

# Lessons learned building tool support for AspectJ

Mik Kersten<sup>1</sup>, Matt Chapman<sup>2</sup>, Andy Clement<sup>2</sup>, Adrian Colyer<sup>3</sup>

<sup>1</sup> University of British Columbia, <sup>2</sup> IBM Hursley Labs, <sup>3</sup> Interface 21

beatmik at acm.org, {mchapman, clemas} at uk.ibm.com, adrian.colyer at interface21.com

## ABSTRACT

A key part of the AspectJ technology is the integrated development environment integration, which makes it possible for developers to use aspect-oriented programming without giving up the tool support to which they are accustomed. In this experience report we summarize our current tool suite, and discuss the lessons we learned extending object-oriented development environments to make crosscutting structure explicit. We also report on the challenges of surfacing aspects in structure views without encouraging misconceptions about the language, our successes and failures in extending object-oriented tools, and our ongoing work in supporting advanced IDE features and exposing the dynamic properties of the language.

## Keywords

Aspect-oriented programming, integrated development environments, language design and implementation, software tools

## 1. INTRODUCTION

Object-oriented programming (OOP) tool support makes the inheritance and encapsulation structure of a system explicit. Many programmers rely on query facilities to find overriding methods, tree views to inspect the system's type hierarchy, or debugger mappings from exception traces to offending source lines. AspectJ [10] is an Aspect-Oriented Programming (AOP) [11] extension to Java [7] that provides the programmer with language support for modularizing crosscutting. The goal of the AspectJ tool support is to make the crosscutting structure of the system explicit.

The AspectJ IDE plug-ins have been crucial to the adoption of the AspectJ language for both practical and pedagogical reasons. They allow programmers to use AspectJ without forcing them to give up the other development tools they already use. Subtle but fundamental assumptions in the object-oriented IDEs' structure models and views made it challenging to implement our AOP extensions to these tools. In this experience paper we report the lessons we learned from integrating aspects into existing IDE's models and views. We also discuss the technical issues that we observed while building facilities for editing, compiling, navigating, documenting, and debugging aspect-oriented programs. Section 2 provides an overview of our current tool suite. Sections 3 & 4 discuss what we learned designing and implementing these tools.

## 2. THE ASPECTJ TOOL SUITE

The first tool we released was a command line compiler. As our user community grew the number of requests for build and development environment integration increased. Users wanted the ability to invoke the AspectJ compiler from within their development environment [9]. In addition, they asked IDE integration that provided the editing and navigation facilities that they were accustomed to in their Java IDEs. In 1999 we set out to design and integrate IDE facilities for working with crosscutting structure.

The first step we took was to show crosscutting structure as navigable textual annotations in Emacs<sup>1</sup>, and as links in Javadoc [6] documentation generated with the ajdoc tool. We met the increasing user demand with plug-ins for the JBuilder<sup>2</sup>, NetBeans<sup>3</sup>, and Eclipse<sup>4</sup> IDEs. Since we were not able to support all IDEs used by our community, we also provided a standalone tool called the AJBrowser for navigating aspects, and an Ant<sup>5</sup> task for integrating the AspectJ compiler into an existing build process. Figure 1 provides a historical perspective on the tool support we released as the language matured and demand grew. Intervals on the timeline indicate periods of active contribution to those tools. Section 4.4 discusses why some tool lasted while others did not.

### 2.1 AspectJ

AOP enables the modular implementation of crosscutting concerns. Such concerns are inherent in complex systems, and are impossible to capture cleanly with OOP. AspectJ is an extension to Java that provides the programmer with language mechanisms that explicitly capture crosscutting structure. The result is similar to the benefits of OOP modularity for object encapsulation and inheritance: easier to maintain code with greater potential for reuse.

Consider the failure handling policy in Figure 2. This simple aspect declares that after a `WSIFException` is thrown in any public method within the `org.apache.wsif` package the exception will be handled by the body of the advice (line 7). The aspect defines a `pointcut` as the set of join points corresponding to every execution of a public method within that

---

<sup>1</sup> <http://www.gnu.org/software/emacs>

<sup>2</sup> <http://www.borland.com/jbuilder>

<sup>3</sup> <http://netbeans.org>

<sup>4</sup> <http://eclipse.org>

<sup>5</sup> <http://ant.apache.org>

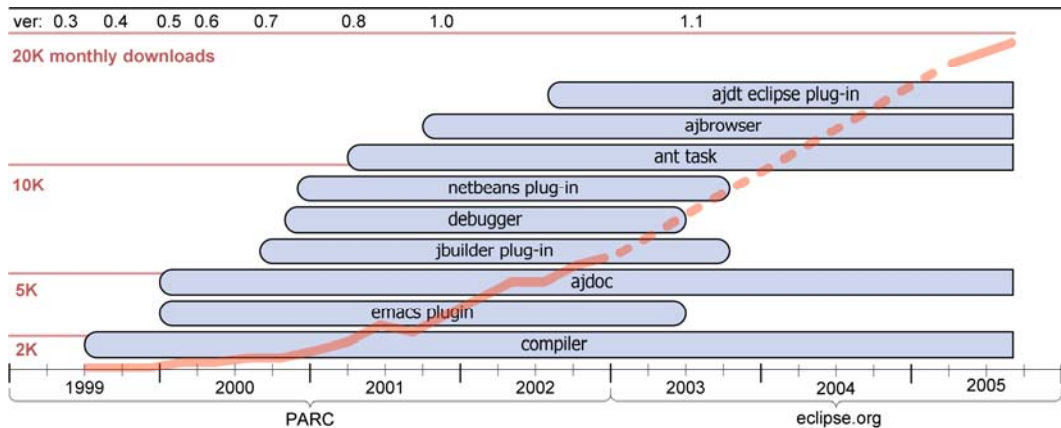


Figure 1. Timeline of AspectJ tool releases (dashed line indicates missing statistics)

