Finding Bugs is Easy *

(Extended Abstract)

David Hovemeyer and William Pugh
Dept. of Computer Science, University of Maryland
College Park, Maryland 20742 USA
{daveho,pugh}@cs.umd.edu

ABSTRACT

Many techniques have been developed over the years to automatically find bugs in software. Often, these techniques rely on formal methods and sophisticated program analysis. While these techniques are valuable, they can be difficult to apply, and they aren't always effective in finding real bugs.

Bug patterns are code idioms that are often errors. We have implemented automatic detectors for a variety of bug patterns found in Java programs. In this extended abstract[1], we describe how we have used bug pattern detectors to find serious bugs in several widely used Java applications and libraries. We have found that the effort required to implement a bug pattern detector tends to be low, and that even extremely simple detectors find bugs in real applications.

From our experience applying bug pattern detectors to real programs, we have drawn several interesting conclusions. First, we have found that even well tested code written by experts contains a surprising number of obvious bugs. Second, Java (and similar languages) have many language features and APIs which are prone to misuse. Finally, that simple automatic techniques can be effective at countering the impact of both ordinary mistakes and misunderstood language features.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification — *Reliability, Validation*

General Terms

Reliability

Keywords

Static analysis, bugs, bug patterns, bug checkers

1. INTRODUCTION

Few people who develop or use software will need to be convinced that bugs are a serious problem. Automatic techniques and tools for finding bugs offer tremendous promise for improving software quality. In recent years, much research has investigated automatic techniques to find bugs in software. Some of the techniques proposed in this research require sophisticated program analysis. This research is valuable, and many interesting and useful analysis techniques have been proposed as a result. However, these techniques have generally not found their way into widespread use.

In our work, we have investigated simple static analysis techniques for finding bugs based on the notion of *bug patterns*. A bug pattern is a code idiom that is likely to be an error. Occurrences of bug patterns are places where code does not follow usual correct practice in the use of a language feature or library API. Automatic detectors for many bug patterns can be implemented using relatively simple static analysis techniques. In many ways, using static analysis to find occurrences of bug patterns is like an automated code inspection. We have implemented a number of automatic bug pattern detectors in a tool called FindBugs (available from `http://findbugs.sourceforge.net`).

In this extended abstract, we will describe some of the bug patterns our tool looks for, and present examples of bugs that our tool has found in several widely used applications and libraries. We hope that the obvious and embarrassing nature of these bugs will convince you of the need for wider adoption of automatic bug finding tools. We also present empirical results which classify (for selected bug patterns) the percentage of warnings reported by the tool that indicate real bugs. We will argue that although some of the bug pattern detectors we evaluated produce false warnings, they produce enough genuine examples of bugs to make the tool useful in practice.

```
// Eclipse 3.0,
// org.eclipse.jdt.internal.ui.compare,
// JavaStructureDiffViewer.java, line 131

Control c= getControl();
if (c == null && c.isDisposed())
    return;
```

Figure 1: Example of null pointer dereference.

2. BUG PATTERNS

Programmers are smart. Therefore, we expect that bugs in software should be subtle, and require sophisticated analysis techniques to find. And, there is no doubt that many bugs are subtle.

However, consider the code shown in Figure 1. This bug

*Supported by NSF grant CCR-0098162 and by an IBM Eclipse Innovation award.

[1]The full version of this paper will be published as [2].

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

Code	Description
CN	Cloneable Not Implemented Correctly
DC	Double Checked Locking
DE	Dropped Exception
EC	Suspicious Equals Comparison
Eq	Bad Covariant Definition of Equals
HE	Equal Objects Must Have Equal Hashcodes
IS2	Inconsistent Synchronization
MS	Static Field Modifiable By Untrusted Code
NP	Null Pointer Dereference
NS	Non-Short-Circuit Boolean Operator
OS	Open Stream
RCN	Redundant Comparison to Null
RR	Read Return Should Be Checked
RV	Return Value Should Be Checked
Se	Non-serializable Serializable Class
UR	Uninitialized Read In Constructor
UW	Unconditional Wait
Wa	Wait Not In Loop

