# AspectJ & AspectWerkz

Aspect Oriented Programming in Java

David Fuchs, Remo Egli

*Aspect Oriented Programming is a powerful programming methodology to modularize a program's structure. In this article, we will first introduce the reader to the benefits of AOP. Different types of AOP are presented. We then explain the basics of the first AOP implementation, AspectJ. Furthermore we will discuss another AOP implementation, AspectWerkz, and explain how AspectWerkz and AspectJ together are evolving into a new AOP language, AspectJ5.*

*Finally, we will contrast the AOP approach of AspectJ/AspectWerkz with two other popular AOP solutions, the Spring framework and PROSE.*

## Introduction to AOP

### Crosscutting Concerns

A *core concern* of a software system is single, specific function the system performs. A core concern of a financial application, for example, could be the processing of financial transactions or the representation of a customer. However, typical software systems also comprise multiple *crosscutting concerns,* secondary functions that affect many of the core concerns. Typical examples of crosscutting concerns include logging, authentication, security policies, or transaction management.

With the advent of the Object-Oriented Programming (OOP) paradigm, the concept of a *class* changed the way complex systems are developed. Classes allow to separate and encapsulate different core concerns, therefore hiding their implementation details. To view a system as a set of collaborating objects allows programmers to modularize code according to the core concerns.

However, crosscutting concerns often span over several unrelated classes. Even though the core concerns of each class may be very different, code needed to perform the secondary crosscutting concerns is often identical. (See figure 1.) The OOP model does not adequately address this behavior. The nature of the crosscutting concerns requires classes to perform tasks other than the core concerns they implement. This has several implications:

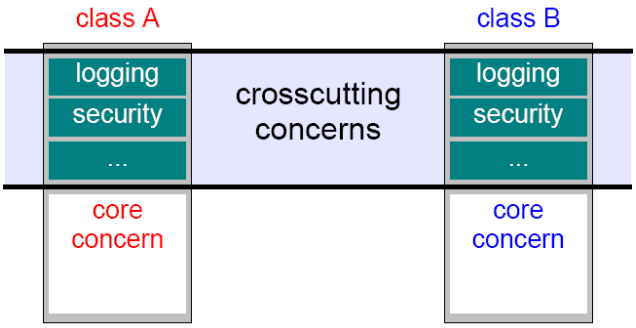


Figure 1: Crosscutting concerns in the OOP model.

- Code is harder to write.
  A developer must think about many concerns simultaneously instead of being able to focus on the main concern. Making errors is thus easier.
- Code is harder to maintain.
  As the implementations of core- and crosscutting concerns are not separated, code becomes bloated, harder to understand and more error-prone. Also, changed requirements of a single crosscutting concern require changes and recompilation of all affected classes.
- A system is harder to evolve.
  Predicting future crosscutting requirements is a very difficult task. If unforeseen requirements emerge, changes or even reimplementation of many parts of the system may become necessary. On the other hand, implementations that try to address even low-probability future requirements tend to become over-designed and confusing.

Most application servers and frameworks address some crosscutting concerns in a modularized way. The Enterprise JavaBeans architecture, for example, handles concerns such as logging and security. The downside of such an approach, however, is that the provided solutions are specific to the framework, thus implementation of the core concerns becomes dependent on the framework in use. Design patterns such as Mix-In Classes or Visitor Pattern allow a developer to defer

a concern's implementation, but the actual control of the operation (e.g., invoking a visitor) still are not separated from the core concern's implementation. AOP provides a powerful and generic solution to separate these concerns.

### AOP fundamentals

An *aspect* is a separately implemented piece of code that is executed ("woven in") at a specific point in the execution of a program. It is the basic unit of modularization of an AOP language, as is the class in OOP languages. Thus, each concern can be implemented separately; in the previous example of a financial application, a developer would write code for logging, authentication and other requirements each as an own, independent aspect. A *weaver* later combines the aspect code with other program code. This principle is depicted in figure 2.

More formally, an AOP language is made up of the following elements:

**Join Point:** A specific point in a program's flow, e.g. the call of a specific method.

**Pointcut:** Set of join points and variable values at these points, e.g. all calls of a given method and the parameter values.