package. In AspectJ's pointcut and advice mechanism, join points are key points in the dynamic execution of the program [13]. These include method and construction execution, class initialization, and field accessors. The `FailurePolicyEnforcement` aspect uses `after` advice to define what action should be taken under these points in the execution. Advice execution can also be specified to happen `before` and `around` a join point. Aspects as simple as this one have been demonstrated to be expressive enough to capture crosscutting concerns such as first failure data capture policies in application servers [2].

OOP implementations corresponding to Figure 2 result in the exception handling code being scattered and tangled across the system. By making these concerns explicit, AspectJ provides the expected benefits of good modularity for crosscutting concerns. The goal of the AspectJ tool support is to show the structure of well-modularized crosscutting concerns.

## 2.2 Crosscutting structure views

The most common objection that arose during our tutorials and presentations suggested that AspectJ's ability to introduce behavior invoked implicitly would compromise program understandability. For example, `around` advice can prevent a method from executing. Following references or imports from the containing file does not help the programmer figure out when this might happen, since aspects crosscut the type structure. Without tool support the only way to know how

aspects affect a particular method is to examine all of the aspects. To address this, the AspectJ tools make crosscutting structure explicit by indicating which advice will affect that method. For example crosscutting links in the Cross References view make it possible to navigate from a method signature to advice that may affect the execution of that method (Figure 2, right hand side). These links are also exposed as inline annotations in the editor ruler (Figure 2, left hand edge). Right-clicking on the annotations displays a context menu with the same relationships. The relationships are shown in both directions, to allow consistent navigation between the source and target of advice.

AspectJ's inter-type declarations, an open class mechanism that originated from Flavors [12], affect the type structure of the system. For example, a class may have new members declared on it. This structure is made explicit in the Cross References view as well (Figure 3). Unlike Java member declarations, advice and some inter-type declarations cannot be invoked explicitly. As such, these declarations are not named. In the Cross References view they are distinguished by their kinds and by the pointcuts that they reference.

When beginners start prototyping aspects their first challenge is to understand the places that the aspect affects. Advice affects the execution of join points. But whereas join points exist in the runtime call graph, the program elements that show up in IDE views correspond to static structure. As a result, the AspectJ structure views indicate all elements that could be effected by

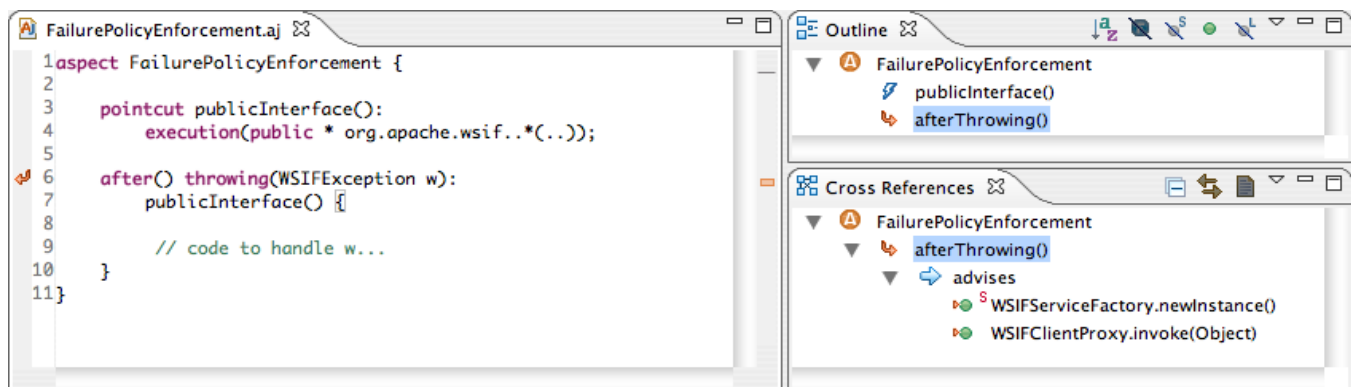


Figure 2. Advised-by annotation in the editor and crosscutting relationships in the Cross References view

advice execution. We call these relationships between static program elements and join points the *join point shadow*. The join point shadow shows how advice affects program elements, and can be thought of as a projection of the dynamic execution of the program onto the static structure. To make it clear where the execution of advice will be decided by a runtime test (e.g. in the case of control flow advice) a question mark is added to the advice icon (Figure 4).