Figure 2: Summary of selected bug patterns.

```
// jdk1.5.0, build 59
// javax.sql.rowset.spi,
// SyncFactory.java, line 325

if(syncFactory == null){
     synchronized(SyncFactory.class) {
        if(syncFactory == null){
            syncFactory = new SyncFactory();
        } //end if
     } //end synchronized block
 } //end if
```

Figure 3: Example of double checked locking.

```
// GNU classpath 0.08,
// java.util,
// Vector.java, line 354

public int lastIndexOf(Object elem) {
  return lastIndexOf(elem, elementCount - 1);
}
```

Figure 4: Example of inconsistent synchronization.

is not subtle, and a relatively simple analysis found not one, but 54 null pointer dereference bugs in Eclipse 3.0 similar to the one shown. Our experience has shown that even well-tested, production software usually contains a significant number of obvious bugs.

Static analysis based on bug patterns represents a useful sweet spot in the design space for bug checking tools. Because bug patterns focus on finding deviations from standard practice, rather than performing deep analysis, they tend to be easy to implement, and their output is generally easy to explain to programmers. With suitable heuristics, automatic detectors for bug patterns can find real bugs without too high a percentage of false warnings.

It is important to distinguish between *bug checkers* and *style checkers*. Bug checkers look for deviations from correct behavior, while style checkers look for deviations from coding style rules. The distinction is that examples of code that violate a style rule are not particularly likely to be bugs (although bugs may often be violations of a style rule). Tools that focus mainly on style, such as PMD[4], are valuable because they help make code easier to understand. However, they tend to be less effective at finding actual bugs, due to the large volume of output they produce. (See Section 3.3.)

In this section we will briefly present a handful of the bug patterns FindBugs can detect, along with some occurrences of those patterns in real code. Figure 2 lists the subset of patterns referred to in this extended abstract. Due to space limitations, we can describe only a few members of this subset. For more complete descriptions and information about analysis techniques, please see the full version of this paper [2].

2.1 Double Checked Locking (DC)

Double checked locking is a design pattern intended for thread safe lazy initialization. An example of double checked locking is shown in Figure 3.

Unfortunately, the double checked locking pattern assumes a sequentially consistent memory model, which isn't true in any major programming language. In Figure 3 it is possible that the writes initializing the `SyncFactory` object and the write to the `syncFactory` field could be reordered (either by the compiler or the processor). Threads which do not acquire the lock may see the object in an inconsistent state.

2.2 Bad Covariant Definition of Equals (Eq)

In this pattern, a class defines an `equals()` method using the type of the class as the parameter, rather than defining the parameter as type `Object`. This covariant version of `equals()` does not override `equals()` in the base `Object` class, and will not be callable by generic container classes such as sets and maps.

2.3 Inconsistent Synchronization (IS2)

A common idiom for making objects thread-safe in Java is for methods to synchronize on the `this` reference when accessing or updating shared state. Programmers sometimes mistakenly omit synchronization of some field references, resulting in data races. The detector for this pattern looks for field accesses that are not guarded by a lock on `this`. Fields usually, but not always, accessed with the lock held are candidate instances of this pattern. In order to reduce false positices, the detector employs several heuristics to infer whether or not the synchronization guarding the field was intentional or incidental.

An example of inconsistent synchronization is shown in Figure 4.

2.4 Null Pointer Dereference (NP)

Dereferencing a null pointer almost always indicates an error. In many cases, null pointer exceptions result from simple typos (such as using the wrong boolean operator following an explicit null comparison), or from incorrect code modification during maintenance. The detector for this pattern catches many obvious null dereference errors. The analysis is intraprocedural, and infers null values arising from explicit null constants and explicit comparisons to null. Although the detector does not find dereferences of null values passed as method parameters, it nonetheless does find a surprising number of real bugs (such as the Eclipse bug shown in Figure 1).

```
// Eclipse 3.0,
// org.eclipse.ui.internal.cheatsheets.views,
// CheatSheetPage.java, line 83
if(cheatSheet != null & cheatSheet.getTitle() != null)
    return cheatSheet.getTitle();
```

Figure 5: A non-short-circuit boolean operator bug.

```
// JBoss 4.0.0RC1
// org.jboss.deployment.scanner
// AbstractDeploymentScanner.java, line 185

// If we are not enabled, then wait
if (!enabled) {
  try {
    synchronized (lock) {
      lock.wait();
...
```

Figure 6: An example of an unconditional wait.

