The Scalability of AspectJ

by

Arjun Singh

B.Sc., University of California, Davis, 2005

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

April 2007

© Arjun Singh, 2007
Abstract

To assess the scalability of using AspectJ, we refactored concerns that crosscut over half of the plug-ins that comprise the Eclipse IDE. Eclipse is a suitable candidate for furthering other studies on AspectJ’s scalability because the system has an additional modularization mechanism typical of large systems that introduces new complexities for defining advice and aspects. We evaluated quantitative and qualitative properties of our AO refactored version of Eclipse and compared them to their equivalents in the original, OO version of Eclipse. Quantitatively, we evaluated execution time and memory usage. Qualitatively, we evaluated changes in scattering, coupling, and abstractions. Our assessment of the scalability of AspectJ shows that using the language in Eclipse resulted in changes in performance and improvements in code similar to those seen in previous studies on the scalability of AspectJ. This leads us to conclude that AspectJ scales up to large systems. We also conclude that it may be necessary for the system to be aware of aspects in order to deal with defining advice that cross system boundaries.
# Table of Contents

Abstract ................................................................. ii  
Table of Contents ................................................... iii  
List of Tables ......................................................... v  
List of Figures ........................................................ vi  
Acknowledgments ....................................................... vii  

Chapter 1  Introduction ............................................. 1  
1.1 Aspect Oriented Programming and AspectJ ....................... 1  
1.2 Motivation ........................................................ 1  
1.3 Goals and Approach ............................................... 2  
1.4 Overview .......................................................... 2  

Chapter 2  Related Work ........................................... 3  
2.1 Using AspectJ for Specific Concerns ............................ 3  
2.2 Beyond Specific Domains and Concerns ......................... 4  
2.3 Using AspectJ for Small Scale Refactorings .................... 4  
2.4 Techniques for Evaluating Software Qualities in AO Systems 5  
2.5 Context ............................................................ 6  

Chapter 3  The Eclipse Code Base ................................ 7  
3.1 System Overview ................................................ 7  
3.2 The Equinox Sub-Project ....................................... 8  
3.3 The Platform Sub-Project .................................. 8  
3.4 The Interaction of Plug-ins .................................. 8  
3.4.1 The Manifest File ........................................... 8  
3.4.2 Uses of the Manifest File ................................ 10  
3.5 Suitability for a Scalability Study ............................ 10
List of Tables

4.1 Possible states of a plug-in. ......................................... 24

6.1 Reduction in scattering for refactored or implemented concerns. ... 36
List of Figures

3.1 An example MANIFEST.MF file. ........................................ 9
4.1 Abstract aspect for logging of handled exceptions. ........... 14
4.2 Concrete sub-aspect for logging of handled exceptions. ...... 15
4.3 The FFDC Aspect. .................................................. 17
4.4 The FFDC Engine class. ............................................ 18
4.5 Plugin contract enforcement aspect. .............................. 20
4.6 Excerpt from an aspect that implements the performance monitoring concern. ........................................ 21
4.7 Aspect that implements the logging of RuntimeException objects within the OSGi plug-in. ......................... 23
4.8 State transition diagram for the state of a bundle, as it appears in [6]. 24
4.9 Aspect in the OSGi plug-in that manages plug-in state. ....... 25
4.10 Advice from the aspect in the OSGi plug-in that implements the performance monitoring concern. ......................... 26
4.11 An excerpt from the aspect in the OSGi plug-in that implements the method security precondition concern. ............. 27
5.1 Example plug-in target in the ant buildfile. ..................... 29
6.1 CPU time comparison. ............................................. 34
6.2 Memory usage comparison. ....................................... 35
Acknowledgments

First I would like to thank my supervisor, Gregor Kiczales. I am grateful for the opportunity to work with you on this project. I can’t thank you enough for the time and effort you spent with me on this work. Working with you has been a rewarding experience from which I’ve learned an incredible amount. Thanks also to Eric Wohlstadteter for being my second reader.

I would also like to thank members of the SPL, in particular Terry Hon, Ryan Golbeck, Andrew Eisenberg, Brett Cannon, and Ducky Sherwood for their continued support. I would also like to say thanks to Stanley Chiu, Man Hon Chan, and Nathan Allen.
Chapter 1

Introduction

1.1 Aspect Oriented Programming and AspectJ

Aspect Oriented Programming (AOP) [18] is a paradigm in which crosscutting in code can be supported in a modular and expressive way. Crosscutting in code arises due to the fact that system decompositions based on Object Oriented (OO) techniques and designs that modularize a system well for some concerns may not be an adequate design for others. This leads to the situation of tangled code, in which implementations of concerns crosscut each other. AOP provides support for crosscutting concerns by allowing developers to implement these concerns in a modular and expressive fashion. The most prominent incarnation of AOP is the AspectJ language [1]. AspectJ is an extension of Java that provides support for AOP related concepts.

1.2 Motivation

Through the use of AspectJ in both academia and industry, it has become clear that work regarding evaluating how well AspectJ can be used for a particular purpose or in a particular domain is needed. Such work helps answer important questions about the language as it evolves. Indeed there have been such studies in the past. Some studies focus on the ability to use the language in a particular domain. An example is [21], in which it is demonstrated that persistence is a concern that can be aspectized in a highly reusable way. Others provide a more objective analysis of using AspectJ in a system. An example is [13], in which it is shown that AspectJ can be used to reduce complexity in middleware. Such previous work involved using AspectJ for small scale refactorings, by which we mean refactoring concerns using AspectJ in a system of a few thousand classes or less. Although such studies are useful for evaluating AspectJ, the limitation of doing so in small scale systems leaves
open questions related to how well AspectJ can be used in large scale systems.

1.3 Goals and Approach

This work provides answers to such questions by evaluating AspectJ when used in a large system. More specifically, in order to further assess the scalability of AspectJ, we refactored crosscutting concerns in the code base of Eclipse. Eclipse is a Java based open source integrated development environment that consists of well over fourteen thousand classes. More importantly, Eclipse contains additional modularization mechanisms for the purpose of dealing with software complexity, as large systems frequently do. Defining advice that cross system module boundaries in light of these additional modularization mechanisms provides a new context in which to evaluate the use of AspectJ. The concerns we refactored by using AspectJ are concerns that are commonly implemented using Aspects by those in the AOP community. This allowed us to use common, or vanilla aspects. By doing so, we can make an evaluation based on work that mirrors that of refactorings often seen by software developers. We evaluate our use of AspectJ on both quantitative and qualitative fronts. Quantitatively, we compare our refactored system to the original system in terms of execution time and memory usage. Qualitatively, we analyze changes in scattering, coupling, and abstractions in the system. The thesis of this work is that AspectJ scales well in that it can be used in large systems, but that, given the context provided by additional modularization mechanisms often present in large systems, such systems may require knowledge of aspects and advice in order to cope with advice that cross system boundaries.

1.4 Overview

The remainder of this document is structured as follows. Chapter 2 outlines related work. Chapter 3 provides a description of the Eclipse code base. The implementation of crosscutting concerns is described in chapter 4. Chapter 5 discusses issues related to defining advice for multiple plug-ins. Chapter 6 discusses our quantitative and qualitative evaluations. Finally, chapter 7 summarizes this work and its contributions.
Chapter 2

Related Work

Other work has addressed the scalability of AspectJ on some level. The context of using AspectJ and how it is evaluated differ from study to study. Some work focuses on showing that the language is capable of providing a modular implementation for specific concerns. Other work hints that AspectJ could be used in a wider, more general and applicable context. Building upon these, some studies used AspectJ for small scale refactorings and evaluated doing so in some way. An overarching theme in such work is techniques used for the evaluation of AspectJ and more generally AOP. This chapter explores these related areas, and places this work in context with other work.