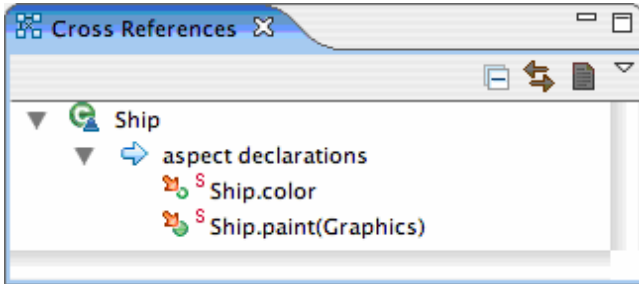


Figure 3. AspectJ declarations in the Cross References view

Some pointcuts crosscut very large portions of the system. Tree views are not effective for showing these because it is difficult to maintain context when scrolling through the hundreds of nodes that can populate the view. The Aspect Visualizer (Figure 5) addresses displaying the global effects of crosscutting by showing advice relationships in a zoomed-out SeeSoft source line-based view [5] inspired by the Aspect Browser [8]. The vertical bars represent source files, with height corresponding to file length. Each line affected by advice is colored according to the aspect-to-color mapping in the right-hand list. The Visualizer can display the crosscutting for an entire project, zoom into a single package, and navigate to the corresponding source code in the editor.

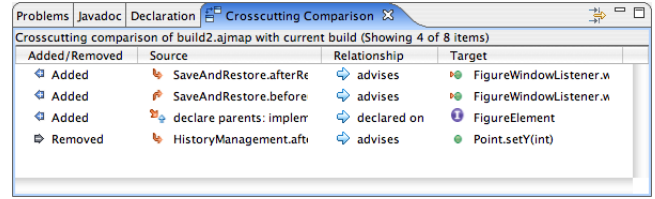


Figure 6. Crosscutting Comparison view

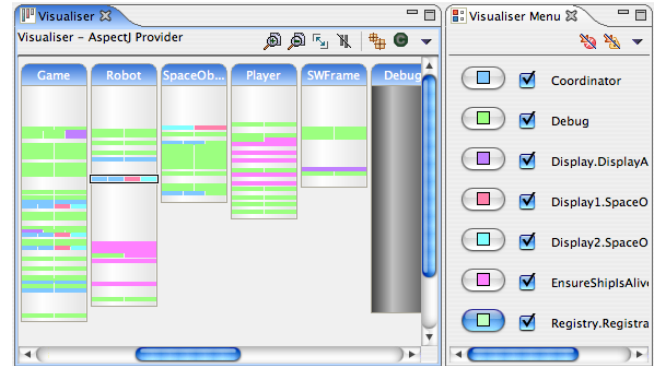


Figure 5. Aspect Visualizer

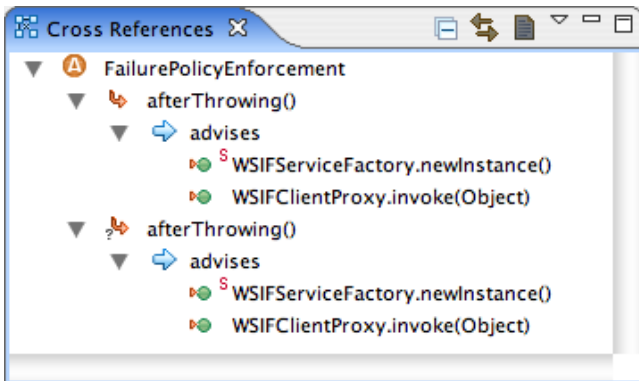


Figure 4. Advice with and without a runtime test

Java developers are accustomed to structured ‘diff’ views that indicate what parts of the object-oriented program have changed. As an aspect-oriented program changes, the changes in pointcuts can have wide-reaching impact on the system. For example, refactoring pointcuts may result in the associated advice affecting more join points than was intended, particularly if some of those join points were added by another team member. The Crosscutting Comparison view (Figure 6) addresses this by comparing the latest version of the program against a check point.

## 2.3 Build and editor support

Since we extended the Java language, the AspectJ language and compiler needed to be compatible with the Java platform. All legal Java programs are legal AspectJ programs, and the bytecodes produced by the ‘ajc’ compiler can run on any Java 2 and later VM. Aspects can be declared in both ‘.java’ and ‘.aj’ source files. Pointcuts can be declared in classes as well as aspects, and inner aspects can be declared in classes. As a result, crosscutting modularity is as primary and visible to the AspectJ programmer as object-oriented modularity, and the AspectJ tools’ role is to present a consistent view of both.

Integrating with the build process means that the AspectJ compiler provides semantic errors and warnings using the same problems list mechanism that IDEs use for Java errors. Since modern IDEs provide eager feedback on syntax errors, AspectJ editor support provides this for aspect members such as pointcuts and advice (Figure 7).

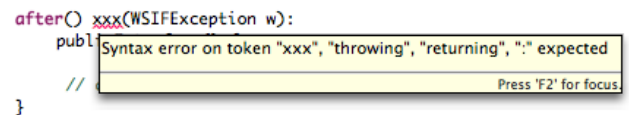


Figure 7. Errors as you type in the AspectJ editor

Additional integration between the compiler and the editor includes code formatting, the organizing of imports, and content assist (Figure 8).

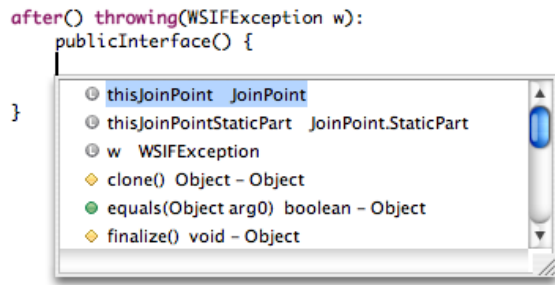


Figure 8. Content assist in the AspectJ editor

### 3. LESSONS LEARNED DISPLAYING CROSSCUTTING STRUCTURE

To expose crosscutting structure, OOP views needed to be extended to show the effects of aspects. Inheritance and encapsulation have clear representations in static structure views, but the core of the AspectJ structure model is dynamic join points. We struggled with ways to model and display an accurate aspect-oriented view of the program. This section reports the key lessons learned from trying to model and display crosscutting structure.