2.5 Non-Short-Circuit Boolean Operator (NS)

Programmers may unintentionally use a non-short-circuiting boolean operator (`&` and `|`) where they intended to use a short-circuiting boolean operator. Because both subexpressions are evaluated unconditionally, unintended behavior may result. An example is shown in Figure 5; this example would also be caught by the NP pattern, but other cases might result in problems not caught by our existing bug detectors, such as an out of bounds array reference.

2.6 Unconditional Wait (UW)

Coordinating threads using `wait()` and `notify()` is a frequent source of errors in multithreaded programs. This pattern looks for code where a monitor wait is performed unconditionally upon entry to a synchronized block. Typically, this indicates that the condition associated with the wait was checked without a lock held, which means that a notification performed by another thread could be missed.

Figure 6 shows an example of an unconditional wait.

3. EVALUATION

In order to evaluate the effectiveness of the bug detectors implemented in FindBugs, we manually evaluated the high and medium priority warnings produced by version 0.8.4 the tool for a subset of the bug patterns[2]. We classified the warnings as follows:

- Some bug pattern detectors are very accurate, but determining whether the situation detected warrants a fix is a judgment call. For example, we can easily and accurately tell whether a class contains a static field that can be modified by untrusted code. However, human judgment is needed to determine whether that class will ever run in an environment where it can be accessed by untrusted code. We did not try to judge whether the results of such detectors warrant fixing, but simply report the warnings generated.

- Some the bug detectors admit false positives, and report warnings in cases where the situation described by the warning does not, in fact occur. Such warnings are classified as *false positives*.

- The warning may reflect a violation of good programming practice but be unlikely to cause problems in practice. For example, many incorrect synchronization warnings correspond to data races that are real but highly unlikely to cause problems in practice. Such warnings are classified as *mostly harmless* bugs.

- And then there are the cases where the warning is accurate and in our judgment reflects a serious bug that warrants fixing. Such warnings are classified as *serious*.

In this extended abstract, we present results for one library and one application:

- rt.jar 1.5.0 build 59: Sun's implementation of the core J2SE libraries

- eclipse 3.0: a popular Java IDE

The full version of the paper presents results for additional applications and libraries.

[2]We used the `-workHard` command line option to eliminate consideration of unlikely exception paths.

Application	Eq	HE	MS	Se	DE	CN
rt.jar 1.5.0 build 59	9	55	259	207	89	73
eclipse-3.0	3	170	1,000	49	23	20

Figure 8: Bug counts for selected other detectors.

Application	KLOC	FindBugs	PMD
rt.jar 1.5.0 build 59	1,183	3,314	17,133
eclipse-3.0	2,237	4,227	25,227

Figure 9: Application sizes and total number of warnings generated by FindBugs and PMD.

3.1 Empirical Evaluation

Figure 7 shows the results of classifying the detectors which can produce false positives. In general, the detectors achieved our goal of having at least 50% of the warnings represent genuine bugs. Only the UW and Wa detectors were significantly less accurate. However, given the small number of warnings they produced and the potential difficulty of debugging timing-related thread bugs, we feel that they performed adequately.

3.2 Other Detectors

Figure 8 lists results for some of our bug detectors for which we did not perform manual examinations. These detectors are fairly to extremely accurate at detecting whether software exhibits a particular feature (such as violating the hashcode/equals contract, or having static fields that could be mutated by untrusted code). However, it is sometimes a difficult and very subjective judgment as to whether or not the reported feature is a bug that warrants fixing in each particular instance. We will simply note that in many cases, these reports represent instances of poor style or design.

