

# Recent Developments in AspectJ™

Cristina Videira Lopes and Gregor Kiczales

Xerox Palo Alto Research Center  
3333 Coyote Hill Rd., Palo Alto CA 94304, USA  
{lopes, kiczales}@parc.xerox.com

**Abstract.** This paper summarizes the latest developments in AspectJ, a general-purpose aspect-oriented programming (AOP) extension to Java. Some examples of aspects are shown. Based on our experience in designing language extensions for AOP, we also present a design space for AOP languages that may be of interest to the AOP community.

## 1 Introduction

Traditionally, programs involving shared resources, multi-object protocols, error handling, complex performance optimizations and other systemic, or cross-cutting concerns have tended to have poor modularity. The implementation of these concerns typically ends up being tangled throughout the code, resulting in systems that are difficult to develop, understand and maintain.

Aspect-oriented programming is a technique that has been proposed specifically to address this problem [3]. In the last couple of years we have been designing aspect-oriented languages. That work lead us to AspectJ™, a general-purpose aspect-oriented extension to Java.

In AspectJ, aspects are programming constructs that work by cross-cutting the modularity of classes in carefully designed and principled ways. So, for example, a single aspect can affect the implementation of a number of methods in a number of classes. This enables aspects to capture the cross-modular structure of these kinds of concerns in a clean way.

## 2 Most Recent Features of AspectJ

In this position paper we illustrate only the basic features of AspectJ. A more comprehensive description of the system can be found in [4]. The basic features are presented with a couple of examples. Consider two classes, Point and Line, with set and get methods (the implementations are not shown), and an aspect that is intended to show the kind of accesses (i.e. read/write/create) that are performed on points and lines:

<pre> class Point {     Point(int x, int y)     void set(int x, int y)     void setX(int x)     void setY(int y)     int getX()     int getY() } </pre>	<pre> class Line {     Line(int x1, int y1,          int x2, int y2)     void set(int x1, int y1,             int x2, int y2)     // also set y1, x2, y2     int getX1()     // also get y1, x2, y2 } </pre>
<pre> aspect ShowAccesses {     static before void Point.set(*), void Line.set(*)                   void Point.setX(*), void Point.setY(*),                   void Line.setX1(*), void Line.setY1(*),                   void Line.setX2(*), void Line.setY2(){         System.out.println("Write");     }     static before int Point.getX(), int Point.getY(),                   int Line.getX1(), int Line.getY1(),                   int Line.getX2(), int Line.getY2() {         System.out.println("Read");     }     static before Point(*), Line(*) {         System.out.println("Create");     } } </pre>	

This aspect contains three weave declarations (“weaves” for short), all of them starting with `static before`. The effect of this aspect is that every time an instance of either of those two classes is invoked or created, a message is printed out on the screen: if the method invoked is one of the set methods, then the string “Write” is printed out; if the method invoked is one of the get methods, then the string “Read” is printed out; and if a constructor is executed, then the string “Create” is printed out.

As the example shows, the weaves apply to elements of classes, such as methods and constructors. Designators name those elements. A method designator has the generic form *Type* *Type.MethodName(Formals)* and a constructor has the generic form *Type(Formals)*, where *Type* is a class or interface name. The character ‘\*’ can also be used to indicate any return type and any list of formal parameters. Fields and class initializers can also be used in weaves (see next example).

The keyword `before` means that the body of those weaves is to be executed before the body of the element (method or constructor) is executed. AspectJ also supports `after` – meaning that the body of the weave is to be executed after the

body of the element is executed – `catch` and `finally` – both of these being similar to Java’s `catch` and `finally` constructs.

All of these (i.e. `before`, `after`, `catch` and `finally`) are called *advise weaves*, in that they annotate the classes’ elements with code wrappers. AspectJ supports another kind of weave called the *new weave*, which extends the classes with new elements. For example,

```
aspect Color {
    static new Color Point.color, Line.color;
    static new void Point.setColor(Color c),
                    Line.setColor(Color c) {color = c;}
}
```

The above aspect extends the classes `Point` and `Line` with new fields of type `Color` called `color` and a new methods called `setColor`. When this aspect is woven, it is exactly as if `color` and `setColor` are members of classes `Point` and `Line`.<sup>1</sup>

The keyword `static` means that no aspect instances are involved. Static weaves are always executed for all instances of the designated classes. The alternative is to have non-static weaves, which require the instantiation of the aspect.<sup>2</sup>

### 3 Properties of Aspects

The basic features explained above are sufficient for explaining three important properties of AspectJ.

