

Aspect-Oriented Programming with AJDT

Andy Clement
IBM UK Ltd
MP 146, Hursley Park
Winchester, Hants. England
+44 1962 816658
clemas at uk.ibm.com

Adrian Colyer
IBM UK Ltd
MP 146, Hursley Park
Winchester, Hants. England
+44 1962 816329
adrian_colyer at uk.ibm.com

Mik Kersten
Intentional Software Corporation
500 108th Avenue NE #1050
Bellevue, WA 98004
+1 425 467 6600
mik at intentsoft.com

ABSTRACT

Tools support has an important role to play in teaching aspect-oriented programming (AOP) by making the crosscutting structure of AOP programs explicit. In this paper we discuss lessons learned from introducing many developers to AOP for the first time, and from developing the AspectJ Development Tools (AJDT) [1] support for Eclipse [2]. To address those lessons we also discuss future plans for improving program understanding and visualization in AJDT.

1. INTRODUCTION

AspectJ [3] is a popular aspect-oriented programming language, and the Eclipse AspectJ Development Tools (AJDT) project provides tool support for editing, building, and debugging AspectJ programs on the Eclipse platform. While leading the development of AJDT we have introduced many people to aspect-oriented programming with AspectJ. As a result, we have come to appreciate that good tool support has a critical role to play both in the development of aspect-oriented programs, and also in assisting newcomers in learning and understanding the concepts and power of aspect-orientation.

In this paper we first describe the current support that AJDT provides for program development and comprehension, and then move on to look at potential future developments that would increase AJDT's power in these areas.

2. PROGRAM COMPREHENSION

AJDT provides several aids to assist programmers in learning and understanding AOP with AspectJ. The most important of these are:

- The outline view and gutter annotations, which show the interaction between advice, inter-type declarations, and other elements in the program.
- The aspect visualizer, which provides a visual overview of the effect of the aspects in the system.
- The debugger, which allows developers to step through AspectJ programs and observe their execution.

2.1 Outline View and Annotations

Figure 1 illustrates the outline view that AJDT provides for the source files in an AspectJ program. For advice within an aspect, the outline view shows the places in the program that will be affected by that advice. The links are navigable, so clicking on them opens an editor directly at the affected location. Similar links are shown for inter-type declarations. The outline view for a source file containing join points where advice will take effect also contains links back to the affecting

advice (in the example of figure 1 for example, the outline view for Line.setP1() will contain a "method advised by" node with a link to the HistoryManagement aspect).

For both new and experienced AspectJ programmers, the outline view is an essential programming aid and feedback tool that can be used to verify that a piece of advice is matching in all the join points it was intended to match, or to give the programmer an early warning that the pointcut is not matching what was intended. This is especially important when programmers are learning the pointcut language.

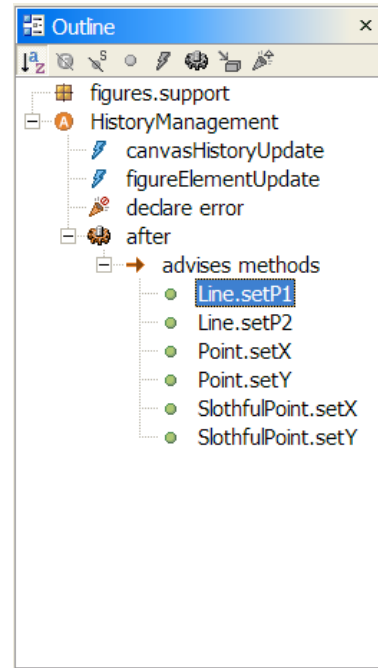


Figure 1 - Outline View in AJDT

The counterparts to the outline view are the in-place gutter annotations displayed in the margin of the editor at the sites affected by advice (see Figure 2).

The hover help for the annotation markers displays the affecting advice and aspect. Using a context menu of the annotation marker, the user can navigate to the affecting advice. The markers give a visual cue to the programmer that one or more aspects interact with the program elements being displayed by the editor.

