Turning Eclipse Against Itself: Improving the Quality of Eclipse Plugins

Benjamin Livshits

Computer Systems Laboratory
Stanford University
Stanford, CA 94305
livshits@cs.stanford.edu

Technical Report

September 14, 2005

Abstract

While many researchers have used Eclipse as a platform for developing software engineering and program analysis tools, Eclipse itself presents an excellent *subject* for analysis and study. Eclipse represents one of the biggest Java projects ever created. While surprisingly robust, Eclipse still suffers from serious bugs that lead to crashes and resource exhaustion.

Eclipse is a collaborative development projects, with its core developers located across multiple continents. Furthermore, hundreds of available plugins are developed by programmers with varying levels of familiarity with the intricacies of Eclipse APIs. As a result of API misuse, complex application-specific bugs are introduced.

In this paper[1] we describe common patterns in Eclipse code and propose lightweight analyses for finding their misuse. Bugs addressed in this paper often do not immediately exhibit themselves and are often discovered after deployment. In our experiments, we find a total of 68 likely errors in Eclipse sources that violate the coding patterns we describe. In addition to these lightweight bug checkers, we also propose the use of Eclipse templates extracted from existing Eclipse code as a way to capture important coding rules.

1 Introduction

A great deal of attention has lately been given to addressing software bugs such as errors in operating system drivers [4, 6] or security errors [10, 16]. These represent critical errors in widely used software and tend to get fixed relatively quickly when found. A variety of static and dynamic analysis tools have been developed to address these high-profile bugs.

Many other errors, however, are specific to individual applications or platforms and are especially prevalent for large software systems such as Eclipse. Repeated violations of these application-specific coding rules, referred to as *error patterns*, are responsible for a multitude of errors.

Due to its extensibility, Eclipse is an excellent example of a project with a lot of hidden rules that must be followed to preserve internal consistency and to avoid bugs. There are hundreds of Eclipse plugins available, with more

[1] A preliminary version of this work was presented at the Eclipse Technology Exchange in March 2005 [11].

Figure 1: Output of the extend super rule checker. Confirmed potential violations are shows in bold.

and more being written every day. Most plugins, however, are written by developers outside of the Eclipse core team, who are significantly less aware of the proper API use patterns. Error patterns tend to be re-introduced into the code over and over by multiple developers working on a project and are a common source of software defects.

The problem of plugin quality has been recognized by the Eclipse foundation, as evidenced by the inclusion of checkers that look for access to internal APIs in version 3.1. However, we believe that there is a potential for much more work towards both detecting potential errors and preventing them in the first place by making it easier to follow coding rules.

In this paper we describe several such coding patterns and lightweight static checkers to find their violations. We also propose adding Eclipse-specific coding templates to make commonly used plugin code easier to write correctly.

2 Patterns in Eclipse Code

In this section we describe three common error patterns in Eclipse APIs we address. The first pattern is an implementation strategy that requires subclass methods to always call the same method in the superclass. The other two patterns fall into the category of complex resource management errors: *lapsed listener errors* and *object disposal rules* both lead to leaks of memory and other operating system resources.

Declaring class(es)	Method	Occurrences
org.eclipse.core.runtime.IProgressMonitor	done	365
java.io.FileOutputStream,...	close	240
org.eclipse.jdi.internal.MirrorImpl	handledJdwpRequest	76
org.eclipse.jdt.internal.corext.refactoring.util.RefactoringFileBuffers,...	release	48
org.eclipse.core.internal.resources.Workspace	endOperation	46
org.eclipse.core.internal.utils.Policy	subMonitorFor	31
org.eclipse.jdt.internal.corext.refactoring.code.CallInliner,...	dispose	18
org.eclipse.core.filebuffers.IFileBufferManager	disconnect	17
org.eclipse.jdt.core.ICompilationUnit	discardWorkingCopy	16
org.apache.tools.ant.Project,...	log	15
org.eclipse.jdt.internal.core.search.indexing.ReadWriteMonitor	exitRead	12

Figure 2: Methods commonly appearing in cleanup code (`finally` blocks).

