Getting started with AspectJ

Article *in* Communications of the ACM · October 2001

DOI: 10.1145/383845.383858 · Source: DBLP

Gregor Kiczales
University of British Columbia - Vancouver
144 PUBLICATIONS **18,282** CITATIONS
SEE PROFILE

Mik Kersten
Tasktop
37 PUBLICATIONS **4,283** CITATIONS
SEE PROFILE

William G. Griswold
University of California, San Diego
235 PUBLICATIONS **10,809** CITATIONS
SEE PROFILE

Getting Started with AspectJ

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

(This paper has been submitted for review to the CACM 2001 special theme section on AOP.)

1 Introduction

Many software developers are attracted to the idea of AOP, but unsure about how to begin using the technology. They recognize the concept of crosscutting concerns, and know that they have had problems with the implementation of such concerns in the past. But there are many questions about how to adopt AOP into the development process. Common questions include: Can I use aspects in my existing code? What kinds of benefits can I expect to get? How do I find aspects? How steep is the learning curve for AOP? What are the risks of using this new technology?

This paper addresses these questions in the context of AspectJ – a general-purpose aspect-oriented extension to Java. A series of abridged examples illustrate the kinds of aspects programmers may want to implement using AspectJ and the benefits associated with doing so. Readers who would like to understand the examples in more detail, or who want to learn how to program examples like these, can find the complete examples and supporting material on the AspectJ web site.[1]

A significant risk in adopting any new technology is going too far too fast. Concern about this risk causes many organizations to be conservative about adopting new technology. To address this issue, the examples in the paper are grouped into three broad categories, with aspects that are easier to adopt into existing development projects coming earlier in the paper. Section 3 presents *development aspects* that facilitate tasks such as debugging, testing and performance tuning of applications. Section 4 presents *production aspects* that implement crosscutting functionality common in Java applications.

These categories are informal, and this ordering is not the only way to adopt AspectJ. Some developers may want to use a production aspect right away. But our experience with current AspectJ users suggests that this is one ordering that allows developers to get experience with (and benefit from) AOP technology quickly, while also minimizing risk.

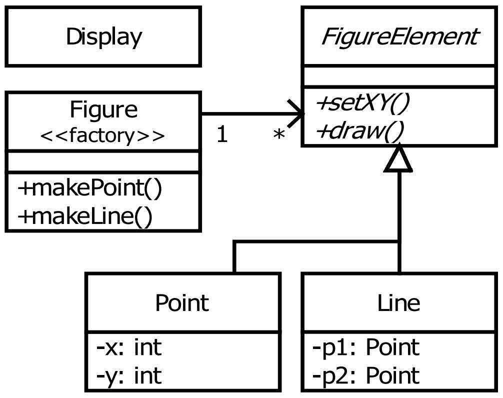
[†] University of British Columbia and Xerox Palo Alto Research Center (gregor@cs.ubc.ca). [‡] Xerox Palo Alto Research Center. [††] University of California, San Diego. This work was partially supported by the Defense Advanced Research Projects Agency under contract number F30602-C-0246.

[1] The AspectJ system, primer, and a complete implementation of these examples are available at http://aspectj.org. (Note to reviewers: We will make a special place on the web site to get these examples. For a few months after the CACM issue comes out this will be prominently featured. Later it will be less prominently featured, but will remain on the site indefinitely.)

2 AspectJ semantics

This section presents a brief introduction to the features of AspectJ used later in the paper. These features are at the core of the language, but this is by no means a complete overview of AspectJ. For a more complete or more detailed understanding of AspectJ, see [2, 3].

The semantics are presented using a simple figure editor system. A `Figure` consists of a number of FigureElements, which can be either `Points` or `Lines`. The `Figure` class provides factory services. There is also a `Display`. Most example programs later in the paper are based on this system as well.



UML for Figure Editor Example.

AspectJ realizes the modularization of crosscutting concerns using join points and advice. *Join points* are well-defined points in the program flow and *advice* define code that is executed when join points are reached.

2.1 The Join Point Model

A critical element in the design of any aspect-oriented language is the join point model. The join point model provides the common frame of reference that makes it possible to define the structure of crosscutting concerns.

