Visualizing and AspectJ-enabling Eclipse Plugins using Bytecode Instrumentation

Chris Laffra
IBM Ottawa Labs
Ottawa, Canada
Chris_Laffra@ca.ibm.com

Martin Lippert
Software Engineering Group
University of Hamburg, Germany
lippert@acm.org

ABSTRACT
Bytecode instrumentation can be used effectively to (a) generate visualizations and (b) to modify the behavior of Eclipse plugins. In this demonstration, we will show two independent techniques that have in common that they obtain their results by modifying the binary representation of a given software system. In the first part of the demo, Chris Laffra will show experiments he performed on visualization of Eclipse plugins in the context of the JikesBT project. In the second part of the demo, Martin Lippert will show how to weave aspects into Eclipse plugins without having access to their source.

Categories and Subject Descriptors
D.1 [**Software**]: Programming Techniques – *Aspect-Oriented Programming.* D.3.2 [**Programming Languages**]: Language Classifications – *AspectJ.* D.3.3 [**Programming Languages**]: Language Constructs and Features – *modules and packages, classes, aspects*

General Terms
Design, Languages, Frameworks, Visualization

Keywords
Eclipse, Plugins, Reflection, Introspection, Visualization, Aspect-Oriented Programming, AspectJ, Cross-Plugin Pointcuts, Modularization

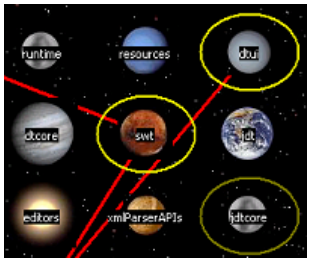
1. ECLIPSE PLUGIN VISUALIZATION
To use a system well, one has to understand its inner workings. Most users and contributors of Eclipse [2] view it as a *black box*. They write their own plugins, connect to a limited set of other plugins, and hope everything works. Novice Eclipse developers often get stuck at questions such as "why does my plugin not load?" The answer to that question lies in the following:

> **Lloyd's Hypothesis:** Everything that's worth understanding about a complex system can be understood in terms of how it processes information. -- Seth Lloyd

Eclipse essentially is a highly componentized Java program. It consists of loosely coupled plugins that collaborate to perform tasks. The plugins extend others by implementing their interfaces. External inputs result in complex chains of events, sometimes in parallel, each racing from the depths of the Eclipse core runtime towards plugins at the border of this living eco-system. Eclipse offers limited mechanisms to understand the event propagations that are so essential in understanding Eclipse better. To obtain a higher insight, we need to add a better reflective layer that can report on a wide variety of information sources, such as plugin activation, class loading, method invocation, and object creation. To provide that extra layer of introspection, two options are available:

1. Run Eclipse on a Java virtual machine in a special debug mode and use standard Java profiling API (JVMPI). The client reacting to the events sits in a separate process and information is transferred using sockets. There is minimal overhead to start the system, but the amount of communication that can be exchanged is heavily restricted, often leading to mandatory filtering at the source VM.
2. Use bytecode instrumentation to modify the class files that define the Eclipse plugins. At critical locations, insert special code that will generate extra "events" when executed. The client code reacting to the events runs in the same process and communication between Eclipse and the client is done as efficiently as possible using normal method calls. The client code can bypass restrictions of JVMPI by creating highly customized visualizations. The events can be easily augmented by extra information obtained by using Eclipse API itself.

Bytecode instrumentation has been used to generate various visualizations that show internal Eclipse communications.



One focuses on plugins and the method calls made between them at a very high level (see at right). Another one records all object allocations and uses a weak hashtable to discover Eclipse memory leaks. Yet another allows for diving really deep into the bowels of Eclipse showing every single method executed (including parameter and return values). Finally, we experimented with making the visualization client an Eclipse plugin. This is "cool" and elegant in itself but also serves a useful purpose. The extension points offered by the plugin allow third parties to enhance the default visualizations with custom ones. Two examples are included in the plugin to do performance profiling and attach sounds to certain events. The source code for the visualization can be downloaded together with the JikesBT project [3].

OOPSLA'03, October 26–30, 2003, Anaheim, California, USA.
ACM 1-58113-751-6/03/0010.