Extend Super	
methods that require `super` to be called	38
calls to these methods	390
filtered calls	19
potential errors (methods not calling `super`)	13
Disposal Rules	
`dispose` methods checked	794
filtered methods	51
potential errors (leaking `dispose` methods)	42
Lapsed Listeners	
subclasses of `ViewPart` checked	81
subclasses with matched listeners	6
subclasses not using listeners	53
subclasses with mismatched listeners	22
potential errors (classes with lapsed listeners)	13
total errors	**68**

Figure 3: Summary of experimental results.

Applying a sound static analysis to find violations of these patterns presents a considerable technical challenge. First, a flow-sensitivity analysis is necessary because the patterns we discuss are highly dependent on the order in which events occur. Second, a powerful alias analysis is necessary because Eclipse APIs refer to the same heap object through multiple access paths. Finally, since the Eclipse code base is so big, scalability presents a major concern. Instead, we propose a lightweight analysis approach that may suffer from both false positives and false negatives, but gives developers almost immediate indication as to where to expand their bug-finding efforts.

2.1 Extend Super Misuse

The *template method* design pattern [13] defines the skeleton of an algorithm in an operation, deferring some algorithm steps to subclasses. Subclasses redefine certain steps of an algorithm without changing the algorithm's structure. A particular variation of this pattern called the *extend super* pattern ensures that a subclass implements the functionality of the superclass by calling methods of the superclass.

The coding idiom used to achieve this in Java is to call `super.m`(...) in

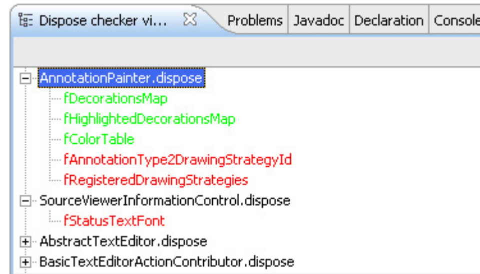

Figure 4: Output of the dispose rule checker showing dispose methods and collections leading to potential leaks.

method `m` [5]. However, when deep class hierarchies are used, developers implementing the subclasses sometimes forget to properly call the superclass implementation, thus breaking API invariants and leading to potential errors later in program execution.

2.2 Object Disposal Rules

While perhaps not as widespread as in C or C++, memory leaks still exist in Java [14]. A Java program can maintain a link to an object that is never used again, causing the garbage collector to never reclaim that object. Finding such memory leaks is difficult, but important because they can gradually cause resource exhaustion in long-running applications.

Languages with explicit resource management such as C++ often uses specific resource allocation disciplines, such as object ownership [7] to specify who is responsible for object deletion. Similar disciplines are necessary in large-scale Java projects that use a lot of operating system resources, such as native GUI elements, fonts, colors, etc. Eclipse documentation suggests a certain well-established resource management discipline [1, 2, 15]. However, this discipline is often violated.

In particular, various Eclipse classes define method `dispose` that is called to dispose of dependent resources. Proving that all resources are properly deallocated is a very difficult task in general. However, some rules of thumb commonly used by developers are relatively easy to check: Eclipse classes often store references to objects in locally allocated collections such as `Vectors` or `HashMaps`. Unless these collections are cleared within method `dispose`, it is possible for superfluous links to objects stored in the collections to exist

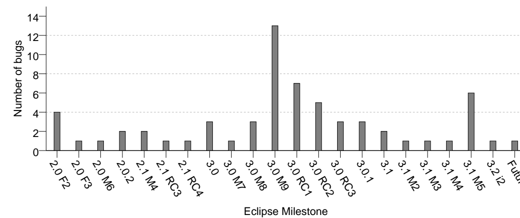

Figure 5: Distribution of lapsed listener errors across Eclipse release milestones.

after the base object has been disposed of, thus leading to memory leaks.

2.3 Lapsed Listeners

Event listeners in Java GUI programs is another common source of memory leaks. Event listeners are a common way to specify actions that should occur when a user interface event such as a mouse click occurs on a given GUI component. This is achieved by *registering* a listener with a GUI component; when the component is destroyed, the listener should be *unregistered*. If a listener is is not unregistered, it will preserve a link to the GUI component. The listener is reached from a global listener table, thus making the potentially large GUI component also reachable and therefore considered live by the garbage collector. This error pattern is referred to in the literature as the *lapsed listener problem* [14]. Lapsed listener errors are quite prominent in Eclipse: our searches on `bugs.eclipse.org` revealed at least 92 bugs in this category. The number of lapsed listener bugs reported for each development milestone shown in Figure 5.

