An overview of AspectJ

6 authors, including:

Gregor Kiczales

University of British Columbia - Vancouver

145 PUBLICATIONS   **23,093** CITATIONS

SEE PROFILE

Mik Kersten

Tasktop

40 PUBLICATIONS   **5,426** CITATIONS

SEE PROFILE

William G. Griswold

University of California, San Diego

253 PUBLICATIONS   **13,533** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project   On DevOps Column for IEEE Software View project

Project   PhD Thesis View project

An Overview of AspectJ

Gregor Kiczales[1], Erik Hilsdale[2], Jim Hugunin[2], Mik Kersten[2],
Jeffrey Palm[2] and William G. Griswold[3]

[1] Department of Computer Science, University of British Columbia,
201-2366 Main Mall, Vancouver, BC V6T 1Z4 Canada
`gregor@cs.ubc.ca`
[2] Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto, CA 94304 USA
`(hilsdale, hugunin, mkersten, palm)@parc.xerox.com`
[3] Department of Computer Science and Engineering, University of California, San Diego
La Jolla, CA 92093 USA
`wgg@cs.ucsd.edu`

Abstract. AspectJ™ is a simple and practical aspect-oriented extension to Java™. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns. In AspectJ's dynamic join point model, join points are well-defined points in the execution of the program; pointcuts are collections of join points; advice are special method-like constructs that can be attached to pointcuts; and aspects are modular units of crosscutting implementation, comprising pointcuts, advice, and ordinary Java member declarations. AspectJ code is compiled into standard Java bytecode. Simple extensions to existing Java development environments make it possible to browse the crosscutting structure of aspects in the same kind of way as one browses the inheritance structure of classes. Several examples show that AspectJ is powerful, and that programs written using it are easy to understand.

1   Introduction

Aspect-oriented programming (AOP) [14] has been proposed as a technique for improving separation of concerns in software.[1] AOP builds on previous technologies, including procedural programming and object-oriented programming, that have already made significant improvements in software modularity.

The central idea of AOP is that while the hierarchical modularity mechanisms of object-oriented languages are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. Instead, we believe that in the

---

[1] When we say "separation of concerns" we mean the idea that it should be possible to work with the design or implementation of a system in the natural units of concern – concept, goal, team structure etc. – rather than in units imposed on us by the tools we are using. We would like the modularity of a system to reflect the way "we want to think about it" rather than the way the language or other tools force us to think about it. In software, Parnas is generally credited with this idea [29, 30].

implementation of any complex system, there will be concerns that inherently crosscut the natural modularity of the rest of the implementation.

AOP does for *crosscutting concerns* what OOP has done for object encapsulation and inheritance—it provides language mechanisms that explicitly capture crosscutting structure. This makes it possible to program crosscutting concerns in a modular way, and achieve the usual benefits of improved modularity: simpler code that is easier to develop and maintain, and that has greater potential for reuse. We call a well-modularized crosscutting concern an *aspect*.[2] An example of how such an aspect crosscuts classes is shown in Figure 1.

AspectJ is a simple and practical aspect-oriented extension to Java. This paper presents an overview of AspectJ, including a number of core language features, the basic compilation strategy, the development environment support, and several examples of how AspectJ can be used.[3] The examples show that using AspectJ we can code, in clear form, crosscutting concerns that would otherwise lead to tangled code.

The main elements of the language design are now fairly stable, but the AspectJ project is not nearly finished. We continue fine-tuning parts of the language, building a third-generation compiler, expanding the integrated development environment (IDE) support, extending the documentation and training material, and building up the user community. We plan to work with that user community to empirically study the practical value of AOP.

The next section describes the basic assumptions behind the AspectJ language design. Section 3 presents the core language. Section 4 outlines the compiler. Section 5 describes the AspectJ-aware tool extensions we have developed. Section 6 shows that AspectJ can capture crosscutting structure in elegant and easy to understand ways. We conclude with a discussion of related and future work. As an overview, detailed design rationale and detailed compiler and tool implementation issues are outside the scope of this paper.

2   Basic Design Assumptions

AspectJ is the basis for an empirical assessment of aspect-oriented programming. We want to know what happens when a real user community uses an AOP language. What kinds of aspects do they write? Can they understand each other's code? What kinds of idioms and patterns emerge? What kinds of style guidelines do they develop? How effectively can they work with crosscutting modularity? And, above all, do they develop code that is more modular, more reusable, and easier to develop and maintain?

Because these are our goals, designing and implementing AspectJ is really just part of the project. We must also develop and support a substantial user community. To

---

[2] AOP support can be added to languages that are not object-oriented. The key property of an AOP language is that it provides crosscutting modularity mechanisms. So when we add AOP to an OO language, we add constructs that crosscut the hierarchical modularity of OO programs. If we add AOP to a procedural language, we must add constructs that crosscut the block structure of procedural programs [4, 6].

[3] This paper is written to correspond with AspectJ version 0.8.

make this possible, we have chosen to design AspectJ as a *compatible* extension to Java so that it will facilitate adoption by current Java programmers. By compatible we mean four things:

- *Upward compatibility* — all legal Java programs must be legal AspectJ programs.
- *Platform compatibility* — all legal AspectJ programs must run on standard Java virtual machines.
- *Tool compatibility* — it must be possible to extend existing tools to support AspectJ in a natural way; this includes IDEs, documentation tools, and design tools.
- *Programmer compatibility* — Programming with AspectJ must feel like a natural extension of programming with Java.

The programmer compatibility goal has been responsible for much of the feel of the language. Whereas our previous AOP languages were domain-specific, AspectJ is a general-purpose language like Java. AspectJ also has a more Java-like balance between declarative and imperative constructs. AspectJ is statically typed, and uses Java's static type system. In AspectJ programs we use classes for traditional class-like modularity structure, and then use aspects for concerns that crosscut the class structure.

There are several potentially valuable AOP research goals that AspectJ is not intended to meet. It is not intended to be a "clean-room" incarnation of AOP ideas, a formal AOP calculus or an aggressive effort to explore the AOP language space. Instead, AspectJ is intended to be a practical AOP language that provides, in a Java compatible package, a solid and well-worked-out set of AOP features.

3   The Language

AspectJ extends Java with support for two kinds of crosscutting implementation. The first makes it possible to define additional implementation to run at certain well-defined points in the execution of the program. We call this the *dynamic crosscutting* mechanism. The second makes it possible to define new operations on existing types. We call this *static crosscutting* because it affects the static type signature of the program. This paper only presents dynamic crosscutting.

Dynamic crosscutting in AspectJ is based on a small but powerful set of constructs. *Join points* are well-defined points in the execution of the program; *pointcuts* are a means of referring to collections of join points and certain values at those join points; *advice* are method-like constructs used to define additional behavior at join points; and *aspects* are units of modular crosscutting implementation, composed of pointcuts, advice, and ordinary Java member declarations.