In AspectJ, join points are certain well-defined points in the execution of the program. AspectJ provides for many kinds of join points, but this paper discusses only one of them: method call join points. A method call join point encompasses the actions of an object receiving a method call. It includes all the actions that comprise a method call, starting after all arguments are evaluated up to and including normal or abrupt return.

Each method call itself is one join point. The dynamic context of a method call may include many other join points: all the join points that occur when executing the called method and any methods that it calls.

2.2 Pointcut Designators

In AspectJ, *pointcut designators* identify collections of join points in the program flow. For example, the pointcut designator:

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

identifies all calls to the method `setX` defined on `Point` objects. Pointcut designators can be composed using a set algebra semantics, so for example:

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

identifies all calls to either the `setX` or `setY` methods defined by `Point`.

Programmers can define their own pointcut designators, and pointcut designators can identify join points from many different classes – in other words, they can crosscut classes. So, for example, the following named pointcut declaration

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

defines a pointcut named `moves` that designates calls to any of the methods that move figure elements.

2.2.1 Property-Based Primitive Pointcut Designators

The previous pointcut designators are all based on explicit enumeration of a set of method signatures. We call this *name-based* crosscutting. AspectJ also provides mechanisms that enable specifying a pointcut in terms of properties of methods other than their exact name. We call this *property-based* crosscutting. The simplest of these involve using wildcards in certain fields of the method signature. For example:

```
calls(void Figure.make*(..))
```

identifies calls to any method defined on `Figure`, for which the name begins with `"make"`, specifically the factory methods `makePoint` and `makeLine`; and

```
calls(public * Figure.* (..)) ||
```

identifies calls to any public method defined on `Figure`.

One very powerful primitive pointcut designator, `cflow`, identifies join points based on whether they occur in the dynamic context of another pointcut. So

```
cflow(moves())
```

designates all join points that occur between receiving method calls for the methods in `moves` and returning from those calls (either normally or by throwing a `Throwable`).

2.3 Advice

Pointcuts are used in the definition of advice. AspectJ has several different kinds of advice that define additional code that should run at join points. *Before advice* runs when a join point is reached and before the computation proceeds, i.e. that runs when computation reaches the method call and before the actual method starts running. *After advice* runs after the computation 'under the join point' finishes, i.e. after the method body has run, and just before control is returned to the caller. *Around advice* runs when the join point is reached, and has explicit control over whether the computation under the join point is allowed to run at all.

```
after(): moves() {
  System.out.println("A figure element moved.");
}
```

2.3.1 Exposing Context in Pointcuts

Pointcut designators can also expose part of the execution context at their join points. Values exposed by a pointcut designator can be used in the body of advice declarations. In the following code, the pointcut exposes three values from calls to `setXY`: the `FigureElement` receiving the call, the new value for `x` and the new value for `y`. The advice then prints the figure element that was moved and its new `x` and `y` coordinates after each `setXY` method call.

```
    pointcut setXYs(FigureElement fe, int x, int y):
      calls(void fe.setXY(x, y));

    after(FigureElement fe, int x, int y): setXYs(fe, x, y) {:
      System.out.println(fe + " moved to (" + x + ", " + y + ").”);
    }
```

2.4 Aspect Declarations

An *aspect* is a modular unit of crosscutting implementation. It is defined very much like a class, and can have methods, fields, and initializers. The crosscutting implementation is provided in terms of pointcuts and advice. Only aspects may include advice, so while AspectJ may define crosscutting effects, the declaration of those effects is localized.

3 Development Aspects

This section presents examples of aspects that can be used during development of Java applications. These aspects facilitate debugging, testing and performance tuning work. The aspects define behavior that ranges from simple tracing, to profiling, to testing of internal consistency within the application.

Using AspectJ makes it possible to cleanly modularize this kind of functionality, thereby making it possible to easily enable and disable the functionality when desired. Section 3.4 presents techniques that make it possible to ensure that the functionality is not included in production builds of an application. These techniques give 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 some testing, but still compile and ship the production code without aspects.