#### 3.1 Crosscutting views made aspect-oriented programs easier to understand

User feedback from tutorials, workshops, and our mailing lists has indicated that the single most effective role that the tools have played is to expose the crosscutting relationships of the program. In contrast to OOP languages which have been successfully used with plain text editors, AspectJ depends heavily on tool support. A developer reading code written in an OOP language can understand the structure of the program by reading the text and following references. A developer reading code written in an aspect-oriented language can not infer the program from local examination of the code, since the crosscutting of aspects can affect the execution of that code. The increasing popularity and reliance on IDEs allowed us to leverage graphical views to make AOP program structure explicit (Figures 2-8).

The early releases of AspectJ IDE support made our users' reliance on the crosscutting structure views clear. At one point a bug prevented the views from displaying certain forms of call site advice, and resulted in confused reports asking if the language semantics had changed. During tutorials we often observed users learning the semantics of AspectJ by inspecting how the structure views show the effects of their first aspects. We have also observed users learning the syntax of pointcuts by inspecting the keyword highlighting and early error indication feature of the AspectJ editor (Figure 7).

#### 3.2 New navigation techniques were required to expose crosscutting

Object-oriented views serve well for showing hierarchies and navigating references. But aspects are about non-hierarchical structure, and the presence of aspect-oriented structure in large

systems can overload the tree views with relationships and links. In Eclipse the editor gutter annotations (Figure 2) suffer from a related problem. The gutter is narrow and only capable of displaying a single icon per source line. If additional annotations are present (e.g., breakpoint indicators) they occlude the advice annotation.

Since the Java IDEs showed program structure in tree views, we needed to extend these views to show crosscutting. But the advice affecting a method could not just appear as a regular child node of the method, since it relates through crosscutting and not by containment. Initially this resulted in our adding relationship nodes (e.g., "advised by") to tree views such as the Outline. To reduce the visual complexity and overload of those views when working on large systems, we moved the relationships out from the structure view and into the Cross References view (Figure 2). A keyboard shortcut also makes it possible to temporarily overlay this view on the editor, for example when a gutter annotation indicates the presence of advice.

Aspects are inherently good at expressing the global properties of a system, whereas object-oriented views tend to focus more on containment and information hiding. Changing a single pointcut can result in every public method of the system being advised. To show this global structure, the Aspect Visualizer (Figure 4) presents a cross-file or cross-package view of the crosscutting. The Crosscutting Comparison view (Figure 5) shows changes to the crosscutting structure between different versions of a project. This allows the programmer to see the crosscutting in the system not limited to a localized portion of the code, and shows the effects of refactoring a pointcut without needing to navigate to other pointcuts or advice.

We also had to provide facilities for helping the programmer maintain context when navigating crosscutting. When updating the body of an advice, the behavior of each of the join points it affects can be of interest. IDEs make it easy to lose context by presenting only the structure relevant to the current focus of the editor. To offset this we first introduced navigation history. Our tree links acted as hyperlinks and navigation could be stepped back and forward (most IDEs now support a similar history feature). This helped, but it did not solve the main problem of needing to see both aspect and affected join point at the same time. The Cross References view addresses this by maintaining a structure view focused on the crosscutting while allowing a second structure view, such as the Outline view in Eclipse, to be synchronized with the editor.

#### 3.3 The tools taught both concepts of and misconceptions about the language

Early on in the development of the IDE support we decided to use Fluid Document technology [4] to present the effects of advice. Figure 9 shows how our prototype made the code of an advice appear at the applicable join point shadow with an animation that fluidly displaced the original code in order to prevent the programmer from losing context by being forced to navigate to the advice. This approach had the benefit of showing both the advice code, and the context in which it would execute. Unfortunately, this also gave the incorrect impression that AspectJ used preprocessor semantics and inserted code into a method, whereas advice does not execute in the same scope as

the method. As a result we did not proceed with this approach. This approach could have potential if combined with a mechanism that shows the separation of scope and the dynamic test that controls the execution of the advice. There is a related caveat with the Aspect Visualizer view (Figure 5) which encourages the developer to think in terms of source lines instead of dynamic points in the execution.

```

void fire() {
    if (!expendEnergy(BULLET_ENERGY))
        return;
    new Bullet(getGame(), xV, yV);
}

void fire() {
    if (traceMethods.getState()) {
        infoWin.println(thisJoinPoint.getSignature());
    }
    if (!expendEnergy(BULLET_ENERGY))
        return;
    new Bullet(getGame(), xV, yV);
}

```

Figure 9 shows the fluid in-lining of a before advice body. The top code block is the original method. The bottom code block shows the method with a before advice body inlined. Annotations include: 'indication of advice' pointing to the if statement in the original code, 'expansion' pointing to the expanded method body, and 'body of before advice' pointing to the inlined advice code.

Figure 9. Fluid in-lining of a before advice body

Making it clear that AspectJ is based on a dynamic model has been the biggest challenge of displaying the structure of AspectJ's crosscutting mechanisms. When developers think in terms of static transformations of source code, they do not learn how to understand and work with aspects as a first-class structure of the system. We have repeatedly observed the preprocessor misconception preventing programmers from attaining an intuitive understanding of AspectJ's semantics. For example, it is awkward to think of join point parameter binding in terms of method parameters. As a result, a driving goal of crosscutting structure views was to discourage notions of inserting code.