This section of the paper presents the main elements of the dynamic crosscutting support in the language. The presentation is informal and example-based.
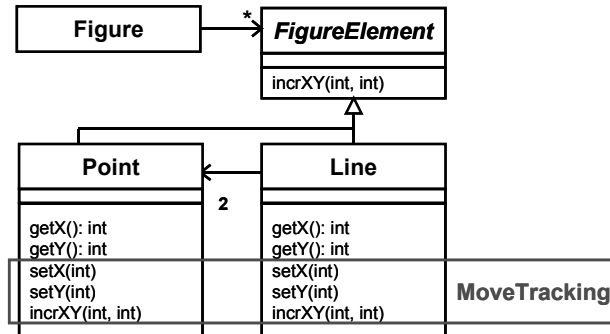
Fig. 1. UML description of a simple figure editor. The box labeled "MoveTracking" shows an aspect that crosscuts methods in the Point and Line classes. This aspect is discussed in detail in Section 3

3.1 Join Point Model

The join point model is a critical element in the design of any aspect-oriented language mechanism. This model provides the common frame of reference that makes it possible for execution of a program's aspect and non-aspect code to be properly coordinated.

In previous work, we have used several different kinds of join point model, including primitive application nodes in a dataflow graph [13, 18, 23, 27] and method bodies [27, 28]. Early versions of AspectJ used a model in which the join points were principled places in the source code.

The dynamic crosscutting elements of AspectJ are now based on a model in which join points are certain well-defined points in the execution of the program. This model gives us important additional expressive power, discussed in Section 3.9. In this model join points can be considered as nodes in a simple runtime object call graph. These nodes include points at which an object receives a method call and points at which a field of an object is referenced. The edges are control flow relations between the nodes. In this model control passes through each join point twice, once on the way in to the sub-computation rooted at the join point, and once on the way back out.

We illustrate join points using a simple figure editor, the kernel of which is shown in Figure 1, and which also serves as a running example. (Complete code from the paper is at aspectj.org/doc/papers/ecoop2001.) Based on these classes, executing the first three lines of code in Figure 2 builds the objects shown below. In this picture large circles represent objects, square boxes represent methods and small numbered circles represent join points. Executing the last line starts a computation that proceeds through the join points labeled below. In each case the join point is first reached just before the action described begins executing. Control passes back through the join point when the action described returns.

```
Point pt1 = new Point(0, 0);
Point pt2 = new Point(4, 4);
Line  ln1 = new Line(pt1, pt2);

ln1.incrXY(3, 6);
```
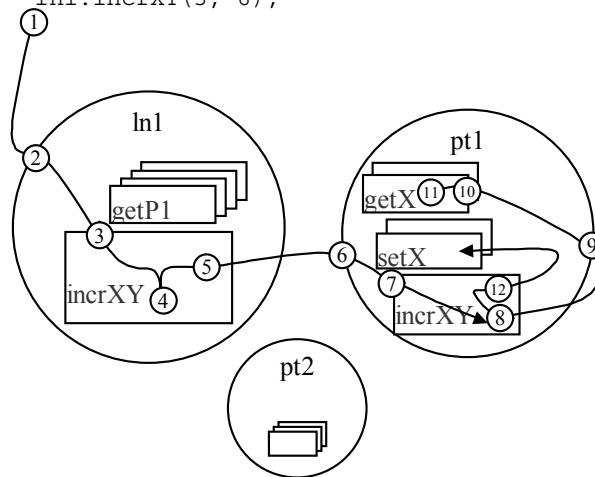

Fig. 2. The first three lines of code build the objects shown as large circles below. The rectangles represent methods.Executing the final line of code starts a computation that includes the sequence of join points shown as small numbered circles.

1. A method call join point corresponding to the `incrXY` method being called on the object `ln1`.
2. A method call reception join point at which `ln1` receives the `incrXY` call.
3. A method execution join point at which the particular `incrXY` method defined in the class `Line` begins executing.
4. A field get join point where the `_p1` field of `ln1` is read.
5. A method call join point at which the `incrXY` method is called on the object `pt1`.
   ...
8. A method call join point at which the `getX` method is called on the object `pt1`.
   ...
11. A field get join point where the `_x` field of point `pt1` is read.

   Control returns back through join points 11, 10, 9 and 8.
12. A method call join point at which the `setX` method is called on `p1`.
   … and so on, until control finally returns back through 3, 2 and 1

The different kinds of join points provided by AspectJ are shown in Table 1. Note that while AspectJ defines a number of kinds of join points, only a few kinds suffice for many programs, and all kinds of join points behave the same with respect to other

Table 1. The dynamic join points of AspectJ. Join points marked with * are not used in further examples in this paper.

kind of join point	points in the program execution at which…
method call constructor call*	a method (or a constructor of a class) is called. Call join points are in the calling object, or in no object if the call is from a static method.
method call reception constructor call reception	an object receives a method or constructor call. Reception join points are before method or constructor dispatch, i.e. they happen inside the called object, at a point in the control flow after control has been transferred to the called object, but before any particular method/constructor has been called.
method execution* constructor execution*	an individual method or constructor is invoked.
field get	a field of an object, class or interface is read.
field set	a field of an object or class is set.
exception handler execution*	an exception handler is invoked.
class initialization*	static initializers for a class, if any, are run.
object initialization*	when the dynamic initializers for a class, if any, are run during object creation.

language features.[4] This substantially reduces the complexity of learning to program with AspectJ.

3.2 Pointcut Designators

A *pointcut* is a set of join points, plus, optionally, some of the values in the execution context of those join points. AspectJ includes several primitive *pointcut designators*. Programmers can compose these to define anonymous or named user-defined pointcut designators. Pointcuts are not higher order, nor are pointcut designators parametric.

---

[4] There is one exception to this rule. It is not possible to define `before` or `around` advice on constructor reception or constructor execution join points.

A simple way to think of pointcut designators is in terms of matching certain join points at runtime. For example, the pointcut designator

```
receptions(void Point.setX(int))
```

matches all method call reception join points at which the Java signature of the method call is `void Point.setX(int)` Intuitively, this refers to every time a point receives a call to change its `x` coordinate. Similarly

```
receptions(void FigureElement.incrXY(int, int))
```

intuitively refers to every time any kind of figure element (i.e. an instance of `Point` or `Line`) receives a call to shift a certain distance.

Pointcuts can be combined using and, or and not operators ('`&&`', '`||`' and '`!`'). The following compound pointcut designator refers to all receptions of calls to a `Point` to change its `x` or `y` coordinate.

```
receptions(void Point.setX(int)) ||
receptions(void Point.setY(int))
```

Primitive Pointcut Designators

AspectJ includes a variety of primitive pointcut designators that identify join points in different ways. Some primitive pointcut designators only identify pointcuts of one kind, for example `receptions` only matches method call reception join points. Others match any kind of join points at which a certain property holds. For example, `instanceof(Point)` matches all join points at which the currently executing object (the value of `this`) is an instance of `Point` or a subclass of `Point`.

These two kinds of join point designators can be combined to identify join points in useful ways. For example:

```
!instanceof(FigureElement) &&
calls(void FigureElement.incrXY(int, int))
```

matches all method calls to `incrXY` that do not come from an object that is a figure element. This will mean calls that come from an object of another type, as well as calls that come from static methods.[5]

The primitive pointcut designators are summarized in Table 2. They are explained further as they are used in the paper.

---

[5] Because there is no currently executing object in static methods, the `instanceof(FigureElement)` will not match such join points.

