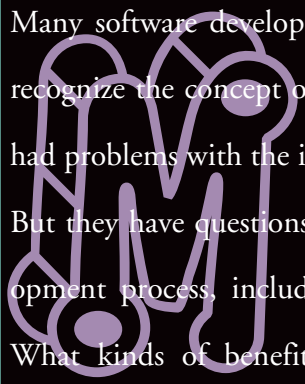
GETTING STARTED WITH ASPECTJ

An aspect-oriented extension to Java enables plug-and-play implementations of crosscutting.

Many software developers are attracted to the idea of AOP—they recognize the concept of crosscutting concerns and know they have had problems with the implementation of such concerns in the past. But they have questions about how to adopt AOP into their development process, including: How to use aspects in existing code? What kinds of benefits can be expected? How steep is the learning curve for AOP? How to begin? These questions are addressed here in the context of AspectJ—a general-purpose AO extension to Java. A series of abridged examples illustrate the kinds of aspects programmers can implement using AspectJ and the benefits associated with doing so. Readers who want to understand the examples in more detail, or want to learn more about AspectJ, can find the complete running examples and additional explanation on the AspectJ.org Web site.



Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold

Concern about the risk of adopting new technology causes many organizations to be reluctant to do so. But simply waiting can lead to rushing to adopt the technology later, which is itself risky. Instead, this article presents a staged approach based on identifying two broad categories of aspects: development aspects facilitate tasks such as debugging, testing, and performance tuning of applications; production aspects implement functionality intended to be included in shipping applications. These categories are informal, and this ordering is not the only way to adopt AspectJ. Some developers will want to use a production aspect right away, but experience with current AspectJ users has shown this ordering allows developers to derive benefits from AOP technology quickly, while also minimizing risk.

AspectJ Semantics

A brief introduction to the features of AspectJ used in this article is presented here. These features are at the core of the language, but this is not a complete overview of AspectJ—for more complete information, see [1, 4]. The semantics are presented using a simple figure editor system—see Figure 1. A `Figure` consists of a number of `FigureElements`, which can be either `Points` or `Lines`. The `Figure` class is also a factory for figure elements. There is a single `Display` on which figure elements are drawn. Most examples in the article are based on this system.

AO languages have three critical elements: a join point model, a means of identifying join points, and a means of affecting implementation at join points. The AspectJ version of each of these are described in more detail here.

The join point model in an AOP language provides the common frame of reference that makes it possible to define the structure of crosscutting concerns. This article describes AspectJ's dynamic join points, in which join points are certain well-defined points in the execution flow of the program. AspectJ has several kinds of join points, but this article discusses only method call join points. A method call join point is the point in the flow when a method is called, and when that method call returns. Each method call itself is one join point. The lifetime of the join point is the entire time from when the call begins to when it returns (normally or abruptly), but execution is at the join point only at the moment the call begins and the moment it returns.

In AspectJ, pointcut designators identify particular join points by filtering out a subset of all the join points in the program flow. For example, the pointcut designator:

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

identifies any call to either the `setX` or `setY` methods defined by `Point`. Syntactically, this code consists of two `call` pointcut designators composed with "or." The syntax of `call` is based on that of Java method signatures: `call(`*`result_type object_type.method_name(arg_type, ...)`*`)` Programmers can define named pointcut designators, and pointcut designators can identify join points from many different classes—in other words, they can crosscut classes. The following code defines a pointcut named `move` that designates any method call that moves figure elements:

```
pointcut move():
  call(void FigureElement.moveBy(int, int))||
  call(void Point.setX(int))          ||
  call(void Point.setY(int))          ||
  call(void Line.setP1(Point)         ||
  call(void Line.setP2(Point);        ||
```

The previous pointcut designators are based on explicit enumeration of a set of method signatures; we call this name-based crosscutting. AspectJ also allows specification of a pointcut in terms of properties of methods rather than their exact name. We call this property-based crosscutting. The simplest of these involve using wild cards in certain fields of the method signature. Others use control flow or other properties to identify join points. Consider:

```
call(void Figure.make*(..))
call(public * Display.*(..))
cflowbelow(move())
```