### 3.4 Surfacing the dynamic properties of crosscutting was difficult

The current tools give the developer no assistance for determining what dynamic conditions will affect the potential execution of advice at a join point shadow. Multiple advice can apply to a single join point. Their execution is specified by ordering rules, or explicitly declared by the programmer. But the structure views do not surface the ordering semantics. There was no simple way to extend the object-oriented views to show this information, and it has remained an open problem<sup>6</sup>.

Part of the problem stems from the fact that the IDEs' views show static structure. While much pointcut matching is static, which is why AspectJ is efficient, some matching has runtime tests (e.g., when a pointcut constrains the join points to only those within the control flow of a particular method execution). We use join point shadows (Section 2.2) to surface all of the places that advice might affect. For example, both method signatures and call sites are of interest for advice on calls. To

<sup>6</sup> For potential solutions see <http://eclipse.org/ajdt/ui.html>

make the programmer aware of dynamic tests we annotate the relationship with a question mark indicating the presence of the test (Figure 4). But the programmer is left to infer what runtime conditions need to be true for the advice to execute. The crosscutting structure views need to become more specific about providing context indicating what runtime tests and conditions will affect the execution of the advice (e.g., by showing the call graph for the control flow constraining a pointcut). Exposing the dynamic properties of crosscutting will involve extending our structure model, which like the IDE's structure models has focused on representing structure that can be easily mapped to source code.

### 3.5 A crosscutting structure model was key, but over-generalizing it was a mistake

From the beginning of the project our use of agile methods helped the tools evolve along with the changing language implementation and IDE platforms. The tools framework grew out of a single IDE implementation, to one generic enough for two Swing<sup>7</sup>-based IDEs, and finally to a GUI-independent framework when we needed to support Eclipse's SWT toolkit.

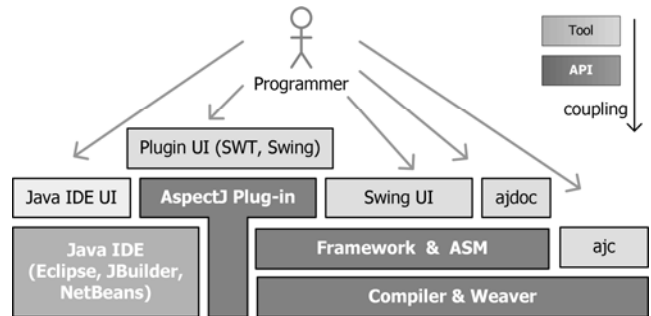


Figure 10. AspectJ tools framework and clients.

The AspectJ Structure Model (ASM) is at the core of the tools framework on which the IDE plug-ins are built. ASM clients expose the model in task-specific views. Unlike the compiler's abstract syntax tree (AST), the model is kept in memory since the crosscutting structure of the entire system must be presented to the user without invoking a search. If the user switches build configurations it is saved to disk. Figure 10 shows how our tools can extend the ASM directly, as ajdoc does, or extend the framework to provide views specific to the look-and-feel of the host IDE. The Framework and ASM have also been extended by someone outside of the core AspectJ team in order to provide support for the JDeveloper IDE<sup>8</sup>.

The ASM represents the crosscutting, inheritance, and referential structure of AspectJ programs (Figure 11). Since structure views represent static program structure, the structure model is a projection of AspectJ's dynamic properties onto the static structure of the system—i.e. from the join points to the join point shadows. The example in Figure 11 depicts pointcut

<sup>7</sup> <http://java.sun.com/products/jfc>

<sup>8</sup> <http://jdeveloperop.dev.java.net>, created in 2004

and advice relationships. Note that the pointcut does not point to affected members, since a pointcut alone does not have a behavioral effect on the program.

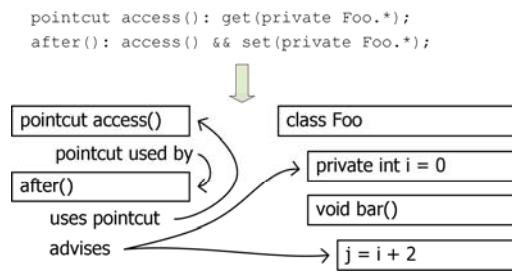


Figure 11. Example of ASM relations

The ASM started out as a string-based map of correspondences between declarations. To incorporate structural relationships for inter-type declarations and other kinds of join point shadows we built a generic data structure composed of program elements and associations relating them. This graph was used for constructing containment, inheritance, referential and crosscutting views. It did not contain a dominant decomposition (the containment hierarchy in most IDEs) and did not require searches to build structure views. It also captured additional relationships, such as the “@see” links from Javadoc. It was intended to be extensible to UML<sup>9</sup>-specific associations or relationships between program elements and structure declared in external resources (e.g., XML<sup>10</sup> files in J2EE<sup>11</sup>). We also started down the road of making the ASM general enough to support a wide range of AOP tools and languages. However, over-generalizing in this way turned out to be a mistake.

