# AJEER: An AspectJ-Enabled Eclipse Runtime

Martin Lippert

Software Engineering Group
University of Hamburg & it-wps GmbH, Germany
lippert@acm.org

## ABSTRACT

There are a number of technologies designed to improve modularity in software systems. The technique presented here combines two of them seamlessly to exploit their respective benefits: Eclipse plugins and AspectJ. The Eclipse runtime is based on the idea of plugins, enabling large systems to be built from smaller components. AspectJ is an AOP-enhanced version of the Java language that allows developers to modularize crosscutting concerns into aspects. While both technologies offer a number of interesting features, their seamless combination is not trivial. Several limitations make it impossible to exploit all the features of the combined technologies. AspectJ-Enabled Eclipse Runtime (AJEER) is designed to overcome these limitations. It integrates load-time weaving for AspectJ into the Eclipse runtime, thus allowing developers to implement aspects that modularize crosscutting concerns beyond the capability of individual plugins. In addition, the dynamic features of the OSGi-based Eclipse 3.0 runtime are preserved in this setting – making it possible to plug AspectJ aspects into and out of the running system dynamically.

## 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, Eclipse Rich Client Platform, Plugin Runtime, Aspect-Oriented Programming, AspectJ, Cross-Plugin Pointcuts, Modularization

## 1. INTRODUCTION

The main goal of the AspectJ-Enabled Eclipse Runtime (AJEER) is to allow developers to use two proven techniques in combination: Eclipse plugins [5] and AspectJ [3] [8]. The idea is to enable large software systems to be built based on the Eclipse Rich Client Platform [5], using AspectJ at the same time to modularize crosscutting concerns across different plugins.


*OOPSLA'04*, Oct. 24–28, 2004, Vancouver, British Columbia, Canada.
ACM 1-58113-833-4/04/0010.

The underlying idea and a preview version of AJEER were presented at OOPSLA 2003 (see [9], [10]). The new OSGi-based runtime of Eclipse 3.0 offers additional advanced features for implementing a combined runtime. These include the option of dynamically adding and removing plugins at runtime, which is of special interest for AJEER because adding and removing aspect-promoting plugins at runtime is not a trivial task.

## 2. LOAD-TIME WEAVING FOR ASPECTS INSIDE AJEER

AJEER allows developers to write separately compiled aspect-promoting plugins to the complete system. Using AJEER, developers do not need to recompile all system plugins if a new aspect appears. They can simply add their aspect-promoting plugin to the set of installed plugins.

This is possible because AJEER adds load-time weaving to the runtime of Eclipse. By doing so, AJEER takes account of the fact that all aspects may be woven into all classes of the running system when classes are loaded.

It is realized by adding the weaving part of the AspectJ 1.2 compiler implementation to the Eclipse runtime. Fortunately, this weaving functionality uses bytecode instead of source code to weave aspects into regular Java classes, which makes it easy to use this functionality at the class-loading level. This is also demonstrated by the preliminary load-time weaving class loader that is now part of AspectJ 1.2 (and is derived from this work).

## 3. DYNAMIC PLUGINS

The new runtime of Eclipse 3.0 is based on the OSGi specification and runs on an OSGi kernel implementation. This allows plugins to be added and removed from the system at runtime. This feature is already used for the Eclipse SDK, where it is possible to add (and partially remove) plugins at runtime without restarting the Eclipse IDE.

### 3.1 Challenges for AJEER

The dynamic features of the OSGi kernel open up quite new possibilities for AJEER. Users would expect to add and remove all kinds of plugins at runtime – even aspect-promoting plugins. This is not a trivial challenge for AJEER since its basic implementation uses load-time bytecode instrumentation to weave aspects into the system (see [9]). Adding an aspect dynamically to the running system (without any further action) would cause subsequently loaded classes to be woven with the new aspect. Previously loaded classes would not be affected by the aspect. Removing an aspect from the system would have similar effects: because the aspect is woven into some already woven classes, this woven code would remain unchanged in the system. This would

result in quite complicated and unpredictable behavior – not what would be expected if aspects were added or removed dynamically.

To overcome these problems, the next generation of AJEER gives special attention to dynamically added and removed aspects. Removing an aspect-promoting plugin at runtime means unweaving the aspect from the system. Adding an aspect-promoting plugin at runtime means weaving this aspect into the system even if affected classes are already loaded.