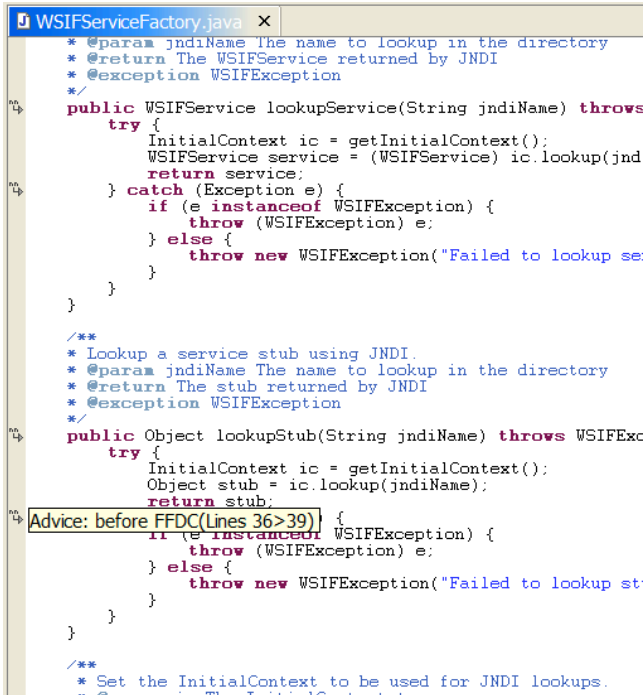


Figure 2 - Gutter Annotations

2.2 Aspect Visualizer

The most powerful tool AJDT provides for understanding the impact of a set of aspects across the whole system is the aspect visualizer. We have found that explaining the value of aspect-orientation to developers through talks and presentations achieves a good degree of comprehension, such that they can understand and explain what a language like AspectJ is capable of. In the abstract, these claims sometimes appear to be too good to be true - but when the audience sees a visual representation of the aspects by looking at the output of the visualizer then interest and excitement really start to take hold.

Figure 5 on page 6 shows the Aspect Visualizer in action. Each aspect in the system appears in the visualizer menu with an associated colour. Visualisation of individual aspects can be turned on or off using the checkboxes in the menu. The main visualization uses a view similar to that produced by the Aspect Browser [4], and the logging example chart used in the AspectJ tutorials, except that rather than showing where scattering exists in a system, it shows all the places affected by advice. Using Seesoft's graphical notation [5] each bar represents a source file in the system; the length of the bar is proportional to the size of the file. Coloured markers appear on the bars to indicate the locations where the program elements in the file are affected by advice. The colour of the bar matches the colour assigned to the aspect in the menu.

To give an overview of the system, the visualizer supports a package mode where all the source files in a given package are folded up into a single bar. It is possible to zoom in and out of the view, and to limit the view to only those classes/packages affected by the selected advice. Double-clicking on an advice marker bar in the visualizer opens the editor on the affected file and line.

2.3 Debugging

Debugging is used not just for tracking down problems, but also for stepping through a program and watching it execute in order to gain an understanding of the program's behaviour. With AspectJ 1.1 and AJDT version 1.1.1 or later it is possible to step through the execution of advice in the debugger window and observe the full flow of the program.

Figure 3 shows the Eclipse debugger in action stepping through the execution of some before advice.

