# Finding Bugs is Easy

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 paper, we describe how we have used bug pattern detectors to find real bugs in several real-world 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.

## 1. INTRODUCTION

Few people who develop or use software will need to be convinced that bugs are a serious problem. Much recent research (e.g., [7, 14, 16, 28, 3, 6, 26, 17, 13]) has been devoted to developing techniques to automatically find bugs in software. Many of these techniques use sophisticated analyses and powerful formalisms.

Our approach to finding bugs is different: we have concentrated on using simple, broad techniques rather than focused, narrow techniques. The principle underlying our work is that we start by looking at actual bugs in real code, and then develop ways to find similar bugs. Our goal has been to better understand what kind of bugs exist in software, rather than advancing the state of the art in program analysis. From this work, we have reached some interesting and sometimes unexpected conclusions.

First, no type of bug has been so "dumb" or "obvious" that we have failed to find examples of it in real code. Every bug detector we have ever implemented has found real bugs. We have been continually suprised by the obviousness of the bugs found using our detectors, even in production applications and libraries.

Second, because of the sheer complexity of modern object-oriented languages like Java, the potential for misuse of language features and APIs is enormous. From our experience we have concluded that even experienced developers have significant gaps in their knowledge which result in bugs.

Third, that automatic bug detection tools can serve an important role in raising the awareness of developers about subtle correctness issues. So, in addition to finding existing bugs, they can help prevent future bugs.

In this paper we will describe some of the bug detection techniques we have used, and present empirical results showing the effectiveness of these techniques on real programs. We will also show examples of bugs we have found to help illustrate how and why bugs are introduced into software, and why commonly used quality assurance practices such as testing and code reviews miss many important bugs.

The structure of the rest of the paper is as follows. In Section 2, we discuss the general problem of using automatic techniques to find bugs in software, and describe a tool, called FindBugs, which uses *bug pattern detectors* to inspect software for potential bugs. In Section 3, we briefly describe the implementation of our tool and some of its features. In Section 4, we evaluate the effectiveness of our bug pattern detectors on several real programs. In Section 5, we offer observations from our experiences and some of our users' experiences putting bug pattern detection into practice. In Section 6, we describe related work. In Section 7 we offer some conclusions, and describe possibilities for future work.

## 2. TECHNIQUES FOR FINDING BUGS

There are many possible ways to find bugs in software. Dynamic techniques, such as testing and assertions, rely on the runtime behavior of a program. As such, they are limited

to finding bugs in the program paths that are actually executed. In contrast, static techniques can explore abstractions of all possible program behaviors, and thus are not limited by the quality of test cases in order to be effective.

Static techniques range in their complexity and their ability to identify or eliminate bugs. The most effective (and complex) static technique for eliminating bugs is a *formal proof* of correctness. While the existence of a correctness proof is the best guarantee that the program does not contain bugs, the difficulty in constructing such a proof is prohibitive for most programs. *Partial verification* techniques have been proposed. These techniques prove that some desired property of a program holds for all possible executions. Such techniques may be complete or incomplete; if incomplete, then the analysis may be unable to prove that the desired property holds for some correct programs. Finally, *unsound* techniques can identify "probable" bugs, but may produce false positives and false negatives.

We have implemented a static analysis tool, called FindBugs, that detects instances of bug patterns in Java programs. Bug patterns are code idioms that are likely to be errors. We have implemented (unsound) detectors for a variety of common bug patterns, and used them to find a significant number of bugs in real-world applications and libraries. In this section we describe the bug patterns we detect, and how we implemented and refined detectors for those patterns.

## 2.1 Bug Patterns

*Bug patterns* are error-prone coding practices that arise from the use of erroneous design patterns, misunderstanding of language semantics, or simple and common mistakes. As developers, we tend to believe that any bugs in our code must be subtle, unique and require sophisticated tools to uncover. However, our experience has shown that while subtle and unique bugs exist, there are also many errors, even in production code, that are blatant, well-understood, and easy to find if you know what to look for. Finding these errors does not require sophisticated or expensive forms of analysis or instrumentation. Many of these errors can be found with trivial static examination, using what we refer to as a *bug pattern detector*.

Bug pattern detectors are not judged by whether they are "right" or "wrong" in any formal correctness point of view. Rather, they are either *effective* or *ineffective* at finding real bugs in real software. Useful bug pattern detectors can only be developed in tandem with continual application and evaluation of their usefulness in finding bugs in software artifacts.

The bug patterns we look for have come from a wide range of sources. Many of the bug patterns are suggested by Java semantics. A simple example is that it is generally an error to dereference a null pointer. A number of books ([1, 27, 5, 19, 9]) describe potential Java coding pitfalls. Several of the bug patterns we implemented in FindBugs were observed in student projects in an undergraduate advanced programming course at the University of Maryland. We have implemented detectors for several bug patterns suggested by users of our analysis tool. In general, bug patterns are found through the process of software development, through the simple observation that bugs often have common characteristics. Whenever a bug is fixed, it is worth looking for other places in the code where a similar bug might be lurking. Bug pattern detectors put this idea into practice.