The latest ASM implementation is, once again, a string-handle-based mapping between elements in a containment hierarchy. The more flexible and extensible generic data structure attempted to solve interesting problems, but was not practical for solving the core problem of supporting high-quality IDE integration. The lack of a dominant hierarchy resulted in multiple graphs representing the entire program structure and causing an excessive memory footprint when used with systems larger than ten thousand classes. In addition, to achieve a deep integration with the Eclipse IDE the update of the structure model needed to happen eagerly as the user edited. In order to make the ASM support incremental update and improve performance we concretized it and returned to using string handles as references to program elements. The relationships still persist in memory, but as a much leaner and easier to incrementally update map between handles corresponding to program elements.

## 4. LESSONS LEARNED EXTENDING OBJECT-ORIENTED IDEs

Modern IDEs provide the programmer with a lot of support for editing and navigating object-oriented programs. We needed to extend the existing functionality in a consistent and elegant way to support AspectJ without getting in the way of the Java tooling. In building this support we iterated through three levels IDE integration, and each of our tools evolved along the path defined by Table 1.

Table 1. Levels of IDE integration

Integration	Goal
1. Invocation & resources	Invoke the AspectJ compiler on project resources and display compiler messages.
2. Editor & views	Provide a custom editor for aspects and new structure views.
3. Structure model	Integrate with the IDE’s structure model, editor, parser, and views, and add new crosscutting-centric views.

The *invocation & resources* integration was our first improvement over the command line compiler and text editor interface. This involved mapping the IDE’s definition of projects and paths to command line parameters understood by our compiler. This level of integration is still used in the form of Ant support by those who do not have IDE support for AspectJ (e.g. IntelliJ users).

The *editor & views* integration improves on the previous level by providing an editor that understands aspects and can support features like keyword highlighting, and views that show crosscutting structure. The JBuilder, NetBeans, and early AJDT plug-ins offered this level of integration. For example, the Outline view in these looked very similar to the IDE’s Outline view, but was a replacement that showed AspectJ declarations and crosscutting relationships. But this approach is fundamentally limited because the advanced IDE features that rely on the IDE’s parser, compiler, and structure model are not supported, and views that mimic the IDE’s views are never up-to-par with the user’s expectations.

The *structure model* integration extends the core model of the IDE to understand AOP semantics. AJDT provides this, and exposes it in features such as eager parsing and content assist. But deep integration is challenging due to the Java-language specific bindings and assumptions made by the IDE’s structure model. As a result, as of the writing of this paper AJDT does not yet integrate deeply enough to offer refactoring support, which is now a commonly expected feature in Java IDEs.

While working through these levels of integration we have struggled with extensibility limitations that result from AOP breaking the core assumptions made by object-oriented IDEs. OOP is about encapsulation and hierarchies, whereas AOP is about structure that crosscuts encapsulation and hierarchies. As a result the underlying data structures of AOP and OOP tools are fundamentally different. In this section we report on the strategies that we developed for providing a clean integration of AOP into OOP IDEs.

<sup>9</sup> <http://www.uml.org>

<sup>10</sup> <http://www.w3.org/xml>

<sup>11</sup> <http://java.sun.com/j2ee/>

## 4.1 Overriding the file extension broke the IDEs' extensibility model

AspectJ is intended to be a seamless integration of AOP and Java. The file extension for AspectJ sources seems like a seemingly small detail, but turns out to have significant implication on the degree to which the integration is seamless. One of the core goals of the AspectJ language is to make crosscutting mechanisms available as a part of the base language. This goal is in contrast to the reflective and XML/annotation-based approaches (e.g., AspectWerkz<sup>12</sup>) in which crosscutting is declared outside of the main language. AspectJ's approach benefits the programmer by providing consistent support for both objects and aspects. However, current IDEs only allow for language extensibility outside of the core program text (e.g., in comments, metadata tags, strings, and new file extensions). In any file named ".java" the tools expect a language that conforms to the Java language specification.

The easiest way to address this is to only allow AspectJ code in separate resources (e.g., ".aj" files with the current syntax or ".xaj" with XML syntax). But this would have only side-stepped the problem. For example, consider an inter-type declaration made in a separate resource. The tool support could try to make the resulting structure clear, but there would still be a confusing and awkward disconnect between the external code for the inter-type declarations and the pure Java declarations. A reference to an inter-type declaration in a ".java" file would be a compiler error. The AspectJ language and tool support makes crosscutting a primary part of the system's architecture and allows aspect declarations in ".java" files.

AspectJ 5 offers a compromise called the @AspectJ style (vs. the code style of ".java") first introduced by AspectWerkz. The @AspectJ style allows declaration of aspects and aspect-members using pure Java syntax by exploiting Java 5 style annotations. This is less disruptive to tools expecting pure Java syntax inside ".java" files, but does not alleviate the major requirement to show crosscutting structure. The @AspectJ style was released recently, and we are still gathering feedback on this approach. But a key enabler is the fact that AJDT can present crosscutting consistently in both styles, and even allow toggling between one style and the other by rewriting the aspect.

## 4.2 No IDE was inherently extensible to AOP

This may seem at odds with the fact that we built IDE plug-ins. But the AspectJ plug-ins do not directly extend the OO structure model of the IDEs. They either replace it or layer onto it. When contrasted with the efforts behind commercial Java tooling, the resources available to the AspectJ project have made this catch up game a slippery slope. Take for example the Eclipse IDE, which offers no facilities for extending its Java model with new semantics. The structure model is the foundation of features such as eager parsing, code assist, and refactoring. Both the core Java tool support and 3<sup>rd</sup> party extensions contain explicit bindings to the Java language, and as such do not inherently

support language extensions. Taking the external-language approach is less intrusive to the other Java tooling (Section 4.1). But this only delays the problem by pushing the aspect-oriented structure out of the way. IDEs' structure models needs to be made semantically extensible to other program structures in order to cleanly integrate crosscutting. So far we have only to layer on this sort of extensibility to the Eclipse IDE, which is open source and has a rich model of Java Structure. As a result, the core AspectJ team's integration efforts are focused on Eclipse.