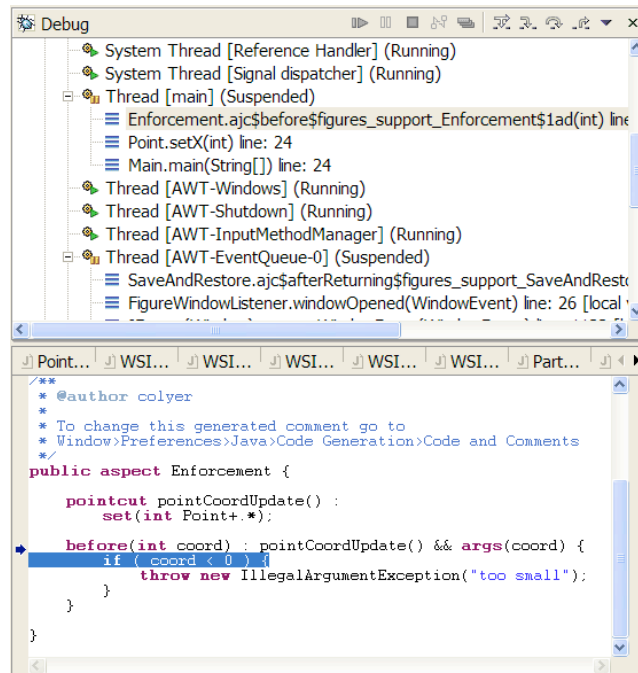


Figure 3 - Debugging Aspects with Source

3. FUTURE DIRECTIONS FOR AJDT

This section discusses future extensions to AJDT that will aid in the development and understanding of AspectJ programs. A complete list can be found at the AJDT project page on Eclipse. Related tool requirements for AspectJ are also discussed in Mik Kersten's paper "Tool Requirements for Commercial Development with AspectJ." [6]

3.1 Tool-tips and Content Assistance

At this stage in the evolution of AOP only a small minority of Java™ programmers have any experience using AOP and AspectJ. For the next few years we can expect that many users of AJDT will be novice or intermediate aspect-oriented programmers. AJDT can do more to help these programmers learn to program in AspectJ:

- * For AspectJ keywords in program source text, the editor should provide a tool-tip containing a slightly expanded version of the text in the AspectJ Quick Reference. This is especially useful for pointcut designators. For example, the tool-tip for the keyword "execution" appearing in a pointcut designator could be "matches if any method with the given signature is executed." And for a "call" pointcut designator the tool-tip would be "matches at the point a call is made to a method with the given signature." This one example would soon clear up novice confusion between call and execution

joinpoints. It should be possible to disable these tool-tips via a preference setting.

- * Eclipse’s Java Development Tools (JDT) provides content assist for Java source that can suggest completions for type, method and variable names by exploiting its knowledge of the program structure and class dictionary. An aspect-aware “Content-assist” facility should offer completions on AspectJ constructs – for example, completions on pointcut names and parameters following the beginning of an advice statement.

- * Parameter hints in Eclipse’s JDT show the expected type and declared parameter names to be passed to the constructor call or method call at the insertion point. AJDT should offer parameter hints for pointcut designators (see for example Figure 4) that show the developer the components of the designator.

```

/**
 * @author colyer
 *
 * To change this generated comment go to
 * Window>Preferences>Java>Code Generation>Code and Comments
 */
public aspect HistoryManagement {

    pointcut canvasHistoryUpdate() :
        call(void Canvas.updateHistory(..));

    declare error: canvasHistoryUpdate() &&
        !within(HistoryManagement) :
        "Only History Mgr should update history";

    pointcut figure [visibility] return-type type-pattern.method-name(arg-pattern)
        execution(void FigureElement+.set*(..));

    after() returning : figureElementUpdate() {
        Canvas.updateHistory();
    }
}

```

Figure 4 - Parameter Hints for execution advice

3.2 Enhanced Debugging Support

The debugging support in AspectJ 1.1.1 does not currently permit the setting of breakpoints inside aspects (you can step into aspects, but not breakpoint in them). Stepping into around advice is also problematic until we have JSR 45 support. These enhancements are needed, but so is a more subtle one: if you look carefully at the stack trace in the debug window of Figure 3 you’ll see that intermediate methods generated by the AspectJ compiler appear. For example, the stack trace entry

```
Enforcement.ajc$before$figures_support_Enforcement$1ad(int)
```

appears in the main thread. This is a compiler implementation detail that shows through to the users of the language and should be suppressed. A common newcomers’ mistake when working with AspectJ is to try and understand it from the perspective of the task the compiler implementation does (weaving), rather than switching to an aspect-oriented mode of thinking that recognizes an aspect is a runtime entity. Therefore any surfacing of such compiler internals should be discouraged. The stack trace entry should be amended to read

```
Enforcement.before(int) [1],
```

where the numeric identifier indicates which of several potential before advice declarations in the aspect is being executed. This numeric identifier is necessary because in the current version of the AspectJ language advice is not named. Extra stack frames that result from callouts to advice are also an implementation detail of the compiler and should not be visible in the default stack view.

