Jguardrail: A Framework for Identifying Possible Errors in Student Java Code*

Ian Finlayson and Stephen Davies
Computer Science Department
The University of Mary Washington
Fredericksburg, VA 22401
ifinlay@umw.edu, sdavies@umw.edu

Abstract

This paper introduces Jguardrail, a tool for identifying potential programming errors in Java programs, especially for beginning programming students. We identified several programming patterns which often lead to bugs in student programs, that are not flagged as warnings by the Java compiler. Jguardrail is a static analysis tool, written with the ANTLR parser framework, which recognizes these patterns and provides warning messages to the programmer. By providing an additional layer of warning reporting, above what the compiler itself provides, Jguardrail aims to help students avoid common programming pitfalls. This paper discusses the patterns Jguardrail provides warnings for, its usage in a CS2 course, and a comparison to other tools.

1 Introduction

This paper presents a tool for identifying common programming errors in Java programs. Over many years of teaching using the Java programming language, we identified several patterns that result in bugs that beginning programming students struggle to find and correct. This tool, named Jguardrail [6], makes a first pass over programs looking for these mistakes and reports them as

warnings, effectively adding an extra layer of warning reporting on top of what the Java compiler offers.

Java is the predominant programming language used in computer science education at the college level [16]. Java has many features to recommend it, and is widely used in industry, but in our opinion the Java compiler does not do a great job of identifying warnings, even compared to compilers like GCC. There are many programming constructs that will certainly, or almost certainly, result in erroneous code which the Java compiler does not flag, even with extra warnings turned on. Jguardrail attempts to give programmers, especially beginners, more diagnostic warnings in such instances.

There has been much prior work relating to the most common mistakes made by novice programmers. Jackson et. al. [10] published a study in which they identify the most common compiler errors encountered by beginning Java students, using the compiler errors that the Java compiler itself gives.

Brown and Altadmri [2] investigated to what degree instructor perceptions of the common mistakes students make are accurate. Some of the mistakes in their study are syntax errors the Java compiler will always catch. Others are mistakes that the compiler does not catch, including many that Jguardrail provides warnings for.

There have been many published articles [1], [12], [13], [15] studying novice programmer's level of understanding of error messages, or works that seek to make such error messages more comprehensible to beginners. Much of this research suggests that these messages may be written for an audience of more experienced developers rather than novices, and that there are improvements to be made in gearing error and warning messages to novices.

Jguardrail differs from these works in that it seeks to actually augment the messages given by the compiler. We have identified mistakes that are not identified by the Java compiler and built a tool which does identify them.

This paper is structured as follows: Section 2 enumerates the patterns that Jguardrail detects as being possible mistakes and issues warnings for. Section 3 discusses the implementation of Jguardrail. Section 4 provides the results of using this tool with students in a CS 2 class. Section 5 compares its warnings to those identified by the Java compiler itself, as well as IDE and linting tools. Section 6 discusses future work and section 7 draws conclusions.

2 Patterns Flagged by Jguardrail

This section will describe the coding patterns flagged by Jguardrail as being either likely sources of bugs or poor programming style.

2.1 String Equality

In Java, there is almost never a reason to use the == operator to compare a string to a literal value. To that end, Jguardrail flags code like this:

```
if (choice == "Quit")
```

The programmer certainly meant to use the `.equals` method instead. This mistake can be difficult to debug since it will seem like string comparisons simply aren't working as the programmer believes they should.

2.2 Empty Control Structures

In this pattern, there is a control structure that has a single semi-colon for its body, as in the following code:

```
while (index < 100);
{
    array[index] = 0;
    index++;
}
```

Here the semi-colon on the end of the line with the while loop forms the body of the loop, meaning the loop is likely to be infinite, as in this case.