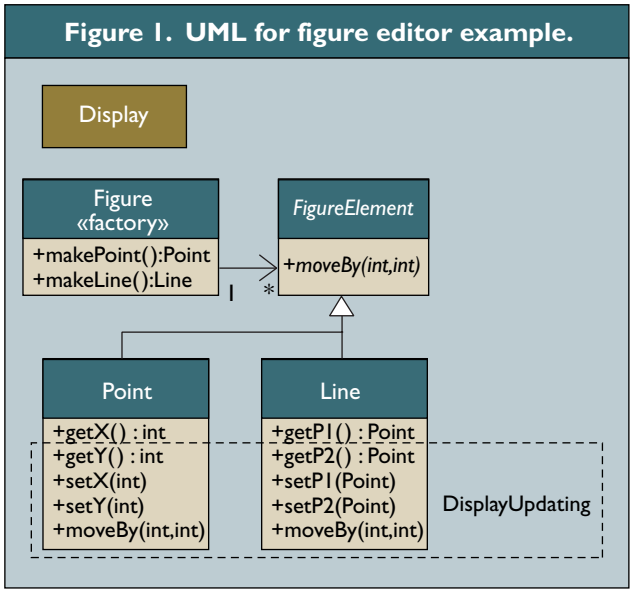
The first designates any call to methods defined on `Figure`, for which the name begins with `"make,"` and which take any number of parameters; effec-

tively the factory methods `makePoint` and `makeLine`. The second identifies any call to a public method defined on `Display`. The third uses the `cflowbelow` primitive pointcut designator and identifies all join points that occur during the execution of methods that move figure elements.

In AspectJ, advice declarations are used to define additional code that runs at join points. Before advice runs at the moment a join point is reached, or in other words just before the method begins running. After advice runs at the moment control returns through the join point, or just after the method has run (and before control is returned to the caller). Around advice runs when the join point is reached, and has explicit control over whether the method itself is allowed to run at all. This advice prints a simple message right after any figure element moves.

```
after(): moves() {
  <code to print message>
}
```

Figure 1. UML for figure editor example.



Pointcut designators can expose certain values in the execution context at join points. Exposed values are called pointcut parameters and can be used in advice declarations. Briefly speaking, the parameter mechanism works using three special pointcut designators: `this`, `target`, and `args`. For example, in `calls(void Point.setX(int))`, both the object receiving the call, corresponding to `Point`, and the new value, corresponding to `int`, can be exposed. The `PointBoundsChecking` aspect later in the article uses this mechanism. The online complete examples explain this in more detail.

Development Aspects

An aspect is a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, constructors, initializers, named pointcuts, and advice. Examples of aspects that can be used during program development are presented here. This kind of aspect defines behavior that ranges from simple tracing, to profiling, to testing of internal consistency within the application, and is used to facilitate debugging, testing, and performance tuning work.

An initial example is a simple tracing aspect that prints messages before certain display operations:

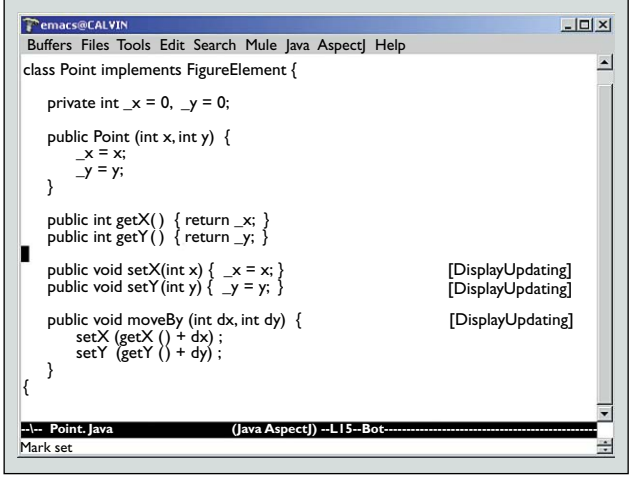
```
aspect SimpleTracing {
  pointcut traced():
      call(void Display.update())    ||
      call(void Display.repaint(..));

  before(): traced() {
      println("Entering:" +
              thisJoinPoint);
      }
  void println(String str) {
      <write to appropriate stream>
  }
}
```

This code first defines a pointcut named `traced`, which identifies calls to several key methods on `Display`—the `update` method and several overloaded `repaint` methods. Before advice on this pointcut uses a helper method of the aspect to print a message. The advice uses the `thisJoinPoint` special variable, which is bound, within advice bodies, to an object that describes the current join point. The overall effect of this aspect is to print a descriptive message whenever the traced methods are called.

Notice that when coded with AspectJ this tracing functionality is modularized—the code is localized and has a clear interface with the rest of the system. Modularization here has the usual benefits. For one, consider changing the set of method calls that are traced. In the AspectJ implementation, this just requires editing the `traced` pointcut and recompiling. The individual methods traced do not need to be edited.