First, aspects can capture cross-cutting design issues. In the `ShowAccesses` example, each weave applies to several methods of two classes. Without AspectJ, the lines of code that print out the messages would be repeated over and over again, and the abstraction behind this aspect would be lost.

Second, AspectJ is more general-purpose than the other AOP languages we have designed. Because this language is fairly generic, aspects can capture a diverse set of cross-cutting design concerns. Aspects can be used in distributed and non-distributed applications for debugging, concurrency control, inter-class protocols, optimizations and for programming many non-functional issues.

Third, aspects can be easily plugged-in and plugged-out of the applications. For example, the `ShowAccesses` aspect is plugged-in simply by invoking the weaver:

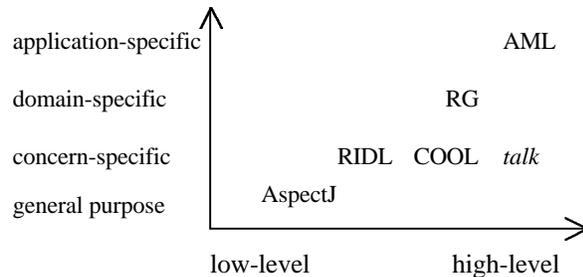
```
% ajweaver Point.java Line.java ShowAccesses.java
```

With AspectJ plugging in and out the code of an aspect involves no editing of the classes, and the code is truly inserted or removed from the executable code (as opposed to its activation being conditional on some flag). Plugging the `ShowAccesses` aspect out is as easy as omitting its file name from the command line above.

---

<sup>1</sup> This example is intended to illustrate the meaning of the *new weave*. It is by no means intended to suggest that the `color` feature of points and lines should be coded as an aspect. The issue of when to use it falls out of the scope of this position paper.

<sup>2</sup> Non-static weaves and the use of aspect instances fall out of the scope of this position paper.



**Fig. 1.** Range of AOP languages' features in some of our work. RIDL is the remote data transfer language in [5]; COOL is the synchronization language in [5]; RG is the image processing system in [6]; AML is the sparse matrix language in [1]; “talk” refers to the language in the slides of the invited talk [2]

## 4 Relation to Previous Work

Having designed a number of different AOP extensions for different purposes, we summarize our work in Fig. 1. The figure presents a two-dimensional space for language design (not specific for AOP). The horizontal axis indicates to what extent the language abstracts away from low-level implementation issues. The vertical axis indicates how specific the language constructs are to a particular domain or problem. The current version of AspectJ, summarized in this position paper, is more general-purpose and more low-level than our previous work.

## References

1. Irwin J., Loingtier J.-M., Gilbert J. et al. *Aspect-Oriented Programming of Sparse Matrix Code*. In proc. *International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE)*, Marina del Rey, USA, 1997.
2. Kiczales G. *AOP: Going beyond objects for better separation of concerns in design and implementation*. Xerox PARC, Slides of invited talk, <http://www.parc.xerox.com/aop/invited-talk/>
3. Kiczales G., Lamping J., Mendhekar A. et al. *Aspect-Oriented Programming*. In proc. *European Conference on Object-Oriented Programming*, Finland, 1997.
4. Lopes C. and Kiczales G. *Aspect-Oriented Programming with AspectJ*. Xerox PARC, Tutorial, <http://www.parc.xerox.com/aop/aspectj/tutorial>
5. Lopes C. V. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Boston, 1997.
6. Mendhekar A., Kiczales G. and Lamping J. *RG: A Case-Study for Aspect-Oriented Programming*. Xerox PARC, Palo Alto, CA. Technical report SPL97-009 P9710044, February 1997.