2.1 Using AspectJ for Specific Concerns

An important way to evaluate AspectJ is to show that it can be used to implement specific concerns. Such work lays the groundwork for further uses of the language. An example of this kind of work is [21], in which Rashid and Chitchyan aspectize persistence in a Java based system that interacts with a database. Their main goal was to evaluate the reusability of aspects that implement the persistence concern, as well as to determine how oblivious developers could be of the aspects. They refactored the system and reported their experiences of doing so, finding that the use of AspectJ in the system to implement the persistence concern was able to modularize the concern in a reusable manor.

Filho et al [14] provide another example of using AspectJ for a specific concern by exploring how well AspectJ modularizes exception handling code. They refactored the code bases of four systems such that the handling of exceptions was implemented with aspects. By using a custom metrics suite to measure changes in various software engineering qualities, Filho et al conclude that AspectJ can be used to modularize exception handling code, but that doing so when the concern
implementation is non-uniform and/or context specific is not beneficial.

2.2 Beyond Specific Domains and Concerns

Other work has involved using AspectJ in more general, more widely applicable ways than for specific concerns. Hannemann and Kiczales [17] developed AspectJ based implementations of the Gang of Four [15] design patterns. They analyzed these AspectJ based implementations by discussing changes in code locality, reusability, composability, and pluggability. In general, they found that changes in these qualities show an increase in modularity when comparing the AspectJ based implementation to the standard OO based implementation, with the extent of the improvement dependent upon the exact pattern. Garcia et al [16] validated the Hanneman and Kiczales (HK) patterns by providing a quantitative study by comparing them to the standard OO pattern implementations using a custom metrics suite designed specifically to evaluate the use of AspectJ. They also found that modularity increased. Cacho et al [9] extended the work of Hanneman and Kiczales by exploring compositions of design patterns implemented in AspectJ. They evaluated how well crosscutting concerns are modularized given intricate pattern interactions through their compositions. They evaluated equivalent AO and OO systems using a quantitative approach similar to [16]. They found that some pattern compositions can modularize concerns, whereas others require complex design decisions. By relating the use of AspectJ to a concept as general as design patterns, such work helps demonstrate that AspectJ can be used for more than just specific concerns.

2.3 Using AspectJ for Small Scale Refactorings

Following this progression of using AspectJ in wider terms, there have been several subsequent studies involving the analysis of using AspectJ in systems of various sizes, in which it is used for small scale refactorings. AO refactoring these systems reflects this wider use of AspectJ, as these systems are larger than simple examples such as JHotDraw [5] that are occasionally used in literature. These systems also involve a variety of refactorings that involve the use of AspectJ.

Kulesza et al [19] explore refactoring the JUnit framework using AspectJ. They refactored several concerns in the system. By reasoning about the separation of concerns and modularity, they argue that using AspectJ resulted in greater understandability and maintainability.

Papapetrou and Papadopoulos [20] implemented three new concerns in a web crawling program. They implemented these three concerns in one copy of the
system using aspects, and another copy using OO techniques. They examined the differences in the development processes of these two systems to observe the impact AspectJ has on these processes. They also examined the modularity of the systems. They found that using AspectJ improved code quality in that the modularity was greater in the AO system. They also found that development time was significantly less in the AO version of the system.

Coady et al [12] refactored concerns in the FreeBSD operating system using an experimental AOP-supported C compiler. Although this work does not use AspectJ, it does deal with evaluating the use of an AOP language based upon AspectJ. By observing changes in modularity, they found that implementing these concerns with aspects leads to better pluggability, a more independent development process of concerns, and better comprehensibility.

Colyer and Clement [13] analyzed the use of AspectJ in an industrial strength middleware system. Their goal was to determine if AspectJ could be used to reduce the inherent complexity present in middleware systems. They refactored several concerns and provided a description of their refactoring process. They concluded that AspectJ could be used to reduce complexity.

Zhang and Jacobsen [23] also address the use of AspectJ in middleware. They used an aspect mining tool to locate concerns whose implementations are scattered and tangled. These concerns were then refactored using AspectJ. The resulting system was analyzed using a custom set of metrics. They concluded that using AspectJ in middleware resulted in negligible performance changes while the customizability of the system was greatly increased.

Cacho et al [8] evaluated the use of AspectJ in reflective middleware. Their goal was to determine if AspectJ can be used to increase the modularity of reflective middleware. Modularity is greatly desired in reflective middleware, as it leads to greater adaptability. After refactoring a middleware system, they used metrics to measure the change in modularity. They determined that using AspectJ in the middleware system led to a large increase in modularity.

2.4 Techniques for Evaluating Software Qualities in AO Systems

One similarity between all related work discussed thus far as well as this dissertation is that each involves the evaluation of AspectJ. Exactly how authors evaluate software qualities of systems in which AspectJ is used differs depending on the context in which they use the language and their preferences in the area of aspect oriented measurements. Given that this work includes a qualitative evaluation of
using AspectJ, this section will explore previous approaches as well as the area of the assessment of aspect oriented technologies.

From the description of previous work provided above, it is apparent that there are different methods that have been used to evaluate software qualities in AO systems. There are two common approaches that present themselves. One is to reason about changes in properties of the system(s) used in the work, the other is to use custom metrics to provide a quantitative basis for the evaluation. Reasoning about changes in software qualities is straightforward; certain software qualities are selected and described in the AO refactored system(s) and compared to their equivalents in the OO or pre-refactored system(s). Using software metrics involves using new metrics that are defined specifically for evaluating the use of AOP.

Software metrics for assessing the use of AspectJ or more generally AOP is a new area of research. There are several studies that provide new definitions for metrics that are well known for OO evaluations. For example, Zhao attempts to provide new definitions for coupling [24] and cohesion [25] such that these quantities account for aspects. Other work focuses on defining entirely new metrics that can provide better reflections of the impacts of using aspects. An example is Sant'Anna et al [22]. They define a number of new metrics in an attempt to better measure the impact that the use of AOP has on separation of concerns, coupling, and cohesion. Another example of this is [10], which also defines new metrics to better understand the impact of using AspectJ. These metrics allow for a quantitative comparison of software qualities in AO systems and their OO equivalents.

2.5 Context

This dissertation extends upon previous work by evaluating AspectJ in similar ways but in a new context. As mentioned previously, Eclipse has modularization mechanisms that introduce new complexities in terms of defining advice. These modularization mechanisms are typical of such large scale systems. Using Eclipse as the basis for our study allows us to evaluate AspectJ in a new context, as systems used in previous studies do not resemble Eclipse in this manor. We evaluate the use of AspectJ in this new context and build upon the conclusions of previous studies.

In terms of the evaluation of software qualities of our AO system, we take the approach of reasoning about changes in software qualities. Although metrics designed for evaluating the use of AOP are appealing, the lack of a definitive work in this area leads us to have a skeptical view of these metrics. Perhaps future work in this young and promising field will lead to such a work; an equivalent for the AO community to the widely recognized [11] in the OO community.
Chapter 3

The Eclipse Code Base

To extend upon previous work and further evaluate the use of AspectJ, we chose to use the language in the code base of the Eclipse integrated development environment (IDE) [3] and assess the impact of doing so. We will refer to the Eclipse IDE code base simply as Eclipse. Eclipse is a large open source project implemented in Java. In addition to being a popular IDE, Eclipse also contains components that provide a robust applications framework. This chapter provides details about this code base. These details provide the knowledge needed to understand chapters 4 and 5.