2.3 Missing Braces on Control Structures

Jguardrail also catches cases where the curly braces for a control structure are not written. Most consider omitting the braces to be poor style. For instance the Google Java Style Guide [4] dictates their use even if blocks contain a single statement. They can also lead to mistakes like this:

```
if (numbers[i] > numbers[i + 1])
    int temp = numbers[i];
    numbers[i] = numbers[i + 1];
    numbers[i + 1] = temp;
```

Here the code will not behave as the author probably intends, since only the first indented line is actually part of the if statement. Jguardrail gives warnings when braces are omitted to encourage good style and avoid such bugs.

2.4 Constant Integer Division

Jguardrail flags code where division is applied between two integers literals which do not divide evenly, as in the following code example:

```
public static double triangleArea(int base, int height) {
    return (1 / 2) * base * height;
}
```

The issue here is that Java sees 1 and 2 as integers and so applies integer division, truncating the result to 0. This can be especially confusing to students coming from languages like Python that convert numbers to floating-point values in cases like this.

2.5 Missing this

If a programmer is attempting to set an instance variable to a parameter value, as in a constructor or setter method, and forget the `this.`, then we create a useless statement such as the following:

```
public Student(String name) {
    name = name;
}
```

Jguardrail issues a warning, as it is likely that the intention was to set an instance variable with the same name instead.

2.6 Variable Shadowing

A similar mistake can be seen when a programmer accidentally shadows an instance variable, such as in the following code:

```
public void setName(String newName) {
    String name = newName;
}
```

Here, instead of setting the existing instance variable `name` to `newName`, we have introduced a new local variable with the same name (which is called shadowing). Jguardrail flags the declaration of variables which shadow instance variables like this, as it can cause difficult to debug issues where variable updates seem not to take effect.

2.7 Void Constructors

We also issue warnings when a "constructor" has accidentally been marked as `void`, as in the following code snippet:

```
public void Student() {
    this.name = "";
    this.grades = new ArrayList<>();
}
```

Since this has a return type, Java sees it not as a constructor, but rather as a method which just happens to be named the same thing as the class it is in. So the constructor will not run when objects are created, leading to a confusing debugging session. Jguardrail detects this for default constructors or those with parameters.

2.8 Uninitialized Variables

Related to the previous issue, many students make mistakes in their programs caused by not initializing instance variables. If class-type instance variables are not initialized, they begin as null references. Students often struggle with `NullPointerExceptions` caused by this issue. Jguardrail checks for instance variables that are not initialized (whether inline or in constructors) and issues warnings for them. This warning is perhaps a little opinionated since Java does guarantee initial values for all variables (unlike C++).

2.9 Mis-capitalized toString

In Java we can control how objects are displayed by overriding a method called `toString`, but this only works when the method is spelled and capitalized correctly. If the student calls the method `tostring`, for instance, Objects will be printed out as just the class name, an `@` character and the memory address the object is stored at. Jguardrail checks for methods like this and issues a warning that the programmer likely meant `toString` instead.

2.10 Missing Breaks in Switch

Finally Jguardrail looks for switch statements where there are missing break statements in the cases, as in the following code:

```
switch (option) {
    case "add":
        System.out.println("Adding");
    case "quit":
        done = true;
}
```

While there are legitimate uses of having switch cases "fall through", such as handling both upper-case and lower-case menu options, or "Duff's Device"[5], most of the time students do this by mistake. Interestingly, this is the only one of these which is caught by the `javac` compiler, and even then only with the `-Xlint:all` flag.

3 Implementation

Jguardrail essentially is a compiler front-end which parses Java code, performs some analyses and then (unlike an actual compiler) stops there. It only produces warnings and leaves actual compilation up to the existing Java compiler.

Jguardrail is implemented using the ANTLR [14] parser framework. ANTLR is a parser generator tool which also comes with example grammars for existing languages such as Java, which we used. This provides us with a syntax tree of the program being analyzed, after lexical and syntax analysis are performed. One of the great things about ANTLR is that the grammar is decoupled from actions taken after parsing. It uses the Visitor Pattern [9] to allow code to walk the parse tree after it is produced. We utilized this to write the analyses as visitors that walk over the tree after the code is parsed, searching for the patterns we identified.