## 2.2 Bug Pattern Detectors

We have implemented 45 bug pattern detectors in our FindBugs tool. All of the bug pattern detectors are implemented using BCEL [4], an open source bytecode analysis and instrumentation library. The detectors are implemented using the Visitor design pattern; each detector visits each class of the analyzed library or application. Broadly speaking, each detector falls into one or more of the following categories:

- Single-threaded correctness issue
- Thread/synchronization correctness issue
- Performance issue
- Security and vulnerability to malicious untrusted code

Our work has focused on understanding bug patterns and the evaluating the feasibility of detecting them, rather than on developing a sophisticated analysis framework. For this reason, we have tried to use the simplest possible techniques to find instances of bug patterns. The implementation strategies used by the detectors can be divided into several rough categories:

- **Class structure and inheritance hierarchy only**. Some of the detectors simply look at the structure of the analyzed classes without looking at the code.
- **Linear code scan**. These detectors make a linear scan through the bytecode for the methods of analyzed classes, using the visited instructions to drive a state machine. These detectors do not make use of complete control flow information; however, heuristics (such as identifying the targets of branch instructions) can be effective in approximating the effects of control flow.
- **Control sensitive**. These detectors make use of an accurate control flow graph for analyzed methods.
- **Dataflow**. The most complicated detectors use dataflow analysis to take both control and data flow into account. An example is the null pointer dereference detector.

None of the detectors make use of analysis techniques more sophisticated than what might be taught in an undergraduate compiler course. The detectors that use dataflow analysis are the most complex; however, we have implemented a framework for dataflow analysis which moves most of the complexity out of the detectors themselves. The most complex detector has 679 lines of Java source code (including blank lines and comments). Almost half of the detectors have less than 100 lines of source code.

| Code | Description |
|---|---|
| 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 |
| OS | Open Stream |
| RR | Read Return Should Be Checked |
| RV | Return Value Should Be Checked |
| UR | Uninitialized Read In Constructor |
| UW | Unconditional Wait |
| Wa | Wait Not In Loop |

**Figure 1: Summary of selected bug patterns**

## 2.3 Selected Bug Pattern Detectors

For space reasons, we will only discuss a handful of our detectors. Each detector/pattern is identified by a short "code", which will be used in the evaluation in Section 4. A table summarizing the bug patterns and detectors described in this paper is shown in Figure 1.

### 2.3.1 Bad Covariant Definition of Equals (Eq)

Java classes may override the `equals(Object)` method to define a predicate for object equality. This method is used by many of the Java runtime library classes; for example, to implement generic containers.

Programmers sometimes mistakenly use the type of their class `Foo` as the type of the parameter to `equals()`:

```
public boolean equals(Foo obj) {...}
```

This covariant version of `equals()` does not override the version in the `Object` class, which may lead to unexpected behavior at runtime, especially is the class is used with one of the standard collection classes which expect that the standard `equals(Object)` method is overridden.

This kind of bug is insidious because it looks correct, and in circumstances where the class is accessed through references of the class type (rather than a supertype), it will work correctly. However, the first time it is used in a container, mysterious behavior will result. For these reasons, this type of bug can elude testing and code inspections.

Detecting instances of this bug pattern simply involves examining the method signatures of a class and its superclasses.

### 2.3.2 Equal Objects Must Have Equal Hashcodes (HE)

In order for Java objects to be stored in `HashMap`s and `HashSet`s, they must correctly implement both the `equals(Object)` and `hashCode()` methods. Objects which compare as equal must have the same hashcode.

Consider a case where a class overrides `equals()` but not `hashCode()`. The default implementation of `hashCode()` in the `Object` class (which is the ancestor of all Java classes) returns an arbitrary value assigned by the virtual machine. Thus, it is possible for objects of this class to be equal without having the same hashcode. Because these objects would likely hash to different buckets, it would be possible to have two equal objects in the same hash data structure, which violates the semantics of `HashMap` and `HashSet`.

As with covariant equals, this kind of bug is hard to spot through inspection. There is often nothing to "see"; the mistake lies in what is missing. Because the `equals()` method is useful independently of `hashCode()`, it can be difficult for novice Java programmers to understand why they must be defined in a coordinated manner. This illustrates the important role tools can play in educating programmers about subtle language and API semantics issues.

Automatically verifying that a given class maintains the invariant that equal objects have equal hashcodes would be very difficult. Our approach is to check for the easy cases, such as

- Classes which redefine `equals(Object)` but inherit the default implementation of `hashCode()`
- Classes which redefine `hashCode()` but not `equals(Object)`

Checking for these cases requires simple analysis of method signatures and the class hierarchy.

### 2.3.3 Inconsistent Synchronization (IS2)

Many Java classes, both in runtime libraries and applications, are designed to be safe when accessed by multiple threads. The most common way to ensure thread safety is for methods of an object to obtain a lock on the object itself. In this way, methods are guaranteed mutual exclusion even when invoked by different threads.