3.1 System Overview

We refactored version 3.2.0 of Eclipse. Conceptually, the system is divided into four sub-projects, each concerned with providing a specific set of functionality which together create the Eclipse IDE. These four are known as the Platform, Equinox, Java development tools (JDT), and Plug-in development environment (PDE) sub-projects.

Each sub-project of Eclipse contains modules, referred to as plug-ins, that implement specific components of these sub-projects. Plug-ins work together to provide the functionality of each sub-project. For example, the PDE sub-project contains plug-ins that implement building support for PDE projects and other plug-ins that implement the PDE user interface. Plug-ins are packaged and deployed as jar files. Each plug-in contains compiled class files and any additional resources associated with the plug-in. Every plug-in also contains a manifest file, commonly named MANIFEST.MF. This manifest file and its importance are discussed later.
3.2 The Equinox Sub-Project

The Equinox sub-project [4] contains a plug-in that is an implementation of the OSGi framework [6]. We will refer to this plug-in as the OSGi plug-in. The OSGi framework is a design document that describes a Java based dynamic module system. The goal of the OSGi framework is to describe a well defined component based system that is capable of tackling the problem of complexity in software systems.

The OSGi framework provides a standardized environment for all sub-components of a system. This is achieved by defining the execution environment, class loading policies, life cycle management, and shared services for all sub-components of the system. In the context of Eclipse, the OSGi plug-in manages execution environments for Eclipse, class loading and resolution between plug-ins, plug-in life cycles, and shared services among plug-ins. Thus the OSGi plug-in manages all other plug-ins and the interactions between them.

3.3 The Platform Sub-Project

The Platform sub-project [7] contains core frameworks and services that other plug-ins can extend to provide additional functionality. In particular, the core runtime plug-in contains many abstract classes and interfaces intended to be used by other plug-ins that extend the Platform sub-project. This plug-in also contains some runtime API. We will refer to this plug-in as the core runtime plug-in.

3.4 The Interaction of Plug-ins

Under the hood, the functionality of the Eclipse IDE is implemented as an interaction between various plug-ins. Upon system startup (e.g. when the user launches the application), the OSGi plug-in is started. This plug-in in turn starts and manages all other plug-ins. When a plug-in is started by the OSGi plug-in, it is referred to as installed or started by the OSGi plug-in.

3.4.1 The Manifest File

A key component of how the framework is able to manage other plug-ins is the manifest file present in each plug-in. In addition to describing the contents of the jar file containing the plug-in, the manifest file provides essential information about the plug-in. Manifest files specify attributes such as dependencies, requirements, services, and many other properties associated with operating within the OSGi framework. These are expressed in the manifest file in the form of attribute:value(s).
An example manifest file is shown in figure 3.1. This example is taken from the org.eclipse.ant.core plug-in. An immediate observation is that the term Bundle is used. This is the name used in the OSGi framework for the components run by the framework. In Eclipse, these are plug-ins. Thus the terms bundle and plug-in are interchangeable. We use the term plug-in, as this is the Eclipse-specific term.

For the purposes of our work, only a subset of the attributes specified in this example MANIFEST.MF file are relevant. Require-Bundle specifies plug-ins required for type resolution and functionality requirements. In the above example, the org.eclipse.ant.core plug-in requires the core runtime, org.eclipse.core.variables, and org.aspectj.runtime plug-ins be installed. An alternative to the Require-Bundle attribute is the Require-Package attribute. This attribute specifies specific packages that are required. The manifest files in many plug-ins use the Require-Bundle attribute to specify dependencies, although both may be used to achieve this goal. Export-Package lists the packages defined in the plug-in that will be visible to other plug-ins. In the above example, the org.eclipse.ant.core, org.eclipse.ant.internal.core, and org.eclipse.internal.core.contentDescriber packages, all of which are defined in the org.eclipse.ant.core plug-in, will be available to other plug-ins. Finally, the Bundle-Activator attribute specifies the class in the plug-in that implements the BundleActivator interface. This interface, defined in the OSGi plug-in, contains a start method that is concretized by implementors of the interface. It is this start
method in the class specified by the Bundle-Activator attribute that will be invoked when the plug-in is started by the OSGi plug-in.

3.4.2 Uses of the Manifest File

The attributes in the Manifest file are used by the framework during both resolving a plug-in and also the runtime class loading procedure. Resolving a plug-in refers to checking all it’s requirements based on attributes within it’s manifest file. The runtime class loading procedure specified in the OSGi framework provides detailed mechanisms that control access to plug-ins and the classes they define. Every plug-in has it’s own Java class loader. The class loader for a plug-in provides access to java.* packages as well as classes defined within that plug-in. Accesses to classes defined outside of java.* packages and the plug-in are strictly controlled based upon the attributes defined by the plug-in’s manifest file, as well as those in the manifest files of the plug-ins that are trying to be accessed.

3.5 Suitability for a Scalability Study

Eclipse is a suitable software system to use in a scalability study of AspectJ for several reasons. First is the obvious reason - it is a large system written in Java. Given that AspectJ is a superset of Java, the most basic requirement of our system is that it be written in Java such that we can use AspectJ without needing to rewrite any of the original code. Size is another important factor given the previous work that addresses the scalability of AspectJ, as this work must extend upon previous scalability studies. Middleware systems typically consist of a few thousand classes. For example, systems used as examples in [13] consisted of approximately three thousand classes. The system used in this study must supersede this by a considerable amount. Eclipse meets this criteria, as it contains well over fourteen thousand classes.

Although the fact that Eclipse is a large Java system is important, the more important quality of Eclipse that makes it a desirable system to use for a scalability study is the fact that it contains an additional modularization mechanism that systems in previous AspectJ scalability studies do not posses. This additional modularization mechanism – the OSGi plug-in’s implementation of the OSGi framework – operates at a more coarse scale than Java classes. Modularizing system components by creating a new entity, the plug-in, as well as defining the framework around this entity introduces new abstractions, dependency and deployment mechanisms, and class loading policies that can create new complexities when defining aspects and advice using AspectJ. Given that such modularization mechanisms are frequently
used in large systems such as Eclipse, evaluating the use of AspectJ in this context allows us to progress the study of AspectJ’s scalability in a more interesting way than simply increasing the size of the code base involved in the study.
Chapter 4

Implementation and Refactoring of Crosscutting Concerns

4.1 Approach

Our approach to using AspectJ in Eclipse involved systematically refactoring the plug-ins by refactoring or implementing specific concerns using common, or vanilla, aspects. These aspects are commonly used by programmers in the AOP community to implement well known crosscutting concerns. We chose to refactor crosscutting concerns that can be refactored using vanilla aspects because we are interested in evaluating the effects of using AspectJ in a large system, not in finding new kinds of aspects or refactorings.

Throughout our systematic refactoring of Eclipse, we focused on ensuring that our work improved the overall quality of the system. Rather than using artificial aspects that simply advise a high number of join points, we focused on using aspects that help improve qualities such as modularity and expressiveness of the code. Using aspects that help improve desirable software qualities mimics some of the goals of typical OO refactoring that occurs in industry, allowing us to formulate a meaningful and relevant evaluation of using AspectJ in Eclipse. Using artificial aspects would not have this similarity to actual refactorings and thus would have undermined the evaluation.

