Tool Requirements for Commercial Development with AspectJ

Mik Kersten (mik-at-intentsoft.com)

Intentional Software Corporation

1. Introduction

Tool support is a key requirement for the commercial adoption of Aspect-Oriented Software Development (AOSD). The AspectJ™ project made tool support a core priority, which helped commercial adoption [Price2001, Suprlin2002]. However as of the AspectJ 1.1 release we have not yet delivered all the tool support that we set out to provide. In addition, the state-of-the-art in Object-Oriented (OO) tools has evolved, which raises the bar for AO tool integration. The following is a discussion of tool requirements that will improve commercial AOSD development with AspectJ.

2. What We Have

2.1. Compiler & Command Line Tools

The AspectJ compiler supports the incremental compilation of Java™ and AspectJ source code. In addition, it accepts aspects in binary form as input, and can be used to weave aspects into existing binaries (JARs). The compiler exposes a command line Ant [Ant2003] task user interface. The current incremental compilation support behaves similar to a Java incremental compiler: when an aspect changes the whole world needs to be rebuilt. The compiler produces JDK1.1+ compatible bytecodes.

AspectJ 1.1 does not include a debugger. We had beta-quality command-line and GUI debuggers prior to 1.1. However, these never reached the quality and feature bar set by modern Java debuggers. Instead we have focused on ensuring that the compiler emits bytecodes that accurately map advice for JSR-45 compatible debuggers [Jsr2003].

2.2. IDE Tools

AspectJ supports extensions to several popular IDEs and offers additional standalone development tools. The AspectJ IDE support provides a representation of aspect-oriented program structure in tree-based structure views, editor annotations, and documentation links. The AspectJ IDE support includes plug-ins for Eclipse [Eclipse2003], Borland®'s JBuilder®, Sun®'s NetBeans, Emacs and JDEE. We also provide a lightweight IDE called the AspectJ Browser, and the AJDoc API documentation tool which extends Javadoc.

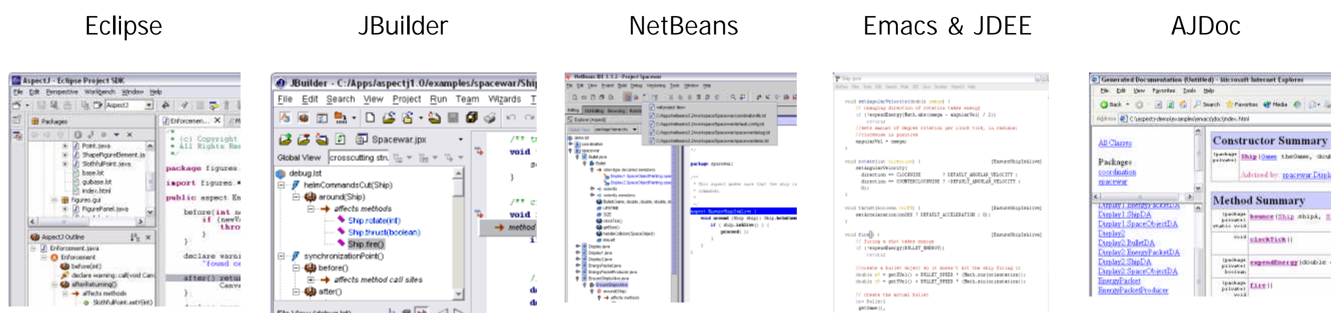
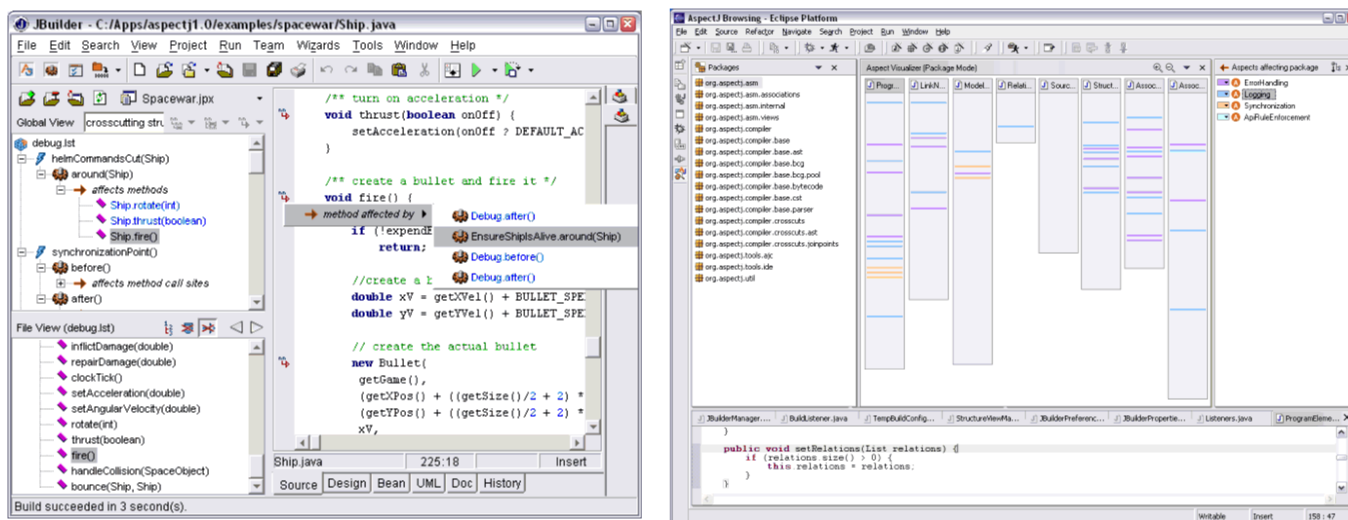