A common category of mistakes in implementing thread safe objects is to allow access to mutable fields without synchronization. The detector for this bug pattern looks for such errors by analyzing accesses to fields to determine which accesses are made while the object's self lock is held. Fields which are sometimes accessed with a self lock held and sometimes without are candidate instances of this bug pattern. We use heuristics to reduce the number of false positives:

- Public fields are ignored
- Volatile fields are ignored
- Fields that are never read without a lock are ignored
- Accesses in object lifecycle methods (such as constructors and finalizers), or in nonpublic methods reachable only from object lifecycle methods, are ignored

In order to avoid reporting fields which are only incidentally locked, we ignore fields for with a high (1/3 or more) proportion of unlocked accesses, according to the formula

$$2(\mathrm{RU} + 2\mathrm{WU}) > (\mathrm{RL} + 2\mathrm{WL})$$

```
// GNU classpath 0.06,
// java.util
// Vector.java, line 354

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

**Figure 2: Example of inconsistent synchronization**

where RU and WU are unlocked reads and writes, and RL and WL are locked reads and writes. This formula gives higher precedence to writes, since they are less frequent than reads.

This detector is more complicated than any other. It uses dataflow analysis to determine where locks are held and to determine which objects are locked, since the analysis relies on being able to determine when a lock is held on the reference through which a field is accessed. We perform redundant load elimination to determine when a value loaded from a field is likely to be the same value as that of an earlier load of the same field; without this analysis, the detector would report false positives for references of the form

```
synchronized (x) {
  ... = x.f;
  x.g = ...
}
```

where x is a field. The detector uses a whole program analysis, since fields may be accessed from classes other than the class in which they are defined.

The inconsistent synchronization detector is interesting because inferring whether or not a class is intended to be thread-safe in the absence of specifications is difficult. The detector relies on the fact that synchronizing on the `this` reference is a common idiom in Java, and uses heuristics to make an educated guess about whether synchronization is intended or incidental.

An example of inconsistent synchronization is seen in Figure 2. This method is in the `Vector` class of the GNU Classpath library, version 0.06. The `Vector` class is specified as being thread safe. Most of the accesses to the class's fields are protected by synchronizing on the the object's `this` reference. In the code shown, the `elementCount` field is accessed without synchronization. The bug is that because of the lack of synchronization, the element count may not be accurate when it is passed to the `lastIndexOf(Object, int)` method (which is synchronized). This could result in an `IndexOutOfBoundsException`.

In the development of the FindBugs tool, we have run it regularly on several applications and libraries, tracking new releases of those packages as they are made available. We have noticed that it is fairly common for programmers not to understand the synchronization requirements of classes when they are performing maintenance (to fix bugs or add new features), and we have seen a number of synchronization bugs introduced this way. Detectors such as the inconsistent synchronization detector can serve as a useful safeguard against the introduction of bugs during code evolution.

#### 2.3.4 Static Field Modifiable By Untrusted Code (MS)

This problem describes situations where untrusted code is allowed to modify static fields, thereby modifying the behavior of the library for all uses. There are several possible ways this mutation is allowed:

- A static non-final field has public or protected access.
- A static final field has public or protected access and references a mutable structure such as an array or `Hashtable`.
- A method returns a reference to a static mutable structure such as an array or `Hashtable`.

For example, Sun's implementation of `java.awt.Cursor` has a protected static non-final field `predefined` that caches references to 14 predefined cursor types. Untrusted code can extend `Cursor` and freely modify the contents of this array or even make it point to a different array of cursors. This would allow, for example, any applet to change the cursors in a way that would affect all applets displayed by the browser, possibly confusing or misleading the user.

#### 2.3.5 Null Pointer Dereference (NP)

Calling a method or accessing an instance field of a null value results in a `NullPointerException` at runtime. This detector looks for instructions where a null reference might be dereferenced.

Our implementation of the detector for this pattern uses a straightforward dataflow analysis. The analysis is strictly intraprocedural; it does not try to determine whether parameters to a method could be null, or whether return values of called methods could be null. The detector takes if comparisons into account to make the analysis more precise. For example, in the code

```
if (foo == null) {
    ...
}
```

the detector knows that `foo` is null inside the body of the `if` statement.

Two types of warnings are produced. Null pointer dereferences that would be guaranteed to occur given full statement coverage of the method are assigned high priority. Those guaranteed to occur given full branch coverage of the method are assigned medium priority. False warnings are possible because the detector may consider infeasible paths.

An example of a null pointer dereference found by the detector is shown in Figure 3. We were surprised to find that a bug this obvious could find its way into a mature, well-tested

```
// Eclipse 2.1.0,
// org.eclipse.jdt.
//   internal.ui.javaeditor,
// ClassFileEditor.java, line 225

if (entry == null) {
  IClasspathContainer container=
    JavaCore.getClasspathContainer(
     entry.getPath(), // entry is null!
     root.getJavaProject());
```

**Figure 3: Example of null pointer dereference**