We looked for well known crosscutting concerns that crosscut multiple plug-ins. This approach allowed us to use and then evaluate AspectJ in the context of the greater modularization mechanism discussed previously. The one exception to looking for concerns across plug-ins is the OSGi plug-in. Given that this plug-in is the central entity of this additional modularization mechanism, we thought it may be interesting and worthwhile to refactor this plug-in as well.
We refactored the Eclipse code base by using the Eclipse IDE 3.2.0, the AJDT tool suite 1.4.0, as well the AspectJ compiler 1.5.3. To locate points in the code base relevant to the concerns we refactored, we used the standard Eclipse search functionality. Our changes to the code typically involved the removal of code. This was done manually. We did not use any additional tools (e.g. aspect mining tools) during our refactoring process.

4.2 Concerns that Crosscut Multiple Plug-ins

4.2.1 Logging of Handled Exceptions

When an exception is caught in a catch block, information related to the fact that the exception occurred may be logged. The implementation of this logging occurs either in the body of the catch block or in methods invoked from those bodies. The logging of handled exceptions is a crosscutting concern because the implementation is scattered throughout many classes, tangled with the implementations of other concerns. The plug-ins implement logging of exceptions by writing to the Eclipse log file. This involves appending a String object that encapsulates information about the exception to the end of the log file. Some plug-ins use a single string for most or all logging of exceptions, while others use different strings for different exceptions.

We refactored plug-ins for which a single, common string was used for most or all logging of handled exceptions. Refactoring logging of handled exceptions that does not use a common string for most or all exception logging would not improve the expressiveness of the code and would result in fragile pointcuts, as these strings are often formed from locally defined data.

We created an abstract aspect to capture the general policy of logging handled exceptions. The code for this aspect is shown in figure 4.1. This aspect resides in the core runtime plug-in. This aspect contains a pointcut named scope, which sub-aspects use as a means to specify the packages, types, and methods for which logging of exceptions should occur. This can be seen on line 5. An abstract method named logThrowable is also defined, which is intended to provide the plug-in specific logging implementation. Sub-aspects provide the plug-in specific details of this method by providing it’s concrete implementation. To accommodate exemptions to this logging policy, two options can be used. One is to use an annotation to mark a class or method as an exemption to the general logging policy. The other is to concretize the additionalExemptions pointcut. Both of these options are used to form the policyExemptions pointcut on line 10. All these elements of the AbstractExceptionLogging aspect are tied together by a before advice, as shown on line 27.
public abstract aspect AbstractExceptionLogging {
    /**
     * Indicates scope of the exception logging policy.
     */
    abstract protected pointcut scope();

    /**
     * Specifies exceptions to general policy.
     */
    protected pointcut policyExemptions():
        within (@ExceptionLoggingExemption *)
        || withinCode (@ExceptionLoggingExemption * *(..))
        || additionalExemptions();

    protected pointcut additionalExemptions();

    /**
     * Method that implements the exception logging policy.
     * This is left as abstract so that each plugin can provide
     * its own implementation.
     */
    abstract protected void logThrowable(Throwable t);

    /**
     * Advice that ties everything together.
     */
    before (Throwable t):
        handler (Throwable+)
        .&& scope()
        && !policyExemptions()
        && args(t) {
        logThrowable(t);
    }
}

Figure 4.1: Abstract aspect for logging of handled exceptions.
public aspect ExceptionLogging extends AbstractExceptionLogging {

protected pointcut scope() :
within(AntCorePreferences) || within(AntPropertyValueProvider) || within(Property);

protected void logThrowable(Throwable t) {
    AntCorePlugin.log(t);
}
}

Figure 4.2: Concrete sub-aspect for logging of handled exceptions.

In terms of using the annotation for specifying exemptions to the general policy, line 11 (part of the definition of policyExemptions) will match join points in entire types that are marked with the ExceptionLoggingExemption annotation. Line 12 will match join points in specific methods marked with the annotation. It is useful to mark a class with the annotation when an entire package should be part of scope, but there are a small number of types for which the general exception logging policy should not be applied. Similarly, it is useful to mark a method with the annotation when a class should be part of scope but there are a small number of methods for which the general policy should not be applied.

Plug-ins refactored for this concern each contain a sub-aspect that provides the plug-in specific details as described above. An example of such a sub-aspect is shown in figure 4.2. Sub aspects are typically similar to this example. Sometimes scope was as broad as entire package names and their sub packages, other times class names or even specific method names were used.

We refactored this concern as follows. For each plug-in, we used the Eclipse search functionality to locate references to logging methods. These search results were examined to determine if they were part of this concern. If any such references to log methods were part of this concern, we created a sub-aspect of the abstract aspect inside the plug-in, added appropriately to scope, defined logThrowable, and removed the logging mechanism in the exception handlers. We used the annotation in order to avoid making the scope pointcut fragile and unexpressive. For example, if 18 out of 20 handlers in a class should be advised by the aspect but 2 should not, then we used the class' type name in scope and the annotation in the class to prevent those 2 from being advised, rather than listing many methods in scope. This
refactoring involved 29.7% of the plug-ins in the SDK. Of these plug-ins, 58% (or 17.2% of all plug-ins in the SDK) contained at least one use of the annotation.

4.2.2 First Failure Data Capture

Throughout the Eclipse code base, exceptions of the type CoreException are used to represent internal failures within the system. These exceptions often propagate through various methods and classes after they are first thrown, possibly being logged at some point. The concept of First Failure Data Capture (FFDC) is the act of capturing relevant information at the first point of a failure [13]. An implementation of FFDC for Eclipse's CoreException objects is useful because identifying the first point of failure when dealing with these exceptions would allow for faster and more accurate bug identification and fixes in the SDK by eliminating the need to perform traces through the Eclipse log file. FFDC is a crosscutting concern because an OO-based implementation would require that each plug-in contain the same logic for keeping track of and logging CoreException objects. Note that this concern is not a feature that already exists in Eclipse, but rather one that we implemented by using AspectJ.

We implemented FFDC for Eclipse by creating an aspect that logs the first time each CoreException object is handled or thrown. This aspect is shown in figure 4.3. The FFDC aspect contains a before advice that advises all handlers of CoreException objects. There is also an after throwing advice that advises join points where CoreException objects are thrown. The bodies of these advices both invoke a method in the FFDCEngine class, shown in figure 4.4. This method checks to see if the CoreException that was just thrown or handled is the same as the last CoreException seen by the aspect. If not, information is added to the Eclipse log file. This information includes clear indications that this exception is a point of first failure. The excluded pointcut on line 4 specifies exemptions to the general FFDC policy.

We added the FFDC aspect and the FFDCEngine class to the core runtime plug-in. To ensure that the advice in the FFDC aspect are defined for all plug-ins, we used the ajc compiler to weave the advice into the jar files containing the plug-ins of Eclipse (this process is described in more detail in chapter 5). The FFDC aspect affected 59% of the plugins in Eclipse.