An interesting potential enhancement to debugging is to allow the use of a pointcut expression to specify all the (break)points where the debugger should stop. We have on occasion simulated this capability by setting a breakpoint in a dummy method called by advice associated with a debug pointcut, and found it to be useful.

3.3 Pointcut Wizard

Mastering the pointcut language is one of the things that newcomers to AspectJ find hardest. We plan to add a “New Pointcut Wizard” to AJDT that guides a user step by step through the process of creating a pointcut. The wizard should encourage good style, and have knowledge of common idioms.

As an example of good style, we prefer pointcut expressions that directly capture the intent of a policy over those that explicitly list matching cases. For example, if a policy requires the addition of behaviour to all public methods of a class, we prefer

```
execution(public * SomeClass.*(..));
```

over

```

execution(int SomeClass.meth1(..)) ||
execution(void SomeClass.meth2(..)) ||
etc.

```

since the latter is more robust in the face of change.

Instead of

```
call(int SomeClass.meth1(String));
```

we prefer

```
call(* SomeClass.meth1(..));
```

for the same reason – unless of course the return type or arguments really do matter.

An example of a common idiom is the use of

```
call(X) && !cflowbelow(X)
```

to capture only outermost calls to the methods matched by “X”.

Early analysis of the decision tree for such a wizard shows that considerable design effort is required to create a complete and usable tool.

3.4 Pointcut Query View

The Pointcut Query View will enable users to explore their program source in support of aspect mining, refactoring and program exploration. It can be left permanently open as a view inside Eclipse (as opposed to a dialog which is dismissed after one usage). It will contain a text input area into which a pointcut expression can be entered (or the pointcut wizard can be used to create the pointcut). The Query View will perform searching within the project classpath to automatically resolve non fully-qualified types used in the pointcut where possible. Where a non fully-qualified type name is ambiguous, the wizard presents an Eclipse “quick-fix” style choice allowing easy selection from amongst the possible options.

When a query has been entered, it can be evaluated. The query view results area displays the matches in a format similar to the Eclipse search results view, and each match allows navigation to the target source location through double-clicking. The query matches can also be visualized using the Aspect Visualizer as discussed in section 2.2 and illustrated in Figure 5. The ability to name and save queries will also be supported.

3.5 Pointcut Reader

The Pointcut Reader is the opposite of the Pointcut Wizard – it acts as an intelligent tool-tip for a pointcut. With the Pointcut Reader turned on, hovering over a pointcut declaration will produce a tool-tip that attempts to explain in “plain English” what the pointcut does. For example, the tool-tip for the pointcut `call(private *.set*(..,int))` might read “a call made to a private method “set*” defined on any type, and whose last argument is an int.” A good pointcut reader able to turn an arbitrary pointcut expression into legible English could be the subject of a project in its own right, but a basic version to get things started could be produced in reasonable time.

3.6 Crosscutting “diff” View

The crosscutting “diff” view is designed to support the “extract to advice/aspect” refactoring. Today this refactoring proceeds as follows:

1. Use a “declare warning” statement to capture all the scattered places in the code where the tangled logic to be refactored currently resides.
2. Determine the policy for when this logic should be executed, and write a pointcut to capture it.
3. Write some advice (before/after/around as appropriate) that contains the logic to be refactored.
4. Compile the program and manually compare the places affected by the new advice (using e.g. the outline view) with the places the tangled logic currently resides.
5. When satisfied as a result of the comparison that the aspect has encapsulated the concern, comment out or remove the tangled code. Run unit tests.

A simple example from the much loved “figures” AspectJ demonstration is the replacement of scattered and tangled calls to `Canvas.updateHistory()` with a “HistoryManagement” aspect. At step 1 a “declare warning” is defined as follows:

```
aspect HistoryManagement {
    pointcut canvasHistoryUpdate() :
        call(* Canvas.updateHistory(..));

    declare warning: canvasHistoryUpdate() &&
        !within(HistoryManagement) :
        "Only History Mgr should update history";
}
```

At step 2, we determine the policy that there should be a history update after any set method executes on a figure element:

```
pointcut figureElementUpdate() :
    execution(* FigureElement+.set*(..));
```