```
// Sun JDK 1.5 build 18,
// java.awt, MenuBar.java, line 164

if (m.parent != this) {
    add(m);
}
helpMenu = m;
if (m != null) {
    ...
```

**Figure 4: A redundant null comparison.**

application. However, this was only one of a significant number of similar bugs we found in Eclipse.

In addition to finding possible null pointer dereferences, this detector also identifies reference comparisons in which the outcome is fixed because either both compared values are null, or one value is null and the other non-null. Although this will not directly result in misbehavior at runtime, it very often indicates confusion on the part of the programmer, and may indicate another error indirectly. (This phenomenon is described in greater detail in [28].) Figure 4 shows some code from the `java.awt.MenuBar` class from Sun JDK 1.5 build 18. Here the code manifests the implicit belief that `m` is not null, because the `parent` field is accessed, and later that it might be null because it is explicitly checked. Because the beliefs contradict, one must be incorrect.

### 2.3.6 Open Stream (OS)

When a program opens an input or output stream, it is good practice to ensure that the stream is closed when it becomes unreachable. Although finalizers ensure that Java I/O streams are automatically closed when they are garbage collected, there is no guarantee that this will happen in a timely manner. There are two reasons why streams should be closed as early as possible. First, operating system file descriptors are a limited resource, and running out of them may cause the program to misbehave. Another reason is that if a buffered output stream is not closed, the data stored in the stream's buffer may never be written to the file (because Java finalizers are not guaranteed to be run when the program exits).

The Open Stream detector looks for input and output stream objects which are created (opened) in a method and are not closed on all paths out of the method. The implementation uses dataflow analysis to determine all of the instructions

```
// DrJava stable-20030822
// edu.rice.cs.drjava.ui
// JavadocFrame.java, line 97

private static File _parsePackagesFile(
  File packages, File destDir) {
  try {
    FileReader fr =
      new FileReader(packages);
    BufferedReader br =
      new BufferedReader(fr);
    ...
    // fr/br are never closed
```

**Figure 5: Example of open stream**

reached by the definitions (creation points) of streams created within methods, and to track the state (nonexistent, created, open, closed) of those streams. If a stream in the open state reaches the exit block of the control flow graph for a method, we emit a warning.

To reduce false positives, we ignore certain kinds of streams. Streams which escape (are passed to a method or assigned to a field) are ignored. Streams that are known not to correspond to any real file resource, such as byte array streams, are ignored. Finally, any stream transitively constructed from an ignored stream is ignored (since it is common in Java to "wrap" one stream object with another).

An example of a bug found by this detector is shown in Figure 5. The `FileReader` object is never closed by the method.

### 2.3.7 Read Return Should Be Checked (RR)

The `java.io.InputStream` class has two `read()` methods which read multiple bytes into a buffer. Because the number of bytes requested may be greater than the number of bytes available, these methods return an integer value indicating how many bytes were actually read.

Programmers sometimes incorrectly assume that these methods always return the requested number of bytes. However, some input streams (e.g., sockets) can return short reads. If the return value from `read()` is ignored, the program may read uninitialized/stale elements in the buffer and also lose its place in the input stream.

One way to implement this detector would be to use dataflow analysis to determine whether or not the location where the return value of a call to `read()` is stored is ever used by another instruction. Because our primary aim is to understand bug patterns and the ways in which they manifest, and not to develop a comprehensive analysis framework, we instead chose a much simpler implementation strategy. The detector for this bug pattern is implemented as a simple linear scan over the bytecode. If a call to a `read()` method taking a byte array is followed immediately by a `POP` bytecode, we emit a warning. As a refinement, if a call to `InputStream.available()` is seen, we inhibit the emission of any warnings for the next 70 instructions. This eliminates

```
// GNU Classpath 0.06
// java.util
// SimpleTimeZone.java, line 780

int length = input.readInt();
byte[] byteArray = new byte[length];
input.read(byteArray, 0, length);
if (length >= 4)
  ...
```

**Figure 6: An example of read return ignored**

some false positives where the caller knows that the input stream has a certain amount of data available.

An example of a bug found by this detector is shown in Figure 6. This code occurs in the class's `readObject()` method, which deserializes an instance of the object from a stream. In the example, the number of bytes read is not checked. If the call returns fewer bytes than requested, the object will not be deserialized correctly, and the stream will be out of sync (preventing other objects from being deserialized).

### 2.3.8 Return Value Should Be Checked (RV)

The standard Java libraries have a number of immutable classes. For example, once constructed, Java `String` objects do not change value. Methods that transform a `String` value do so by returning a new object. This is often a source of confusion for programmers used to other languages (such as C++) where string objects are mutable, leading to mistakes where the return value of a method call on an immmutable object is ignored.

The implementation of the detector for this bug pattern is very similar to that of the Read Return detector. We look for calls to any memory of a certain set of methods followed immediately by `POP` or `POP2` bytecodes. The set of methods we look for includes

- Any `String` method returning a `String`
- `StringBuffer.toString()`
- Any method of `InetAddress`, `BigInteger`, or `BigDecimal`
- `MessageDigest.digest(byte[])`
- The constructor for any subclass of `Thread`

This detector shows how an automatic tool can be an effective remedy to a common misconception about API semantics.

### 2.3.9 Uninitialized Read In Constructor (UR)

When a new object is constructed, each field is set to the default value for its type. In general, it is not useful to read a field of an object before a value is written to it. Therefore, we check object constructors to determine whether any field is read before it is written. Often, a field read before it is initialized results from the programmer confusing the field with a similarly-named parameter.

```
// JBoss 3.2.2RC3
// org.jboss.security.auth.callback
// ByteArrayCallback.java, line 26

public ByteArrayCallback(String propmpt)
{
  this.prompt = prompt;
}
```

**Figure 7: Example of uninitialized read in constructor**

```
// JBoss 3.2.2RC3
// org.jboss.deployment.scanner
// AbstractDeploymentScanner.java,
// line 185

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

**Figure 8: An example of an unconditional wait**

An example of a bug found by this detector is shown in Figure 7. In this case, the error is simply a misspelling of the name of the constructor's parameter.