**Advice:** Code to be executed when a join point is reached.

**Aspect:** Advice and a non-empty set of pointcuts.

Note that AOP and OOP are not competing paradigms. AOP is orthogonal to OOP in that it addresses the encapsulation of crosscutting concerns, whereas OOP separates and encapsulates core concerns.

### Types of AOP

The weaver is an important element of AOP as it composes independently implemented concerns to form the final system. Depending on when code weaving takes place, different types of AOP are distinguished:

**Run-time AOP:** Advice code is woven in during runtime, without the need for application restarts. This requires a modified VM [1] or "proxy objects" [2] that trace and intercept method calls, field access etc.

**Load-time AOP:** A modified class loader weaves in advice code when classes are loaded into the VM. Aspects can be selected at load time, and no recompilation is necessary when changing the selection.

**Compile-time AOP:** The weaver modifies the Java source or the Java-bytecode. A compile-time weaver outputs "normal" Java-bytecode which will run on any VM. The compiled code is also very efficient, as all aspects are statically compiled in the bytecode, eliminating the indirections introduced by proxy objects or modified VMs.

## AspectJ

AspectJ is an open-source, compile-time AOP implementation for Java. Developed in the mid-90s at Xerox PARC and thus the first AOP language, it is now an openly developed eclipse.org project [3] and released under the Mozilla Public License. A development plugin for eclipse is also available [4].

AspectJ is an *extension* of the Java language, adding to it the pointcut, advice, and aspect constructs. It therefore needs its own compiler, *ajc*. Furthermore, AspectJ includes *type-modification constructs* that allow to modify the static structure of a program, i.e., add new fields, methods, and supertype declarations to classes and even interfaces. While this is a very powerful mechanism, it must be noted that it arguably violates the encapsulation property of OOP.

### AspectJ Example

A very simple sample aspect tracing calls to any method with name `myMethod` in `MyClass` is shown below.

```
public aspect Logger{

    // named pointcut definition
    pointcut: logHere():
        call( * MyClass.MyMethod(..) );

    // advice for above pointcut
    before: logHere(){
        System.out.println("Entering "
            +thisJoinPoint);
    }
}
```

When woven into any program, the program will print something similar to `Entering call(void MyPackage.MyClass.myMethod())` each time `myMethod` is called, before method execution. This is a good example of how AOP can be used to relieve a programmer from the tedious task of adding debug code to a program and eventually commenting it out again for the release build.

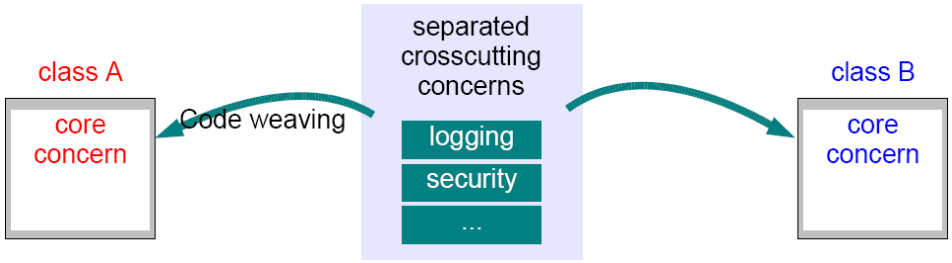


Figure 2: AOP implementation of crosscutting concerns

### Aspect structure

The structure of an AspectJ aspect resembles that of a Java class. Within the aspect, advice code and pointcuts are defined. Like Java classes, aspects can also contain constants, fields and methods and may be marked as `public`, `abstract`, or `final`.

A wide range of pointcuts is supported by AspectJ, including

- method call and execution
- constructor call and execution
- get and set access to a field
- exception handler execution
- lexical-structure based pointcuts to select join points within the lexical scope of classes or methods
- control-flow based pointcuts to select join points within the control flow of a given method

Pointcuts can also be combined using the logical operators `||`, `&&`, and `!`. Pointcuts can be *named* (as in the example), or *anonymous*, in which case the pointcut definition is specified directly in the advice. The joinpoints selected by a pointcut are described by the pointcut type (e.g., `get` or `call`) and a *pattern* containing package, class, and method/field signatures. Thus, patterns to choose a method or field, respectively, look as follows.