Jguardrail does a separate tree-walk for each of the patterns that it catches, using a system in which new ones can be easily added to the system. Each of these overrides the needed `visit` methods ANTLR produces. For example, to catch division between two integer constants, we visit tree nodes which perform division. From there we simply check if both operands are integer constants and, if so, whether they divide evenly or not.

Some of the checks were slightly more complicated to perform. For instance to look for uninitialized instance variables, we have to make two passes. In the first we note all of the instance variables that a class has. If it's initialized inline, we skip over it, but if not, we add it to a list. In the second pass, we scan through all the constructors making sure that each one initializes all of the instance variables it needs to. Two passes are needed as constructors do not necessarily come after all instance variables. Moreover this analysis must be done using a stack as Java allows for nested classes. So we in fact keep a stack of lists of uninitialized variables to ensure that it works correctly in this case as well.

Each of these analyses records any warnings that should be issued about the code, which are displayed at the end of the program. We sort them by line number and pattern the output after that given by the `javac` compiler.

4 Evaluation

This tool was used in two sections of a CS 2 class during the Spring 2024 semester. The class uses the Java programming language and introduces concepts of object-oriented design and analysis. This class was taught using a command-line interface in which students write programs in the Vim text editor and compile using the `javac` command, on a shared server operated by the

department.

With the goals of helping students avoid errors, and of learning how often students actually write the patterns Jguardrail checks for, we set up our system so that Jguardrail was automatically invoked when the `javac` command was run. This was done by creating a shell script in `/usr/local/bin` called `javac` which calls Jguardrail followed by the real `javac`, which is in the `/usr/bin` directory. Because `/usr/local/bin` comes first in a default `$PATH` variable, students got the Jguardrail warnings by default.

We also decided to write our script in such a way that if Jguardrail *did* find any warnings, it does not compile the code at all. This way, warnings must be fixed before the code can be run. This behavior is similar to the `-Werror` flag that the GCC compiler provides.

We instrumented Jguardrail so that each time a warning was issued, we saved a record of it so that we can collect usage statistics. This machine is used by students in many classes, but here we will look specifically at the 43 students in the two sections of the CS 2 class.

The student who triggered the fewest warnings triggered 19 of them. The student who triggered the most triggered 597. The mean was 124.3 and the median value was 93. This was across a semester's worth of work.

Table 1 contains data on how many times each of the warnings was triggered by the students in the CS 2 class. The warnings are listed in the same order as they are described in Section 2.

Table 1: Warnings issued by Jguardrail over one semester of a CS 2 class

Pattern	**Times Issued**
String Equality	125
Empty Structures	25
Missing Braces	218
Integer Division	0
Missing this	12
Local Shadowing	312
Void Constructor	65
Uninitialized Vars	4580
tostring	0
Missing Break	7

As can be seen, students are much more likely to trigger some of these warnings than others. The integer division and mis-capitalized `toString` method

were never triggered at all. The integer division one was actually triggered by a more advanced student using the server, but not in the CS 2 class population.

The use of uninitialized instance variables is a clear outlier. It is possible that Jguardrail is a little over-zealous in its enforcement of the practice of initializing all instance variables. However it certainly leads to bugs in student programs and we believe being a little extra rigorous is not a bad thing at this level. Additionally Java already ensures that all local variables are initialized before being used, so enforcing this for all variables provides a consistent rule for students.

We believe that the use of Jguardrail in this class did make a difference in the amount of bugs in student code. Even setting aside the use of uninitialized instance variables, many of the other warnings were issued a significant number of times and can cause difficult bugs if not addressed. It also made helping students who encounter these issues easier on the instructor. For example in the case of shadowing a local variable, we can explain the warning to them and how to fix it, rather than need to help them debug why their variables are not being updated as they expect.