### 2.3.10 Unconditional Wait (UW)

Predicting the ordering of events in a multithreaded program can be difficult. Therefore, when waiting on a monitor, it is a good (and usually necessary) practice to check the condition being waited for before entering the wait. Without this check, the possibility that the event notification has already occurred is not excluded, and the thread may wait forever.

The detector for this bug pattern uses a linear scan over the analyzed method's bytecode. It looks for calls to `wait()` which are preceeded immediately by a `monitorenter` instruction and are not the target of any branch instruction.

An example of a bug found by this detector is shown in Figure 8. The `enabled` field may be modified by multiple threads. Because the check of `enabled` is not protected by synchronization, there is no guarantee that `enabled` is still true when the call to `wait()` is made. This kind of code can introduce bugs that are very hard to reproduce because they are timing-dependent.

### 2.3.11 Wait Not In Loop (Wa)

Java's `Object.wait()` method waits on a monitor for another thread to call `notify()` or `notifyAll()`. Usually, the thread calling `wait()` is waiting for a particular condition to become true. The most robust way to implement a condition wait is to put it in a loop, where the waited-for condition is checked each time the thread wakes up. This coding style accommodates false notifications arising when a single

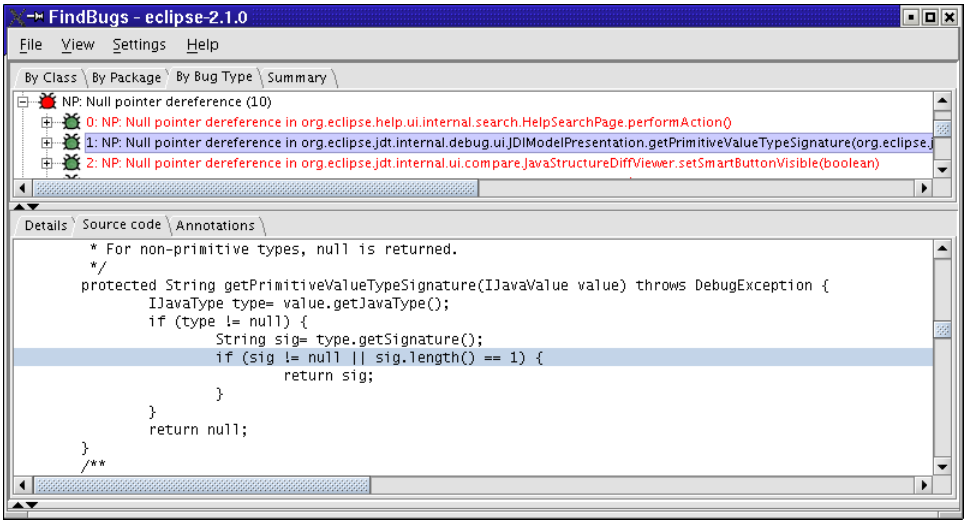


**Figure 9: Screenshot of FindBugs**

monitor is used for multiple conditions. False notifications can also be caused by the underlying native synchronization primitives.

While it is possible to correctly use `wait()` without a loop, such uses are rare and worth examining, particularly in code written by developers without substantial training and experience writing multithreaded code.

## 3. IMPLEMENTATION

FindBugs is available under an open source license. Downloads and documentation are available on the web at `http://findbugs.sourceforge.net`.

We have implemented several front ends to FindBugs:

- A simple batch application that generates text reports, one line per warning.
- A batch application that generates XML reports.
- An interactive tool that can be used for browsing warnings and associated source code, and annotating the warnings. The interactive tool can read and write the XML reports. A screenshot of the interactive tool is shown in Figure 9.

We have also developed several tools for manipulating the XML reports. These tools allow us to perform actions such as comparing the difference in warnings generated on two different versions of an application.

FindBugs users have contributed two additional front-ends: a plugin that integrates FindBugs into the Eclipse[12] IDE, and a task for running FindBugs from the Apache Ant[2] build tool.

## 4. EVALUATION

It is easy to apply our bug pattern detectors to software. However, evaluating whether the warnings generated correspond to errors that warrant fixing is a manual, time-consuming and subjective process. We have made our best effort to fairly classify many of the warnings generated. Some of the situations that arise include:

- 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 warned about 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 *harmless bugs*.
- Some warnings do not accurately reflect the problem with the code they identify. However, the code may be non-standard or dubious in a way that is, at best, confusing and at worst, wrong in some way other than that identified by the detector. Such warnings are classified as *dubious*.
- 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 section, we report on our manual evaluation of several warning categories reported by FindBugs on the following applications/libraries:

- GNU Classpath, version 0.06
- rt.jar from Sun JDK 1.5.0, build 18
- Eclipse, version 2.1.0
- DrJava, version stable-20030822
- JBoss, version 3.2.2RC3
- jEdit, version 4.1

GNU Classpath [15] is an open source implementation of the core Java runtime libraries. rt.jar is Sun's implementation of the APIs for J2SE [18]. Eclipse [12] and DrJava [10] are popular open source Java integrated development environments. JBoss [20] is a popular Java application server. jEdit [21] is a programmer's text editor.

