Editor: Christof Ebert ■ Alcatel ■ christof.ebert@alcatel.com

Static Code Analysis

Panagiotis Louridas

Wool has a tendency to collect static electricity and thus to attract dust and lint. Developers know that programs have a similar tendency to attract defects and, furthermore, that many of them aren't visible to compilers. In the 1970s, Stephen Johnson, then at the Bell Laboratories, wrote Lint, a tool to examine C source programs that had compiled without errors and to find bugs that had escaped detection.[1]

There are many ways to reduce the number of bugs in a program. Writing tests is one, and tools such as JUnit help programmers do this.[2] Research tells us that code reviews are probably the best way to eliminate bugs. Unfortunately, getting the right people together to study programs and identify problem areas in them takes a lot of time. Code review teams also need practice (perhaps even training) to do the work right. It's therefore impossible to use them on a project's complete code base.

Moreover, the earlier we find bugs, the easier it is to fix them. Ideally, we would like to catch errors when we make them, or as close afterward as possible, and not ipso facto with reviewing or testing.

Most errors fall into known categories, as people tend to fall into the same traps repeatedly. It's exactly the predictability of people's fallibility that gives tools such as Lint a chance. Lint worked by looking for known error patterns. It didn't try to execute the program and compare actual with expected behavior, which is the dynamic analysis that we do when we test. Instead, Lint trawled through the program source trying to match patterns. Such tools are called *static checkers*. They check our programs for errors without executing them, in a process called static code analysis.

Programmers usually employ static checkers after compilation and before testing. In this way, they work with a program that has an initial indication of correctness (because it compiles) and try to avoid well-known traps and pitfalls before measuring it against its specifications (when it's tested). Static checking is relatively painless, although it can be humbling—especially the first time. Lint's success has given today's programmers many descendant tools, both open source and proprietary, that target different languages and operating systems.

Editor's introduction

Bugs are disturbing. This holds in summer with the insect types and year-round with the software types. While developers have long deployed reviews, inspections, and different testing strategies to find software bugs, they don't yet widely use the available semiautomatic defect-detection techniques. Panagiotis Louridas explains here how static code-analysis tools are used and what defects they can detect. As usual, the column compares several open source tools together with a commercially available tool. They can help you reduce the number of bugs of all the software types—security, memory, data typing, and so on—before you deploy the more expensive verification techniques.

—Christof Ebert

Static code checking in Java

To see static code checking in action, let's start with the coding horror in figure 1. Although this program compiles, it fails to do what the programmer wants. The programmer intends to read a string from the user, substi-

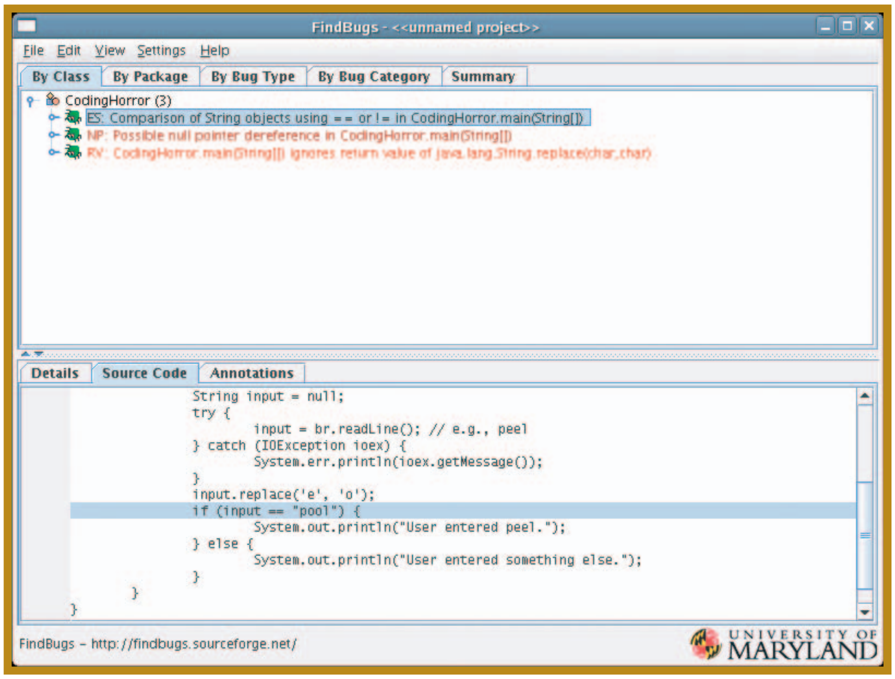
```
1  import java.io.InputStreamReader;
2  import java.io.BufferedReader;
3  import java.io.IOException;
4
5  public class CodingHorror {
6
7     public static void main(String args[]) {
8
9        InputStreamReader isr = new InputStreamReader(System.in);
10       BufferedReader br = new BufferedReader(isr);
11       String input = null;
12       try {
13          input = br.readLine(); // e.g., peel
14       } catch (IOException ioex) {
15          System.err.println(ioex.getMessage());
16       }
17       input.replace('e', 'o');
18       if (input == "pool") {
19          System.out.println("User entered peel.");
20       } else {
21          System.out.println("User entered something else.");
22       }
23    }
24 }
```

Figure 1. A coding horror.

tute all "e" characters with "o" characters, then check whether the substitution results in the string "pool." This will never be the case, no matter what input we provide.