The tool did cause some issues for students, however. Code in the textbook used for the class occasionally violates the rules we enforce, which can be frustrating for a student who may not understand why code given to them by the text may fail to compile. Jguardrail is rather an "opinionated" tool in that it flags things as errors which may be considered subjective style decisions. It raises the question of what is considered "good code" which different instructors will certainly have different opinions on. However we do believe it served to help students avoid bugs in their programs.

5 Comparison to Other Tools

In this section we compare Jguardrail to other tools which report warnings on Java code. We compared the OpenJDK `javac` compiler itself, as well as the Eclipse[8], NetBeans[7] and IntelliJ[11] IDEs. Interestingly these IDEs each provide more warnings than the compiler itself provides. We also tried the Sonar Lint[17] linting tool plugin for IntelliJ, which provides more warnings on top of IntelliJ itself. We also tried the SpotBugs[18] plugin for IntelliJ, but it did not provide any more warnings than IntelliJ itself did so it is not present in the results.

Table 2 shows which of the checks that Jguardrail performs are caught by these tools. Again, these checks are listed in the same order as they are described in Section 2 of this paper.

Table 2: Comparison of tools which offer warnings for Java programs

Pattern	**javac**	**Eclipse**	**NetBeans**	**IntelliJ**	**Sonar**
String Equality			✓	✓	–
Empty Structures			✓	✓	–
Missing Braces					
Integer Division				✓	–
Missing this		✓	✓	✓	–
Local Shadowing		✓[1]	✓		✓
Void Constructor		✓		✓[2]	–
Uninitialized Vars				✓	–
tostring					✓
Missing Break	✓[3]				✓

[1] Eclipse gives a warning for the code we used to test local variables shadowing instance variables, but the warning was that the local variable was never used. This warning does not clearly indicate the underlying mistake.

[2] IntelliJ provides a warning for our test which says that the method (with the same name as the class) is never called. This indicates something is wrong, but is not especially clear as to the mistake being made.

[3] The javac compiler only provides warnings for missing break statements with the `-Xlint:all` flag passed to the compiler.

The – entries for Sonar are because that tool was used as a plugin for IntelliJ. So the cases where IntelliJ already provided warnings were still provided with the Sonar lint tool installed.

The IntelliJ IDE, especially with the Sonar lint tool installed, catches almost all of the cases Jguardrail catches. The Checkstyle tool [3] was also tried and performs similarly to Sonar. However, in our experience most students ignore the warnings given by IDEs such as IntelliJ for a few reasons. One, they are fairly inconspicuous and many students don't even really notice them. Two, these IDEs allow you to run the program if it compiles, whether or not warnings are given. Students will usually choose to run and test their code rather than read the warnings. Finally IDEs and style checkers use warnings to suggest refactoring programs in ways students may find confusing. For example, IntelliJ suggests that switch statements be replaced with switch expressions which students may not have seen.

Jguardrail is designed to be a code analysis framework specifically for beginning students. It does this by focussing on mistakes that we have observed beginning students making, making the warning messages clearer than some other tools provide, and making the warnings more visible. In our usage of the

tool, we went so far as to dis-allow running of the program until warnings are resolved.

Jguardrail is also a command-line tool. In an environment, such as ours, where we encourage students to use the command-line to write and compile programs, Jguardrail provides many more warnings than the `javac` compiler itself.

6 Future Work

There are several areas in which Jguardrail can be expanded. Of course more warnings can be added to the tool. We would like to issue warnings for code which is improperly indented. That way if the code structure the programmer sees does not match the structure the compiler sees, the student will be given a warning. This will be a little trickier to implement using ANTLR than the other analyses, but is possible.

We will also be making the tool more flexible by providing a means for users to determine which of the patterns should be considered errors (disallowing compilation), warnings (allowing compilation), or disabled entirely. This will allow instructors to determine which patterns they want to be highlighted as potential mistakes for their students.