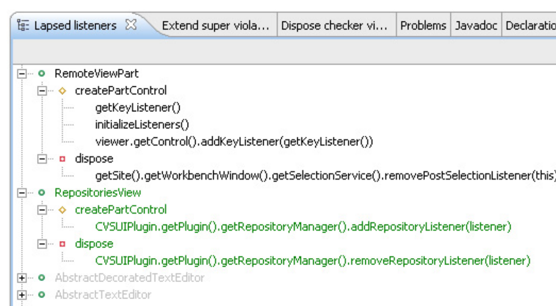

Figure 6: Output of the lapsed listener checker. For each method, calls in green are matched; others are potential mismatches.

2.4 Cleanup Templates

Matching method pairs represent one of the most commonly used temporal patterns in software. While pairs such as ⟨`fopen`, `fclose`⟩ are widespread in C, similar method pairs are present in Eclipse code as well. Eclipse APIs have a number of such methods pairs throughout the code, including ⟨`register`, `unregister`⟩, ⟨`beginCompoundEdit`, `endCompoundEdit`⟩, etc.

A common coding idiom pertaining to using method pairs consists of making sure that the second method is placed within a `finally` block. Given a pair of methods ⟨`A`, `B`⟩, the following code is common:

```
A(...);
try {
    ...
} finally {
    B(...);
}
```

To ensure that method B is called on *all* execution paths, the call to B is placed within the `finally` block. Placing it outside the `finally` block may lead to B not being called when an exception is thrown. This coding idiom in Java is explored in more detail in Weimer et.al. [18].

We take this concept further by creating a set of *coding templates* common to plugin development. According to the Eclipse documentation, "templates are a structured description of coding patterns that reoccur in source code", and thus represent a perfect mechanism for our purposes. The machinery built into Eclipse is responsible for template expansion.

3 Experimental Results

Eclipse JDT APIs expose abstract syntax trees of Java programs and make tasks such as examining the statements in a method or looking for specific method calls easy to accomplish. Our checkers perform local intraprocedural analysis to find likely violations. Our implementation consists of three special-purpose checkers addressing each of the error patterns described in the previous section as well as an analysis for finding good candidates to be included as Eclipse-specific code templates.

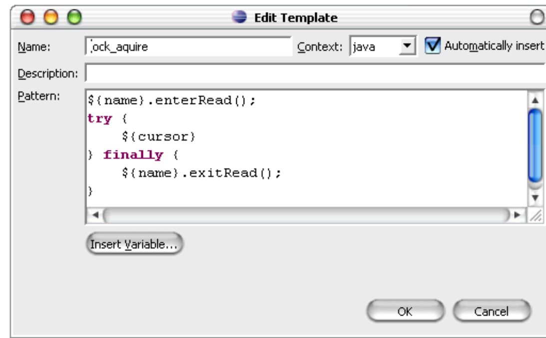

Figure 7: Template creation for ⟨`acquireRead`, `exitRead`⟩ pattern.

We summarize the results of running our static checkers on 20 large plugins from the Eclipse 3.0 code base in Figure 3. A total of 68 likely errors is reported. Unlike many other types of bugs, these errors are very difficult to verify statically by examining the code and are best validated either through the use of dynamic analysis or by original code developers. We used our best judgement about the code to arrive at the final bug count. Applying our tools to all the plugins took on the order of several minutes on an AMD Athlon XP 2500+ machine.

3.1 Extend Super

This checker finds implementations of methods such as `hookControl` and others that do not include a call to the `super`'s method on all paths through the method. This has allowed us to expose error cases where `super` is either called conditionally or not called at all. The results are presented to the user for verification, as shown in Figure 1.

3.2 Disposal Rules

To find potential memory leaks caused by collections that are not deallocated, we find all methods `dispose` that have collections defined in the same class. Classes that fail to use *all* those collections in the code of `dispose` are reported as potential sources of leaks. While false positives are possible, the answer computed by our checker provides a pretty strong indication that there may be a memory leak. The user is presented with a listing of offending `dispose` methods and collections that need to be cleared for each, as shown

in Figure 4. The output is color-coded to simplify the code auditing process: collections shown in green are mentioned in `dispose` at least once, the ones in red are not.

