Aspect Oriented Programming

Aleksi Kallio

Helsinki 19th April 2004

Principles of Programming Languages seminar

UNIVERSITY OF HELSINKI
Department of Computer Science

Contents

1 Abstract

Aspect Oriented Programming is a programming methodology that improves modularisation of crosscutting concerns in the program source code. By modularising concerns into aspects programmer can improve the clarity and maintainability of source code.

2 Motivation for AOP

The art of software engineering can be seen as a task of modeling a given problem domain. Given the fact that real life problems often display a great variety of possible points of view and multitude of related and unrelated concerns, it can be argued that modeling tools must allow the programmer to separate different points of view and unrelated concerns in the program source code.

In AOP terminology, a concern is a particular goal, concept or area of interest [Lad02]. Unrelated concerns are often referred to as crosscutting concerns. This reflects the way of seeing a piece of software as a multidimensional object, which can be observed from different angles and where unrelated functionalities are thought to be in an orthogonal setting. A good example of crosscutting concerns are application logic and logging. Logging is usually just a way of documenting the high level program execution and has no effect on the actual logic the application implements. However in the low level program code execution the two concerns are tightly coupled because all writes to log must happen immediatly after the actual action is executed. It can be argued that because of low level coupling, many modern programming languages also force to couple these two concerns in higher level, how unrelated they may be.

Crosscutting concerns are a source of code tangling and code scattering [Lad02]. Code tangling means that often in a single point of source code many unrelated concerns are present, making the code harder to understand. Code scattering, on the other hand, refers to fact that crosscutting concerns by their nature are scattered throughout the code, making code management very difficult. Tangling and scattering cause poor traceability, lower productivity, less code resuse, poor code quality and more difficult evolution [Lad02]. Aspect Oriented Programming was introduced to better encapsulate crosscutting concerns and improve the clarity and maintainability of source code.

Besides pragmatical benefits, AOP can also improve the design of the high level program architecture. It has been argued that having one dominant programming paradigm is not always beneficial, because of an effect called "the tyranny of the dominant decomposition". Single method of decomposition cannot satisfy the demands of a tough modeling task. Decomposition along a single dimension is often inadequate, resulting in reusability and traceability problems [TOHJ99].

3 The Field of AOP

The field of aspect programming is still quite unstructured and non-matured. However it is widely agreed that two major mainstreams can be seen: program transformers and composition filters. Composition filters are older than the concept of AOP. They do not modify the source code of the program, but instead by using different dynamic mechanisms modify the messages that are passed between program components. This kind of dynamic weaving was seen as the most promising method for future aspect tools, but implementing it has proved to be surprisinly challenging.

Program transformers are aspect tools that instrument the original source code with aspect code to produce the final program code. Such instrumentation is often cited as weaving. Three main classes of program transformers are: builtin aspect structures, composition based transformers and reflexive transformers. Builtin aspect structures can be found from non-aspect oriented languages, such as *synchronized*-structure in Java language. Composition based transformers modify components according to the demands of aspects. Examples of such tools are AspectJ and D^2AL. D^2AL is a programming methodology and aspect language that incorporates UML based abstractions to create the distribution of components in a distributed system. AspectJ is discussed in more detail later. Reflexive program transfomers employ run-time reflexive operations to modify components. Example of such a solution is SMove. SMove introduces so-called reflective components, which present two kinds of interfaces: the base-interface that provides the primary functionality and the meta-interface that allows the client application to negotiate aspects of how the primary functionality is provided.

Another way to classify AOP tools is a division between black box and clear box implementations [FF]. Black box AOP is approach where program components are "black boxes" ie. their internal structure is invisible for the AOP tool. In clear box AOP the internals of components are visible and subject to modification to AOP tool. Of course there are varying levels of granularity, as it is advisable to hide the lowest level details even in clear box AOP.

4 AspectJ

The best known aspect tool is AspectJ, a general purpose AOP implementation from Xerox PARC [Lad02]. AspectJ is a composition based program transformer that uses static weaving of aspect code into Java source code to produce final programs. The core concepts of AspectJ are aspects, point cuts, join points and advice. Aspects are encapsulated concerns, that crosscut the main program logic. Join points are the points of main logic where control is passed to aspects. For example, poincut can be the points of exiting from a method, where the logging aspect is given the control. Advice is a code snippet to be executed in a pointcut. In the previous case, advice would be simply a call to logging component.

