Technical Report CS2016-08

Empirical Study of Tools to Assist Java Programmers in Finding Bugs

Andreas Bach Landgrebe

Submitted to the Faculty of
The Department of Computer Science

Project Director: Professor John Wenskovitch
Second Reader: Dr. Gregory M. Kapfhammer

Allegheny College
2016

I hereby recognize and pledge to fulfill my responsibilities as defined in the Honor Code, and to maintain the integrity of both myself and the college community as a whole.

Andreas Bach Landgrebe

ANDREAS BACH LANDGREBE. Empirical Study of Tools to Assist Java Programmers in Finding Bugs.
(Under the direction of Professor John Wenskovitch.)

ABSTRACT

As a programmer, it is inevitable to encounter logic based bugs and it is imperative to be able to fix the issues. One approach to this situation is to use tools that helps detect logic bugs. Tools can assist programmers to find logic errors so they can spend time on different tasks in their projects. This paper presents an empirical study to examine three tools FindBugs, PMD, and Checkstyle used to find logic based bugs in Java programs.

Acknowledgment

I like to acknowledge:

Prof. John Wenskovitch, for continuing advice and support, and for relentlessly forcing me to do my best.

Dr. Gregory M. Kapfhammer, for continuing advice and support as a mentor and a professor.

Dr. Robert Cupper, for starting me on the path I follow today.

Every other member of the Allegheny faculty, for unparalleled excellence throughout my education.

My colleagues, for advice, support, and patience throughout my Allegheny career.

To all of the students who took time out of their busy schedule to participate in my empirical study

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

When programmers write code in Java programming language, there is a high chance that some of the programs do not run as expected. There are many options to fix this issue. One approach is to look back into the source code and read it line-by-line and try to find out what the issue is. This approach is time consuming and tiresome and may not always work. An alternative approach is to use a tool such as FindBugs, PMD, or Checkstyle to help finding the issue.

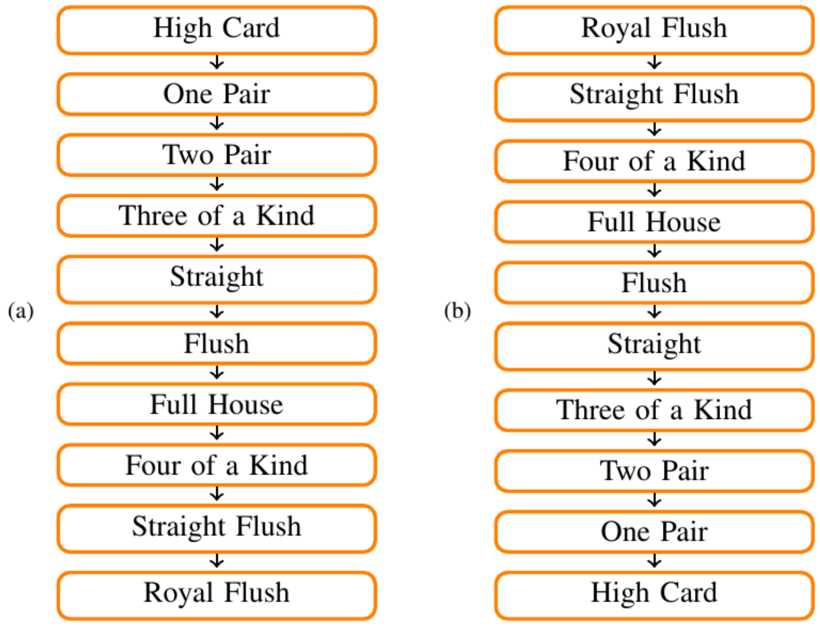


Figure 1.1: The Figure shows the representation of poker hands - (a) starts with the lowest hand (b) starts with the highest hand. The highest hand is the winner of the game, and therefore the program code should use the representation (b) to decide the winner.