3.1 Tracing, logging, profiling

A first example is a simple tracing aspect that just prints a simple message at the specified method calls. Continuing with the figure editor example, one such aspect might simply trace whenever points are moved.

```
  aspect SimpleTracing {
    pointcut tracedCalls():
      calls(void FigureElement.draw(GraphicsContext));

    before(): tracedCalls() {
      System.out.println(“Entering: ” + thisJoinPoint);
    }
  }
```

This code makes use of the `thisJoinPoint` special variable. Within all advice bodies this variable is bound to an object that describes the current join point. The effect of this code is to print a line like the following every time a figure element receives a draw method call:

```
Entering: call(void FigureElement.draw(GraphicsContext))
```

To understand the benefit of coding this with AspectJ consider changing the set of method calls that are traced. With AspectJ, this just requires editing the definition of the `tracedCalls` pointcut and recompiling. The individual methods that are traced do not need to be edited.

When debugging, programmers often invest considerable effort in figuring out 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 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 isn't being used. This is done by writing an aspect specifically for that tracing mode, and removing that aspect from the compilation when it is not needed.

This ability to concisely implement and reuse debugging configurations that have proven useful in the past is a direct result of AspectJ modularizing a crosscutting design element – the set of methods that are appropriate to trace when looking for a given kind of information.

3.1.1 Profiling and Logging

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, the following aspect will count the number of calls to the `rotate` method on a `Line` and the number of calls to the `set*` methods of a `Point` that happen within the control flow of those calls to `rotate`:

```
aspect SetsInRotateCounting {
  int rotateCount = 0;
  int setCount    = 0;

  before(): calls(void Line.rotate(double)) {
    rotateCount++;
  }
  before(): calls(void Point.set*(int)) &&
            cflow(calls(void Line.rotate(double))) {
    setCount++;
  }
}
```

3.2 Pre/post conditions

Many programmers use the “Design by Contract” style popularized by Eiffel [1]. 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 {
  pointcut setXs(int x):
        calls(void FigureElement.setXY(x, int)) ||
        calls(void Point.setX(x));

  pointcut setYs(int y): ...;

  before(int x): setXs(x) {
    if ( x < MIN_X || x > MAX_X ) )
      throw new IllegalArgumentException ("x is out of bounds.");
  }

  before(int y): setYs(y) {
    ...
  }
}
```

implements the bounds checking aspect of pre-condition testing for operations that move points. Notice that the `setXs` pointcut designator refers to all the operations that can set a point’s `x` coordinate; this includes the `setX` method, as well as “half of” the `setXY` method. In this sense the `setXs` pointcut can be seen as involving very fine-grained crosscutting – it names the the `setX` method and half of the `setXY` method.

Even though pre- and post-condition testing aspects can 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 good control over this decision.

3.3 Contract enforcement

The property-based crosscutting mechanisms can be very useful in defining more sophisticated contract enforcement. One very powerful use of these mechanisms is to identify method calls that, in a correct program, should not exist. For example, the following aspect enforces the constraint that only the well-known factory methods can add an element to the registry of figure elements. Enforcing this constraint ensures that no figure element is added to the registry more than once.

```
static aspect RegistrationProtection {
  pointcut registers():
    calls(void Registry.register(FigureElement));

  pointcut canRegister():
    withincode(static * FigureElement.make*(..));

  before(): registers() && !canRegisters() {
    throw new IllegalAccessException("Illegal call " + thisJoinPoint);
  }
}
```

This aspect uses the `withincode` primitive pointcut designator to denote all join points that occur within the body of the factory methods on `FigureElement` (the methods with names that begin with `"make")`. This is a property-based pointcut designator because it identifies join points based not on their signature, but rather on the property that they occur specifically within the code of another method. The before advice declaration effectively says "signal an error for any calls to register that are not within the factory methods."

3.4 Configuration Management

Configuration management for aspects can be handled using a variety of "make-file like" techniques. To work with optional aspects, the programmer can simply define their make files to either include the aspect in the call to the AspectJ compiler or not, as desired.

Developers who want to be certain that no aspects are included in the production build can do so by configuring their make files so that they use a traditional Java compiler for production builds. To make it easy to write such make files, the AspectJ compiler has a command-line interface that is consistent with ordinary Java compilers.

4 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 and follow with property-based aspects. Name-based production aspects tend to affect only a small number of methods. For this reason, they are a good next step for projects adopting AspectJ. But even though they tend to be small and simple, they can often have a significant effect in terms of making the program easier to understand and maintain.

4.1 Change Monitoring

The first example production aspect supports the code that refreshes the display. The role of the aspect is to maintain a dirty bit indicating whether or not an object has moved since the last time the display was refreshed.

Implementing this functionality as an aspect is straightforward. The `testAndClear` method is called by the display code to find out whether a figure element has moved recently. This method returns the current state of the `dirty` flag and resets it to `false`. The `moves` pointcut captures all the method calls that can move a figure element. The after advice on `moves` sets the `dirty` flag whenever an object moves.