Table 2. Primitive pointcut designators. Any …*TypeName* position does normal sub-type matching. Any …*id* position does matching by string equality. See section 3.9 for information about more sophisticated wild card matching in these positions

`calls(`*signature*`)`
`receptions(`*signature*`)`
`executions(`*signature*`)`
 	
Matches call/reception/execution join points at which the method or constructor called matches *signature*. The syntax of a method signature is:
 	
  `ResultTypeName RecvrTypeName.meth_id(`*ParamTypeName*`, …)`
 	
The syntax of a constructor signature is:
 	
  `NewObjectTypeName.new(`*ParamTypeName*`, …)`

`gets(`*signature*`)`
`gets(`*signature*`)[`*val*`]`
`sets(`*signature*`)`
`sets(`*signature*`)[`*oldVal*`]`
`sets(`*signature*`)[`*oldVal*`][`*newVal*`]`
 	
Matches field get/set join points at which the field accessed matches the signature. The syntax of a field signature is:
 	
  `FieldTypeName ObjectTypeName.field_id`

`handles(`*ThrowableTypeName*`)` Matches exception handler execution join points of the specified type.

`instanceof(`*CurrentlyExecutingObjectTypeName*`)`
`within(`*ClassName*`)`
`withincode(`*signature*`)`
 	
Matches join points of any kind at which the currently executing:
   - object is of type `CurrentlyExecutingObjectTypeName`
   - code is contained within `ClassName`
   - code is contained within the member defined by the method or constructor
signature

`cflow(`*pointcut_designator*`)` Matches join points of any kind that occur strictly within the dynamic extent of any join point matched by pointcut_designator.
`callto(`*pointcut_designator*`)` Matches method call join points that in one step lead to any reception or execution join points matched by pointcut_designator.

`staticinitializations(TypeName)`
`initializations(TypeName)`
 	
Matches class or object initializations of the specified type.

User-defined Pointcut Designators.

User-defined pointcut designators are defined with the `pointcut` declaration. The declaration:

```
pointcut moves():
  receptions(void FigureElement.incrXY(int, int)) ||
  receptions(void Line.setP1(Point))              ||
  receptions(void Line.setP2(Point))              ||
  receptions(void Point.setX(int))                ||
  receptions(void Point.setY(int));
```

defines a new pointcut designator, `moves()`, that identifies whenever a figure element receives a call of a method that can move it. User-defined pointcut designators can be used wherever a pointcut designator can appear.

3.3 Advice

Advice is a method-like mechanism used to declare that certain code should execute at each of the join points in a pointcut. AspectJ supports *before*, *after,* and *around* advice. Additionally, there are two special cases of after advice, *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point. Both before advice and all three kinds of after advice are strictly additive with respect to the normal computation at the join point. Around advice has the special capability of selectively preempting the normal computation at the join point. This advice framework is based on the declarative method combination mechanism in CLOS [11] (which itself was modeled on the demon methods of Flavors [8, 10, 31]).

Advice declarations define advice by associating a code body with a pointcut, and a time, relative to each join point in the pointcut, when the code should be executed. The advice declaration

```
after(): moves() {
  flag = true;
}
```

defines after advice on the pointcut `moves()`. The '`()`' between '`after`' and the '`:`' means the advice has no parameters. The effect of this declaration is to ensure that the `flag` variable is set to true whenever a figure element finishes handling a move method call. (The declaration of the variable is shown in the example in Section 3.4.)

A simple model for the behavior of advice is in terms of runtime dispatch. (Section 4 outlines the techniques the compiler uses to ensure that most if not all of the matching overhead happens at compile time.) Upon arrival at a join point, all advice in the system are examined to see whether any apply at the join point. Any that do are collected, ordered according to specificity (described in Section 3.5), and executed as follows:

1. First, any `around` advice are run, most-specific first. Within the body of an around advice, calling `proceed()` invokes the next most specific piece of `around` advice, or, if no `around` advice remain, goes to the next step.
2. Then all `before` advice are run, most-specific first.
3. Then the computation associated with the join point proceeds.
4. Execution of `after returning` and `after throwing` advice depends on how the computation in step 3 and prior `after returning` and `after throwing` advice terminate.
   - If they terminate normally, all `after returning` advice are run, least specific first.
   - If they terminate by throwing an exception, all `after throwing` advice that match the exception are run, least specific first. (This means `after throwing` advice can handle exceptions thrown by less specific `after returning` and `after throwing` advice.)
5. Then all `after` advice are run, least-specific first.
6. Once all `after` advice have run, the return value from step 3, if any, is returned to the innermost call to proceed from step 1, and that piece of `around` advice continues running.
7. When the innermost piece of `around` advice returns, it returns to the surrounding `around` advice.
8. When the outermost piece of `around` advice returns, control continues back from the join point.

3.4 Aspects

Aspects are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a form similar to that of class declarations. Aspect declarations may include pointcut declarations, advice declarations, as well as all other kinds of declarations permitted in class declarations.

The following declaration defines an aspect that implements the behavior of keeping track of whether a figure element has moved recently. This aspect might be used by the screen update mechanism to find out whether anything has changed since the last time the screen was updated. (More sophisticated versions of this aspect will be presented as the paper proceeds.)

```
aspect MoveTracking {

  static boolean flag = false;

  static boolean testAndClear() {
    boolean result = flag;
    flag = false;
    return result;
  }
```

```
pointcut moves():
  receptions(void FigureElement.incrXY(int, int))  ||
  receptions(void Line.setP1(Point))                ||
  receptions(void Line.setP2(Point))                ||
  receptions(void Point.setX(int))                  ||
  receptions(void Point.setY(int));

after(): moves() {
  flag = true;
}
}
```

Advice of an aspect are similar to methods in that they have access to all members of the class. So in this case the after advice can reference the static variable `flag`.

Aspect Instances

In AspectJ, the default behavior of non-abstract aspects is to have a single instance. Advice run in the context of this instance. The `aspect` declaration accepts a modifier, called 'of' that provides other kinds of aspect instance behavior. Discussion of this functionality is outside the scope of this paper.

3.5 Aspect Precedence

In general, more than one piece of advice may apply at a join point. The different advice can come from different aspects or even the same aspect. The relative order in which such advice executes is well defined. The ordering is based on the fact that aspects are the primary units of crosscutting functionality. So advice ordering, or specificity, is resolved with respect to the relative precedence of the aspects in which the advice is defined.

For two pieces of advice, a_1 and a_2, defined in aspects A_1 and A_2 respectively, the relative specificity is determined as follows:

- If A_1 and A_2 are the same, whichever piece of advice appears first in that aspect declaration's body is more specific. This rule exists because one aspect may need to define multiple advice that apply at the same join point. This commonly happens when there are matching before and after advice, but it can also happen with two pieces of advice of the same kind.
- If A_1 directly or indirectly extends A_2, then a_1 is more specific than a_2. This rule is a natural extension of method overriding rules in OO languages. It supports the common case where the related advice are defined in aspects that naturally exist in an extends relationship. (Section 3.8 discusses aspect inheritance and overriding in more detail.)
- If the declaration of A_1 includes a `dominates` modifier that mentions A_2, then a_1 is more specific than a_2. This rule exists because, in some cases, the programmer needs to control precedence between aspects that do not exist in an extends relationship.