4.2.3 Plug-in Contract Enforcement

Each plug-in in Eclipse contains one sub-class of the Plugin class. Each of these subclasses overrides methods of Plugin that are used by the OSGi plug-in to manage
public aspect FFDC {
    protected pointcut ffdcScope(): within(org.eclipse..*);

    protected pointcut excluded():
        within(FFDC+)
        || within(FFDCEngine+)
        || within(PlatformActivator);

    before(CoreException c):
        ffdcScope()
        && !excluded()
        && handler(CoreException+)
        && args(c) {
            logThrowable(thisJoinPointStaticPart.toShortString(), c);
        }

    after() throwing(CoreException c):
        ffdcScope()
        && !excluded()
        && !handler(*) {
            logThrowable(thisJoinPointStaticPart.toShortString(), c);
        }

    /**
     * Method invoked by advice in this aspect. This method
     * is called rather than directly interacting with
     * FFDCEngine so that subaspects may implement their own
     * functionality if needed.
     */
    protected void logThrowable(String joinPointStr, Throwable th) {
        FFDCEngine.logThrowable(joinPointStr, th);
    }

    after(Plugin activator):
        execution(void PlatformActivator.start(..))
        && this(activator) {
            FFDCEngine.setLog(activator.getLog());
        }
}

Figure 4.3: The FFDC Aspect.
public class FFDCEngine {
    /**
     * The last exception that has been seen.
     */
    private static Throwable lastSeen = null;

    /**
     * Platform log used to log first failures.
     */
    private static ILog log = null;

    /**
     * Method used to set the engine's ILog object. Expected
     * to be called before any other methods are used.
     */
    public static void setLog(ILog pluginLog) {
        log = pluginLog;
    }

    /**
     * Method provided for logging potential first failure
     * data points. If the th parameter has not been seen
     * before, it is logged as a failure via the eclipse
     * platform log.
     */
    public static void logThrowable(String joinPointStr,
                                     Throwable th) {
        if (th != lastSeen) {
            log.log(new Status(IStatus.ERROR,
                                "org.eclipse.ffdc",
                                th.hashCode(),
                                "Error in " + joinPointStr, th));
            lastSeen = th;
        }
    }
}

Figure 4.4: The FFDCEngine class.
that plug-in’s lifecycle. These methods are not intended to be invoked by any entity other than the OSGi and core runtime plug-ins. Although the comments in the Plugin class clearly indicate this, any plug-in can violate this policy.

We implemented an aspect that provides policy enforcement for the Plugin class. This aspect is shown in figure 4.5. This aspect declares calls of lifecycle methods outside of specific packages in the OSGi and core runtime plug-ins as errors. Lifecycle methods are defined by the restrictedOperations pointcut on line 15. Various pointcuts are used to specify packages and classes that are allowed to access these methods, the union of which is represented by the accessAllowed pointcut on line 9. These two pointcuts are used with the declare error construct of AspectJ on line 6. This means access to lifecycle methods of a plug-in outside of the packages specified by accessAllowed will show up as errors during the system build.

The implementation of this concern involved adding this aspect to the core runtime plug-in. This concern, as with the FFDC concern, required the build process described in chapter 5.

4.2.4 Performance Monitoring

The core runtime plug-in provides the PerformanceStats class, which encapsulates the implementation of performance monitoring. Typical usage consists of retrieving an instance of the class by invoking one of it’s static methods. A method in this class is then invoked to start performance monitoring before a particular block of code. This is followed later by calling a method in the class to stop the performance monitoring after that block of code. These calls are frequently at the start and end of methods. Although the implementation of performance monitoring is already modularized in PerformanceStats, the calls to invoke it crosscut classes throughout several plug-ins.

We refactored the use of PerformanceStats such that each plug-in that needs to implement the performance monitoring of methods has an aspect containing one or more around advice. These around advice invoke the start of performance monitoring, proceed with the execution of the advised method, and then invoke the end of performance monitoring. An example advice from the performance monitoring aspect in the org.eclipse.workbench.ui plug-in is shown in figure 4.6.

These aspects replaced the scattered OO implementation, which we found using the Eclipse search functionality to find references of the PerformanceStats type. We removed such references manually. This refactoring involved 5.9% of the plug-ins in the SDK.
public aspect PluginRestrictions {
    /**
     * Declare an error if a restricted operation is being
     * accessed outside the set of allowed join points.
     */
    declare error: ! accessAllowed() && restrictedOperations():
        "Restricted operation of class Plugin being invoked.";

    pointcut accessAllowed(): frameworkScope() || runtimeScope();

    /**
     * Join points where the Plugin class' lifecycle methods are
     * called.
     */
    pointcut restrictedOperations():
        call( Plugin.new(..) )
        | call(void Plugin.shutdown() )
        | call(void Plugin.startup() )
        | call(void Plugin.start() )
        | call(void Plugin.stop() );

    /**
     * Join points in the org.eclipse.osgi project.
     */
    pointcut frameworkScope():
        within org.eclipse.core.runtime.adaptor.*
        | within org.eclipse.core.runtime.internal.adaptor.*
        | within org.eclipse.core.runtime.internal.stats.*
        | within org.eclipse.osgi..*
        | within org.osgi..*

    /**
     * Join points in the org.eclipse.core.runtime project
     */
    pointcut runtimeScope():
        within org.eclipse.core.internal.preferences.legacy.*
        | within org.eclipse.core.internal.runtime.*
        | within org.eclipse.core.runtime.*
        | within org.eclipse.policy.*
        | within org.eclipse.policy.annotations.*;
}

Figure 4.5: Plugin contract enforcement aspect.
void around (PackageExplorerPart part):
    execution (void PackageExplorerPart.createPartControl (..))
    && this(part) {
    final PerformanceStats stats
    = PerformanceStats.getStats(PERF_CREATE_PART.CONTROL, part);
    stats.startRun();

    proceed(part);

    stats.endRun();
}

Figure 4.6: Excerpt from an aspect that implements the performance monitoring concern.

4.2.5 The Singleton Design Pattern

A design pattern used in the Eclipse code base is the Singleton design pattern [15]. To participate in this pattern, a class implements a method that provides access to its singleton instance. The implementation of this pattern exists in several classes across the code base.

We refactored the OO-based implementation of this design pattern using the AOP-based HK version. In this version of the Singleton pattern, there is an abstract aspect that contains the Singleton interface, which is used to designate classes as participants in the pattern using the declare parents construct. The aspect advises calls to the constructor of such classes with an around advice. The call proceeds if such a call has not been invoked previously, with the aspect storing the newly created object. If this has already happened, then the call to the constructor does not proceed and the existing object is returned. Plug-in specific sub-aspects define exceptions to the general policy. Code for this concern is identical to that in [17].

The abstract aspect was placed in the core runtime plug-in, with each plug-in that participates in the pattern containing a sub-aspect. Using aspects to implement the singleton pattern allowed for the removal the OO based implementation from classes participating in the pattern. This involved searching for and then removing declarations of getInstance() methods, which was done using the Eclipse search functionality. Calls to getInstance() methods were manually changed to calls to the appropriate class constructor. This refactoring involved 8.9% of the plug-ins in the SDK.
4.3 Concerns that Crosscut the OSGi Plug-in

As discussed previously, we thought it would be a worthwhile endeavor to refactor the OSGi plug-in. This section describes these refactorings. For reasons discussed in chapter 5, concerns in the OSGi plug-in had to be refactored separately from the concerns presented thus far, despite some similarities that exist.

4.3.1 Logging of Handled RuntimeException Objects

The crosscutting behavior observed with respect to this concern is that when exceptions of the type RuntimeException are caught in handlers, information related to the fact that the exception has occurred is logged. Logging is implemented in this plug-in by invoking methods of the Profile class. Such invocations specify the exact message to log using a String parameter. This parameter was either the name of the method in which the exception was being handled, or the name of this method with the string "-loop" appended, depending upon the context in which the exception was handled. Both cases are covered by the aspect, shown in figure 4.7. The pointcut loggingPoints, defined on line 3, handles the former of these cases. The pointcut customLoggingPoints, defined on line 10, handles the latter. Note that some of the less relevant details of the aspect in figure 4.7 have been omitted for space concerns (this is indicated by the use of ellipses).