3.3 Lapsed Listener

The lapsed listener checker looks for mismatches in the way `createPartControl` and `dispose` method make listener registration and unregistration calls by pattern matching `add{T}Listener` and `remove{T}Listener` calls, where `T` is the listener type. An example of checker output is given in Figure 6. Methods shown in green represented matching listener registration and unregistration calls.

3.4 Cleanup Templates

To determine a set of methods that are commonly used in cleanup code, we wrote an analysis to process all `finally` blocks and rate methods called from within the block in order of frequency. The most common "cleanup" methods are listed in Figure 2.

Most of the methods listed in Figure 2 corresponds to one or more coding pattern in Eclipse code and we created Eclipse templates to match these patterns. Methods such as `log` which are commonly used for recording exception information are not really part of a well-formed pattern. In many cases, several patterns correspond to a single method. For instance, in the case of `endOperation`, we provide several templates corresponding to common uses of the `Workspace` class.

The process of template creation is shown in Figure 7. While templates are easy to create and use, there are several inherent shortcomings that Eclipse templates have. These issues can be addressed as part of future work.

- Pattern expansion needs to be made more context-sensitive by taking the *types* of variables involved into account. In other words, only offering to expand the ⟨`aquireLock`, `exitLock`⟩ template on a variable of type `ReadWriteMonitor` would make template expansion more useful.
- Pattern expansion needs to be made more context-sensitive by being aware of already existing code. For example, if a call to `exitLock` already exists, there is no need to insert it again.

4 Related Work

There has been a lot of interest recently in the general subject of code pattern discovery [3, 17] and enforcement [6, 9]. However, the Eclipse code base represents a relatively novel study subject. Several recent projects target the development of programmer assistant technology for Eclipse plugin development, including the Strathcona example recommendation tool [8] and Prospector tool for synthesizing API usage code [12].

5 Conclusions

This paper describes usage patterns in Eclipse code and lightweight static checkers that allowed us to find a total of 68 likely bugs in 20 plugins from the Eclipse code base. Misuse of application-specific coding patterns are a common source of errors in large software systems developed by multiple programmers. The sheer complexity and scale of the problem makes a sound, whole-program analysis prohibitively expensive and justifies the use of lightweight tools that may suffer from both false positives and negatives.

In addition to finding errors in Eclipse code, we also explored the use of programmer assistant technology for plugin development. We describe an approach for discovering common resource cleanup patterns, which are subsequently converted into code templates that Eclipse can insert into the code during development.

References

[1] SWT: The standard widget toolkit. part 2: Managing operating system resources. http://www.eclipse.org/articles/swt-design-2/swt-design-2.html.

[2] User interface resources. http://dev.eclipse.org/viewcvs/index.cgi/~checkout~/org.eclipse.platform.doc.isv/guide/jface_resources.htm?rev=1.14&content-type=text/html.

[3] G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, Portland, Oregon, 2002. ACM Press.

[4] T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. Technical Report MSR-TR-2004-08, Microsoft, 2004.

[5] Cedric. Don't call super. http://www.beust.com/weblog/archives/000077.html, 2004.

[6] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implentation*, pages 1–16, 2000.

[7] D. Heine and M. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Conference on Programming Language Design and Implementation*, pages 168–181, June 2003.

[8] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 117–125, 2005.

[9] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.

[10] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D.-T. Lee, and S.-Y. Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the 13th conference on World Wide Web*, pages 40–52, May 2004.

[11] V. B. Livshits. Turning Eclipse against itself: Finding bugs in Eclipse code using lightweight static analysis. Eclipsecon '05 Research Exchange, Mar. 2005.

[12] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Mining Jungloids: Helping to navigate the API jungle. In *Proceedings of the ACM SIGPLAN'05 Conference on Programming Language Design and Implementation*, 2005.

[13] S. A. Stelting and O. Maassen. *Applied Java Patterns*. Prentice Hall, 2001.

[14] B. A. Tate. *Bitter Java*. Manning Publications Co., 2002.

[15] tazzzzz. SWT and memory management. http://www.blueskyonmars.com/archives/2003/10/20/swt_and_memory_management.html, 2003.

[16] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 3–17, Feb. 2000.

[17] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms For The Construction And Analysis Of Systems*, pages 461–476, Apr. 2005.

[18] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 419–431, Oct. 2004.