- In all other cases the relative specificity between a_1 and a_2 is undefined. This is the most common case – two conceptually and semantically independent aspects define advice that apply at the same join point – and the programmer does not need to control the relative ordering of such advice.

The following mobility aspect is an example of the use of the `dominates` modifier. This simple aspect implements a global flag that freezes all figure elements so that they cannot move. The aspect works by checking the flag before any move operation, and simply doing a "quiet abort" of the operation if moves are disabled.

```
aspect Mobility dominates MoveTracking {
  private static boolean enableMoves = true;

  static void enableMoves()  { enableMoves = true;  }
  static void disableMoves() { enableMoves = false; }

  around() returns void: MoveTracking.moves() {
    if ( enableMoves ) {
      proceed();
    }
  }
}
```

It would not make sense for this aspect to extend `MoveTracking`, because it doesn't define a more specialized version of the move tracking functionality. But it is essential that it have precedence over `MoveTracking`, so that it can abort a move before it gets registered. Note that the code for this aspect shows that one aspect can refer to a pointcut defined in another aspect in the same way that static fields are referred to in Java.

3.6 Pointcut Parameters

In many cases it is useful for advice to have access to certain values that are in the execution context of the join points. For example, a more sophisticated version of the move tracking aspect might record the specific figure elements that have moved recently rather than just a single bit saying that some figure element has moved recently.

AspectJ provides a parameter mechanism that makes it possible for advice to see a subset of the values in the execution context of join points. This mechanism operates in both advice and pointcut declarations. In advice declarations values can be passed from the pointcut designator to the advice. In pointcut declarations values can be passed from the constituent pointcut designators to the user-defined pointcut designator. In both cases, the flow of values is from the right of the ':' to the left. The net effect is that values made available by primitive pointcut designators can be used in the body of advice.

For example, the following piece of advice has access to both the object receiving the method call and the argument to that call:

```
    before(Point p, int nval):
        receptions(void p.setX(nval)) {
      System.out.println("x value of " + p +
                           " will be set to " + nval + ".");
    }
```

The parameter mechanism uses a combination of positional and by-name matching. The list of parameters to the left of the ':' declares that this piece of advice has two parameters, of type `Point` and `int`, named `p` and `nval` respectively. Then, to the right of the colon, those two parameter names can be used in the same position that a type name would normally appear to say that the parameter should get the corresponding value. So, the `p` and `nval` in `p.setX(nval)` mean that the effective signature is `Point.setX(int)`, and that `p` should get the object receiving the call, and `nval` should get the value of the first argument to the call.

Definition and use of parameters works in a similar way in user-defined pointcuts. In this code

```
    pointcut incrXYs(FigureElement fe):
        receptions(void fe.incrXY(int, int));

    after(FigureElement figElt): incrXYs(figElt) {
        <'figElt' is bound to the figure element here>
    }
```

the pointcut declaration says that `incrXYs(FigureElement)` exposes a single parameter, of type `FigureElement`, and that it is the receiver of the `incrXY` method call. The advice declaration says that `figElt` should be bound to the first parameter of `incrXYs`, which is the figure element being moved. Note that the name of the parameter in the pointcut declaration does not have to be the same as within the advice declaration.

Values can be exposed from other primitive pointcut designators as well. A common case is to use `instanceof` with a parameter to provide access to the object making a call, as follows:

```
    pointcut gets(Object caller):
        instanceof(caller) &&
        (calls(int Point.getX())     ||
         calls(int Point.getY())     ||
         calls(Point Line.getP1())   ||
         calls(Point Line.getP2()));
```

The primitive pointcut designators expose values as suggested by the naming convention in Table 2. The `RecvrTypeName` position in method signatures exposes the object receiving the method call and so on.

Static Typing of Receiver
In a *highly polymorphic* pointcut designator like `moves`, there is no common super type that accepts all of the method calls in the pointcut (i.e. there is no type that accepts all of `incrXY`, `setP1`, `setP2`, `setX` and `setY`). That means it isn't possible to write `moves` to expose the figure element that is moving by simply

plugging a common parameter into the receiver position of each receptions. One cannot write something like:

```
pointcut moves(FigureElement fe):
    receptions(void fe.incrXY(int, int)) ||
    receptions(void fe.setP1(Point))     ||
    receptions(void fe.setP2(Point))     ||
    ...
```

because `setP1` is not defined on `FigureElement`. Instead, the object receiving the calls must be picked up using `instanceof` as follows:

```
pointcut moves(FigureElement fe):
    instanceof(fe) &&
    (receptions(void FigureElement.incrXY(int, int)) ||
     receptions(void Line.setP1(Point))              ||
     receptions(void Line.setP2(Point))              ||
     receptions(void Point.setX(int))                ||
     receptions(void Point.setY(int)));
```

Access to Return Values

In some cases, `after returning` advice may want to access the value being returned through the join point. This is done with special syntax, to make it clear that the return value is only present in after returning advice:

```
after(Point p) returning (int x):
    receptions(int p.getX()) {
  System.out.println(p + " returned " +
                     x + " from getX().");
}
```

Parameters and proceed

Within an `around` advice that has parameters, `proceed` accepts parameters with the same signature as the `around` advice itself. Calling `proceed` with different actual values for those parameters will cause all remaining advice and the rest of the computation to see the new values. This can be used to implement advice that does pre-processing on the values as follows:

```
around(int nv) returns void:
    receptions(void Point.setX(nv)) ||
    receptions(void Point.setY(nv)) {
  proceed(Math.max(0, nv));
}
```

The effect of this advice is to ensure that any method call to change the x or y coordinate of a point has its parameter clipped to greater than zero before the change proceeds.

3.7 Reflective Access to Join Point

To make certain kinds of advice easier to write, AspectJ provides simple reflective access to information about the current join point. Within the body of an advice declaration, the special variable `thisJoinPoint` is bound to an object representing the current join point. The join point object provides information common to all join points, such as what kind of join point it is and the signature of the surrounding method. It also provides information specific to each kind of join point, i.e. a field reference join point provides access to the field signature.

3.8 Inheritance and Overriding of Advice and Pointcuts

To support aspect-libraries, AspectJ provides a simple mechanism of pointcut overriding and advice inheritance. To use this mechanism a programmer defines an *abstract aspect*, with one or more *abstract pointcuts*, and with advice on the pointcut(s). This, then, is a kind of library aspect that can be parameterized by aspects that extend it. For example, the following defines a simple library of tracing functionality.

```
abstract aspect SimpleTracing {

  abstract pointcut tracePoints();

  before(): tracePoints() {
    printMessage("Entering", thisJoinPoint);
  }
  after(): tracePoints() {
    printMessage("Exiting", thisJoinPoint);
  }

  void printMessage(String when, JoinPoint tjp) {
    code to print an informative message
    using information from the join point
  }
}
```

Using the library aspect in a specific situation just requires extending the aspect and supplying a concrete definition for the abstract pointcut.

```
aspect IncrXYTracing extends SimpleTracing {

  pointcut tracePoints():
      receptions(void FigureElement.incrXY(int, int));

}
```

Making the abstract pointcut concrete in the sub-aspect has the effect of inheriting the advice declaration from the super-aspect into the sub-aspect. If the sub-aspect includes a `dominates` modifier, that modifier affects the precedence of the inherited advice.

