An AspectJ-Enabled Eclipse Core Runtime Platform

Martin Lippert

Software Engineering Group
Faculty of Computer Science
University of Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg, Germany
++4940 42883 2306
lippert@acm.org

ABSTRACT

Separation of concerns and modularity are key elements of software engineering. The work described here presents a combination of two proven techniques that help improve both of these elements: the Eclipse Core Runtime Platform, which introduces plugins to Java programming as a kind of module concept on top of packages, and aspect-oriented programming using AspectJ, which aims to improve the modularity of crosscutting concerns. The work presents a combination of these two technologies in an AspectJ-enabled version of the Eclipse Core Runtime Platform. Unlike the standard implementation of the Eclipse Core Runtime Platform, the AspectJ-enabled implementation allows aspects to modularize crosscutting concerns beyond the boundaries of plugins (without the need for recompilation across plugins). It allows crosscutting concerns to be modularized by means of aspects and plugins while using the enhanced but compatible version of the Eclipse Core Runtime Platform as promoted by the Eclipse project.

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

Keywords

Eclipse, plugin runtime infrastructure, aspect-oriented programming, AspectJ, cross-plugin pointcuts, modularization

1. INTRODUCTION

Many approaches to software engineering aim to improve separation of concerns and modularity [6], [7]. This work focuses on two promising approaches that are implemented for Java: the Eclipse Core Runtime Platform, which provides a plugin mechanism for Java [3], and aspect-oriented programming via AspectJ, which is designed to modularize crosscutting concerns [2], [4], [5], [1]. While these two approaches seem to be orthogonal to each other, their combination would appear to be powerful.

But what happens if we try to combine these two approaches? What if we want to develop large applications on top of the Eclipse Core Runtime Platform using the Java dialect AspectJ? What, for example, if we want to develop the Eclipse system itself using the aspect-oriented programming features of AspectJ?

Typically applications developed using AspectJ have to be completely compiled or woven via the AspectJ compiler. The AspectJ compiler takes the parts of the system (libraries, sources and classes) and produces a complete woven system. This basic assumption 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 rich-client platform together with AspectJ. The goal is to allow developers 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. AN ASPECT-ENABLED ECLIPSE CORE RUNTIME PLATFORM

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 is typically done via customized class loaders. These load the bytecode of the class to be loaded out of the .class file and weave the aspect hooks and calls into this bytecode. The woven bytecode is subsequently given to the VM for actual definition of the class.

Although load-time weaving functionality was not available for the AspectJ 1.0 language, the bytecode weaving implementation of AspectJ 1.1 allows load-time weaving to be realized for the complete AspectJ 1.1 language. A weaving class loader could be

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

implemented by simply using the second part of the AspectJ compiler implementation, where the weaving is done at bytecode level.

2.1 A Weaving Runtime

In contrast to the common way of enhancing the Eclipse system via additional plugins, the runtime architecture of Eclipse cannot, unfortunately, be AspectJ-enabled by an additional plugin. The reason for this is that the load-time weaving has to be injected at class-loading time, which is not designed to be modified or enhanced via plugins in the Eclipse system. Thus the idea of an AspectJ-enabled Eclipse Core Runtime is based on modified versions of the basic core plugins of Eclipse instead of additional plugins.

2.1.1 Load-Time Bytecode Modification For Eclipse

One way to introduce the weaving functionality into the Eclipse system is to insert a basic load-time bytecode modification hook at the classloading 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 can be used by a 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. How could this be achieved?

2.1.3 The Aspect Extension Point

A neat way of promoting aspects at startup time is to use the general Eclipse mechanism of *Extension* and *Extension Point* for this purpose. While the mechanism is used by the Eclipse system to let plugins add parts (extensions) to the system at predefined points (extension points), it is already used by the core runtime to determine which applications are available (via the extension point org.eclipse.core.runtime.applications).

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.

2.2 Summary

We have discussed the basic integration of a load-time aspect weaving functionality into the Eclipse Core Runtime Platform. We presented an approach enabling the basic weaving functionality of AspectJ 1.1 to be used to implement this. 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.

So far, the static view of the system is complete. Aspects were woven into classes that are loaded by the Eclipse system. 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.

3. ACKNOWLEDGMENTS

I wish to thank Jim Hugunin and Wes Isberg of the AspectJ project for their help in implementing the weaving class loader and improving weaving performance. My special thanks go to Axel Schmolitzky for his comments on earlier drafts of this work.

4. REFERENCES

[1] AOSD Web Site. http://www.aosd.net/.

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

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

[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] G. Kiczales, E.Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In J. Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer-Verlag.

[6] D. L. Parnas. *Information Distribution Aspects of Design Methodology*. IFIP Congress Preprints. 1971.

[7] 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.