To see why, we can use FindBugs (http://findbugs.sourceforge.net), a popular open source static code checker for Java, developed at the University of Maryland. After running FindBugs on the program in figure 1, we get the screen in figure 2. It shows three possible bugs:

- In the comparison at line 18 of the source code, the == operator compares object references, not the objects themselves. So, unless the two strings we compare are stored in the same object, the comparison will fail.
- If, for some reason, the read operation at line 13 fails, `input` will be null, and the program will try to call the method `replace` on a null object. (I must admit that, when preparing this example, I hadn't planned for this bug involving the use of a possibly null pointer.)
- Because strings are immutable in Java, `replace` doesn't modify the original string. Instead, it returns a new string with the results of any replacements it carries out. The program simply ignores the result string



```
CodingHorror (3)
   ES: Comparison of String objects using == or != in CodingHorror.main(String[])
   NP: Possible null pointer dereference in CodingHorror.main(String[])
   RV: CodingHorror.main(String[]) ignores return value of java.lang.String.replace(char,char)
```

```
      String input = null;
      try {
         input = br.readLine(); // e.g., peel
      } catch (IOException ioex) {
         System.err.println(ioex.getMessage());
      }
      input.replace('e', 'o');
      if (input == "pool") {
         System.out.println("User entered peel.");
      } else {
         System.out.println("User entered something else.");
      }
   }
}
```

Figure 2. CodingHorror output in the FindBugs static code checker.

Code Checker Resources

For details on FindBugs, see "Finding Bugs Is Easy" by David Hovemeyer and William Pugh (*ACM Sigplan Notices*, Dec. 2004, pp. 92–106). In addition, IBMdeveloperWorks has a two-part article on it (http://www-128.ibm.com/developerworks/java/library/j-findbug1/, http://www-128.ibm.com/developerworks/java/library/j-findbug2/); the CodingHorror presented here is inspired by similar snippets there.

PMD Applied, by Tom Copeland (Centennial Books, 2005), describes PMD in detail.

Diomidis Spinellis's "Bug Busters" (*IEEE Software*, vol. 23, no. 2, 2006, pp. 92–93) looks at code analysis tools from a wide software engineering perspective.

In any project, some modules are more critical than others; moreover, not all bugs are worth fixing—a topic that Edward N. Adams addresses in "Optimizing Preventive Service of Software Products" (*IBM J. Research and Development*, vol. 28, no. 1, 1984, pp. 2–14). For a discussion of different ways to predict error-prone modules and thus direct attention to them (for instance, conducting code reviews), see Christof Ebert and Ekkehard Baisch, "Industrial Application of Criticality Predictions in Software Development" (*Proc. 9th Int'l Symp. Software Reliability Eng.* (ISSRE 98), IEEE CS Press, 1998, pp. 80–89).

Piet Hein (1905–1996), a Danish poet and scientist with wide-ranging interests, wrote short poems known as "grooks." "The Road to Wisdom," a grook that Donald Knuth brought to the attention of the computer science community, is always an elegant reminder:

The road to wisdom? — Well it's plain
and simple to express:
Err
and err
and err again
but less
and less
and less.

in line 17. The remainder of the program continues to use the original string. This bug is arguably the most subtle of the set.

The example might seem contrived, but such bugs can wreak havoc on large projects, especially when deadlines loom and fatigue kicks in. In general, static code checkers can find many kinds of bugs—FindBugs has more than 200 bug patterns. They're usually run as part of the build process, and their output comes in several formats that users can view through a Web browser.

Background

Static code checkers in Java come in two flavors: those that work directly on the program source code and those that work on the compiled bytecode. Each has its own advantages. When you work directly on the program source, your checking mirrors the exact program code written by the programmer. Compilers optimize code, and the resulting bytecode might not mirror the source. On the other hand, working on bytecode is considerably faster, which is important on projects containing more than a few tens of thousands of lines of code.

Although each code checker works in its own way, most share some basic traits. They read the program and construct some model of it, a kind of abstract representation that they can use for matching the error patterns they recognize. They also perform some kind of data-flow analysis, trying to infer the possible values that variables might have at certain points in the program. This is how, for instance, FindBugs found the second bug of our CodingHorror.