3.9 Property-Based Crosscutting

The pointcuts presented above are all defined in terms of an explicit enumeration of method signatures. Although this is appropriate in many cases, we have found that it is useful to be able to define a pointcut by means of certain other properties of join points. To enable such property-based crosscutting, AspectJ includes two kinds of features, wildcarding in pointcut designators and control-flow based pointcut designators.

Wildcarding in Pointcut Designators. AspectJ includes a very simple wildcarding mechanism in pointcut designators. Examples of what this mechanism allows the programmer to say are:

- `receptions(* Point.*(..))`
  matches receptions of calls to any method defined on the class `Point` (i.e. `incrXY(int, int)`, `getX()`, `getY()`, `setX(int)`, `setY(int)`).[6]
- `receptions(Point.new(..))`
  matches receptions of calls to any constructor for an object of type `Point` (i.e. the `Point(int, int)` constructor).
- `receptions(public * com.xerox.scanner.*.*(..))`
  matches receptions of calls to any public method defined on any type in the `com.xerox.scanner` package.
- `receptions(* Point.get*())`
  matches receptions of calls to any method defined on `Point` for which the the id starts with `get` and which accepts zero arguments – i.e. the nullary getters `getX()` and `getY()`

Control-Flow Based Crosscutting. AspectJ also includes two primitive pointcut designators that allow picking out join points based on whether they are in a particular control-flow relationship with other join points. In order to do this, these designators differ from others in that they accept pointcut designators as parameters.

The `cflow(pcd)` pointcut designator matches all join points that are strictly within the dynamic extent of the join points matched by *pcd*. The points matched by *pcd* itself are not matched by `cflow(pcd)`. A canonical use of `cflow` is to distinguish between top-level versus recursive calls of a method. So, for example,

```
pointcut moves(FigureElement fe): <as above>;

pointcut topLevelMoves(FigureElement fe):
    moves(fe) && !cflow(moves(FigureElement));
```

---

[6] This will also match calls to methods defined in the class `Object`. If the programmer explicitly wants to exclude these they could write: `receptions(* Point.*(..)) && !receptions(* Object.*(..))`.

The definition of `topLevelMoves` reads as any join point matched by `moves`, but not within the control flow of `moves`. In other words, if the move operation invokes another move operation recursively, that recursive operation will not be matched.

4 Implementation

This section briefly outlines the current language implementation.

The main work of any AOP language implementation is to ensure that aspect and non-aspect code run together in a properly coordinated fashion. This coordination process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. As is the case with most other language features, aspect weaving can be done by a special pre-processor [7], during compilation, by a post-compile processor [14], at load time, as part of the virtual machine, using residual runtime instructions, or using some combination of these approaches.

The AspectJ language design strives to be silent on the issue of when aspect weaving should be done. We provide a compiler-based implementation of the language that does almost all weaving work at compile-time. This exposes as many programming errors as possible at compile time and avoids unnecessary runtime overhead. Certain special cases of advice involve residual dispatch overhead at runtime.

The compiler uses a pay-as-you-go implementation strategy. Any parts of the program that are unaffected by advice are compiled just as they would be by a standard Java compiler.

The compiler transforms the source program in three ways: the body of every advice declaration is compiled into a standard method, parts of the program where advice applies are transformed to insert static points corresponding to the dynamic join points, and code to implement any residual dynamic dispatch is inserted at those static points.

4.1 Compilation of Advice Bodies

Every before or after advice body is compiled into a standard method and the advice is run by a call to the method from appropriate points in the code. This potentially means that the use of advice will add the overhead of a single method-call. But, these methods are always either static or final, so they can easily be inlined by most JVMs [11]. This means there should generally be no observable performance overhead from these additional method calls.

An around advice body is compiled into one method body for each corresponding static point in the code. This allows us to pass the needed state for the around efficiently on the call-stack and to implement the `proceed` statement without needing to use Java's reflection mechanisms. This implementation strategy trades an increase in bytecode size for significantly reduced runtime overhead.

4.2 Corresponding Method

The compiler transforms 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. This transformation is only performed for join points that might have advice, not all join points. So, for example, in a program that has advice on the pointcut designated by `gets(int Point._x)`, the compiler would transform references of the form `p._x`, where p is a `Point`, to `Point.$jp$0$(p)`, and add the following method to the class `Point`:

```
private static int $jp$0(Point obj) {
  return obj._x;
}
```

Once this corresponding method has been generated, `before` and `after` advice are implemented by making the corresponding method call the advice methods, as needed.

There are many cases, including this one, where the compiler will add additional method calls in order to create corresponding methods. This happens for method call, method call reception and field access join points. Extra method calls are also added as part of the implementation strategy for around advice. The overhead of these methods is small in any JVM, and again since they are all static or final, they will be optimized away by good JVMs. We expect a future version of the AspectJ compiler to provide optimizing modes that will eliminate some of these minor overheads.

4.3 Dynamic Dispatch

The use of certain pointcut designators, like `cflow`, `callsto`, and `instanceof`, can require a run-time test to determine whether a particular corresponding method actually matches a particular join point designator. In such cases, the corresponding method includes residual testing code that guards the execution of the advice. This overhead is relatively small.

5   Tool Support
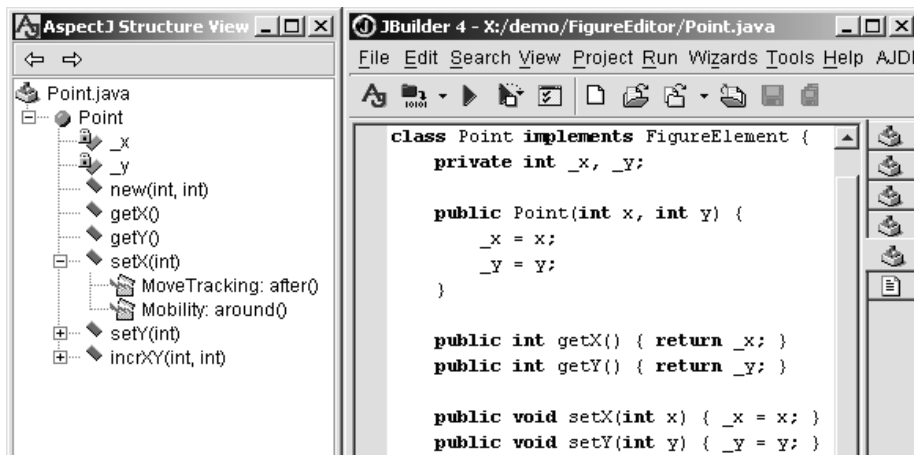
Fig. 3. A portion of the screen when using the AJDE extension to JBuilder 4. The main window on the right shows the code for the class `Point`. The structure view on the left shows the class `Point`, and shows that `setX`, `setY` and `incrXY` are all crosscut by advice; `setX` is further expanded to show what advice crosscuts it. The user can click on the advice to jump there.

In object-oriented programming, development tools typically allow the programmer to easily browse the class structure of their programs. Such support enables the programmer to see the inheritance and overriding structure in their program, as well as seeing compact representations of the contents of individual classes [13, 18, 23].