In step 3 we write the after advice that implements the policy:

```
after() returning : figureElementUpdate() {
    Canvas.updateHistory();
}
```

Finally at step 4, we can now see the need to compare the places matched by the declare warning (a call-based pointcut) with the places advised by the aspect (an execution based pointcut). This is where the Crosscutting “diff” View comes in. It compares the results from a query (perhaps a named and saved query created by the pointcut query view), with the locations advised either by an aspect as a whole, or by an individual piece of advice in the aspect. This allows the user to

easily see places in the code that the aspect does not match the tangled logic. It also shows the places where the aspect-based policy implementation is providing advice, but the tangled logic was not. Figure 6 on page 6 shows an example of such a visual comparison.

3.7 Refactoring

Refactoring is becoming a standard part of modern Java IDEs. There are two challenges for AJDT with respect to refactoring: the first is to extend the existing refactorings in the catalogue to be aspect-aware, and the second is to contribute additional aspect refactorings.

An example of the first challenge is the most basic of all refactorings – rename. In a Java program, it is possible to track down all references to the subject of the rename and update them accordingly. In an AspectJ program it is not so simple; the renamed element may now receive advice it didn’t previously (because the new name matches a pointcut expression), or may lose advice that previously applied. The Eclipse 2.2 development plan promises to open up the refactoring APIs so that AspectJ-specific deltas and messages can be added when a refactoring operation is triggered.

Additional aspect refactorings could include ‘basic’ aspect refactorings such as “Extract to advice”, and “Move to inter-type declaration”. More sophisticated refactorings might for example recognize usage of the observer design pattern and replace it with the aspect alternative [7].

3.8 Navigator Views & Search

Many of the planned extensions to AJDT rely on enhancements to the underlying AspectJ support. In AspectJ 1.1, the AspectJ compiler is implemented as an extension of the Eclipse JDT compiler. The compiler’s AST is extended to incorporate AspectJ elements, but the Java model from which the majority of the Eclipse views are built is not extended. Integrating with the Java model in this way will enable better integration of AspectJ constructs into the various Java views provided by Eclipse such as the Package Explorer, Type Hierarchy, and the search engine.

4. Scaling to Larger Systems

As well as contributing to the development of AJDT, some of the authors are investigating the application of AOSD techniques to large-scale middleware systems. The systems under study can comprise of in the region of 10-15 thousand classes. Such systems contain multiple internal components that themselves comprise of multiple packages. Here we find a need to express queries and pointcut expressions in terms of those component names, and to have a visualization mode that shows crosscutting across system components at the top level (a level above the currently supported packages view).

Applying AOSD at this scale is the subject of ongoing research.

5. Related Work

Tool support for AspectJ is also provided in Emacs [8], NetBeans [9] and JBuilder [10]. The Aspect Browser project provided the inspiration for the visualizer in AJDT. The Aspect Mining Tool [11] and the Extended Aspect Mining Tool [12] support discovery of aspects in existing code. JQuery [13] is a sophisticated querying tool for exploring Java projects. FEAT [14] supports both querying and concern modelling.

6. SUMMARY

AJDT provides a rich set of features which aid in the development and understanding of aspect-oriented programs built using AspectJ. These include the Aspect Visualizer, Outline View, Editor support, and debugger.

Planned extensions to AJDT will further enhance the support it provides both to novice and experienced AO programmers. We are also investigating the issues raised by scaling AspectJ and AJDT to work with large-scale systems.

7. ACKNOWLEDGMENTS

Thanks to all the team past and present who have worked on AspectJ and helped to make it the tremendous tool that it is.

Thanks to the IBM® Extreme Blue™ team who contributed the first version of the Aspect Visualizer.

The notion of a permanently open query view as opposed to a search dialog was influenced by discussions with Bill Harrison, Harold Ossher, and Peri Tarr of IBM's T.J. Watson Research Lab, and with the JQuery and FEAT teams.

IBM and Extreme Blue are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product or service names may be trademarks or service marks of others.