For example, let us say we are writing a program in Java that will decide the winner of a poker game. Figure 1.1 shows the different poker hands that a player can have. Figure 1.1 (a) shows the sequence of hands starting with the lowest hand 'high card' and ending with the highest hand 'royal flush', where figure 1.1 (b) starts with the highest hand 'royal flush' and ends with the lowest hand 'high card'. The program is written so it looks at poker hands starting at the beginning of the representation and continues to go through the poker hands in the representation until a match between the poker hand and the representation is found. If there is a match the program will select the hand as the highest hand. Using the representation in Figure 1.1 (a) the program will start with the 'high card'. A poker hand always has a highest card so it will stop here and select the highest card as the poker hand. This is not correct because there could also be 'one pair', 'two pair', 'three of a kind' or any of the other hands listed in the representation. Using this representation will not necessarily find the highest poker hand. Using the representation in Figure 1.1 (b) the program will go through the representation starting with the highest hand, and select the first match, which will give the highest hand and this is correct.

Figure 1.1: The Figure shows the representation of poker hands - (a) starts with the lowest hand and (b) starts with the highest hand. The highest hand is the winner of the game, and therefore the program code should use the representation (b) to decide the winner.

1.2 Current State of the Art

At present, the process of locating logic based bugs is mostly manual. However, there are tools to assist in finding logic based bugs. Many of these tools have been developed using techniques such as syntactic pattern matching, data flow analysis, type systems, model checking, and theorem proving [37].

1.3 Definitions

logic based bug is defined as a bug in a program that causes it to operate incorrectly. The program will not terminate abnormally or crash. This is considered to be fault of commission. An example of a logic based bug is comparing String objects with the == operator compared to the .equals() method. When it comes to objects, when using the == operator, it is testing for references equality. This means that it is checking if the two are the same object. The .equals() method tests for value equality. This means that it is testing if the two objects are logically "equal".

Source code analysis is defined as: "Source code analysis is the process of extracting information about a program from its source code or artifacts generated from the source code using automatic tools" [11]. The idea of source code analysis is imperative in order to figure out how programs works. Source code analysis is also used to locate bugs in programs.

Static program analysis of software is defined as analysis that is completed without actually executing programs and is usually performed as part of a Code Review. The analysis attempts to highlight possible errors within the source code. The three tools analyzed in this paper FindBugs, PMD, and Checkstyle are examples of static program analysis. These tools intend to identify errors in Java code without running the program.

Dynamic Program Analysis is defined as the analysis of computer software that is performed by executing programs on a real or virtual processor. Tools using Dynamic Program Analysis are also known as program monitors because they watch and report the program's behavior.

Fault of omission is when some key aspect of the code is missing. For example, when a variable is not initialized.

Fault of commission is when there are one or more incorrect assignments in a program. For example, when a variable is initialized to the wrong value.

Compiler bug is defined as a bug that is detected by the compiler when compiling the source code and it prevents it from being executable. An example of a compiler bug is writing a line of source code without a semi colon (;) at the end of the statement.

Run-time bug is defined as a bug that occurs during the execution of a program. In contrast to logic based bugs, a run-time bug will terminate or crash the program.

False Positive is defined as a condition that is incorrectly reported as an error. For example the tools FindBugs, PMD, or Checkstyle could report a false positive if they scan through source code and highlight lines in the code that actually do not contain any bug.

False Negative is defined as a condition where a test result improperly indicates no presence of an error. For example tools FindBugs, PMD, or Checkstyle could report a false negative if they scan through source code and fail to highlight lines of the code that actually hold bugs.

Cyclomatic Complexity is defined as a measurement used to indicate the complexity of a program. This metric calculates the number of linearly independent paths through a program's source code. The tool PMD is able to measure the complexity of a program by calculating the number of linearly independent paths in a program's source code.

Inefficient Code is when complex code can be re-written to be simpler. An example of inefficient code in Java is using the String.trim().length() to check if a string is empty. This approach creates a new String object just to check its size. Instead a static function that loops through the string should be used as this would be a much more efficient way to check if a string is empty.

Bug checker is defined as a static analysis tool used to find code that does not conform to specific properties and which may cause the program to misbehave at runtime [37]. An example of a bug checker is the tool FindBugs.

Style checkers examine code to determine if it contains non-conformance to particular coding style rules [37]. Enforcing style checkers create a consistent style throughout programs and projects. This makes it easier for developers to understand and maintain code for a particular project. Two examples of style checkers are PMD and Checkstyle.