We are also beginning work on an IDE plugin for Jguardrail, initially for IntelliJ. The goal here is not only to provide warnings for the things Jguardrail provides warnings for, but also to allow the option of making fixing warnings mandatory before programs can be run.

7 Conclusion

Jguardrail is a framework for identifying programming patterns which will, or are very likely to, lead to incorrect Java programs. It currently provides warnings for ten such patterns, which are not caught by the Java compiler itself. This tool can be extended to catch other such patterns.

The study we conducted shows that most of these are actually encountered in real student code at the CS 2 level. Because Jguardrail catches these mistakes at compile time, students should be in a better position to fix the underlying mistake instead of needing to debug the symptoms of the mistake. We can also use this data to inform our teaching, highlighting the issues that we see occurring the most for our students.

Overall we believe that the needs of novice programmers are different from those of experienced professionals. We have by and large moved away from using programming languages geared towards beginners (such as Pascal or Scheme) in favor of those used in industry. However, that doesn't mean we

need to use the same tools around those languages. We believe that it is possible to develop tooling that does a better job of catching and reporting mistakes that beginning students most commonly face, and Jguardrail is a step in that direction.

References

[1] Brett A. Becker et. al. "Effective compiler error message enhancement for novice programming students". In: *Computer Science Education* 26.2-3 (2016), pp. 148–175.

[2] Neil C.C. Brown and Amjad Altadmri. "Investigating novice programming mistakes: educator beliefs vs. student data". In: *Proceedings of the Tenth Annual Conference on International Computing Education Research*. ICER '14. 2014, pp. 43–50.

[3] CheckStyle. *CheckStyle*. `https://checkstyle.sourceforge.io/`.

[4] Google Corporation. *Google Java Style Guide*. `https://google.github.io/styleguide/javaguide.html`.

[5] Tom Duff. *Explanation, please!* `https://www.lysator.liu.se/c/duffs-device.html`.

[6] Ian Finlayson. *Jguardrail Code Repository*. `https://github.com/IanFinlayson/jguardrail`.

[7] Apache Software Foundation. *NetBeans Homepage*. `https://netbeans.apache.org/front/main/index.html`.

[8] Eclipse Foundation. *Eclipse Homepage*. `https://eclipseide.org/`.

[9] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st ed. Addison-Wesley Professional, 1994.

[10] James Jackson, Michael Cobb, and Curtis Carver. "Identifying top Java errors for novice programmers". In: *Proceedings frontiers in education 35th annual conference*. IEEE. 2005, T4C–T4C.

[11] JetBrains. *IntelliJ Homepage*. `https://www.jetbrains.com/idea/`.

[12] Yoshitaka Kojima, Yoshitaka Arahori, and Katsuhiko Gondow. "Investigating the difficulty of commercial-level compiler warning messages for novice programmers". In: *International Conference on Computer Supported Education*. Vol. 2. SCITEPRESS. 2015, pp. 483–490.

[13] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. "Compiler error messages: what can help novices?" In: *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*. SIGCSE '08. 2008, pp. 168–172.

[14] Terence Parr. *The definitive ANTLR 4 reference*. The Pragmatic Bookshelf, 2013, pp. 1–326.

[15] James Prather et al. "On Novices' Interaction with Compiler Error Messages: A Human Factors Approach". In: *Proceedings of the 2017 ACM Conference on International Computing Education Research*. ICER '17. 2017, pp. 74–82.

[16] Robert M. Siegfried et al. "Trends Of Commonly Used Programming Languages in CS1 And CS2 Learning". In: *2021 16th International Conference on Computer Science and Education (ICCSE)*. 2021, pp. 407–412. DOI: 10.1109/ICCSE51940.2021.9569444.

[17] Sonar. *Sonar Lint Homepage*. https://www.sonarsource.com/products/sonarlint/.

[18] SpotBugs. *SpotBugs Homepage*. https://spotbugs.github.io/.