Data-flow analysis is especially important for vulnerability checking—an increasingly important area for code checkers. Whenever a program accepts input from a user, there's a possibility (albeit remote) that a cracker can use the input to subvert the system. Buffer overflows have been a cracker's favorite inroad until recently, when SQL injection seems to have overtaken the top place for program sore spots. In both cases, it's important to be able to trace the input flow from users through the program.

No code checker can ever assure us that a program is correct—such guarantees aren't possible. In fact, no code checker is complete or sound. A complete code checker would find all errors, while a sound code checker would report only real errors and no false positives. The percentage of false to true positives is an important indicator of a code checker's suitability for our programs—it makes sense to examine a checker's behavior in our work before committing to it in a whole project.

Although code checkers rest on the idea that human fallibility is somewhat predictable, they can't take all possible bugs into account. Many checkers, including FindBugs, let programmers define their own rules for the checker to use. In this way, if developers know that they're particularly prone to some kinds of bugs, they can guard against them by writing custom bug detectors.

Eliminating bugs doesn't ensure high program quality. Many code quality metrics exist. Cyclomatic complexity, a well-known one, assumes that overly complex programs are difficult to maintain and more likely to contain errors.[3] Among umpteen measures for object-oriented programs, Shyam Chidamber and Chris Kemerer's suite is the most widely cited.[4] Special tools for calculating such metrics exist, but code checkers can implement some of them on the side. For other code-checker resources and guidelines, see the sidebar.

Table 1

Static code checkers

Features	FindBugs	Checkstyle	PMD	Klocwork K7
Version	0.9.7	4.1	3.6	7.0.4.15
Works on	Bytecode	Source	Source	Bytecode and source
Languages	Java	Java	Java	Java, C++
Interface	GUI, command line, plugin	Command line, plugin	Command line, plugin	GUI, command line, plugin
Detects security vulnerabilities	Few	No	Few	Many
Stack overflow analysis	No	No	No	Yes (C++)
Custom checkers	Yes	Yes	Yes	Yes (C++)
Architectural analysis	No	No	No	Yes
Metrics	No	Few	No	Many
Web-based project management	HTML reports	HTML reports	HTML reports	Yes
Size	3.6 Mbytes	6.8 Mbytes	49.1 Mbytes (of which 48.6 Mbytes is documentation)	Depends on the configuration; about 250 Mbytes for a comprehensive Java installation
License	GNU Lesser General Public License (LGPL)	GNU LGPL	Berkeley Software Distribution-style license	Proprietary

A bunch of code checkers

Besides FindBugs, Checkstyle (http://checkstyle.sourceforge.net) and PMD (http://pmd.sourceforge.net/) are also popular open source tools for Java. Checkstyle started as a tool for checking compliance with coding standards, but it has evolved considerably and can now check for many coding problems. PMD is similar to FindBugs, so it might be useful to try both and see what best fits a particular project.

The Klocwork K7 (www.klocwork.com) suite is a proprietary solution that works on bytecode to find defects and security vulnerabilities and on source code to perform metrics and architectural analysis. It therefore lets developers spot different kinds of problems at different detail levels. A license for five seats offering full functionality for projects up to a half-million lines of code costs US$19,975.

All static code checkers I've examined let users filter the messages and warnings at different levels of sophistication. This feature is important because false positives can be a big problem; you need a way to reduce the noise a wealth of warnings produces. Only comprehensive testing of the tools will give developers a clear idea of how to optimize their potential at finding bugs for any given project.

Table 1 compares four tools, but it's only a starting point.

Programming is arguably one of the toughest jobs in project development. No machine can substitute for good sense, a solid knowledge of the fundamentals, clear thinking, and discipline, but bug detection tools can help developers. In a recent article,[5] Niklaus Wirth opined that "never do programs contain so few bugs as when no debugging tools are available." Caveat emptor.

References

1. S. Johnson, *Lint: A C Program Checker*, tech. report 65, Bell Laboratories, Dec.1977.
2. P. Louridas, "JUnit: Unit Testing and Coding in Tandem," *IEEE Software*, vol. 22, no. 4, 2005, pp. 12–15.
3. T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, 1976, pp. 308–320.
4. S.R. Chidamber and C.F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, 1994, pp. 476–493.
5. N. Wirth, "Good Ideas, through the Looking Glass," *Computer*, vol. 39, no. 1, 2006, pp. 28–39.

Panagiotis Louridas is a grid software engineer at the Greek Research and Technology Network and a researcher at the Eltrun Software Engineering and Security research group of the Athens University of Economics and Business. Contact him at louridas@{grnet, aueb}.gr.