For AspectJ, we are developing analogous support for browsing aspect structure. This enables the programmer to see the crosscutting structure in their programs. It works by showing a bi-directional coupling between aspects (and their advice), and the classes (and their members) that the advice affect. Figure 3 shows one of the extensions we have made to JBuilder 4. This extension to the structure view tool allows the programmer to easily see a summary of all the crosscutting affecting the class Point. If the structure view window is focused on an aspect, it will show all the targets of that aspect's advice.

A second kind of environment extension provides a more light-weight reminder of the aspect structure. This extension works by annotating the source code, as seen in the editor, with an indication of whether aspects crosscut that code. Figure 4 shows how we have extended emacs with this functionality. The automatically generated annotations name the aspects that crosscut the method. A keystroke command can be used to pop up a menu of the advice, choose one, and jump to it.

We currently support AspectJ-aware extensions to emacs, JBuilder and Forte for Java. Additional tool support includes debugger extensions to understand that advice, display it correctly on the stack etc., as well as extensions to Javadoc [17] to make it understand crosscutting structure and generate appropriate hyper-links etc.
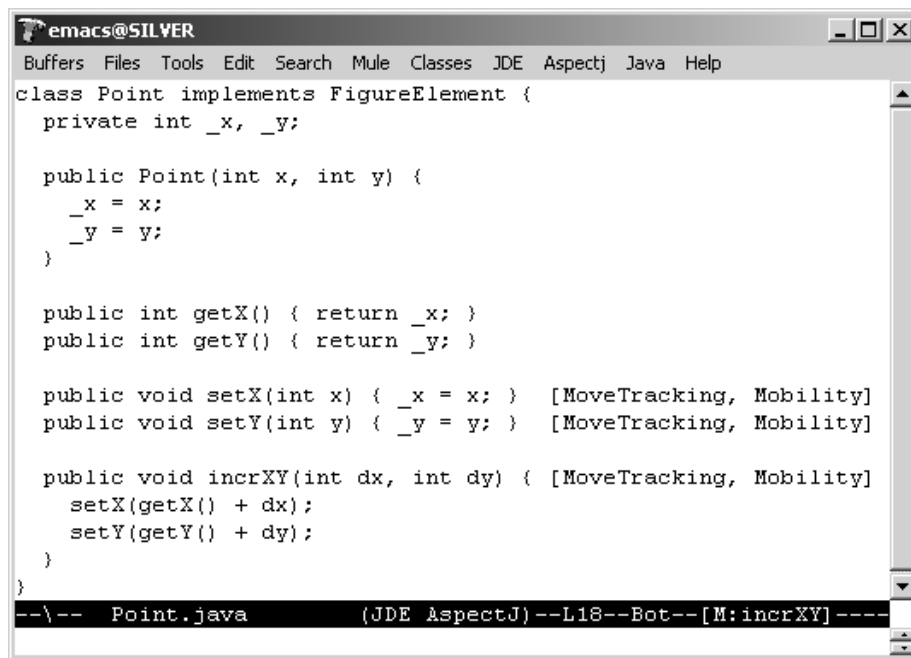
Fig. 4. A portion of the screen when using the AspectJ-aware extension to emacs. The text in [Square Brackets] following the method declarations is automatically generated, and serves to remind the programmer of the aspects that crosscut the method.

All of these extensions work by consulting a database that is maintained by the compiler. Once the API stabilizes, we intend to make it public so that others can develop tools that use it as well.

6 Understanding Crosscutting Structure

One of the most important questions we must answer is how easy is it to program with AspectJ. In particular, is crosscutting structure, implemented with AspectJ, something that appears easy to understand and work with? We do not yet have enough experience to say for sure, but our experience to date suggests that the answer is yes.

6.1 Modular, Concise and Explicit

Consider the following simple aspect, which is not part of the figure editor example. This aspect implements a simple error logging functionality, in which every public method defined on any type in the `com.xerox.printers` package logs any errors it throws back to its caller.

```
aspect SimpleErrorLogging {
  Log log = new Log();

  pointcut publicEntries():
      receptions(public * com.xerox.printers.*.*(..));

  after() throwing (Error e): publicEntries() {
    log.write(e);
  }
}
```
This aspect appears to be better than the plain Java implementation of the same functionality in several ways:

- The aspect is *more modular*. In the ordinary Java implementation, code for this aspect would be spread across every public method.
- The aspect is *more concise*. In the plain Java version something like six lines of code would be added to each public method to wrap the body in a "try… catch…" statement.
- The aspect is *more explicit*. In this code, the *structural invariant* underlying the crosscutting is clear. A quick look at the code is all it takes to understand that all the public methods defined in the com.xerox.printers package should do error logging.

Consider a program maintenance scenario in which a programmer must work with a system with this kind of functionality. In the plain Java implementation, the programmer would discover the logging code one method at a time. After seeing several such methods the programmer might guess that logging was being done by all public methods of that class or perhaps even the package. But they would have to make careful use of a tool like grep to be sure. And of course they might not even make this guess.

In the AspectJ implementation, every public method would carry an annotation, similar to that in Figures 3 and 4, so that when the programmer look at the first public method they would see an annotation, something like that in Figures 3 and 4, which would tell them that the method was crosscut by the SimpleErrorLogging aspect. They could quickly go to the aspect, read the ten lines of code, and understand the intent of the code. The structural invariant underlying the functionality – that all public methods must log – would be clear and enforced.

Aspects that use explicit enumeration of method signatures can also be more modular, concise and explicit than their plain Java counterparts. Consider the now familiar moves pointcut:

```
pointcut moves(FigureElement fe):
    instanceof(fe) &&
    (receptions(void FigureElement.incrXY(int, int)) ||
     receptions(void Line.setP1(Point))              ||
     receptions(void Line.setP2(Point))              ||
     receptions(void Point.setX(int))                ||
     receptions(void Point.setY(int)));
```

Even though this pointcut is enumeration rather than property-based, putting the complete set of method signatures in a single place makes the crosscutting structure

explicit and clear. When reading the `MoveTracking` aspect it is easy to tell what invariant it preserves – whenever something moves it records that fact. Writing the `Mobility` aspect in terms of `MoveTracking.moves`, makes it clear that multiple aspects of the implementation crosscut all the move operations. The IDE support ensures that when we happen to be looking at the `setX` method for `Point`, we see that `Mobility` and `MoveTracking` crosscut there. Navigating to either aspect will show their structure and the fact that `Mobility` is defined in terms of MoveTracking.

This clarity is preserved when enumeration-based crosscutting is used together with property-based crosscutting. This is evident in the `topLevelMoves` pointcut.

```
pointcut topLevelMoves(FigureElement fe):
    moves(fe) && !cflow(moves(FigureElement));
```

Our experience is that the `cflow` pointcut designator takes only a short while for people learning AspectJ to learn, and once they do so, they find it quite easy to understand this code. It is certainly much easier than to understand what is going on from the middle of the classic tangled implementation of this functionality.

Clear explicit crosscutting structure can come from the way multiple advice declarations interact as well. In the `SimpleTracing` aspect of Section 3.8, there are two advice declarations:

```
before(): tracePoints() {
  printMessage("Entering", thisJoinPoint);
}
after(): tracePoints() {
  printMessage("Exiting", thisJoinPoint);
}
```

Even without knowing what join points `tracePoints` will match, we understand something important about the structure of this code – the entering and exiting messages happen in pairs, on the way into and back out of join points matched by `tracePoints`.