## 4.3 Semantic extensibility requires openness and structure-aware features

Our first IDE plug-in after Emacs extended the Microsoft Visual J++ 6 IDE<sup>13</sup>. Its extensibility APIs are similar to those currently available in VisualStudio.NET. However, the IDE was closed-source and not self-hosted on the APIs (i.e. the internal implementation was not based on them). It was not possible to extend the IDE beyond the limited use cases that the IDE's developers had planned for. The release of the pure-Java JBuilder 3.5 was promising. We released the first IDE plug-in on JBuilder's much broader open APIs. The breadth and stability of these APIs helped bring AspectJ into the hands of real developers [14], but the APIs were not designed for incorporating new modularity ideas and language extensions. JBuilder is not open source, and was missing critical extension points. For example, to make the file structure view work we had to walk the Swing component tree, and overwrite the Java structure specific view.

The first NetBeans release was encouraging since the IDE was open source. Open sources proved invaluable for figuring out how to extend the IDEs in a way that was not originally planned. However, due to an overly general architecture and not enough focus on extensible Java tooling it was more difficult to build the NetBeans plug-in than to build the JBuilder plug-in. The Eclipse 2 release combined openness, broad APIs, a much deeper self-hosting on those APIs, a more complete plug-in component model, and more features for viewing and manipulating OOP structure. Eclipse was also the only IDE to provide an open source compiler, AST, and structure model, which offered opportunity for a deep integration. However, the advanced tool features of Eclipse are a double-edged sword for language extensions. They set the feature bar very high. For example, Eclipse users new to AspectJ often expect refactoring to work for AspectJ as seamlessly as it works for Java.

## 4.4 The cost of supporting multiple IDEs was justified by a broad outreach

We released support for 7 versions of JBuilder, 5 versions of NetBeans, and 4 versions of Eclipse. Supporting multiple IDEs was costly. But it helped establish AspectJ as standard for AOP on Java rather than as an extension to a single IDE, and helped ensure that the command line compiler remained de-coupled from any IDE. Also, we could not initially choose one IDE up-front because we did not know what our users' platform of

---

<sup>12</sup> <http://aspectwerkz.codehaus.org>

---

<sup>13</sup> <http://msdn.microsoft.com/vjsharp>

choice would be. More recently we have prioritized deep integration with a single IDE, but have maintained the APIs that make extensibility with other IDEs possible (Section 3.5).

The porting of plug-ins to newly released APIs of a given IDE has been more straightforward than we expected. Our main problem has been maintaining a single release that is backwards compatible with older releases. Users want the latest bug fixes even if they are stuck on an older version of the IDE, while others expected immediate support for new IDE releases. Supporting both the latest and older versions imposed a drag on evolution, was time consuming, and involved marginal solutions such as checking the IDE version at runtime and invoking the appropriate API call reflectively. The only complete solution is to maintain multiple release streams, one for each IDE release, and this is the approach we have resorted to now.

One good domain for AOP is enterprise applications [3]. To enable AspectJ use for these developers we needed to support the enterprise versions of the IDEs (JBuilder® Enterprise Edition, Sun Java Studio Enterprise built on the NetBeans platform, and IBM® Rational® Application Developer for WebSphere® Software built on the Eclipse platform). In theory, this should not have required extra work since all plug-ins making correct use of the extensibility model should interoperate with the extended enterprise IDEs. However, testing and updates specific to the enterprise editions were necessary to ensure compatibility. This additional work was necessary, in part because enterprise project configurations involve more kinds of resources, and because the enterprise editions can depend on core platform features that are not used by the standard edition. Unlike the early adopters who always downloaded the latest free standard version of their IDE, the enterprise developers are often stuck on a version until their organization decides to upgrade. Another problem was that we did not use the enterprise versions of the tools internally. Close dialog with enterprise versions users helped us understand the expectations they had of the integration.

## 4.5 Integrating with existing UIs was more important than creating new ones

Numerous IDE views can be affected by the presence of aspects, for example:

- Document outline: additional members
- Content assist: additional members and special join point variables
- Inheritance tree: additional super types can be declared by aspects
- Debugger thread tree: additional stack frames appear for generated methods

Our bug report database indicates that the most important part of clean integration is not getting in the way of existing Java tool support, but showing the relevant crosscutting information when needed. New users curious to try AspectJ are not willing to continue using it if they have to sacrifice their existing Java support. To further improve integration quality we set a policy of zero-configuration after install (e.g. by automatically configuring preference settings). In addition, we had to provide IDE-specific UIs for toggling whether or not AspectJ was

enabled for a particular project. Although effort spent on improving integration was at the expense of adding features it was critical for getting feedback from use on real systems. For example, the first release of JBuilder support with zero-configuration resulted in many more bug reports than we had ever received for the tool. This made us realize that users were not bothering to submit bugs if they did not get far when first trying the tool.

