Turning Eclipse Against Itself: Finding Bugs in Eclipse Code Using Lightweight Static Analysis

V. Benjamin Livshits

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

While some commonly occurring error patterns in Java are addressed by static tools such as FindBugs[5], complex software systems are full of rules that developers must follow. These application-specific rules are often not expressed in any way other than code comments and often are not enforced, leading to hard-to-detect bugs later in the program execution.

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 intricacies of Eclipse APIs, thus causing the introduction of complex application-specific bugs. Bugs addressed in this paper do not immediately exhibit themselves and are often discovered after deployment.

In this paper we describe some common error patterns in Eclipse code and propose, Checklipse, a lightweight analysis tool for finding these errors; Checklipse is implemented as an Eclipse plugin and leverages Eclipse's JDT APIs to analyze Java code. In our experiments, we find a total of 68 likely errors in Eclipse sources that follow the error patterns described in this paper.

1 Error Patterns in Eclipse

In this section we describe three common error patterns in Eclipse APIs addressed by Checklipse. 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.

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 or by calling different methods. 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 where to spend their bug-finding efforts.

1.1 Extend Super Misuse

The *template method* design pattern 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 method `m`.

However, when deep class hierarchies are used, developers implementing the subclasses sometimes forget to properly call the superclass implementation, thus breaking API invariants[3] and leading to potential errors later in program execution.

1.2 Object Disposal Rules

While not as widespread as in C or C++, memory leaks still exist in Java[6]. 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 kinds of memory leaks is difficult, but important because they can gradually cause resource exhaustion in a long-running applications such as Eclipse, leading to instability and crashes.

Languages with explicit resource management such as C++ often uses specific resource allocation disciplines, such as object ownership[4] to specify who is responsible for object deletion. Although this is less common, 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 APIs suggest a certain resource management discipline that is often misused[1, 2, 7]. Various Eclipse types 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 `Vector`s or `HashMap`s. Unless these collections are cleared in method `dispose`, it is possible for superfluous links to objects stored in the collections to exist *after* the base object has been disposed of, thus leading to memory leaks.

1.3 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*[6]. Lapsed listener errors are quite prominent in Eclipse: our searches at `bugs.eclipse.org` revealed at least 92 bugs in this category.

2 Implementation and Experiments

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 perform. Our checkers perform local intraprocedural analysis to find likely violations. Our implementation of Checklipse consists of three special-purpose checkers addressing each of the error patterns described in the previous section:

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.

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

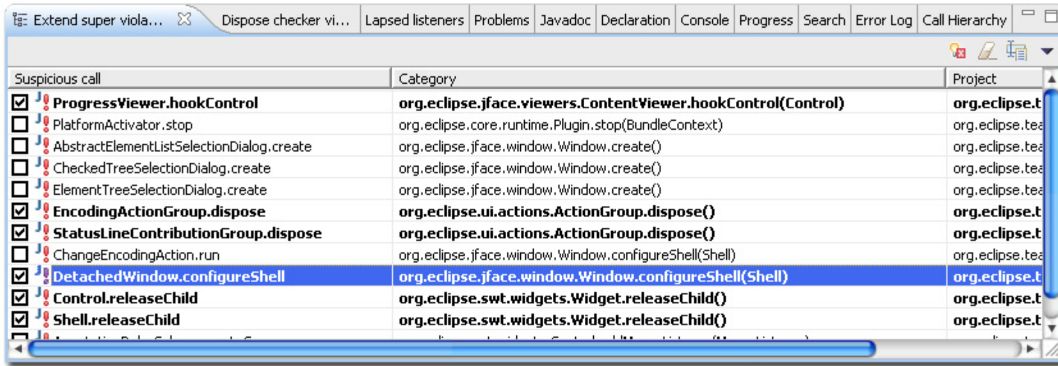

Figure 1: Output of the extend super rule checker. Confirmed potential violations are shows in bold.

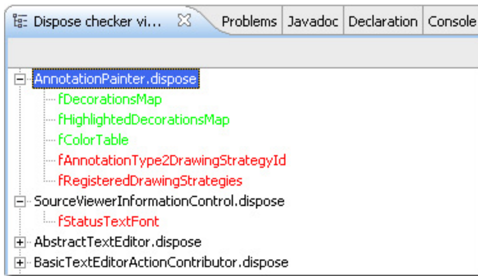

Figure 2: Output of the dispose rule checker showing dispose methods and collections leading to potential leaks.

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 2. 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.

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 ??. Methods shown in green represented matching listener registration and unregistration calls.

We summarize the results of running Checklipse on 20 large plugins from the Eclipse code base in Figure 4. 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.

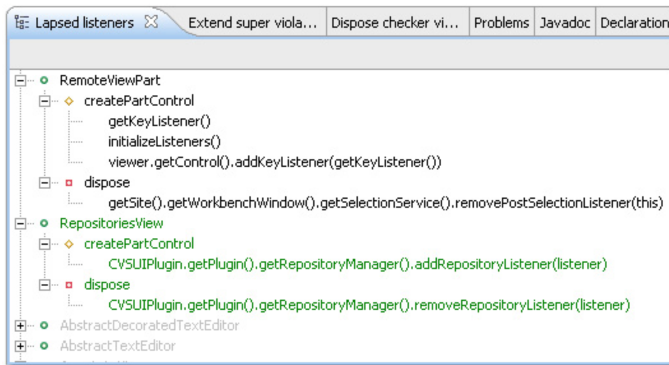

Figure 3: Output of the lapsed listener checker. For each method, calls in green are matched; others are potential mismatches.

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 4: Summary of experimental results.

3 Conclusions

In this paper we have described Checklipse, a lightweight static analysis tool that has 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. We have successfully explored three important error patterns in Eclipse APIs that lead to broken logical constraints and resource leaks. Our preliminary experience with Checklipse makes us believe that our approach is a fruitful one and will yield more errors when applied to other complex error patterns.

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.platfor%m.doc.isv/guide/jface_resources.htm?rev=1.14&content-type=text/html`.

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

[4] 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 (PLDI)*, pages 168–181, 2003.

[5] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proceedings of the Onward! Track of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2004.

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

[7] tazzzzz. SWT and memory management. `http://www.blueskyonmars.com/archives/2003/10/20/swt_and_memory_managem%ent.html`, 2003.