```
aspect MoveTracking {
  private static boolean dirty = false;

  public static synchronized boolean testAndClear() {
    boolean result = dirty;
    dirty = false;
    return result;
  }

  pointcut moves():
    calls(void FigureElement.setXY(int, int)) ||
    calls(void Line.setP1(Point))             ||
    calls(void Line.setP2(Point))             ||
    calls(void Point.setX(int))               ||
    calls(void Point.setY(int));

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

Even this simple example serves to illustrate some of the important benefits of using AspectJ in production code. Consider implementing this functionality with ordinary Java: there would likely be a helper class that contained the `dirty` flag, the `testAndClear` method as well as a `setFlag` method. Each of the methods that could move a figure element would include a call to the `setFlag` method. Those calls, or rather the concept that those calls should happen at each move operation, are the crosscutting concern in this case.

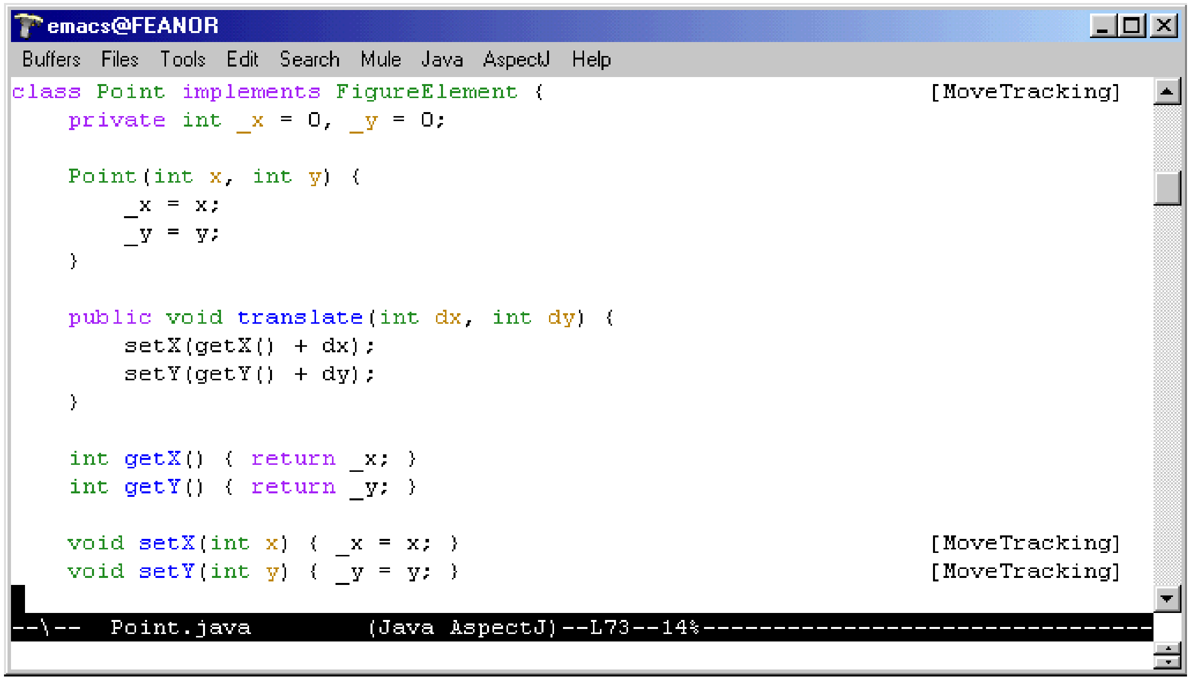
The AspectJ implementation has several advantages over the standard implementation:

The structure of the crosscutting concern is captured explicitly. The `moves` pointcut clearly states all the methods involved, so the programmer reading the code sees not just individual calls to `setFlag`, but instead sees the real structure of the code. As shown in Figure 2 the IDE support included with AspectJ, will 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 record not just that some figure element moved, but rather to record exactly which figure elements 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 record that object. ([3] presents a detailed discussion of various ways this aspect could be expected evolve.)

The functionality is easy to plug in and out. Just as with development aspects, production aspects may need to be removed from the system, either because the functionality is no longer needed at all, or because it is not needed in certain configurations of a system. 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 the existing methods, this advice in this aspect will still apply. In the ordinary Java implementation the programmer would have to remember to add the call to `setFlag` in the new overriding method. This benefit is often even more compelling for property-based aspects (see Section 4.4).



```
class Point implements FigureElement {                          [MoveTracking]
    private int _x = 0, _y = 0;

    Point(int x, int y) {
        _x = x;
        _y = y;
    }

    public void translate(int dx, int dy) {
        setX(getX() + dx);
        setY(getY() + dy);
    }

    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) { _x = x; }                                [MoveTracking]
    void setY(int y) { _y = y; }                                [MoveTracking]

--\--  Point.java        (Java AspectJ)--L73--14%-------------------------------
```

Figure 2. A snapshot of 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 provide commands to jump to the advice from the method and vice-versa.

4.2 Synchronization

Another good use of name-based production aspects is to implement 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.

The following example shows how the readers and writers synchronization pattern presented in [4] can be implemented using AspectJ. This aspect uses the `eachobject` feature of the language to ensure that each object to which this aspect applies will have its own instance of the aspect, and therefore its own count of active and waiting readers and writers. This means that the synchronization constraints of this aspect will apply on a per-object basis, which is appropriate for this pattern.

```
aspect RegistryReaderWriterSynchronizing
       of eachobject(instanceof(readers() || writers()) {
  pointcut readers():
        calls(Vector Registry.elementsNear(int, int));

  pointcut writers():
        calls(void Registry.add(FigureElement)) ||
        calls(void Registry.remove(FigureElement));

  protected int activeReaders,  activeWriters,
                waitingReaders, waitingWriters;

  before(): readers() { beforeRead();  } //these helper
  after():  readers() { afterRead();   } //methods of the
  before(): writers() { beforeWrite(); } //aspect are
  after():  writers() { afterWrite();  } //defined below

  protected synchronized void beforeRead() {
    ++waitingReaders;
    while (!(waitingWriters == 0 && activeWriters == 0)) {
      try { wait(); } catch (InterruptedException ex) {}
    }
    --waitingReaders;
    ++activeReaders;
  }

  protected synchronized void afterRead()   { ... }
  protected synchronized void beforeWrite() { ... }
  protected synchronized void afterWrite()  { ... }
}
```

4.3 Context Passing

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 following code adds after advice that runs only when the factory methods of `Figure` are called in the control flow of a method on a `ColorControllingClient`.

```
aspect ColorControl {
  pointcut CCClientCflow(ColorControllingClient client):
    cflow(calls(* client.* (..)));

  pointcut makes(FigureElement fe):
    calls(fe Figure.make*(..));

  after (ColorControllingClient c, FigureElement fe):
      makes(fe) && CCClientCflow(c) {
    fe.setColor(c.colorFor(e));
  }
}
```

This aspect 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. This is a benefit common to many property-based aspects – while the aspect is short and affects only a modest number of benefits, the complexity the aspect saves is potentially much larger.

4.4 Consistent Behavior Across a Large Number of Operations

This example aspect shows how a property-based aspect can be used to provide consistent handling of functionality across a large set of operations. This aspect ensures that all public methods of the `com.xerox` package log any errors they throw to their caller. The `publicCalls` pointcut captures the public method calls of the package, and the after advice runs whenever one of those calls returns throwing an exception. The advice logs the exception and then the throw resumes.