All of the applications and libraries we used in our experiments, with the possible exception of GNU Classpath, are commercial-grade software products with large user communities. Except for rt.jar, we used stable release versions. (The version of rt.jar we analyzed is from a non-public build.)

None of the analyses implemented in FindBugs is particularly expensive to perform. On a 1.8 GHz Pentium 4 Xeon system with 1 GB of memory, FindBugs took no more than 13 minutes to run all of the bug pattern detectors on any of the applications we analyzed. To give a sense of the raw speed of the analysis, the version of rt.jar we analyzed contains 11,051 classes, is about 30 MB in size, and required 10 minutes to analyze. The maximum amount of memory required to perform the analyses was approximately 500 MB. We have not attempted to tune FindBugs for performance or memory consumption.

## 4.1 Empirical Evaluation

Figure 10 shows our evaluation of the accuracy of the detectors for which there are clear criteria for deciding whether or not the reports represent real bugs. All of the detectors evaluated found at least one bug pattern instance which we classified as a real bug.

The null pointer dereference detector found the largest number of genuine bugs. The numbers shown do not include warnings generated by the detector for redundant null comparisons, many of which also indicate real problems in the code.

It is interesting to note that the accuracy of the detectors varied significantly by application. For example, the detector for the RR pattern was very accurate for most applications, but was largely unsuccessful in finding genuine bugs in Eclipse. The reason is that most of the warnings in Eclipse were for uses of a custom input stream class for which the `read()` methods were guaranteed to return the number of bytes requested.

| Application | Eq | HE | MS |
|---|---|---|---|
| classpath-0.06 | 4 | 19 | 32 |
| rt.jar 1.5.0 build 18 | 13 | 83 | 162 |
| eclipse-2.1.0 | 1 | 58 | 659 |
| drjava-stable-20030822 | 0 | 78 | 196 |
| jboss-3.2.2RC3 | 0 | 21 | 165 |
| jedit-4.1 | 0 | 7 | 19 |

**Figure 11: Bug counts for other detectors**

Our target for bug detectors admitting false positives was that at least 50% of reported bugs should be genuine. In general, we were fairly close to meeting this target. 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. We also found that these detectors were much more successful in finding errors in code written in undergraduate courses, which illustrates the role of tools in steering novices towards correct use of difficult language features and APIs.

It is worth emphasizing that all of these applications and libraries (with the possible exception of GNU Classpath) have been extensively tested, and are used in production environments. The fact we were able to uncover so many bugs in these applications makes a very strong argument for the need for automatic bug checking tools in the development process. Static analysis tools do not require test cases, and do not have the kind of preconceptions about what code is "supposed" to do that human observers have. For these reasons, they usefully complement traditional quality assurance practices.

Ultimately, no technique short of a formal correctness proof will ever find every bug. From our evaluation of FindBugs on real applications, we conclude that simple static tools like bug pattern detectors find an important class of bugs that would otherwise go undetected.

## 4.2 Other Detectors

In Figure 11 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 judgement as to whether feature is a bug that warrants fixing in each particular instance. We will simply note that at best, these reports represent instances of poor style.

# 5. ANECDOTAL EXPERIENCE

In this section, we report on some of our experience in using this tool in practice. In addition, we report some information about the experience of an external organization which has adopted our tool. These reports are not rigorous, but simply anecdotal reports of our experience.

## 5.1 Our Experience

| | classpath-0.06 | | | | | rt.jar 1.5.0 build 18 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | warnings | serious | harmless | dubious | false pos | warnings | serious | harmless | dubious | false pos |
| DC | 0 | — | — | — | — | 6 | 83% | 0% | 0% | 16% |
| IS2 | 18 | 72% | 16% | 0% | 11% | 52 | 30% | 63% | 0% | 5% |
| NP | 7 | 85% | 0% | 0% | 14% | 21 | 95% | 0% | 0% | 4% |
| OS | 9 | 22% | 33% | 22% | 22% | 5 | 0% | 0% | 0% | 100% |
| RR | 7 | 100% | 0% | 0% | 0% | 10 | 100% | 0% | 0% | 0% |
| RV | 11 | 45% | 0% | 0% | 54% | 2 | 100% | 0% | 0% | 0% |
| UR | 3 | 100% | 0% | 0% | 0% | 3 | 100% | 0% | 0% | 0% |
| UW | 2 | 0% | 0% | 0% | 100% | 6 | 33% | 0% | 0% | 66% |
| Wa | 2 | 0% | 0% | 0% | 100% | 6 | 16% | 0% | 0% | 83% |

| | eclipse-2.1.0 | | | | | drjava-stable-20030822 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | warnings | serious | harmless | dubious | false pos | warnings | serious | harmless | dubious | false pos |
| NP | 43 | 93% | 0% | 6% | 0% | 0 | — | — | — | — |
| OS | 16 | 6% | 6% | 18% | 68% | 5 | 40% | 0% | 40% | 20% |
| RR | 22 | 4% | 0% | 0% | 95% | 0 | — | — | — | — |
| RV | 9 | 100% | 0% | 0% | 0% | 0 | — | — | — | — |
| UR | 0 | — | — | — | — | 1 | 100% | 0% | 0% | 0% |
| UW | 0 | — | — | — | — | 3 | 66% | 0% | 0% | 33% |