When debugging, programmers often invest considerable effort in determining a good set of trace points to use when looking for a particular kind of problem. When debugging is complete—or appears to be complete—it is frustrating to have to lose that investment by deleting trace statements from the

Figure 2. A snapshot of a 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. The editor also provides commands to jump to the advice from the method and vice versa. AspectJ support for JBuilder and Forte4J provide similar functionality.



code. The alternative of just commenting them out makes the code look bad and can cause trace statements for one kind of debugging to get confused with trace statements for another kind of debugging. With AspectJ it is easy to both preserve the work of designing a good set of trace points and disable the tracing when it is not being used. Simply write an aspect for each particular tracing mode and remove that aspect from the compile configuration (or "makefile") when it is not needed.

This clean modularization gives developers who have reason to be conservative about new technology adoption a strong intermediate position from which to start using AspectJ. They can use AspectJ for debugging and other development aspects, but still compile and ship the production code without aspects. The makefiles can even be written so that they use a traditional Java compiler for production builds, which ensures no aspects can be present in such builds.

There are many sophisticated profiling tools available on the market. These can gather a variety of information and display the results in useful ways. But sometimes programmers want very specific profiling or logging behavior. In these cases it is often possible to write a simple aspect similar to the ones above to do the job. For example, an aspect could use the following pointcut declaration to identify the particular calls to `setX` or `setY` of a point within the control flow of calls to `moveBy`:

```
call(void Point.set*(int)) &&
cflow(call(void *.moveBy(..)))
```

Many programmers use the "Design by Contract" style popularized by Eiffel [3]. In this style of programming, explicit pre-conditions test that callers of a method call it properly and explicit post-conditions test that methods properly do the work they are supposed to. AspectJ makes it possible to implement pre- and post-condition testing in modular form. For example, this code

```
aspect PointBoundsChecking {
  before(Point p, int x):
      call(void p.setX(x)) {
    checkX(p, x);
    }
}
  before(Point p, int y):
    <same for y>

  before(Point p, int x, int y):
      call(void p.moveBy(x, y)) {
    checkX(p, p.getX() + x);
    checkY(p, p.getY() + y);
  }
}
```

implements pre-condition testing for operations that move points. (Note that it makes use of `checkX` and `checkY` helper methods that are not shown.) Even though pre- and post-condition testing aspects may often be used only during testing, in some cases developers may wish to include them in the production build as well. Again, because AspectJ makes it possible to cleanly modularize these crosscutting concerns, it gives developers easy control over this decision.

Property-based crosscutting can be used to define more sophisticated contract enforcement. One such use is to identify method calls that violate a design invariant of the program. For example, the following pointcut can be used to enforce the constraint that only the factory methods can create new figure elements.

```
call(FigureElement.new(..)) &&
!withincode(* Figure.make*(..));
```

The `call` pointcut designator identifies any call to `new` that makes figure elements. The `withincode` primitive pointcut designator identifies all join points that occur lexically within the body of

the factory methods on `FigureElement`. The conjunction therefore identifies illegal constructions. Note that this provides more expressive power than the Java access protection qualifiers, because in this code, not even the `Point` and `Line` classes themselves can call their constructors.

Production Aspects

This section presents examples of aspects that are inherently intended to be included in production builds of an application. Again, we begin with named-based aspects. Because these aspects tend to affect only a small number of methods they are a good next step for projects adopting AspectJ. But even though they tend to be small and simple, they often have a significant effect in terms of making the program easier to understand and maintain.

The first example production aspect handles updating the display whenever a figure element moves. Implementing this functionality as an aspect is straightforward. We reuse the `move` pointcut defined earlier, and after advice on `move` informs the display it needs to be refreshed whenever an object moves.

```
aspect DisplayUpdating {
  pointcut move(): <as above>;
  after(): move() {
    Display.needsRepaint();
  }
}
```