8. REFERENCES

1. *Eclipse AspectJ Development Tools project*: <http://www.eclipse.org/ajdt>.
2. *Eclipse.org - Main Page*: <http://www.eclipse.org>.
3. Kiczales, G., et al., *Getting Started with AspectJ*. Comm. ACM, 2001. **44**(10): p. 59--65.
4. Griswold, W.G., Y. Kato, and J. Yuan, *Aspect Browser: Tool Support for Managing Dispersed Aspects*. First Workshop on Multi-Dimensional Separation of Concerns in Object-oriented Systems, OOPSLA, 1999.
5. Eick, S.G., J.L. Steffen, and E.E. Sumner, *Seesoft - A Tool For Visualizing Line Oriented Software Statistics*. IEEE Transactions on Software Engineering, 1992. **18**(11).
6. Kersten, M., *Tool Requirements For Commercial Development With AspectJ*. Commercialization of AOSD Workshop, Held in conjunction with the 2nd International Conference on AOSD, 2003.
7. Hannemann, J. and G. Kiczales. *Design Pattern Implementation in Java and AspectJ*. in *Proceedings of the 17th ACM Conference on Object-oriented programming, systems, languages and applications*. 2002: ACM Press.
8. *AJDE for Emacs and JDEE*: <http://aspectj4emacs.sourceforge.net/>.
9. *AJDE for SunONE/NetBeans*: <http://aspectj4netbean.sourceforge.net/>.
10. *AJDE for JBuilder*: <http://aspectj4jbuidr.sourceforge.net/>.
11. *The Aspect Mining Tool*: <http://www.cs.ubc.ca/~jan/amt>.
12. *Extended Aspect Mining Tool*: <http://www.eecg.utoronto.ca/~czhang/amtex>.
13. Janzen, D. and K. De Volder, *Navigating and Querying Code Without Getting Lost*. Proceedings 2nd International Conference on Aspect-Oriented Software Development, 2003: p. 178-187.
14. Robillard, M., *FEAT: A tool for locating, describing, and analyzing concerns in source code*: <http://www.cs.ubc.ca/labs/spl/projects/feat/>.

