

AO Tools: State of the (AspectJ™) Art and Open Problems

Mik Kersten (beatmik-at-acm.org)

Palo Alto Research Center

3333 Coyote Hill Road

Palo Alto, CA 94304

1. Where We Were

Early in the AspectJ project we noticed two common problems raised by users as barriers to adoption. “Support my development environment” was a frequent mailing list request from those who were not willing or able to leave their Java IDE. “How do I know what aspect affects my code?” was a question that came up at every tutorial that we gave. As a result, the mission of AspectJ tool support became presenting aspect-oriented (AO) program structure and tool functionality consistently across multiple development environment platforms.

2. Where We Are Today

AspectJ supports extensions to several popular IDEs and offers additional standalone development tools. These tools provide a representation of aspect-oriented program structure in tree-based structure views, editor annotations, and documentation links. But user’s demands continue and we’re getting requests for refactoring support, UML and other high-level views, AO structure mining, and more dynamic information. Many of these requests have solutions that are not necessarily AspectJ-specific. Users want to be able to find, browse, and manipulate the crosscutting structure of their system whether it is declared in AspectJ or not. However, the existing AspectJ framework offers some experience in integrating AO tools into existing development environments.

2.1. The AspectJ Tools Framework

The AspectJ tools suite includes IDE support for Eclipse, JBuilder, NetBeans, Emacs and JDEE. In addition to a command line compiler we also provide a lightweight IDE called the AspectJ Browser, the AJDoc API documentation tool, and an Ant task.

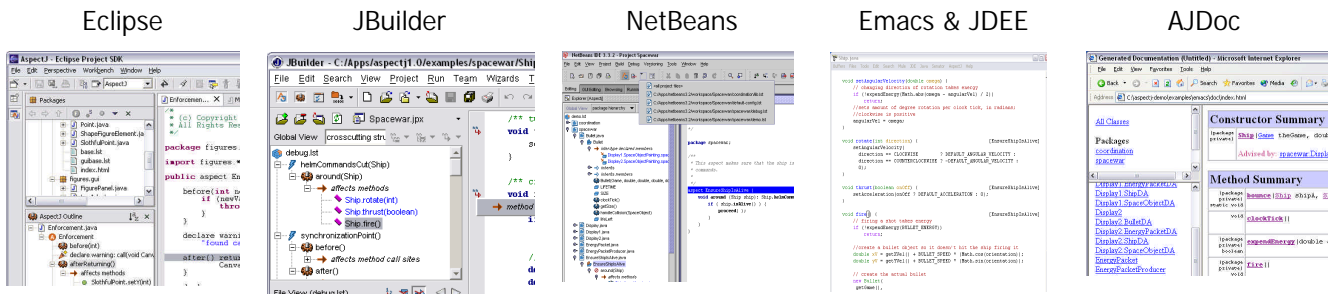


Figure 1: Examples of AspectJ Tool Support

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 IDEs the crosscutting structure is represented as GUI widgets, in Emacs as text annotations, and in AJDoc as hyperlinks.

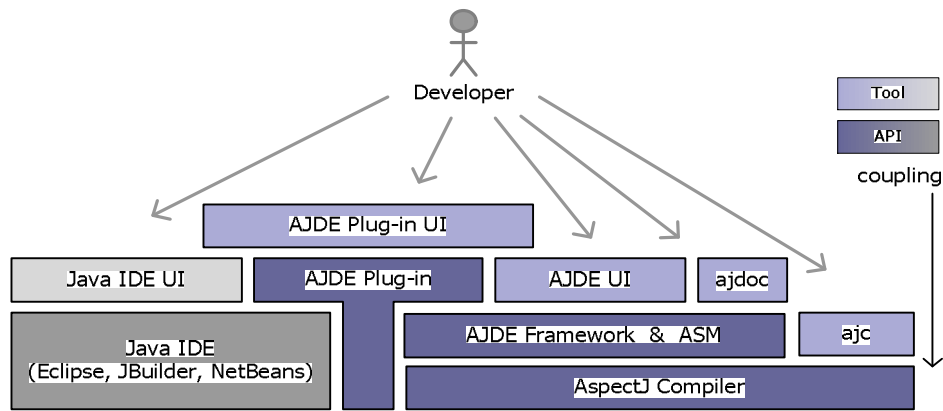


Figure 2: Tools Architecture Overview

2.2. 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. At the core of the AspectJ are joinpoints. 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 joinpoints. 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. 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.