1.4 Goals of the Project

There are two significant goals for this project. The first goal is to perform the empirical study of the tools and this goal is a prerequisite for the second goal. The second goal is to determine whether new tools should be developed or additional features could be added to one or more of the existing tools. If none of the three tools FindBugs, PMD, and Checkstyle are successful in assisting the programmer in finding logic based bugs then it may be recommended that a new tool should be developed. However, if at least one of the three tools are able to successfully assist in locating the bugs, then the tool may be the future tool used to assist in such a tedious and time-consuming task.

1.5 Thesis Outline

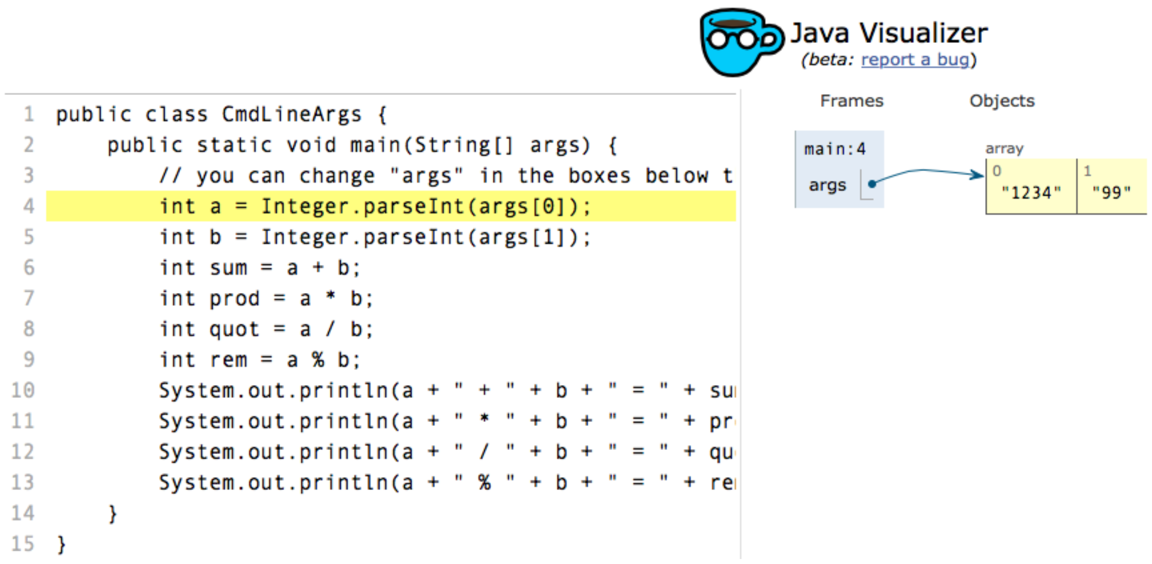
This paper consists of eight chapters which cover topics for my senior project. Chapter 2 will discuss the related research work that is on-going in the field of assisting in locating logic bugs. Chapter 3, 4, and 5 will describe the three tools FindBugs, PMD, and Checkstyle accordingly including their features and functionality. Chapter 6 outlines the methodology I have used for this project including how each participant conducted the study and descriptions of the buggy programs I have developed. In chapter 7 the results of this empirical study are presented and discussed and this is used to form the conclusion. Chapter 8 is the conclusion of my study and it also includes suggestions for further studies.

Chapter 2

Related Work

Automated and semi-automated analysis of source code has remained a topic of intense research for more than 30 years [11]. It helps programmers to work more efficiently and enforces rules to use the same programming patterns. This paper will explore and compare the effectiveness of some of these tools. In this chapter, some additional tools are mentioned

2.1 Java Visualizer



```
public class CmdLineArgs {
    public static void main(String[] args) {
        // you can change "args" in the boxes below t
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        int sum = a + b;
        int prod = a * b;
        int quot = a / b;
        int rem = a % b;
        System.out.println(a + " + " + b + " = " + su
        System.out.println(a + " * " + b + " = " + pr
        System.out.println(a + " / " + b + " = " + qu
        System.out.println(a + " % " + b + " = " + re
    }
}
```

Figure 2.1: Java Visualizer used to go through a Java program line by line

In Table 2.1, the Java Visualizer is used to go through a Java program line by line and provide a visualization of the program steps on the right hand side.