| | jboss-3.2.2RC3 | | | | | jedit-4.1 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | warnings | serious | harmless | dubious | false pos | warnings | serious | harmless | dubious | false pos |
| IS2 | 2 | 50% | 0% | 0% | 50% | 1 | 0% | 100% | 0% | 0% |
| NP | 10 | 100% | 0% | 0% | 0% | 0 | — | — | — | — |
| OS | 2 | 100% | 0% | 0% | 0% | 1 | 100% | 0% | 0% | 0% |
| RR | 0 | — | — | — | — | 1 | 100% | 0% | 0% | 0% |
| RV | 2 | 0% | 0% | 0% | 100% | 0 | — | — | — | — |
| UR | 2 | 50% | 0% | 0% | 50% | 2 | 50% | 0% | 50% | 0% |
| UW | 1 | 100% | 0% | 0% | 0% | 1 | 100% | 0% | 0% | 0% |
| Wa | 0 | — | — | — | — | 2 | 50% | 0% | 0% | 50% |

**Figure 10: Evaluation of false positive rates for selected bug pattern detectors**

Our null pointer exception detector was partially inspired by a bug in Eclipse 2.1.0 that was fixed in 2.1.1 (bug number 35769). (This bug is shown in Figure 3.) Our detector accurately identifies this error as well as a number of similar errors in the current releases of Eclipse.

Early, non-public builds of Sun's JDK 1.4.2 included a substantial rewrite of the StringBuffer class. Our tool found a serious data race in the `append(boolean)` method, and our feedback to Sun led them to fix this error in build 12.

In build 32 of Sun's JDK 1.5, a potential deadlock was introduced into the `om.sun.corba.se.impl.orb.ORBImpl` class. In this case, the `get_next_response()` calls `wait()` while holding a lock that is also needed by the `notifyORB()` method, which is responsible for notifying the waiting thread. We discoved the bug using a detector which looks for calls to `wait()` made with more than one lock held. Even though it is possible to write correct code that calls `wait()` while holding a lock, this idiom is error prone and warrants careful inspection when used. When we reported the issue to Sun, it was confirmed to be a bug, fixed internally, and the fixed version is slated to be part of the next JDK 1.5 beta release.

We applied our tool to the Java implementation of the International Children's Digital Library [11], and found a number of serious synchronization and thread problems. In talking to the developers, some of the errors we found closely paralleled a bug in an earlier version of the software that they had spent substantial time trying to diagnose and fix in the previous months. When we applied our tool to an earlier version of their software, we immediately found the error they had spent substantial time on.

We have used our tool in a senior level undergraduate course that covers many advanced Java topics, including threads and distributed programming. In the current semester, we are encouraging students to use FindBugs as part of their development process for their projects. We have also applied the tool to project submissions from previous semesters. Our tool found many errors, including problems that one would never expect to find in production code, such as

- executing the run method of a Thread rather than starting the thread, and
- performing wait() or notify() on an object while not holding a lock on that object.

We did not expect that the detectors we wrote particularly for undergraduate course work would find many problems in production code. However, we did find a case in Eclipse where a thread object was created but never run.

One of the nice features of the Java programming language is that it is impossible to "forget" to release a lock. However, JCP JSR-166 introduces a new set of more powerful concurrency abstractions to the Java programming language. These new concurrency mechanisms include locks which do not enforce that acquire and release calls are balanced. We worked with the JSR-166 expert group to develop a detector that generate a warning if an lock acquisition was not obviously matched with a release on all program paths (including paths arising from exceptions). When run on the code developed as part of JSR-166, we generated only 9 warnings on 78 uses of acquire that corresponded to places where non-standard locking was being performed. We didn't find any errors, but the care that went into the crafting of that library made it unlikely that there would be any to find.

## 5.2 User Experience

Since we made FindBugs available under an open source license, it has been used by a variety of individuals and organizations. One of the users is an organization developing a Geographic Information System application. The application comprises over 200 packages, 2,000 classes, and 400,000 lines of code.

The first time FindBugs was applied to this code base, it reported around 3,000 warnings. The developers on the project spent about 1 week addressing these issues, eventually getting the total number of warnings down below 1,000. This shows that, for this application, a significant number of the warnings issued by FindBugs were judged by the developers to be worth fixing. The project leader notes that they are still fixing some of the remaining warnings. Some of the specific types of errors reported by FindBugs that the developers have fixed are hashcode/equals problems, serialization issues, unchecked method return values, unused fields and constants, mutable static data, and null pointer dereferences.

Two specific errors caught by FindBugs illustrate how automatic tools can be useful for finding subtle bugs in code that appears to be reasonable on the surface. For example, the developers found the following problem stemming from an incorrect boolean operator:

```
if ((tagged == null) && (tagged.length < rev))
```

The second error resulted from confusion between an instance field and an identically named method parameter:

```
public void setData(String keyName, String valName,
  HashMap hashMap)
{
  if (hashMap != null)
    this.hashMap = hashMap;
  else
    this.hashMap = new HashMap(true);

  if (hashMap.size() > 0) {
    ...
}
```

Both of these examples *look* like they ought to work, making it hard to find them by human inspection.