- `<return_type> <package>.<class>.<method> (<parameter_type>)`
- `<field_type> <package>.<class>.<field>`

The wildcards `*` (for return types and names) and `..` (for method parameters) may be used in a pattern. The detailed pattern syntax can be found in the AspectJ Programming Guide [5].

AspectJ provides various ways to associate an advice to a pointcut:

- a `before` advice is executed just before the joinpoint is reached, as in above example
- `after returning` advice is executed after the joinpoint, if no exception occurs
- `after throwing` advice runs after an exception has been thrown
- `after` advice is always executed after a joinpoint, comparable to `finally`
- an `around` advice replaces the advised code. Using `proceed()`, the advice can call the original code.

Pointcuts not only pick out join points, they can also expose part of the execution context at their join points. Values exposed by a pointcut can be used in the body of advice declarations. An advice for some method therefore has access to the method's parameters and the object executing the method.

## AspectWerkz

AspectWerkz [6] is an aspect oriented framework for the Java language. It utilizes bytecode modification to weave classes at compile-time or load-time. Experimental support for run-time weaving is also available. AspectWerkz is based on pure Java and is not a new language like AspectJ. AspectWerkz is free and open source (GNU Lesser General Public License), sponsored by BEA Systems.

### Aspects

As in AspectJ, the aspect is AspectWerkz' unit of modularity for crosscutting concerns. An aspect contains pointcuts, advices and introductions (AspectWerkz' analog of AspectJ's type modification constructs). Any Java class can be an aspect, no specific interface must be implemented. The only constraint is that the class has either no constructor at all or one of following two:

- A default no-argument-constructor

- A constructor that takes an `org.codehaus.aspectwerkz.AspectContext` instance as its only parameter

Since aspects are regular Java classes, it is possible to reuse aspects by marking them as `abstract` and inheriting from them. Pointcuts and bindings between pointcuts and advice are configured using either Java annotations within the class file or by external XML configuration files. (See figures 3 and 4.) The two possibilities are equivalent and can be used together.

Figure 3: Annotation style aspect

```
@Aspect
public class MyAspect{

    //pointcut definition
    @Expression(call( * MyPackage.MyClass.
        myMethod(..)))
    Pointcut MyPointcut;

    //before advice definition
    @Before(myPointcut)
    public void myBeforeAdvice() {
        // do some stuff
    }
}
```

Figure 4: XML-configured aspect

```
<aspect class=MyAspect>

    <pointcut name=MyPointcut
        expression=call( * MyPackage.MyClass.
            myMethod(..))/>

    <advice name=MyBeforeAdvice
        type=before
        bind-to=MyPointcut/>

</aspect>
```

### Pointcuts

A lot of joinpoint types supported by AspectWerkz. (The list is almost identical to that of AspectJ.) The joinpoints which should be selected by a pointcut are described by the same joinpoint selection pattern language as in AspectJ. Similar to AspectJ, joinpoints can be combined using the boolean AND, OR, and NOT operators.

In AspectWerkz, pointcuts are defined as fields or in special cases as methods in an aspect class. A method-definition is required if the pointcut takes arguments. Pointcuts can be named or anonymous.

### Advices

Advice code can be executed `before`, `after` or `around` (instead of) the pointcut code. It is also possible to distinguish between a successful method call (`after returning`) and an exception (`after throwing`). Advices are regular methods in aspect classes with a specific signature defined by the AspectWerkz specification.

### Introductions

The goal of introductions is adding code to existing classes. The implementation in AspectWerkz is using mix-in classes. The code of the mix-in class is "mixed" into all classes which are picked out by a pointcut. Mix-ins can simulate multiple inheritance by adding interfaces and implementations to existing classes.

Any regular class implemented as an inner class within an aspect can be a mix-in. The only requirement is that this class must consist of at least one interface and an implementation of this interface.

## AspectJ5

As described in the previous sections, there are a lot of similarities between AspectJ and AspectWerkz. For example, the supported joinpoint types and joinpoint selection are almost identical. On the other hand, both projects have particular strengths and use somewhat different approaches. The AspectJ language supports efficient compile-time weaving, while the Java-based AspectWerkz also offers more flexible load-time weaving and provides a simple annotation-based or XML-based development style.