6.2 The Role of IDE Technology

IDE technology plays an important role in these scenarios. In the course of preparing the paper we encountered a bug in which `MoveTracking` and `Mobility` had inconsistent moves pointcuts. The bug was immediately apparent, because the environment showed numerous methods tagged with the `Mobility` and `MoveTracking` aspects and one method tagged with just `MoveTracking`.

Because it is now standard practice for OO programmers to use some kind of IDE support and because it is so easy to incorporate AspectJ support into an IDE, we believe it is reasonable to expect programmers to have IDE support available for such scenarios.

The ability of the IDE to present the structure of the program depends on the degree to which the code declaratively captures that structure. OO IDEs do a good job of presenting inheritance structure because code in OO programming languages

captures inheritance explicitly. The AspectJ IDE support works well because code in AspectJ captures crosscutting explicitly.

7 Related Work

In earlier work we proposed aspect-oriented programming [24] and presented three examples of domain-specific [1] AOP languages [5] that we had developed. AspectJ differs from those three systems in that it is a general-purpose language, it is integrated with Java, it has a dynamic join point model, and we are developing a full compiler, rather than just a pre-processor.

7.1 Other Work in AOP

Adaptive Programming [27] provides a special-purpose declarative language for writing class structure traversal specifications. Using this language prevents knowledge of the complete class structure from becoming tangled throughout the code. Adaptive Components [36] build on adaptive programming by using similar graph-language techniques to allow flexible linking of aspectual components and classes. This makes aspectual components reusable. AspectJ supports reusable aspects using the pointcut-overriding and advice-inheritance mechanism, neither of which requires a special graph language.

Composition Filters [28] wrap objects inside of filters that operate on the messages the objects receive. The filters have crosscutting access to the messages received by an object. But attachment of filters to objects is done as part of class definitions, so composition filters are less well suited than AspectJ for crosscuts that involve more than one class.

De Volder has proposed a logic meta-programming (LMP) approach that can serve as kind of an AOP language toolkit [32, 33]. In this approach, the equivalent of our pointcut designators use logical queries to specify crosscuts. This approach can take advantage of unification to define parametric pointcut designators. It supports higher-order pointcut designators as well. We have considered extending AspectJ with this kind of power, but have decided not to do so, in order to keep the language simpler and easier for Java developers to learn quickly. We may re-consider this issue in release 2.0 or later; we believe the current pointcut designator syntax leaves us room to do so in an upward compatible way.

7.2 Multi-Dimensional Separation of Concerns

Subject-oriented programming is a means for composing and integrating disparate class hierarchies (subjects), each of which might represent different concerns [9]. More recent work on multi-dimensional separation of concerns (MDSOC) [2, 19] is intended to separate concerns along multiple dimensions at once. Hyper/J [15] is a specific proposal for MDSOC. Hyper/J works by having the programmer write two kinds of meta-declarations: The first describes how to slice concerns out of a set of

classes; the second describes how to re-compose those concerns into a new program. Hyper/J has the potential to slice a concern out of code without re-factoring the classes. By comparison, in AspectJ the separation of crosscutting concerns is done in the original code, by writing it as an aspect. We believe re-factoring the code with an aspect will be easier to maintain than slicing concerns out, but it is too soon to know.

7.3 Reflection

Computational reflection [19-22, 26, 38] enables crosscutting programs. For example, it is possible to write a small piece of meta-code that runs for all methods. Smalltalk-76 included meta-level functionality [12]. CommonLoops and 3-KRS proposed different meta-level architectures for OO languages [3] PCL provided the first efficient metaobject-protocol [25]. Much of the research in reflection has explored varying the meta-level architecture to support different kinds of crosscutting [2] and to achieve flexibility without sacrificing performance [34].

With the exception of reflective access to `thisJoinPoint`, AspectJ has been designed so that the semantics of advice is not a meta-programming nor a reflective semantics. In particular, AspectJ the identifiers in pointcut designators do not refer to program representation or interpreter state – they do not involve reification.

7.4 Object-Oriented Programming

Flavors [35], New Flavors [37], CommonLoops [16] and CLOS [34] all support multiple-inheritance, declarative method combination and open classes. C++ supports multiple inheritance [35]. While AspectJ includes elements of these, AspectJ also provides more powerful and modular support for crosscutting than can be achieved with these features.

Declarative method combination, as in the CLOS line of languages, is not sufficient for AOP, because it lacks the pointcut mechanisms that enable crosscutting.

Ordinary multiple-inheritance (MI) is not sufficient for AOP for two reasons. First, a single aspect can include advice for all the different participants in a multi-class interaction. Using MI, a separate mixin-class must be defined for each participant class. Second, aspects work by *reverse-inheritance* – the aspect declares what classes it should affect rather than vice-versa. This means that adding or removing aspects from the system does not require editing affected class definitions.

Completely unstructured open classes, as in CLOS and its ancestors, enable some degree of crosscutting modularity, but they do so in a totally unstructured way. In AspectJ, classes and aspects are modular units, even if an aspect can crosscut classes.[7]

---

[7] Flavors, New Flavors and CLOS use the Common Lisp module system, called the package system. It is typically used in only very coarse-grained ways, certainly not at the level of single classes as in Java, and usually not even at the level of single packages in Java.

7.5 Other Work

Walker and Murphy have proposed a system based on implicit context that is also intended to improve separation of concerns [37]. Implicit context is similar to AspectJ in that the separation is made explicit in the source code. But it differs from AspectJ in that it provides reflective access to the entire call history of a system. Thus explicit context can reason about a wider dynamic context than is possible with `cflow`. AspectJ programmers could write aspects to manually gather call history information and thereby duplicate some explicit context functionality.

Implicit parameters provide dynamically scoped variables within a statically typed Hinley Milner framework [16]. Implicit parameters are lexically distinct from regular identifiers, and are bound by a special construct whose scope is dynamic, rather than static as with let. Implicit parameters have some of the power of using cflow to pass dynamic context. Implicit parameters are more powerful, in that the binding they create can be set from any reference site. But they do not have explicit crosscutting modularity support because references to the parameter are still spread throughout the code. Many implementations of Scheme provide the fluid-let construct that dynamically binds variables by side-effect, and then re-instates the previous binding after evaluation of the body is completed.

8   Future Work

We plan to use AspectJ as the basis of an empirical assessment of aspect-oriented programming. We want to develop a real AOP user community, and work with them to understand the practical effects of AOP. Our main focus now is building up and supporting the user community. To enable this, we are focusing on fine-tuning the language design and improving the quality of the compiler, IDE extensions and documentation.

The compiler has three main limitations that we are currently working on: it uses javac as a back-end rather than generating class files directly; it requires access to all the source code for the system; and it performs a full recompilation whenever any part of the user program changes. We believe we know how to build an incremental compiler that will perform reasonably well on modestly large systems, but fast incremental compilation for a language like AspectJ is definitely an area for future research.

In the tools area, we plan to support more IDEs. We will also make a crosscutting structure browsing API that will allow others to develop tools that understand AspectJ code. Our existing navigation model doesn't capture the structure of `cflow` pointcuts as well as we would like, so this will also be an area for future work.