Finally, the lead developer notes that FindBugs has been helpful in large part because of its ease of application; it can be run on the entire application in a few minutes.

## 6. RELATED WORK

There has been substantial work in using bug pattern detection for finding errors in C and C++ code. Lint, of course, is the grandfather of all the C error finders [22]. The Metal system, which works by simulating program executions with finite state automata, has found thousands of bugs in the Linux kernel [16, 28]. The ASTLog tool [8], which looks for syntactically suspicious code patterns, has been extended into the PREfast tool [23], used extensively within Microsoft as a bug pattern detector.

There are a fair number of books and web sites that describe good programming style and bug patterns in Java [1, 27, 5, 19, 9]. However, these are prose descriptions, not automated tools (and not all bug patterns can be successfully automated).

Several implementations of Java code checkers exist. One of the more complete implementations is PMD [25], which checks for patterns in the abstract syntax trees of parsed Java source files. PMD implements checks for several of the bug patterns which we used in this paper, such as hashcode/equals and double-checked locking. However, many of the detectors we have implemented, such as inconsistent synchronization and mutable static fields, are not present in PMD. Like our tool, PMD uses Visitors to implement its pattern detectors. One nice feature of PMD is that patterns may be implemented as XPath expressions, so that new checks can be added without the need to write a new Visitor class. PMD is based on source code and is more suited to checking stylistic rules than for checking low-level code features such as access to fields. In its current form, PMD analyzes each source file in isolation.

## 7. CONCLUSIONS AND FUTURE WORK

We believe that bug patterns are a very effective and practical technique for finding bugs in software. In the future, we would like to incorporate new bug pattern detectors into FindBugs. In particular, we would like to find ways to make bug patterns easy to specify, ideally to the point where developers could add detectors for new patterns. All of our conclusions regarding error-prone APIs in the Java libraries also apply to custom applications, so providing a way for development organizations to add their own project-specific bug patterns is an important goal. Towards this end, we have experimented with using regular expression patterns to match bytecode sequences, with encouraging results.

We would also like to refine our existing bug detectors to make them more accurate. One possibility would be to add user-tunable analysis parameters which could be adjusted for the characteristics of a particular application. Another possibility would be to investigate more sophisticated kinds of analysis, such as taking interprocedural information into account.

## Acknowledgments

Many of the ideas for bug patterns to look for came from conversations and email discussions with Joshua Bloch and Doug Lea, in addition to the books and resources cited [1, 27, 5, 19, 9]. We would also like to thank J. Keller for valuable feedback.

## 8. REFERENCES

[1] Eric Allen. *Bug Patterns In Java*. APress, 2002.

[2] Apache Ant, http://ant.apache.org/, 2003.

[3] Thomas Ball and Sriram K. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3, Portland, Oregon, January 2002.

[4] The Byte Code Engineering Library, http://jakarta.apache.org/bcel/, 2003.

[5] Joshua Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2002.

[6] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice & Experience*, 30:775–802, 2000.

[7] Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for object-oriented programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, July 2002.

[8] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In *USENIX Conference on Domain Specific Languages*, pages 229–241, Santa Barbara, 1997.

[9] Michael C. Daconta, Eric Monk, J. Paul Keller, and Keith Bohnenberger. *Java Pitfalls*. John Wiley & Sons, Inc., 2000.

[10] DrJava, http://www.drjava.org/, 2003.

[11] A. Druin, Ben Bederson, A. Weeks, A. Farber, J. Grosjean, M.L. Guha, J.P. Hourcade, J. Lee, S. Liao, K. Reuter, A. Rose, Y. Takayama, L., and L Zhang. The international children's digital library: Description and analysis of first use. Technical Report HCIL-2003-02, Human-Computer Interaction Lab, Univ. of Maryland, January 2003.

[12] Eclipse, http://www.eclipse.org/, 2003.

[13] David Evans. Static Detection of Dynamic Memory Errors. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 44–53, Philadelphia, Pennsylvania, May 1996.

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In PLDI'02 [24], pages 234–245.

[15] GNU Classpath, http://www.gnu.org/software/classpath/, 2003.

[16] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A System and Language for Building System-Specific, Static Analyses. In PLDI'02 [24], pages 69–82.

[17] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the International Conference on Software Engineering*, pages 291–301, May 2002.

[18] Java(tm) 2 Platform, Standard Edition, http://java.sun.com/j2se/, 2003.

[19] Collected java practices. http://www.javapractices.com.

[20] JBoss, http://www.jboss.org/, 2003.

[21] jEdit, http://www.jedit.org/, 2003.

[22] S.C. Johnson. Lint, a c program checker. In *UNIX Programmer's Supplementary Documents Volume 1 (PS1)*, April 1986.

[23] Jonathan D. Pincus. Helping you succeed with PREfix & PREfast. http://research.microsoft.com/specncheck/docs/pincus.ppt, 2001.

[24] *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[25] PMD, http://pmd.sourceforge.net/, 2003.

[26] Nicholas Sterling. WARLOCK — a static data race analysis tool. In *Proceedings of the USENIX Annual Technical Conference*, pages 97–106, Winter 1993.

[27] Bruce Tate. *Bitter Java*. Manning Publications, 2002.

[28] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, November 2002.