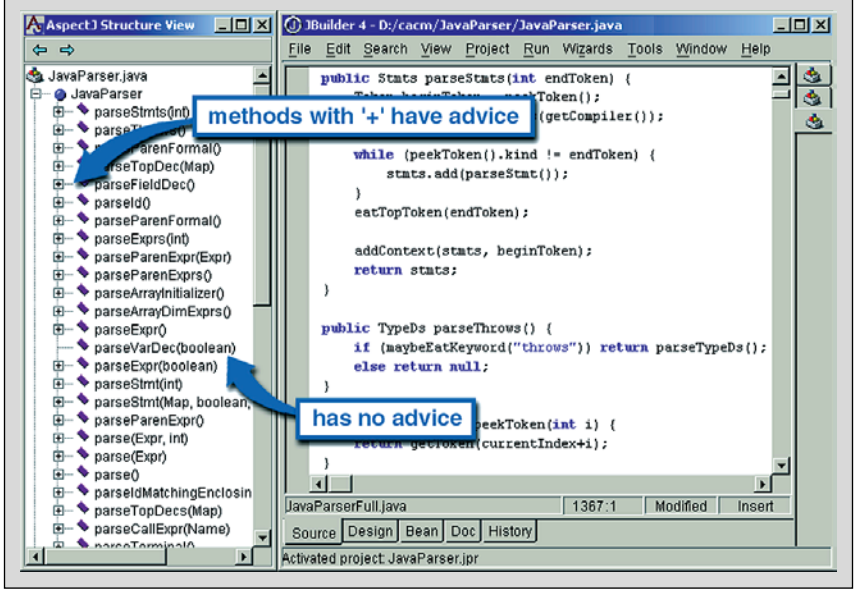
Even this simple example serves to illustrate some of the important benefits of using AspectJ in production code. Consider the code one would have to write without AspectJ. Each of the methods that could move a figure element would include a call to `Display.needsRepaint()`. Those calls, or rather the concept that those calls should happen at each move operation, are the crosscutting concern in this case. The AspectJ implementation has several advantages over the standard implementation:

- The structure of the crosscutting concern is captured explicitly. The `move` pointcut clearly states all the methods involved, so the programmer reading the code sees not scattered individual calls to `needsRepaint`, but instead sees the overall structure of the code. As shown in figures 2 and 4, AspectJ extensions to existing IDE tools automatically remind the programmer that this aspect advises each of the methods involved.
- Evolution is easier. If, for example, the aspect needs to be revised to inform the display exactly which element moved the change would be entirely local to the aspect. The pointcut would be updated to expose the object being moved, and the advice would be updated to pass that object. (Several ways this aspect could be expected to evolve are presented in [1].)
- The functionality is pluggable. Like development aspects, production aspects may need to be removed from the system, either because they are no longer needed at all, or because they are not needed in certain system configurations. Because the functionality is modularized in a single aspect this is easy to do.
- The implementation is more stable. If, for example, the programmer adds a subclass of `Line` that overrides existing methods, this aspect will still work properly. In the ordinary Java implementation the programmer would have to remember to add the call to `needsRepaint` in the overriding method. Later we will see that this benefit can be even more compelling for property-based aspects.

Figure 3. An aspect that controls factory methods based on dynamic call context.

```
aspect ColorControlling {

  /**
   * All join points dynamically within methods
   * on ColoringClient.  Exposes the client.
   */
  pointcut clientCflow(ColoringClient client):
    cflowbelow(call(* client.*(..)));

  /**
   * All figure element factory calls.
   */
  pointcut makes():
    call(FigureElement Figure.make*(..));

  after (ColoringClient c) returning (FigureElement fe):
      make() && clientCflow(c) {
    fe.setColor(c.colorFor(fe));
  }
}
```

Figure 4. Examining the partially refactored parser using the AspectJ extensions to JBuilder. This view allows us to quickly see the structure of the aspect and how it affects the `JavaParser` class.



Another good use of name-based production aspects is implementing synchronization policies. These aspects are similar to change monitoring, except that the work done by the advice tends to be more complex, and these aspects usually use paired before and after advice to handle the synchronization work.

For example, in order to implement the readers and writers synchronization pattern [2], the programmer would define two pointcuts, named `reader` and `writer`, and then would define appropriate before and after advice on those pointcuts. Such an implementation reflects the structure of the synchronization rules more clearly than the normal scattered implementation.

The crosscutting structure of context passing can be a significant source of complexity in Java programs. Consider implementing functionality that would allow a client of the figure editor (a program client rather than a human) to set the color of any figure elements that are created. Typically this requires passing a color, or a color factory, from the client, down through the calls that lead to the figure element factory. All programmers are familiar with the inconvenience of adding a first argument to a number of methods just to pass this kind of context information.

Using AspectJ, this kind of context passing can be implemented in a modular way. The aspect in Figure 3 defines after advice that runs when the factory methods of `Figure` are called from within the control flow of a method on `ColoringClient`. This enables the client to set the color of any figure elements created. The aspect shown in Figure 3 affects only a small number of methods, but note that the non-AOP implementation of this functionality might require editing many more methods; specifically, all the methods in the control flow from the client to the factory would have to pass the client. This is a benefit common to many property-based aspects—while the aspect is short and affects only a modest number of methods, the complexity it saves is potentially much larger.