2.2 Jlint

Jlint is a static code analyzer that checks Java source code to attempt to find bugs. Throughout the code, it looks for inconsistencies and synchronization problems. It is done by performing data flow analysis and building a lock graph. Jlint is a simple but highly effective static analyzer that checks Java programs for several common errors, such as null pointer exceptions and overflow errors [6]. The original version Jlint1 was just a Java program checker that could check packages in an automated manner. Jlint1 was extended to Jlint2 to fully support synchronization. [5].

2.3 Bandera

Bandera is an integrated collection of program analysis and transformation components [17]. Bandera takes Java source code and outputs a program model in the input language for several existing verification tools [17].

Bandera evaluates the model by checking the Java source code. It provides tool to support definition and managing collections of requirements [18].

2.4 ESC/Java2

ESC/Java2 (Extended Static Checker for Java) is the main (extended) static checker for the JML (Java Modeling Language) [14]. The purpose of the extension is to include extra constructs for specifying an object-oriented module such as frame properties, data groups, and ghost and model fields [14].

At first, it was an experimental compile-time program checker that found common programming errors. The checker is powered by verification-condition generation and automatic theorem-proving techniques [20] [21].

2.5 Summary of Related Work

There are a number of tools that assist programmers in working more effectively. Three of these have been researched in this paper, but apart from these there are many other tools. The tools that are presented in this paper may not detect a possible bug (false negative) or may highlight a point in the source code that may not be the bug (false positive). My proposed study is to evaluate three tools for four Java programs that have been written to include a bug. This will provide Java programmers information about tools to assist in detecting logic based bugs.

Chapter 3

FindBugs

FindBugs is a free and open source program that allows Java programmers to get assistance in locating bugs in Java source code. It has been developed at the University of Maryland and was initially released in 2006. The version that has been used in this project is FindBugs release 3.0.1.

This tool is considered to be a bug checker tool. The reason for this is that its primary job is to assist programmers in finding bugs in their code.

All of the bug pattern detectors are implemented using BCEL [4]. BCEL is an open source bytecode analysis and instrumentation library [25]. FindBugs is further developed constantly. There have been several front end implementation added onto generating FindBugs. Some of these include XML that reports possible bug findings and plugins for several Integrated Development Environments (IDE). Some of the IDEs that have been contributed as front ends include the Eclipse IDE which was used for this empirical study, and IntelliJ IDEA. Another front end implementation that was added onto FindBugs include a task for running FindBugs from the Apache Ant Build tool.

Since FindBugs 2.0, there have been some major new features. One of these new features is the idea of bug rank which means ranking according to detected severity of a bug. This feature gives possible bugs a rank from 1 - 20. These possible bugs are grouped into categories. Rank 1-4 is considered to be the scariest. Rank 5-9 is considered to be scary. Rank 10-14 is considered to be troubling. Rank 15-20 is considered to be of concern.