Figure 1: AspectJ Tool Extensions

Each of these tools is an extension of the AspectJ Development Environment (AJDE) framework. The framework provides compilation facilities, a structure model that can be used to generate AO views, an abstract UI and event model, and a concrete GUI for Swing extensions. The goal for each of the framework extensions is to provide a seamless UI integration with its host platform. For example, in the

2.3. Crosscutting Structure Views

Object-oriented programming enables the programmer to modularize hierarchical structure. As a result, modern Java development tools present this hierarchical structure. AOSD enables the programmer to modularize crosscutting structure. As a result, the AspectJ tools must present this crosscutting structure. Join points are the core AspectJ's crosscutting model. These are key points in the dynamic call graph such as method calls, exception handler executions, class initializers, and fields sets. The programmer specifies when an aspect's advice execute by declaring a set of join points. Since the invocation of advice is implicit, without tool support users complained that they did not know what advice would affect the execution of a particular method when they were working on that method. When dealing with OO structure a programmer can follow references and inheritance declarations made in a particular method and its class. AspectJ's power comes from the ability to express crosscutting concerns in a single module that affects many different parts of the system. This same power means that you need a global understanding of the system to understand where aspects may apply, which is particularly challenging with large commercial systems. One solution is to show this crosscutting structure as navigable annotations in-line with the places that they affect. This allows the programmer to navigate from a method to the advice that affects it. The crosscutting structure can also be navigated in the other direction in order to find out where an aspect applies. Figure 2 shows crosscutting structure (blue links) in both a tree view and as inline annotations. In Figure 3 all of the files in a package are show, and the crosscutting of different aspects is highlighted with colored source locations. This crosscutting structure is also made explicit in the other tools like Emacs (as text annotations) and AJDoc (as hyperlinks).



Figures 2 & 3: Crosscutting Structure Views

3. What We Need

While providing the core tool support and IDE plug-ins took all of our development time on the AspectJ project at PARC [Parc2003], we paid attention to feature requests made both by users and ourselves. The following is a list of unimplemented features that we considered as important to commercial development with AspectJ. This list is a work-in-progress. In order to continue improving the support for commercial development with AspectJ this list needs to be completed, the items expanded, and then prioritized for implementation.

3.1. Refactoring

Refactoring [Fowler1999] support is becoming a commodity feature in Java IDEs [Eclipse2003]. The overall value of AspectJ adoption is reduced when it comes at the price of useful features such as refactoring support. Java refactorings should be aspect-aware (e.g. "rename method" should update references within advice bodies and pointcut designators). AspectJ-specific refactoring need to be explored and then supported (e.g. "use supertypes and patterns where possible" in a pointcut designator). As the size of a system grows so does the overall benefit of refactoring, making it an important candidate for commercial tool support.

3.2. Structure Mining

Refactoring support enables structured edits of the system. Sometimes these use simple heuristics to help improve modularity (e.g. "use sypertype where possible"). The adoption of AspectJ for a large system results in improved modularity for crosscutting concerns. However, the programmer is not assisted in untangling the crosscutting into well-modularized aspects. Tools that encapsulate the heuristics of tangled crosscutting (e.g. the same two methods are called wherever some exception is handled) should automate some the process of improving the modularity of large OO systems with AspectJ. Crosscutting structure mining support will speed up the benefit of adopting AspectJ for a project.