4.3.2 Plug-in State Management

The OSGi plug-in is responsible for maintaining the state of each plug-in. This state represents the current point in the plug-in’s lifecycle. Possible states are shown in table 4.1.

Changes in plug-in state are triggered by specific actions, which are clearly articulated in the OSGi framework. This is an implementation of a finite state machine, which is another kind of crosscutting concern that is often implemented with an aspect by AspectJ developers. This finite state machine can be described visually, as shown in figure 4.8.

We created an aspect for the OSGi plug-in that contains a variety of before and after advice that control the modification of plug-in state. This implementation allowed for a direct translation from the description of plug-in state in the OSGi framework. Part of this aspect is shown in figure 4.9. In this excerpt from the plug-in state aspect, triggers for changing a plug-in’s state to INSTALLED are defined by the pointcuts install, update, and refresh. These correspond to the edges going into the INSTALLED state in figure 4.8. These pointcuts are used in the after advice on lines 13 and 19. The aspect also includes a declare error statement that makes any
public aspect RuntimeExceptionLogging {

...  

pointcut loggingPoints():
    within(ContentHandlerFactory)
    || within(MultiplexingFactory)
    || within(MultiplexingURLStreamHandler)
    || withincode(*
        StreamHandlerFactory.findAuthorizedURLStreamHandler(...));

pointcut customLoggingPoints():
    withincode(Object
        MultiplexingFactory.findAuthorizedFactory(••))
                || withincode(ContentHandler
        ContentHandlerFactory.findAuthorizedContentHandler(••))
                || withincode(URLStreamHandler
        StreamHandlerFactory.findAuthorizedURLStreamHandler(...));

before(Exception e):
    loggingPoints()
    && !customLoggingPoints()
    && handler(Exception)
    && args(e) {
        logException(thisJoinPointStaticPart, e);
    }

before(Exception e):
    customLoggingPoints()
    && handler(Exception)
    && args(e) {
        logException(thisJoinPointStaticPart, e, "— loop");
    }

private void logException(JoinPoint.StaticPart jp,
    Exception e) {
    logException(jp, e, null);
}

private void logException(JoinPoint.StaticPart jp,
    Exception e, String customMessage) {
    ...
}

Figure 4.7: Aspect that implements the logging of RuntimeException objects within
the OSGi plug-in.
<table>
<thead>
<tr>
<th>Plug-in State</th>
<th>State Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Installed</td>
<td>The plug-in has been successfully installed in the framework.</td>
</tr>
<tr>
<td>Resolved</td>
<td>All Java classes that the plug-in requires are available, meaning that the plug-ins it depends on are already installed in the framework. This state indicates that the plug-in is either ready to be started or has stopped.</td>
</tr>
<tr>
<td>Starting</td>
<td>The plug-in is being started. This means that the plug-in’s BundleActivator.start method has been called, but has not yet returned.</td>
</tr>
<tr>
<td>Active</td>
<td>The plug-in has successfully started and is running.</td>
</tr>
<tr>
<td>Stopping</td>
<td>The plug-in is being stopped. The BundleActivator.stop method has been called but the stop method has not yet returned.</td>
</tr>
<tr>
<td>Uninstalled</td>
<td>The plug-in has been uninstalled from the framework. It cannot move into another state.</td>
</tr>
</tbody>
</table>

Table 4.1: Possible states of a plug-in.

Figure 4.8: State transition diagram for the state of a bundle, as it appears in [6].
public aspect StateManagement {
    /*
     * Pointcuts and advice for changing the state to INSTALLED
     */
    pointcut install():
        execution(AbstractBundle.new(..));
    pointcut update():
        execution(boolean Bundle.*.resetResolvedBundle());
    pointcut refresh():
        execution(void AbstractBundle.refresh())
        && !within(SystemBundle);
    after(AbstractBundle bundle):
        (install() || update())
        && target(bundle) {
            bundle.setState(Bundle.INSTALLED);
        }
    after(AbstractBundle bundle): refresh() && target(bundle) {
        if (bundle.getState() == Bundle.RESOLVED) {
            bundle.setState(Bundle.INSTALLED);
        }
    }
    declare error:
        set(* AbstractBundle.state) && !within(StateManagement):
        "Modification to state of bundle outside of StateManagement";
}

Figure 4.9: Aspect in the OSGi plug-in that manages plug-in state.

change of a plug-in's state outside of the aspect an error. We found the OO based implementation by searching for references to the state field in the AbstractBundle class in the OSGi plug-in. We removed this implementation manually.

4.3.3 Performance Monitoring

The Profile class contains several static members used to implement performance monitoring for the OSGi plug-in. Typically, references to Profile related to the performance monitoring concern consist of a conditional statement to check if performance monitoring is enabled, and if so, calling a method that invokes the start of monitoring. A separate method is invoked later to stop the monitoring. These calls
pointcut monitorPoints():
  execution(static Object
    EclipseStarter.run(String[], Runnable))
  || execution(static BundleContext
    EclipseStarter.startup(String[], Runnable))
  || execution(void
    BundleContextImpl.startActivator(BundleActivator))
  || execution(void Framework.initialize(FrameworkAdaptor));

before(): monitorPoints() {
  Signature signature = thisJoinPointStaticPart.getSignature();
  Profile.logEnter(signature.getDeclaringTypeName() + "." + signature.getName() + "()";
}

after(): monitorPoints() {
  Signature signature = thisJoinPointStaticPart.getSignature();
  Profile.logEnter(signature.getDeclaringTypeName() + "." + signature.getName() + "()";
}

Figure 4.10: Advice from the aspect in the OSGi plug-in that implements the performance monitoring concern.

were often found at the start and end of methods. Thus the performance monitoring of methods in the OSGi plug-in is a crosscutting concern, as such invocations of these methods in the Profile class are scattered through the plug-in. We refactored such uses of Profile by removing them and placing them into an aspect. We did so by searching for references to the SecurityManager class and manually moving the implementation of the concern into an aspect. This aspect contains a variety of pointcuts and advice, examples of which are shown in figure 4.10.

4.3.4 Method Security Preconditions

Another crosscutting concern in the OSGi plug-in is the invocation of methods in the SecurityManager class at the beginning of methods. The SecurityManager class is used to perform necessary permission checks at the start of methods that require such checks. We refactored these calls by moving them to an aspect. This aspect uses several before advice that invoke the necessary calls to SecurityManager. These calls are advised by the aspect itself with an around advice that performs a not-null check for the singleton instance of SecurityManager. A portion of this aspect is shown in figure 4.11. We found the OO based implementation by using the Eclipse