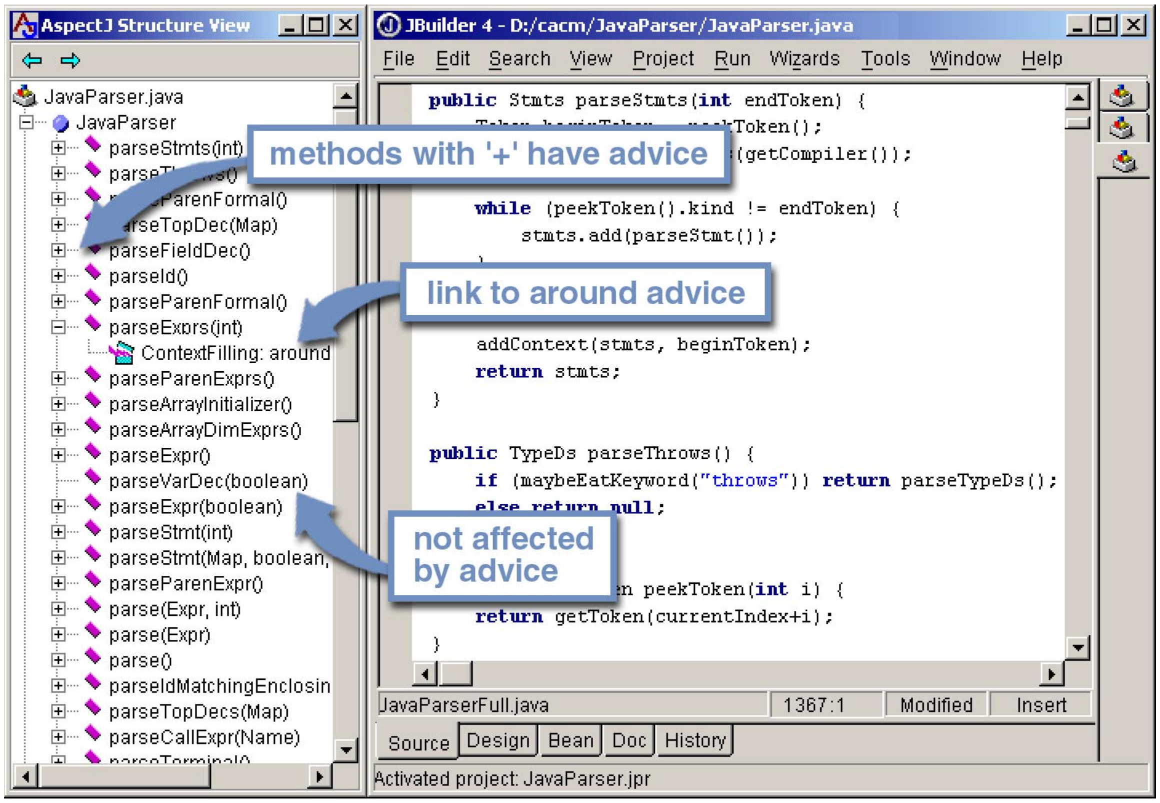
```
aspect PublicErrorLogging {
  Log log = new Log();

  pointcut publicMethodCalls ():
    calls(public * com.xerox.*.*(..));

  after() throwing (Error e): publicMethodCalls() {
    log.write(e);
  }
}
```

In some cases this aspect can log an exception twice. This happens if code inside the `com.xerox` package itself calls a public method of the package. In that case this code will log the error at both the outermost call into the `com.xerox` package and the re-entrant call. The cflow primitive pointcut can be used in a nice way to exclude these re-entrant calls:

```
  after() throwing (Error e): publicMethodCalls() &&
                              !cflow(publicMethodCalls()) {
    log.write(e);
  }
```



```
public Stmts parseStmts(int endToken) {
    Token beginToken = ...kToken();
    ...(getCompiler());

    while (peekToken().kind != endToken) {
        stmts.add(parseStmt());
    }

    addContext(stmts, beginToken);
    return stmts;
}

public TypeDs parseThrows() {
    if (maybeEatKeyword("throws")) return parseTypeDs();
    else return null;


...n peekToken(int i) {
    return getToken(currentIndex+i);
}
```

Figure 3. Examining the partially re-factored code with the AJDE extension to JBuilder. The method names marked with lightning? Are advised by the aspect. This view allows us to quickly see the structure of this larger aspect. Being able to see this perspective of the aspect is very helpful to designing and debugging property-based aspects like this one.