The goal of AspectJ5 is to combine the complementary strengths of the two projects to produce a single, powerful aspect-oriented programming platform. In January 2005, the two development teams announced that the AspectWerkz developers will join the AspectJ project to bring the key features of AspectWerkz to the AspectJ platform.

AspectJ5 is thus the successor of AspectJ, enriched with a lot of new features. The most important modification is the integration of load-time weaving, an essential part of AspectWerkz. Another useful feature is the new annotation-based style of development which is called @AspectJ-annotation in AspectJ5. This feature allows a programmer to write aspect definitions in a simple way directly into the source code files.

It is important to note that there is only one language, one semantic and one weaver, but two different development styles which can be mixed in a single project. Users familiar with the current AspectJ code style can continue to use it in AspectJ5 projects. New users can utilize the new, simpler annotation style.

AspectJ5 is still a fully open-source project on the eclipse.org platform, backed by IBM and BEA Systems.

# Other AOP solutions

Finally, we will compare AspectJ5 to two other prominent AOP solutions: Spring AOP, and PROSE.

### Spring AOP

Spring is a powerful and effective Java application framework, with AOP being one among many abilities. Spring uses a proxy-based approach to provide run-time weaving; load- or compile-time weaving are not supported. Spring AOP is based on pure Java, and no special compiler or modified VM are required.

In contrast to AspectJ5, Spring AOP can only advise method and constructor execution. Spring's goal is not to provide a full-fetched AOP solution, but rather a tight integration of the AOP features into the framework. As the Spring website puts it, "Spring AOP will never strive to compete with AspectJ or AspectWerkz to provide a comprehensive AOP solution. ... [AspectJ and Spring] are complementary, rather than in competition." AOP advice is specified using the "normal" Bean Definition XML files, and the framework manages advice and pointcuts, easing administration. As a downside, aspects must implement interfaces provided by the framework.

Version 1.1 of the Spring framework provides a closer integration of AspectJ aspects. The framework is now able to configure AspectJ aspects just like ordinary Java classes. For future releases, the support of AspectJ pointcut syntax for Spring AOP is planned. Also, the export of some Spring services (e.g., transaction management) as AspectJ aspects for use without the Spring framework has been announced.

### PROSE

PROSE is a middleware platform for dynamic run-time adaptation. It is an open-source project of the ETH Zurich's Computer Science Department. The main goal of PROSE is the ability to change an application's code during run-time, enabling updates or bug fixes to be deployed in a running system without restart. In contrast, AspectJ5 (as well as Spring AOP) represents a programming paradigm addressing the issues discussed in the first section of this article.

PROSE is based on pure Java, but requires a modified Java VM to intercept calls to methods and fields. Currently, Sun JVM and IBM Jikes RVM are supported. A powerful ability of the PROSE platform is *atomic weaving* that atomically enables all join points of an aspect at the same time, ensuring the application's consistency. Load- or compile-time weaving are not available.

A wide range of join points are supported, the selection of join points is done with regular expressions and thus very flexible. Additionally, filtering mechanisms allow execution of advice code based on various properties of the execution point.

An eclipse plug-in for easy aspect development as well as a PROSE workbench to visualize and control aspects in a VM are also available.

# Conclusion

AspectJ, the first AOP implementation for Java, allows developers to modularize crosscutting concerns. Its key benefits are easier-to-write and easier-to-evolve systems and good performance. AspectWerkz strengths are the more flexible load-time weaving support and the annotations-based and XML-based development styles. Other AOP solutions, such as Spring or PROSE, have other main objectives and are therefore not in competition to AspectJ/AspectWerkz.

Combining the excellence of both AspectJ and AspectWerkz, AspectJ5 could evolve into an even more comprehensive and powerful AOP solution with an ever-growing base of users and expertise.

# References


[1] G. Alonso, T. Gross, A. Nicoară et al. Various papers on PROSE. `http://www.iks.inf.ethz.ch/publications/publications/aosd02.ps`; `http://www.iks.inf.ethz.ch/publications/publications/aosd03.ps`; `http://www.iks.inf.ethz.ch/publications/PROSE-ASMEA05.pdf`

[2] *Spring AOP.* `http://www.springframework.org/docs/reference/aop.html`

[3] `http://eclipse.org/aspectj/`

[4] `http://eclipse.org/ajdt/`

[5] `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`

[6] `http://aspectwerkz.codehaus.org/`