We started by making AspectJ features as visible as possible (e.g., there was an AspectJ menu that provided build commands). As our plug-ins improved, access to AspectJ features was integrated with the corresponding Java features. In general, extending existing views worked better than adding new ones. We were able to populate the standard Outline view with the structure of aspects, instead of using a custom version of the Outline view which did not look or behave identically to the standard version provided by the IDE. The new structure is placed inline with Eclipse's editing and navigation features, and indicates when the user might want to pop up a view or menu indicating the crosscutting structure. We only provide additional views (Figures 4-6, Cross References, Crosscutting Comparison and Visualiser) to support crosscutting-centric navigation of inspection of the system.

## 4.6 Integrated debuggers were more extensible than expected

In 2001 we released a JPDA<sup>14</sup>-based debugger with a command-line and GUI interface. However, we were not able to get the quality and features of the debugger up to the expectation set by debuggers in existing IDEs, and discontinued the standalone debugger after 1.0. Extending existing debuggers turned out to be easier than extending the IDE's core structure model because of the extra extensibility that results from requirements on debuggers to support breakpoints in non-Java languages. For example, to support Java Servlets, a debugger must map bytecodes to corresponding Java source embedded in ".jsp" files (JSR-45<sup>15</sup>).

We were able to support the use of standard Java debuggers on AspectJ by encoding the source line mappings in the bytecodes. However, Java debuggers expose the implementation of the AspectJ language instead of the language semantics. For example, extra frames corresponding to advice call-outs show on the stack. Nevertheless, this allowed us to consider the additional functionality as an additional user interface layer over the existing debugger functionality. User feedback on debugger support has indicated the need to preserve both views. The standard Java debugger views turns out to be useful for the cases where seeing exactly what executes is more important than seeing clean AspectJ language abstractions, for example in resource-constrained environments.

---

<sup>14</sup> <http://java.sun.com/products/jpda>

<sup>15</sup> <http://www.jcp.org/en/jsr/detail?id=45>

## 5. SUMMARY

AspectJ provides tool support that exposes the aspect-oriented structure of programs by extending object-oriented IDEs. In this experience paper we reported the way in which we surfaced crosscutting structure by extending the object-oriented IDEs. From our user community we learned about the need for seamless integration with the IDE, the importance of providing mechanisms for viewing the global effects of aspects and the need for maintaining context when navigating crosscutting structure. Whereas we succeeded at showing the static mapping of join points to structure views, we have not yet managed to fully expose the dynamic properties of crosscutting. We also learned to choose mechanisms for displaying crosscutting carefully, since these have the ability to both teach the language and to encourage misconceptions about it. The most difficult problems we encountered resulted from the lack of extensibility beyond OOP that is an inherent limitation of the tool platforms. But we are continuing to improve the depth and integration of the AspectJ tool support, and in the process hope to provide additional experience on making object-oriented tool platforms more extensible.

## 6. ACKNOWLEDGEMENTS

Thanks to Gregor Kiczales and Gail Murphy in helping select and distill the lessons learned, the AspectJ and AJDT teams who continue to improve the tool support, and currently include Adrian Colyer, Jonas Bonér, Andrew Clement, George Harley, Helen Hawkins, Wes Isberg, Sian January, Mik Kersten, Alexandre Vasseur, Julie Waterhouse Park, and Matthew Webster. Additional thanks go to Erik Hilsdale and Jim Hugunin, former members of the AspectJ team, who played a key role in helping design the tool support, and to our user community whose feedback continues to teach us about AOP.

## 7. REFERENCES

1. Beck, Kent: Test-Driven Development By Example, Addison-Wesley, 2003.
2. Colyer, A., Clement, A., Bodkin R., Hugunin, J.: Practitioners report: Using AspectJ for component integration in middleware. In: Proceedings of the Conference on Aspect-Oriented Software Development (AOSD). Lancaster, UK (2004)
3. Colyer, A., Clement, A.: Large scale AOSD for middleware. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Anaheim, California (2003)
4. Chang, B.W., Mackinlay, J.D., Zellweger, P.T. and Igarashi, T.: A Negotiation Architecture for Fluid Documents. In: Proceedings of the ACM Symposium on User Interface Software and Technology, pp. 123-132 (1998)
5. Eick, S.G., Steffen, J.L., Sumner, E.E.: Seesoft - A Tool For Visualizing Line Oriented Software Statistics. In IEEE Trans. on Software Engineering, Vol. 18, N. 11 (1992)
6. Friendly, L.: Design of Javadoc. In: The Design of Distributed Hyperlinked Programming Documentation (IWHDD). Springer-Verlag, Montpellier, France (1995)
7. Gosling, J., Joy, B., and Steele, G., Bracha, g.: The Java Language Specification. Second Edition. Addison-Wesley, Reading, Massachusetts (2000)
8. Griswold, W.G., Kato, Y. and Yuan, J.J.: Aspect browser: Tool support for managing dispersed aspects. In First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems, OOPSLA (1999)
9. Kersten, M., Murphy, G.: Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-Oriented Programming. In: Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM, Denver, Colorado (1999)
10. Kiczales, G., et al.: An Overview of AspectJ. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland (2001)
11. Kiczales, G., et al.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Finland (1997)
12. Moon, D.: Object-oriented programming with Flavors. In *Conference on ObjectOriented Programming Systems Languages and Applications*, pp.1-8 (1986)
13. Masuhara, H. and Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Springer-Verlag, Spain (2002)
14. Price, R.: Real-world AOP Tool Simplifies OO Development. Java Report, September Issue (2001)