Beyond the 1.0 release we plan to explore new kinds of pointcut designators based on dataflow properties of the program. Our goal with this functionality, which we call dflow, is to be able to capture crosscuts such as the extent of a value and control boundary crossings.

9   Summary

AspectJ is a seamless aspect-oriented extension to Java. Programming with AspectJ feels like a small extension of programming with Java. AspectJ programs are largely ordinary Java programs in which we use ordinary Java for class-like modularity, and use aspects to implement crosscutting modularity.

Implementing crosscutting concerns using AspectJ benefits in three ways over a plain Java implementation of the same functionality: the implementation is more modular and concise; the structure of the crosscutting is captured in a more explicit form; and because of the first two properties, programming environment tools can help the programmer navigate and understand that structure.

Looking forward, our goal is to work with the AspectJ user community to assess the benefits of using aspect-oriented programming, in more complex systems as well as to continue to explore language design, methodological and other issues.

10 Acknowledgements

References

1. Proceedings of the Conference on Domain-Specific Languages (DSL). USENIX,Santa Barbara, California, USA (1997)
2. Bobrow, D.G., et al.: CommonLoops: Merging Lisp and Object-Oriented Programming. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Portland, Oregon (1986) 17-29
3. Cannon, H.: Flavors: A non-hierarchical approach to object-oriented programming. Symbolics Inc.(1982)
4. Coady, Y., G. Kiczales, and M. Feeley: Exploring an Aspect-Oriented Approach to Operating System Code. In: Position paper for the Advanced Separation of Concerns

Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Minneapolis, Minnesota, USA (2000)

5. DeVolder, K.: Aspect-Oriented Logic Meta Programming. In: Meta-Level Architectures and Reflection, Reflection'99. Springer, Saint-Malo, France (1999) 250-272

6. Filman, R.E. and D.P. Friedman: Aspect-Oriented Programming is Quantification and Obliviousness. In: Position paper for the Advanced Separation of Concerns Workshop at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Minneapolis, Minnesota, USA (2000)

7. Friendly, L.: Design of Javadoc. In: The Design of Distributed Hyperlinked Programming Documentation (IWHD). Springer-Verlag, Montpellier, France (1995)

8. Goldberg, A.: Smalltalk-80: The Interactive Programming environment. Addisson-Wesley, Reading MA (1984)

9. Goldberg, A. and D. Robson: Smalltalk-80: The Language and Its Implementation. Addison-Wesley, (1983)

10. Green, T.R.G. and M. Petre: Usability analysis of visual programming environments: a 'cognitive dimensions' approach. Journal of Visual Languages and Computing. 7,2. (1996) 131-174

11. Griswold, D.: The Java HotSpot Virtual Machine Architecture. Sun Microsystems, Inc.(1998)

12. Ichisugi, Y., S. Matsuoka, and A. Yonezawa: RbCl: A reflective object-oriented concurrent language without a run-time kernel. In: International Workshop on New Models for Software Architecture (IMSA): Reflection and Meta-Level Architecture. Tama City, Tokyo (1992) 24-35

13. Irwin, J., et al.: Aspect-Oriented Programming of Sparse Matrix Code. In: Proceedings of the International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE). Springer, Marina del Rey, CA, USA (1997) 249-256

14. Kiczales, G., et al.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland (1997)

15. Kiczales, G. and L. Rodriguez: Efficient Method Dispatch in PCL. In: LISP and Functional Programming. ACM Press, Nice, France (1990) 99-105

16. Lewis, J., et al.: Implicit Parameters: Dynamic Scoping with Static Types. In: Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Boston, Massachusetts (2000) 108-118

17. Lieberherr, K.J.: Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston (1996)

18. Lopes, C.V. and G. Kiczales: D: A Language Framework for Distributed Programming. Technical Report SPL97-010, P9710047. Xerox Palo Alto Research Center,Palo Alto, CA (1997)

19. Maes, P.: Concepts and Experiments in Computational Reflection. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Orlando, Florida (1987) 147-155

20. Masuhara, H., S. Matsuoka, and A. Yonezawa: Designing an OO reflective language for massively-parallel processors. In: Position paper for the workshop on Object-Oriented Reflection and Metalevel Architectures at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Washington, DC (1993)

21. Matsuoka, S., T. Watanabe, and A. Yonezawa: Hybrid group reflective architecture for object-oriented concurrent reflective programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer, Geneva, Switzerland (1991) 231-250

22. McAffer, J.: The CodA MOP. In: Position paper for the workshop on Object-Oriented Reflection and Metalevel Architectures at the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA),. Washington, DC (1993)

23. Mendhekar, A., G. Kiczales, and J. Lamping: RG: A Case-Study for Aspect-Oriented Programming. Technical Report SPL97-009, P9710044. Xerox Palo Alto Research Center,Palo Alto, CA (1997)
24. Mezini, M. and K.J. Lieberherr: Adaptive Plug-and-Play Components for Evolutionary Software Development. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Vancouver, British Columbia, Canada (1998) 97-116
25. Moon, D.A.: Object-Oriented Programming with Flavors. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Portland, Oregon (1986) 1-8
26. Okamura, H., Y. Ishikawa, and M. Tokoro: Metalevel Decomposition in AL-1/D. In: International Symposium on Object Technologies for Advanced Software. Springer Verlag, (1993) 110-127
27. Ossher, H., et al.: Subject-Oriented Composition Rules. In: Proceedings of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Austin, Texas (1995) 235-250
28. Ossher, H. and P.L. Tarr: Hyper/J: multi-dimensional separation of concerns for Java. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, Limerick, Ireland (2000) 734-737
29. Parnas, D.L.: On the Criteria To Be Used in Decomposing Systems Into Modules. Communications of the ACM. 15,12. (1972) 1053-1058
30. Parnas, D.L.: Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs. Lecture Notes in Computer Science. Programming Methodology. (1974)
31. Shneiderman, B.: Direct Manipulation: A step beyond Programming languages, In: Human-Computer Interaction: A Multidisciplinary Approach, R.M. Baecker and W.A.S. Buxton, Editors. Morgan Kaufmann Publishers, Inc.: Los Altos, CA (1983) 461-467
32. Smith, B.C.: Reflection and Semantics in a Procedural Language, PhD Thesis. M.I.T(1982)
33. Smith, B.C.: Reflection and Semantics in LISP. In: Proceedings of the Symposium on Principles of Programming Languages (POPL). ACM, (1984) 23-35
34. Steele, G.L.: Common Lisp the Language. 2nd ed. Digital Press, (1990) 1029
35. Stroustrup, B.: The C++ Programming Language. 3rd ed. Addison-Wesley, (1997)
36. Tarr, P.L., et al.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proceedings of the International Conference on Software Engineering (ICSE). ACM, Los Angeles, CA (1999) 107-119
37. Walker, R. and G. Murphy: Implicit Context: Easing Software Evolution and Reuse. In: Proceedings of the Conference on Foundations of Software Engineering (FSE). ACM, San Diego, California (2000)
38. Watanabe, T. and A. Yonezawa: Reflection in an object-oriented concurrent language. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, San Diego, CA (1988) 306-315