The following aspect is taken from work on the AspectJ compiler. The aspect 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`.

```
aspect ContextFilling {
  pointcut parses(JavaParser jp):
        calls(* jp.parse*(..) &&
       !calls(Stmt parseVarDec(boolean)); // var decs
                                          // are tricky

  around(JavaParser jp) returns ASTObject: parses(jp) {
    Token beginToken = jp.peekToken();
    ASTObject ret = proceed(jp);
    if (ret != null) jp.addContext(ret, beginToken);
    return ret;
  }
}
```

This example exhibits a property found in many aspects with large property-based pointcuts. In addition to a general property based pattern – `calls(* jp.parse*(..))` – it includes an exception to the pattern – `!calls(Stmt parseVarDec(boolean))`. The exclusion of `parseVarDec` happens because the parsing of variable declarations in Java is too complex to fit with the clean pattern of the other `parse*` methods. Even with the explicit exclusion this aspect is a clear expression of a clean crosscutting modularity. Namely that all `parse*` methods that return `ASTObjects`, except for `parseVarDec` share a common behavior for establishing the parse context of their result.

The process of writing an aspect with a large property-based pointcut, and of developing the appropriate exceptions can clarify the structure of the system. This is especially true, as in this case, when refactoring existing code to use aspects. When we first looked at the code for this aspect, we were able to use the IDE support provided in AJDEforJBuilder to see what methods the aspect was advising as compared to where we had manually previously manually coded the functionality. We used the AJDE structure view shown in Figure 3 and scrolled through the code. We quickly discovered that there were a dozen places where the aspect advice was in effect but we had not written the manual coding of the context functionality. Two of these were bugs in our prior non-AOP implementation of the parser. The other ten were needless performance optimizations. So in this case refactoring the code to express the crosscutting structure of the aspect explicitly made the code more concise and eliminated latent bugs.

5 Conclusion

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.

Adoption of AspectJ into an existing Java development project can be a straightforward task and incremental task. One path is to begin by using only development aspects, going on to using production aspects after building up experience with AspectJ. Adoption can follow other paths as well. For example, some developers will benefit from using production aspects right away.

AspectJ enables both name-based 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 can have small or large scale.

Using AspectJ results in clean well-modularized implementations of crosscutting concerns. When written as an AspectJ aspect the structure of a crosscutting concern is explicit and easy to understand. Aspects are also highly modular, making it possible to develop plug-and-play implementations of crosscutting functionality.

AspectJ provides more functionality than is covered by this short article, but these examples should provide a sense of the kinds of aspects it is possible to write using AspectJ. But we recommend that programmers read the on-line AspectJ documentation and examples carefully before deciding to adopt AspectJ into a project.

6 Acknowledgements

We thank the AspectJ users most of all.

Tom Roeder contributed the initial implementation of parser context aspect. Yvonne Coady, Kris De Volder, Chris Dutchyn, Jan Hanneman and Gail Murphy made extensive comments on early drafts of the paper.

References

1. *Bug-Free O-O Software: An Introduction to Design by Contract*, in *Web page at http://eiffel.com/doc/manuals/technology/contract/page.html*. 1999, Interactive Software Engineering.
2. *The AspectJ Primer*, in *Web page at http://aspectj.org/docs/primer*. 2001, The AspectJ Team.
3. Kiczales, G., et al. An Overview of AspectJ. *15th European Conference on Object Oriented Programming (ECOOP)*. Springer. June 2001.
4. Lea, D., *Concurrent Programming in Java: Design Principles and Patterns*. 2nd Edition ed: Addison-Wesley. 1999.