Property-based aspects can also be used to provide consistent handling of functionality across a large set of operations. One common idiom is to define functionality that should be common to all the public methods of a package. Typical examples include logging, billing, and error-checking behavior. Such aspects, typically use a property-based pointcut such as `call(public * com.xerox.*.*(..))`. This pointcut identifies any call to a public method of a type defined in the com.xerox package. Behavior defined in terms of such a pointcut is robust during program evolution. If a new public method is added, it will automatically get the advice.

In some cases it is important to distinguish between initial and recursive calls to a method or set of methods. It might, for example, be important to only do error checking for initial calls into a package. In such cases the `cflowbelow` pointcut designator can be used to distinguish the initial calls. For example, `move() && !cflowbelow(move())` will exclude any call to move methods that occurs during the execution of move methods. So it excludes calls

to `setX` that happen during a `moveBy`.

Sometimes aspects consist of a property-based crosscut with a small number of exceptions. For example, in work on the AspectJ compiler we have developed an aspect that advises about 35 methods in the `JavaParser` class. The individual methods handle each of the different kinds of elements that must be parsed. They have names like `parseMethodDec`, `parseThrows`, and `parseExpr`. The role of the aspect is to ensure that each of these parse methods fills in the parse context in a consistent way. The aspect includes the following pointcut, as well as around advice to fill in parse context.

```
pointcut parse(JavaParser jp):
  call(* jp.parse*(..)) &&
  !call(Stmt parseVarDec(..));
```

Note that in addition to the general pattern—`call(* jp.parse*(..))`—it includes an exception—`!call(Stmt parseVarDec(..))`. The exception happens because the parsing of variable declarations in Java is too complex to be handled in the same way as the other `parse*` methods. We find that even with a small number of exceptions, such aspects are a clear expression of crosscutting modularity. In this case, that all `parse*` methods, except for `parseVarDec`, share a common behavior for establishing the parse context of their result.

The process of writing an aspect with property-based crosscutting and exceptions can also help clarify the design structure of the system. This is especially true when refactoring existing code to use aspects. During development of the parse context aspect, we used the AspectJ support provided for JBuilder, shown in Figure 4, to compare the aspect crosscutting to where we had previously manually coded the functionality and incrementally refactored the two to understand exactly what the aspect should crosscut and do.

Conclusion

AspectJ is a simple and practical AO extension to Java. Using AspectJ results in clean modular implementations of crosscutting concerns such as tracing, contract enforcement, display updating, synchronization, consistency checking, protocol management and others. When written as an aspect the structure of a crosscutting concern is explicit and easy to reason about. Aspects are also modular, making it possible to develop plug-and-play implementations of crosscutting functionality.

AspectJ enables both name- and property-based crosscutting. Aspects that use name-based crosscutting tend to affect a small number of other classes. But despite their small scale, they can often eliminate significant complexity compared to an ordinary Java implementation. Aspects that use property-based crosscutting range from small to large scale. Features of Aspect J not presented provide additional power for modularizing crosscutting concerns.

Adoption of AspectJ into an existing project can be a straightforward and incremental task. One path is to begin with development aspects, and move on to production aspects only after building up experience with AspectJ. Other paths are possible, depending on the needs of the project. Programmers interested in adopting AspectJ are encouraged to read the online documentation and examples carefully before doing so. C

References

1. Kiczales, G., et al. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2001.
2. Lea, D. *Concurrent Programming in Java: Design Principles and Patterns, 2d ed*. Addison-Wesley, 1999.
3. Meyer, B. *Object-Oriented Software Construction.* Prentice Hall, 1989.
4. *The AspectJ Primer*; aspectj.org/doc/primer.

Gregor Kiczales (gregor@cs.ubc.ca) is Professor and NSERC, Xerox, Sierra Systems Chair of Software Design at the University of British Columbia, and Principal Scientist at Xerox PARC.
Erik Hilsdale (hilsdale@parc.xerox.com) is a Research Staff Member at Xerox PARC.
Jim Hugunin (hugunin@parc.xerox.com) is a Research Staff Member at Xerox PARC.
Mik Kersten (mkersten@parc.xerox.com) is a Research Staff Member at Xerox PARC.
Jeffrey Palm (palm@parc.xerox.com) is a Research Staff Member at Xerox PARC.
William G. Griswold (wgg@cs.ucsd.edu) is Associate Professor of Computer Science at the University of California, San Diego.

This work was partially supported by DARPA under contract number F30602-C-0246.