3.3. Better Structure Views

The AspectJ Browser and JBuilder® plug-in provide a tree-based global structure navigating tool (Figure 2). This tool needs to be integrated into the Eclipse platform, and extended to update it's model using eager parsing and the result of incremental builds. The resulting model should also expose a search-based user interface that handles Java elements correctly (e.g. inter-type declared members are included in a structured search of the target type) as well as exposing crosscutting structure (e.g. find all pointcut designators referring to this method). A common complaint of AspectJ structure views has been that they hint little at runtime behavior. Advice execution order on a particular join point should be made specific by the tool. Additional constraints such as cflows and dynamic tests should be viewable inline at the places that the advice might affect. Higher-level views (e.g. Figure 3) and UML notation views should be supported for both viewing and editing. The main benefit of making the crosscutting structure explicit in these views is reducing the learning curve, and improving the comprehensibility of crosscutting modularity in a large system.

3.4. More Compiler Support

AspectJ builds will speed up once the compiler understands AO dependencies so that it can behave in an incremental way when aspects are being edited (e.g. a local change to the body of an advice should only recompile the file with the advice). Load-time weaving (useful for application servers) should be supported using the existing bytecode weaving architecture. Eager parsing and incremental structure model generation are also features that involve improving the compiler support, and will enable the structure views to stay in synch with the code.

3.5. Project Configurations

Project build configuration is currently done using AspectJ's ".lst" files. These have two failings: they're based on inclusion and as such make it cumbersome to unplug aspects (e.g. the default configuration for my project should exclude the profiling aspect), and they're non-standard. Ant-based [Ant2003] build configuration based on exclusion properties should work better for unplugging aspects from large systems, and would be based on a standard. In addition, in an IDE that has a rich model of project dependencies such as Eclipse [Eclipse2003] support for project dependencies on aspects needs to be improved.

3.6. Debugging, Profiling & Testing

AspectJ needs better debugger support. Once the IDEs support JSR-45 we'll be most of the way there. However, runtime structure views will need to be extended to understand crosscutting. The thread tree should show the running advice in a way that is similar to the structure views showing what methods are affected by advice. The programmer should be able to issue commands related to the execution of advice (e.g. break on this advice whenever it is within the control flow of that method). Profiling tools could be extended to let the programmer express constraints based on the join point model (e.g. profile the length of the execution in the control flow below this method).

The Eclipse and JBuilder IDEs provide support for unit testing Java methods using JUnit [JUnit2003]. Although there has been little exploration for what it means to unit test advice, once some methodology is specified the IDE support should integrate unit testing for aspects (e.g. execute unit test for this advice in conjunction with all the unit tests for the methods that it applies to).

4. Conclusion

We're on the way to providing commercial-level support for AspectJ development. That's not to say that commercial development isn't being done on with the current AspectJ tools. However, since AspectJ allows the programmer to capture system-wide crosscutting concerns, the tools play an important role for displaying the effects of those concerns. Just as OO structure is made explicit by modern Java IDEs, crosscutting structure needs to be made explicit across the development lifecycle and corresponding tool support. Also, the increasing feature expectation set by IDEs needs to be met in order to keep the bar for adopting AspectJ low. Finally, tools support that facilitates integration, understanding, and usability of AspectJ on large systems is needed to ease adoption for existing commercial projects.

5. References

Vaughn Spurlin October 2002

[Price2001] Price, Richard. "AspectJ, 0.8b5: Real-world AOP tool simplifies OO development". In Java Report, September 2001, http://www.adtmag.com/java/article.asp?id=4712&mon=9&yr=2001

[Suprlin2002] Vaughn Spurlin. "Aspect-Oriented Programming with Sun(tm) ONE Studio". In Sun ONE Studio Developer Resource Page, October 2002, http://forte.sun.com/ffj/articles/aspectJ.html

[Ant2003] Apache Ant Project, http://ant.apache.org

[Jsr2003] Sun® Java Community Process, "JSR-45: Debugging Support for Other Languages", http://www.jcp.org/aboutJava/communityprocess/review/jsr045

[Eclipse2003] Eclipse Open Source Project, http://eclipse.org

[Parc2003] Palo Alto Research Center AspectJ Project, http://www.parc.com/groups/csl/projects/aspectj/

[Fowler1999] Fowler, Martin. "Refactoring: Improving the Design of Existing Code", Addison-Wesley Co., Inc, Reading, MA, 3rd printing Nov 1999

[JUnit2003] JUnit Project, http://junit.org