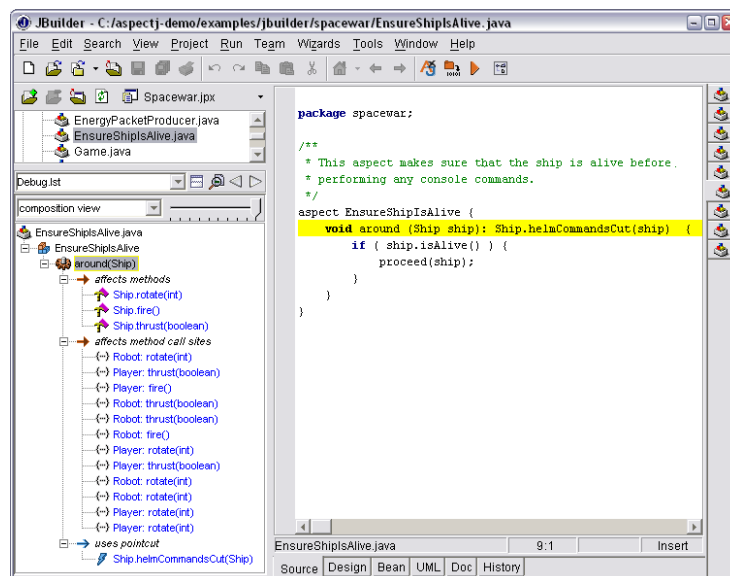


Figure 3: Crosscutting Structure in AJDE for JBuilder

2.3. The Abstract Structure Model

The Abstract Structure Model (ASM) represents the crosscutting, inheritance, and referential structure of AspectJ programs. The AJDE tools expose the structure of the model in task-specific views. The lifecycle of the ASM is longer than that of the AST in order to support structure with errors and to provide the AJDE framework with multiple live views. The model is kept in memory and it's footprint is small enough to be kept live for large programs. The public ASM API is generic enough to support extensions for structure declared in another

programming language (e.g. C#), structure defined outside of the primary language (e.g. design documents and XML descriptors), and emergent structure (e.g. for crosscutting structure mining).

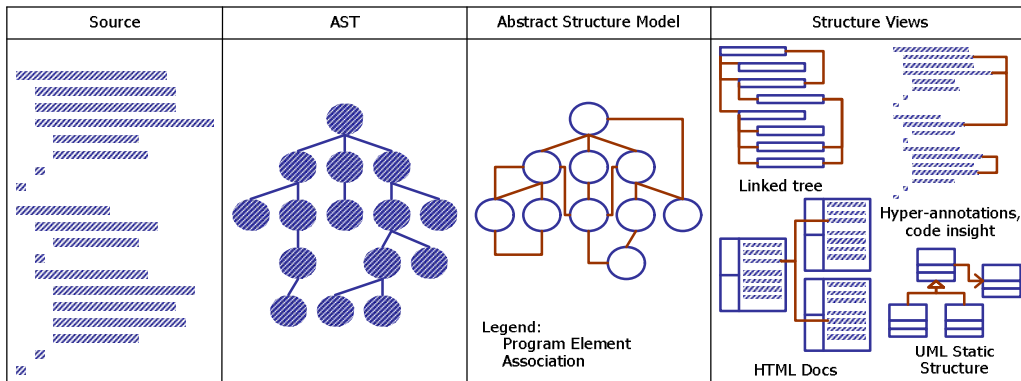


Figure 4: Relation of the ASM to Source, AST, and Views

3. Where We Are Going

3.1. New AspectJ Tools

AspectJ users are starting to expect the advanced tool functionality that is appearing in Java IDEs such as Eclipse. Requested features such as refactoring, code-insight, and structure-aware version control require the corresponding views understand the AO extensions to the IDE's structure model. One of the goals of AspectJ 1.1 is to integrate the ASM into Eclipse's structure model, implement some of these features, and provide a platform on top of which future advanced AO tooling can be developed.