## 3.2 Runtime-Like Weaving for AJEER

One way to realize adding and removing aspect-promoting plugins from a running system would be to implement real runtime weaving, as in the AspectWerkz system (see [4]). This is not a trivial task because the AspectJ language offers features that make runtime weaving for the complete language quite difficult (like inter-type declarations). Not even the class hot-swapping features of the new JVMTI interface (provided by JDK 1.5) are capable of handling these changes to class definitions.

To avoid AJEER supporting only a subset of the AspectJ language, we allow aspect-promoting plugins to be added and removed from the running system in a slightly different way. Instead of swapping class definitions at the VM level, AJEER utilizes the dynamic features of the Eclipse runtime's OSGi layer.

### *3.2.1 Removing Aspect Plugins at Runtime*

Weaving an aspect at load time results in a dynamic dependency between the plugin that promotes the aspect and the plugin that contributes the load-time woven class. Removing the plugin that promotes the woven aspect at runtime causes the OSGi layer to stop the dependent plugins, too – and thus all plugins that have loaded a class into which the aspect has been woven.

AJEER is now able to restart these plugins, which causes the original versions of those plugins to be reloaded. And since the aspect is plugged out of the system, the plugin is reloaded without the aspect being woven into the classes of that plugin. All remaining aspects are woven again into the classes of that plugin via the normal load-time weaving mechanism of AJEER.

### *3.2.2 Adding Aspect Plugins at Runtime*

Adding an aspect is slightly more complicated. AJEER must determine which classes that are already loaded into the system would be affected by the new aspect. This is done by keeping track of loaded classes. AJEER is able to analyze these classes to determine whether a class would be affected by the aspect[1]. If such classes are found, the corresponding plugins are reloaded via the OSGi mechanism.

# 4. WHAT THE AUDIENCE WILL SEE

Those attending the demo will see the latest running version of AJEER, which is currently built on the OSGi-based runtime of Eclipse 3.0. They will see several demo applications, including

- a reimplemented version of Chris Laffra's Eclipse Monitor (showed last year at OOPSLA in the static bytecode modification version),
- an IBM research project that draws heavily on the AspectJ-enabled runtime,
- and a demo from the Parallax project (see [11]) which relies on aspect-promoting Eclipse plug-ins to address middleware-specific crosscutting concerns at different MDA-levels of abstraction throughout the development life cycle of distributed middleware-mediated applications.

There will also be an opportunity to take a look under the hood of the implementation.

# 5. SUMMARY AND OUTLOOK

We have discussed the integration of load-time aspect-weaving functionality into the new OSGi-based runtime of Eclipse 3.0 including its dynamic features. The enhanced runtime is fully compatible with the original implementation. This allows applications to be built on top of the Eclipse Rich Client Platform while using AspectJ at the same time.

Achieving good runtime performance and low overhead in memory consumption are still challenges for AJEER. Here there is room for future improvement. AJEER is open-source and can be downloaded at [1].

# 6. ACKNOWLEDGMENTS

I wish to thank the AspectJ team for their help in implementing the weaving class loader and improving weaving performance and all early adopters for their feedback and patience, especially Raul Silaghi, Sabine Hauert, Steven Rohall and Chris Laffra.

# 7. REFERENCES

[1] AJEER homepage: http://www.martinlippert.com/.

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

[3] AspectJ Team. AspectJ homepage. http://www.eclipse.org/aspectj/.

[4] AspectWerkz. http://aspectwerkz.codehaus.org/

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

[6] E. Hilsdale, J. Hugunin. Advice Weaving in AspectJ. In *Proceedings of AOSD '04*. ACM press. 2004.

[7] 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.

[8] 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.

[9] C. Laffra, M. Lippert. Visualizing and AspectJ-enabling Eclipse Plugins using Bytecode Instrumentation. In *OOPSLA '03 Companion*, ACM press, 2003.

[10] M. Lippert. An AspectJ-enabled Eclipse Core Runtime Platform. In *OOPSLA '03 Companion*, ACM press, 2003.

[11] Software Engineering Laboratory at the Swiss Federal Institute of Technology in Lausanne: *The Parallax Project*. http://parallax-lgl.epfl.ch/, August 2004.

[1] In the first naive implementation, by using a combination of the fastmatch functionality (see [6]) and repassing the original bytecode of the class to the weaver to detect possible effects.