26
```java
public aspect MethodSecurityPreconditions {
    void around(SecurityManager sm):
        within(MethodSecurityPreconditions)
        && call(* SecurityManager.check*(..))
        && target(sm) {
            if (sm != null) {
                proceed(sm);
            }
        }
        return;
}

before(String key):
    ( execution(static String
        FrameworkProperties.setProperty(..))
    || execution(static String
        FrameworkProperties.clearProperty(..))
    && args(key, ..) {
        checkPropertyWriteAccess(key);
    }
}

private void checkPropertyWriteAccess(String key) {
    System.getSecurityManager().checkPermission(
        new PropertyPermission(key, "write");
}

Figure 4.11: An excerpt from the aspect in the OSGi plug-in that implements the method security precondition concern.

search functionality to find references to SecurityManager, removing the relevant uses manually.

27
Chapter 5

Defining Advice that Cross System Boundaries

Defining aspects and advice for concerns that cross plug-ins results raises interesting issues related to the use of AspectJ in large systems. This chapter describes our experiences of defining advice that cross system boundaries and discusses the consequences of doing so.

5.1 Build Process

As mentioned previously, all aspects and refactorings were done using the Eclipse IDE. For the aspects that affected only one plug-in, e.g. those created during the refactoring of the OSGi plug-in, the normal AJDT tools could be used to build such plug-ins. However, for aspects involving concerns that cross plug-in boundaries, e.g. the aspect that implements the FFDC concern, a different method for building the system was used. The reason for taking this approach was because the advice defined in these aspects are intended to be defined for multiple plug-ins. Simply using the AJDT tools is not an adequate method for building the system with these aspects. The AJDT tools do actually provide a mechanism for achieving this goal: for the project containing the aspects with advice that are intended to be defined for multiple plug-ins, that project's inpath would have to list the jar files for all other plug-ins. However, given the number of plug-ins and their size, this is simply not a practical process for building the entire system of the plug-ins (Eclipse runs out of memory).

To build projects involving aspects that contain advice intended to be defined for multiple plug-ins, we used a two step process. First, we built and exported the each plug-in from within the Eclipse IDE. Note that any aspects defined for only
specific plug-ins will be woven during the building of the plug-in (compile time weaving is used). Exporting the plug-in produces a jar file containing the plug-in and any additional resources it may require. The second step in the build process was to use an ant buildfile that makes use of the ajc ant target. This ant buildfile contains one target for each plug-in. Each such ant target is used to weave in aspects that contain advice for multiple plug-ins. To do this, the inpath, argfiles, classpath, and outjar options for the ajc ant target are used. This exposes the types in the jar file specified in the inpath option to the aspects in the file listed in the argfiles option. The outjar option simply specifies the name of the output jar. Within the target, the classpath element is used to specify the path to aspectjrt.jar as well as all files in the plug-in directory of Eclipse. All plug-ins are needed on the classpath for type resolution. An example target from the ant buildfile is shown in figure 5.1. In this figure, full path names have been omitted for space concerns. The entire ant buildfile was generated by a python script.

5.2 The Effects of Defining Advice that Cross Plug-in Boundaries

The process for building the plug-ins just outlined was successful in weaving in advice code at compile time. However, doing so had interesting and unintended consequences.

Recall that the OSGi plug-in will use each plug-in's manifest file to manage
the plug-in within the overall system. Specifically, each manifest file contains entries that detail dependencies, which types defined within the plug-in are viewable to other plug-ins, and other vital information. During the normal building of a plug-in that takes place when doing so from within the Eclipse IDE, the PDE tools incorporate the use of these manifest files into the build process. For example, suppose we are building plug-ins A and B from within the Eclipse IDE. If plug-in A contains a reference to a class that is defined in plug-in B, but plug-in B is not listed as a dependency in plug-in A’s manifest file, then an error will be produced during the building process.

By using the ajc ant task during our build process to weave advice intended to be defined for multiple plug-ins into the jar files of various plug-ins, uses of the FFDCEngine class were added to plug-ins advised by the FFDC aspect. Depending on the plug-in, this may have woven in references to a class that could not be resolved by using manifest files, leading to errors at run time. These errors happen at run time because we do not use the manifest file during the second part of the build process. Recall that the FFDCEngine class resides in the core runtime plug-in. Weaving in uses of this class introduces a (possibly new) dependency on the core runtime plug-in for the plug-in that the advice body was woven into. Since FFDCEngine resides in the core runtime plug-in and most plug-ins already depend on it, this is not an issue for the majority of plug-ins. However, there are two cases in which this issue did affect plug-ins. These are outlined in the following two subsections.

5.2.1 Dependency Errors

Weaving in references to FFDCEngine into a plug-in whose manifest file does not indicate a dependency on the plug-in or package containing FFDCEngine (meaning the core runtime plug-in), the OSGi plug-in throws a NoClassDefFoundError exception at run time. Even if the core runtime plug-in is installed in the system when the use of FFDCEngine is encountered in the plug-in in question, this reference to FFDCEngine will result in an error. This is due to the OSGi plug-in’s management of plug-ins and the overall class loading procedure, as described earlier.

5.2.2 Bootstrapping Errors

There is a small subset of the plug-ins that the OSGi plug-in starts prior to starting the core runtime plug-in. Plug-ins will fall into this subset for either of two reasons. One is that a plug-in is listed as a dependency in the core runtime plug-in’s manifest file. The other reason is that a plug-in is listed prior to the core runtime plug-in in
the config.ini file of Eclipse. This file specifies certain properties of Eclipse. Among these properties is a list of plug-ins that should be started by the OSGi plug-in by default. Recall that a plug-in must be started by the framework before it's classes can take part in the class loading process. These plug-ins that are started before the core runtime plug-in is started will thus not be able to resolve references to FFDCEngine. As such, weaving in references to this class will result in the OSGi plug-in throwing a NoClassDefFoundError exception at run time.

5.2.3 Consequences

These dependency and bootstrapping issues meant that we could not weave our FFDC aspect into plug-ins that the core runtime plug-in depends on or into plug-ins that the framework starts prior to starting the core runtime plug-in. These dependency and bootstrapping issues lead us to believe that for AspectJ to be used in large systems, the system must be aware of aspects in some way. Having the awareness and capability of dealing with aspects and advice would allow the system to deal with such issues in an appropriate manner. Exactly how this knowledge of aspects and advice presents itself will vary from system to system. It is highly likely that it will include a modification to the build system and also the runtime system.

The issue of using AspectJ within dynamic component based frameworks such as the OSGi framework is a new and interesting area. Solutions of this problem must be dynamic. For example, modules that contain aspects intended to advise other modules may be added or removed dynamically. Solutions of this problem must also deal with issues such as circular dependencies. In the context of Eclipse, the AspectJ community has addressed this need by creating the Eclipse OSGi aspects incubator project [2]. This project introduces support for managing aspects such that dependency and bootstrapping issues are no longer a problem. This is achieved by altering the class loading and dependency mechanisms of the system to include support for the use of aspects.
Chapter 6
Evaluation

To assess the use of AspectJ in Eclipse, it is necessary to evaluate our AO refactored version of the system. Specifically, we must consider how using AspectJ in the code base impacts the system both quantitatively and qualitatively. This is achieved by comparing quantitative and qualitative properties in the AO system and comparing them to their equivalents in the OO system.

6.1 Quantitative Evaluation

Quantitatively, we evaluated execution time and memory usage. Evaluating the change in execution time allows us to observe how any overhead introduced by the use of AspectJ affects the speed of execution. Similarly, evaluating the change in memory usage allows us to observe any impact on the memory footprint of the system.

6.1.1 Format of Data Collection

There are several plug-ins created and maintained by Eclipse developers that contain performance tests for Eclipse. These performance tests are implemented as JUnit test suites. Each such JUnit test suite invokes individual JUnit test cases that each use the plug-ins of a given installation of Eclipse to record information related to the performance of these plug-ins. Each test case is run multiple times in order to accumulate a good sample set of data. We used these test suites as well as a collection of scripts to measure, record, and process data about execution time and memory usage. We used most but not all of these JUnit test suites written to evaluate performance. The reason we did not use a small number of these test suites is that they tested plug-ins that were not refactored and thus would not provide any interesting insights for our evaluation of AspectJ. We ran tests on a machine with