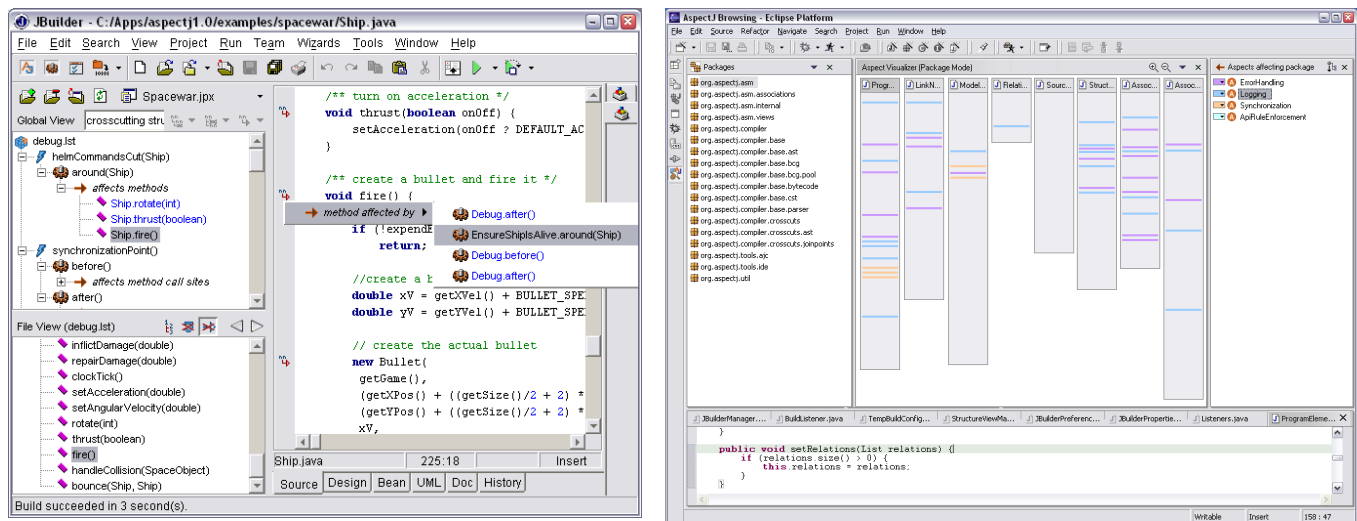


Figure 5: Multiple Task-Specific and Global Structure Views

3.2. Open Problems

The current trend in development tools is moving towards providing structured views of systems and structured editing facilities. One example is the availability of Java refactoring and UML modeling on the Eclipse platform. However, the structured features of these tools are based the OO principles of hierarchical information hiding. In practice OOP-based modeling fails to capture structure that crosscuts:

- The primary system modularity (e.g. joinpoints and aspects)
- Enterprise application resources (declarations and references in other resources, e.g. XML)
- Development tasks (e.g. content, presentation, and application development)

The lack of a consistent model of program structure has lead to numerous redundant solutions that surface inconsistent and incomplete views in today's development tools. Crosscutting structure is present in complex software systems whether it is explicitly declared in the primary programming language, supporting domain-

specific languages, or documented. The inability of current tools to deliver whole-system structure views is a result of the limitations of OO structure modeling. A unified model that captures crosscutting structure and presents workflow-specific views is needed because:

- Enterprise applications incorporate many highly interrelated resource types. These resource types are not compiled collectively in order to validate the intended relationships.
- The development process is separated into isolated but interdependent roles that share concerns.

We need extensions to models such as the ASM in order to support this additional structure. The resulting unified model needs to be editable in order to allow consistent refactoring across the different application resources. We also need new crosscutting-aware views in order to provide better global visualization, design-level perspectives, refactoring and mining support, and runtime structure information. We need better user interfaces in order to present this structure in-line with development tasks. These model extensions and corresponding views are not only needed by future AspectJ-specific tools, but common to those tools that set out to help the programmer manage the crosscutting structure of a large system.

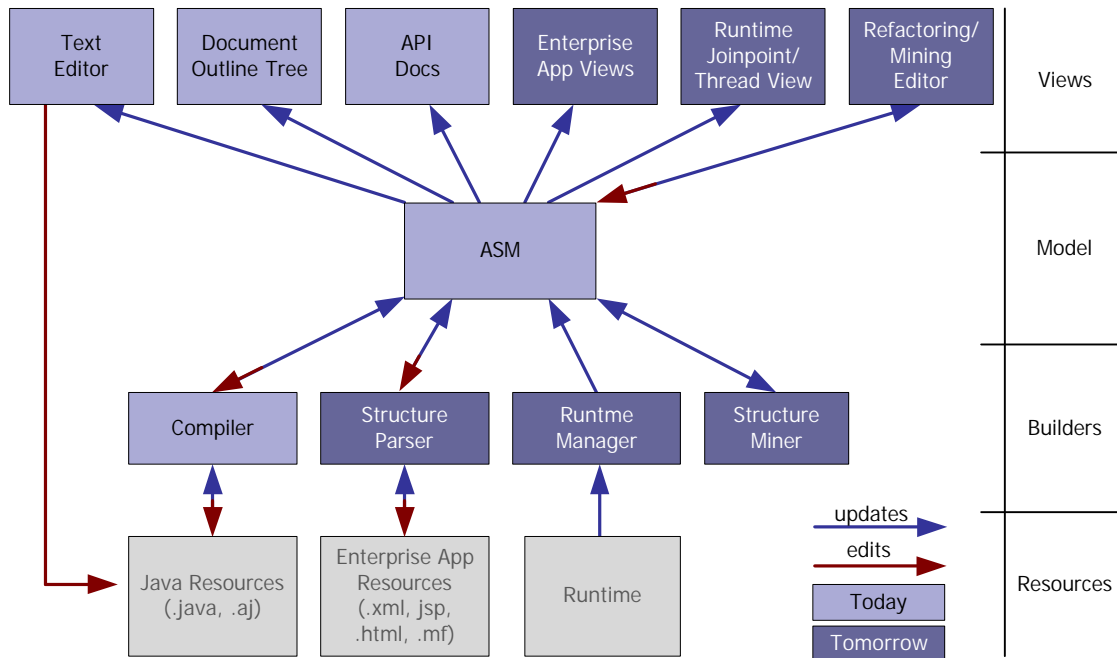


Figure 6: Current AspectJ Tools and Open Problems