3.3 Comparison with PMD

In Figure 9, we list the total number of thousands of lines of source code for each benchmark application[3], the total

[3]Note that the figure for rt.jar is low because not all of its source code is available in the standard public distribution.

	rt.jar 1.5.0 build 59				eclipse-3.0			
code	warnings	serious	mostly harmless	false pos	warnings	serious	mostly harmless	false pos
DC	88	100%	0%	0%	88	100%	0%	0%
EC	8	100%	0%	0%	19	57%	0%	42%
IS2	116	44%	47%	7%	63	61%	22%	15%
NP	37	100%	0%	0%	70	78%	7%	14%
NS	12	25%	66%	8%	14	78%	21%	0%
OS	13	15%	0%	84%	26	46%	0%	53%
RCN	35	57%	0%	42%	69	40%	11%	47%
RR	12	91%	0%	8%	39	38%	0%	61%
RV	7	71%	0%	28%	8	100%	0%	0%
UR	4	100%	0%	0%	4	50%	50%	0%
UW	6	50%	0%	50%	7	28%	0%	71%
Wa	8	37%	0%	62%	12	25%	0%	75%

Figure 7: Evaluation of false positive rates for selected bug pattern detectors.

number of high and medium priority warnings generated by FindBugs version 0.8.4, and the number of warnings generated by PMD version 1.9 [4][4]. In general, FindBugs produces a much lower number of warnings than PMD when used in the default configuration. Undoubtedly, PMD finds a significant number of bugs in the benchmark applications: however, they are hidden in the sheer volume of output produced.

We do not claim this comparison shows that FindBugs is "better" than PMD, or vice versa. Rather, the two tools focus on different aspects of software quality

4. WHY BUGS OCCUR

We have been actively working on FindBugs for a year and a half, and have looked at a huge number of bugs. Based on this experience, we can offer the following observations on why bugs occur. These are by no means exhaustive, but may provide some food for thought.

Everyone makes dumb mistakes. This is perhaps the most fundamental theme of our work so far. Detectors for the most blatant and dumb mistakes imaginable routinely turn up real bugs in real software. One way to explain this phenomenon is that a huge number of bugs are just one step removed from a syntax error. For example, many of the null pointer bugs we've seen fall into this category: the programmer intended to use the `&&` operator, but mistakenly used the `||` operator. We even found a bug in code written by Joshua Bloch, author of *Effective Java* [1]: the bug, a class with an `equals()` method but no `hashCode()` method, was a violation of one of the proscriptions laid out in his book. The lesson here is that even the best programmers are not perfect 100% of the time. Bug checkers take static checking further than the compiler, and are able to catch errors that the compiler ignores.

Java offers many opportunities for latent bugs. The hashcode/equals and covariant equals bug patterns are examples of *latent* bugs: they don't affect the correctness of a program until a particular kind of runtime behavior occurs. For example, a class with a covariant equals method will work correctly until someone puts it into a map or set. Then, rather than failing in an obvious manner (such as throwing an exception), the program will quietly perform the wrong computation. Similarly, there are a number of patterns and requirements (such as those for a serializable class) that are not checked by the compiler but simply result in runtime errors when violated. Bug checkers help increase the visibility of some of these latent errors.

Programming with threads is harder than people think. We did a study of concurrency errors in Java programs [3], and found that misuse of concurrency (such as deliberate use of data races to communicate between threads) is widespread. The problem here is that programmers are not as scared of using threads as they should be. Java tends to hide many of the potential negative consequences of concurrency glitches. For example, a data race cannot cause a program to violate type safety, or corrupt memory. However, due to the aggressive reordering of memory accesses by modern processors and JVMs, programming with data races is a very bad idea. Concurrency bugs are especially problematic because they can be extremely difficult to reproduce. Bug checkers can help prevent concurrency errors before they make it into deployed code.

5. CONCLUSIONS

Static analysis based on bug patterns finds an important class of bugs in production code, and is a very useful complement to traditional quality assurance practices such as code inspections and testing.

6. RELATED WORK

For references to related work, please see the full version of the paper [2].

7. REFERENCES

[1] J. Bloch. *Effective Java Programming Language Guide.* Addison-Wesley, 2002.

[2] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Notices*, December 2004.

[3] D. Hovemeyer and W. Pugh. Finding concurrency bugs in Java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, St. John's, Newfoundland, Canada, July 2004.

[4] PMD, http://pmd.sourceforge.net, 2004.

[4]For PMD, we used the suggested rulesets: basic, unusedcode, imports, and favorites.