed such as design and usage patterns.

AOP and AspectJ have a lot of advantages and I especially like the way it helps with the coding and debugging such as the way logging and tracing can easily be implemented.

References

- AspectJ version 1.1a1, http://www.aspectj.org

- Ivan Kiselev, Aspect-Oriented Programming with AspectJ, 2001, Sams, ISBN: 0-672-32410-5

- Nicholas Leidenfrost, An Introduction to Aspect Oriented Programming with AspectJ

- Elizabeth A. Kendall, Aspect-Oriented Programming in AspectJ