2. ASPECTJ-ENABLED ECLIPSE CORE RUNTIME

Many approaches to software engineering aim to improve separation of concerns and modularity [6]. The idea of an AspectJ-enabled Eclipse Core Runtime focuses on two promising approaches that are implemented for Java: the Eclipse Core Runtime Platform [2], and aspect-oriented programming via AspectJ [1], [4]. While these two approaches seem to be orthogonal to each other, their combination appear to be powerful.

The combination of these two technologies for Java is not trivial. Typically applications developed using AspectJ have to be completely compiled or woven via the AspectJ compiler. This way the compiler works breaks with the modularization approach used via plugins. When we develop plugins, the compiler typically knows all the source code of the plugin itself and the bytecode of the required plugins – and no more than that. As a result, aspects could only define pointcuts that are completely inside a single plugin (they can define more, but the weaving functionality of the aspect compiler will find only those targets of the pointcut that are inside the plugin where the aspect is defined). This is not enough. We would like to be able to define pointcuts that are beyond the boundaries of plugins. This would allow us to use Eclipse as a general application ("rich-client") platform together with AspectJ. Just think of any large AspectJ-based application being developed as a set of plugins. The goal is to not let the plugin technology limit the capabilities AspectJ provides. Developers should be enabled to design aspects for pointcuts that may crosscut plugin boundaries (like object boundaries) and let them modularize and implement those aspects into their own plugins.

2.1 A Load-Time Weaving Eclipse Runtime

An AspectJ-enabled Eclipse Core Runtime that integrates load-time weaving can solve the problem. The basic idea of load-time weaving of aspects is to let the aspect be woven into classes at the time the classes are loaded into the VM (in the case of Java). This can be realized via customized class loaders. Such a class loader loads the bytecode of each class and weaves the aspect hooks and calls into this bytecode. The woven bytecode is subsequently given to the VM for actual definition of the class.

2.1.1 Load-Time Bytecode Modification For Eclipse

One way to introduce the load-time weaving functionality into the Eclipse system is to insert a basic load-time bytecode modification hook at the class loading mechanism of Eclipse. This hook allows us to inject the weaving functionality exactly where the bytecode of a class is loaded without greatly modifying the class loaders of Eclipse.

2.1.2 Inserting AspectJ Bytecode Weaving

The load-time bytecode modification hook provided by the modified runtime is used by a weaving plugin to insert the bytecode weaving functionality of AspectJ 1.1. This plugin just weaves class per class as they are loaded into the system. Therefore the plugin needs to know all aspects plugged into the system at each startup.

A neat way of promoting aspects at startup time is to use the general Eclipse mechanism of *Extension* and *Extension Point* for this purpose. We can introduce a new extension point called "aspects" that lets other plugins define, in their plugin.xml description, the aspects they want to promote for weaving.

2.1.3 Dynamic Dependencies

Apart from the static view of the system, the runtime behavior of the Eclipse plugin infrastructure plays an important role when aspects are defined inside plugins and should be woven into classes of other plugins. The AspectJ load-time weaving plugin can take care of these issues and ensure that the dynamic dependencies between load-time woven plugin code inside different plugins is mapped onto the general plugin dependency mechanisms of Eclipse.

2.2 Status of Work

All of this is implemented and working for the current version of Eclipse (2.1) and AspectJ (1.1). The modified runtime is fully compatible with the original implementation in a way that the complete Eclipse platform including the Java IDE is working on top of it without any adaptations. For more information on that take a look at [5].

3. ACKNOWLEDGMENTS

Martin wish to thank Jim Hugunin and Wes Isberg for their help in implementing the weaving class loader. Special thanks from Martin go to Axel Schmolitzky for his comments on earlier drafts of the work.

4. REFERENCES

[1] AspectJ Team. AspectJ home page. http://www.eclipse.org/aspectj/.

[2] Eclipse Project. http://www.eclipse.org/eclipse/.

[3] The Eclipse Monitor sample shipped with JikesBT Project. http://www.alphaworks.ibm.com/tech/jikesbt.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Longtier, J. Irwan. Aspect-Oriented Programming. In *Proceedings of ECOOP'97*, Springer-Verlag LNCS 1241, June 1997.

[5] Martin Lippert home page. http://www.martinlippert.com/.

[6] D. L. Parnas. On the criteria to be used in decomposing systems into modules. In *Communications of the ACM*, volume 15, pages 1053-1058, 1972.