Aspect uses a so-called weaver, which is an aspect compiler that compiles classes and aspects into standard Java bytecode [Lad02]. In weaving the main logic is instrumented with approriate calls to aspect logic.

As an example we present a source code of a counting aspect implemented with AspectJ. Purpose of this aspect is to modularise the concern of counting calls to database query methods. Aspect introduces one pointcut called *findCall*, which captures all points in the program code where *Database*-methods with a name beginning with *find* are called. Aspect includes also an advice, which calls *Counter*-class before database queries.

```
aspect CountFinds {
   pointcut findCall(): call(* Database.find*(..));

   before() returning: findCall() {
       Counter.increaseFindCount();
   }
}
```

Aspects acts a the basic unit of modularisation. Inside it, there is a definition for a pointcut called *findCall*. It is defined to consists of call to methods that have the following properties: return type can be anything, the method is in class *Database*, the name begins with *find* and the parameter list can be anything. Aspect introduces also one advice that executes before given pointscuts, just calling counting logic.

Besides method calls, also execution of method bodies, execution of exception handler, type of executing object and location in relation to other pointcuts can be used for defining pointcuts. Pointcuts can also be handled similarly to sets. If we wanted to capture calls that begin with *query* or *executeQuery*, we could have defined the pointcut in the following manner.

```
aspect CountFinds {
   pointcut findCall():
call(* Database.query*(..)) ||
call(* Database.executeQuery*(..));

   before() returning: findCall() {
       Counter.increaseFindCount();
   }
}
```

We can also capture the context of the pointcut in the advice. The next example demonstrates it, and also uses *after*-advice, which is executed after the pointcut.

```
aspect CountFinds {
```

```
   pointcut findCall(String query):
call(* Database.find*(String)) &&
args(query);

   before() returning: findCall() {
       Log.add("making query: " + query);
   }

   before(String query) returning: findCall(query) {
       Log.add("made query: " + query);
   }
}
```

Advice can gain information about the active pointcut by using a special variable called *thisJoinPoint*. Aspects in AspectJ are singletons by default. However, also per-object aspects can be used. They associate a unique aspect instance for each object. The system automatically looks up the aspect instance associated to the object, and uses the instance as the execution context.

5 Synthesis

The intent of AOP is to modularise something that would otherwise be scattered and tangled into source code: "AOP can be understood as the desire to make quantified statements about the behaviour of programs, and to have these modifications hold over programs written by oblivious programmers" [FF]. Using this definition, many methods that have no AOP roots also qualify as aspect tools: for example mixins with multiple inheritance and event based publish and subscribe [FF].

In a high level AOP can be described as creating independent but crosscutting components and composing the unified program. Composition is matching units that describe the same concent, reconciliating the differences in descriptions and integrating the units to produce the unified whole [TOHJ99]. Thus the main difference between traditional programming and AOP is that components can be crosscutting and therefore the process of composition is much more demanding.

AOP is used successfully at least with web application architectures. For example authentication and authorisation can be implemented effectively with AspectJ. In general, aspects are at their best in distributed environments, where many more-or-less related concerns have effect on software design and where the maintainability of source code is exceedingly important.

6 Summary

The purpose of AOP is to allow modularisation of functionality that is tangled and scattered into program source code. The motivation is that by using quantified statements, aspects can be weaved into main program logic and implementers of main logic need not be concerned with aspect functionality. This is how AOP enhances modularisation and code readability and traceability.

References

FF Filman, R. and Friedman, D., Aspect-oriented programming is quantification and obliviousness. *Workshop on Advanced Separation of Concerns, OOPSLA 2000.*

Lad02 Laddad, R., I want my AOP! (part 1) - Separate software concerns with aspect-oriented programming. *Java World.* URL `http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.%html`.

TOHJ99 Tarr, P. L., Ossher, H., Harrison, W. H. and Jr., S. M. S., N degrees of separation: Multi-dimensional separation of concerns. *International Conference on Software Engineering*, 1999, pages 107–119, URL `citeseer.ist.psu.edu/tarr99degrees.html`.