a 3 GHz Pentium IV processor and 1 GB memory running SUSE Linux 10.1.

6.1.2 Execution Time

Figure 6.1 shows the change in execution time. Each pair of bars represents the CPU time for a specific JUnit test suite, with one of these bars representing the quantity for the OO system and the other representing the quantity for the AO system. The values for these test suites is formed by averaging the data of all individual JUnit test cases within that test suite. We group the individual tests in this way in order to provide a clear and comprehensible summary or their results.

It must be noted that in order to achieve an accurate comparison between the OO and AO versions of the system, we commented out the logging functionality in the FFDCEngine class. This step was taken due to the fact that the OO system contains no implementation of this concern. Comparing the OO and AO systems with this logging in place would not produce a good comparison as the results are affected by the extra writing to the Eclipse log file that takes place in FFDCEngine.

CPU time for the AO version of the system is near equivalent of the CPU time for the OO version. Differences for most test suits are near one percent of the CPU time for the OO version. The greatest difference occurs with the jdt.text performance test suite, in which the difference is over three percent of the OO time for that test suite. These numbers are within a reasonable margin of error of our data collection methods.

6.1.3 Memory Usage

Figure 6.2 shows the change in memory usage. As with execution time, these changes are presented by grouping results by test suite, with each pair of bars showing the average for memory usage for all tests within that test suite.

There appears to be a negligible impact on memory usage. Differences in quantities of memory usages between the AO and OO systems also appear to be within a reasonable margin of error. This is not surprising given that none of our changes to the code base involve use of memory intensive operations.

6.2 Qualitative Evaluation

This section provides a qualitative analysis of our use of AspectJ. This analysis is formed by exploring the impact that using AspectJ in the system has on the software engineering qualities discussed below.
Figure 6.1: CPU time comparison.
Figure 6.2: Memory usage comparison.
### 6.2.1 Scattering

Table 6.1 shows the reduction in scattering of the concerns. For each concern, we calculated this quantity for the implementation of that concern by determining the number of source files involved in the implementation in both versions of the system. This is shown in columns 1 and 2 of table 6.1. Column 3 shows the reduction of scattering as a percentage of the OO quantity. Overall, scattering was greatly reduced. Using an aspect-based implementation rather than a scattered, OO-based implementation that crosscuts various modules within the system caused a large decrease in the number of modules involved with the implementation of the concern. This decrease in number of modules is not as great for the logging of handled exceptions concern as it is for other concerns because the use of the annotation in advised classes to denote policy exceptions means that more modules than just the aspects are used in the implementation.

### 6.2.2 Coupling

Coupling between classes in the system decreased due to the use of AspectJ. Defining metrics that represent software attributes, such as coupling, for aspect oriented systems is a new and growing area of research. It is not always clear how to best measure a metric such as coupling in an aspect oriented system and compare it to its equivalent in a corresponding object oriented system; there does not yet exist a definitive work for metrics and AO systems. Nevertheless it is still possible to observe fine grain changes in coupling by reasoning about the changes in the code base.
made during our refactoring. In the OO-based implementation, modules that are
crosscut by other concerns contain code that implements the crosscutting concerns;
the OO-based implementation suffers from tangling. For example, all modules that
the logging of handled exceptions concern crosscuts contained references to classes
that implement logging. By removing or not using this tangled implementation and
using a more modularized aspect-based implementation instead, coupling between
modules in the system is reduced because removing or not using tangled code means
that references to certain classes that provide the implementation of the crosscutting
concerns are removed or omitted.

6.2.3 Abstractions

Using AspectJ in the Eclipse code base introduced new abstractions into the system.
For example, consider the refactoring of the singleton pattern. By creating an
abstract aspect to encapsulate the general policy involved in using the pattern, we
added an abstraction that previously did not exist in the system. Each concern that
we refactored or implemented introduced one or more additional abstractions into
the system for that concern.

6.2.4 Other Observations

We observed that using AspectJ in Eclipse would work well with the overall Eclipse
development process. Abstract aspects can be used to provide a modular implemen­
tation of policies that affect multiple plug-ins, with individual plug-ins providing
plug-in specific details related to these policies. This decomposition of policy imple­
mentation works well with Eclipse because it allows requirements for policies that
are applicable to multiple plug-ins (e.g. our logging of handled exceptions concern
described earlier), to be captured in a modular way in an abstract aspect. Addition­
ally, having specific details specified at the individual plug-in level allows the correct
teams of developers to provide these details, which is highly appropriate given that
those developers are most familiar with the code they maintain.

From our observations of the changes in the qualities discussed in this section,
we can reason about several qualities that are related to conclusions of previous
studies on AspectJ’s scalability as outlined in section 2. Reduction in scattering
leads to greater modularity, as the implementations of the crosscutting concerns
are now in fewer modules. This also makes maintainability easier, as developers
can more easily interact with the implementations of concerns. The decrease in
coupling also leads to greater modularity and maintainability since modules are more
independent. The additional abstractions introduced by the use of AspectJ lead to
greater modularity and program comprehension, as the additional modules allow a more expressive implementation that more clearly resembles program designs.
Chapter 7

Conclusion

To assess the scalability of AspectJ, we systematically refactored the code base of the Eclipse IDE. We chose to use Eclipse because it contains modularization mechanisms that operate at a more coarse grain than Java classes. This allows us to extend upon previous scalability studies on AspectJ by evaluating the use of the language in a new context. This context exists because of the extra modularization mechanisms.

Our work involved the use of vanilla aspects, allowing us to base our evaluation on refactorings that mirror those typically seen within the AO community. We refactored concerns that crosscut multiple plug-ins. Doing so allowed us to use AspectJ in the context of the additional modularization mechanisms mentioned above. We refactored logging of handled exceptions, performance monitoring, and uses of the singleton pattern. We also implemented the FFDC and plug-in contract enforcement concerns using AspectJ. In addition to these, we also refactored the OSGi plug-in, as we thought doing so would be worthwhile since it is the central entity related to these extra modularization mechanisms. In this plug-in we refactored logging of handled RuntimeException objects, plug-in state management, performance monitoring, and method security preconditions.

Defining advice with the intent of advising multiple plug-ins led to bootstrapping and dependency issues. In order to build the system with all aspects in place, we used a custom build process that used an ant buildfile with the ajc ant task. This resulted in possibly new dependencies being added to plug-ins. These dependencies, if not addressed by the manifest files of the involved plug-ins, would result in bootstrapping and dependency issues. This leads us to conclude that for AspectJ to be used in large systems, the underlying system may require knowledge of aspects and advice in order to deal with issues such as these.

We evaluated AspectJ both quantitatively and qualitatively. Quantitatively, we examined the execution time and memory usage of both the AO and OO systems. There are negligible changes in these quantities. Qualitatively, we evaluated
changes in scattering, coupling, and abstractions in the system. By observing these properties in the AO system and comparing them to their equivalents in the OO system, it is clear that scattering was reduced, as was coupling (at least on a fine grain level). The AO system had more abstractions that provided a more expressive implementation of the concerns we implemented or refactored.

By building upon previous scalability studies of AspectJ and evaluating the language within a new context, we found that AspectJ scales well in that it can be used in large systems. However, the underlying system may require knowledge of aspects and advice in order to deal with advice that cross system boundaries.
Bibliography


[19] Uirá Kulesza, Cláudio Sant’Anna, and Carlos Lucena. Refactoring the junit framework using aspect-oriented programming. In OOPSLA ’05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming,