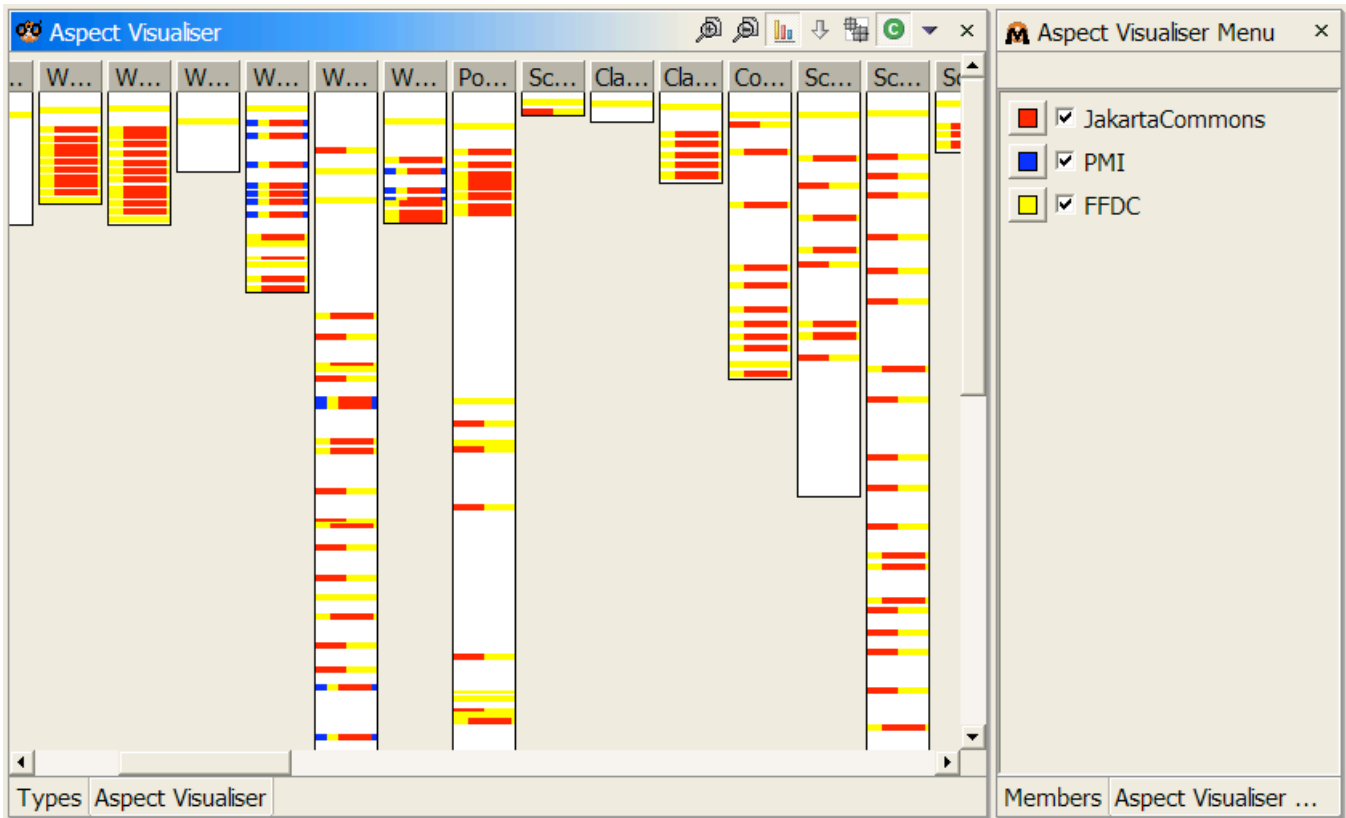


Figure 5 - Aspect Visualization in AJDT

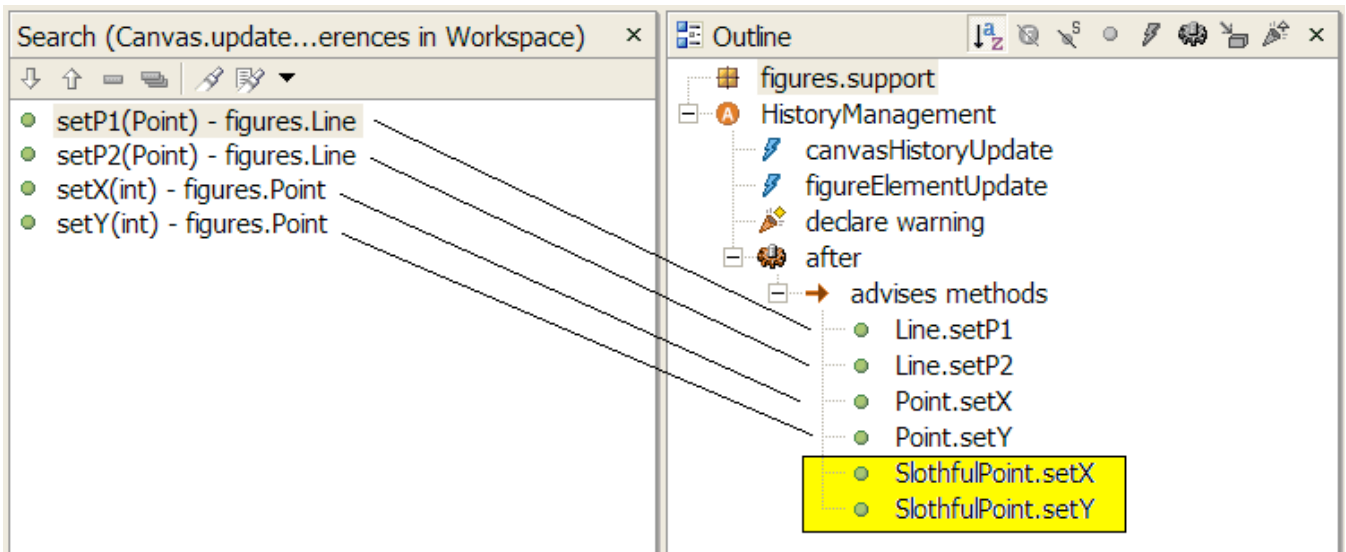


Figure 6 - Crosscutting Diff