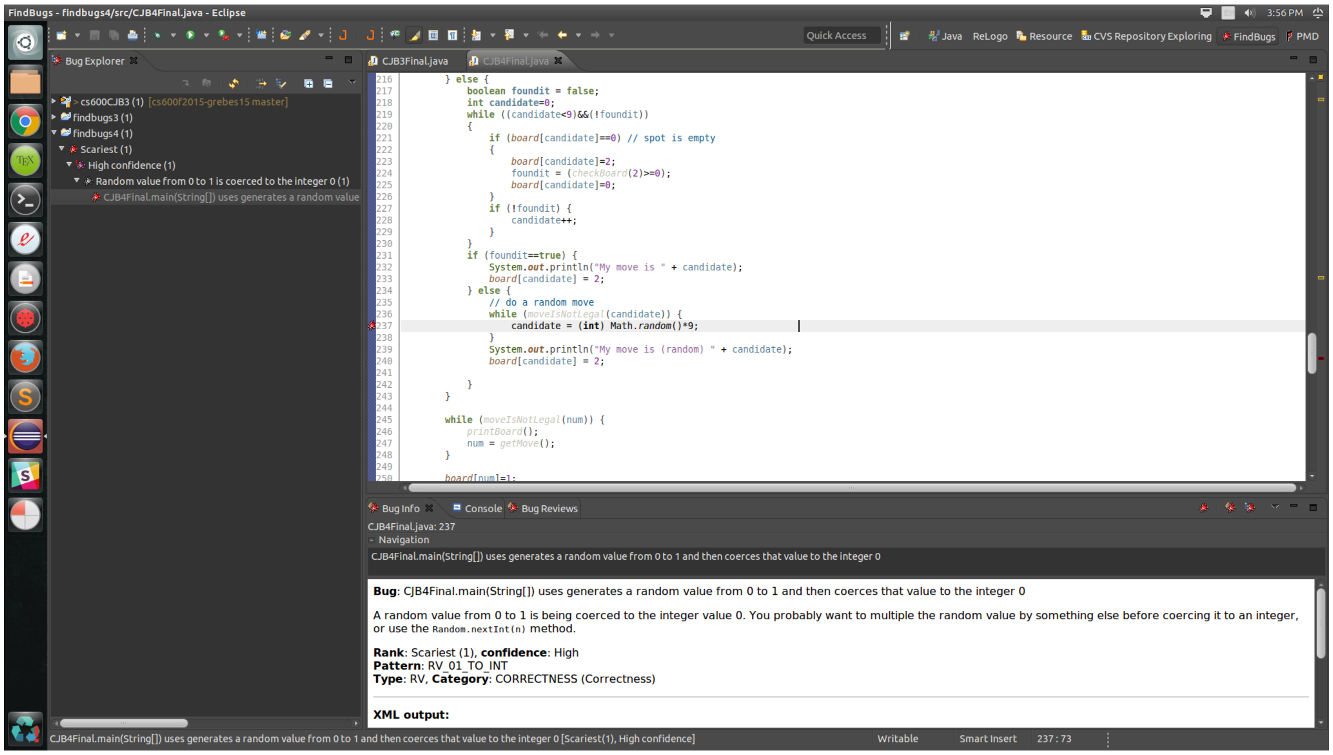
FindBugs uses a series of ad-hoc techniques designed to balance precision, efficiency, and usability [37]. One of FindBugs approaches is to match source code to "well known" bad or suspicious programming practices. In some cases, FindBugs also uses data flow analysis to check for bugs.

Internally FindBugs analyze the generated Java bytecode for bug patterns. This means that the code has to be compiled prior to FindBugs doing its analysis.

The current FindBugs version categorizes bugs in 9 categories: Bad practice (88), Correctness (149), Dodgy code (79), Experimental (3), Internationalization (2), Malicious code vulnerability (17), Multithreaded correctness (46), Performance (29), and Security (11). There are 424 checks in total. The number in parenthesis is the current number of checks in that category. Some of these categories contains checks for are actual bugs like IL_INFINITE_LOOP: "This loop does not seem to have a way to terminate (other than by perhaps throwing an exception)," whereas other categories merely identifies coding styling issues as for example NM_FIELD_NAMING_CONVENTION: "Names of fields that are not final should be in mixed case with a lowercase first letter and the first letters of subsequent words capitalized". Both examples are taken from the Correctness category.

FindBugs is highly configurable. FindBugs can be expanded by writing custom bug detectors for Java [37]. If the programmers are familiar with Java bytecode they can write a new FindBugs detector in as little as a few minutes.

Figure 3.1 is a screenshot taken that shows the FindBugs tool being used in the empirical study. Inside of the Eclipse Integrated Development Environment, FindBugs points to a specific line of source code in the program that FindBugs finds


FindBugs - findbugs4/src/CJB4Final.java - Eclipse

Bug Explorer

- > cs600CJB3 (1) [cs600f2015-grebes15 master]
- findbugs3 (1)
- findbugs4 (1)
  - Scariest (1)
    - High confidence (1)
      - Random value from 0 to 1 is coerced to the integer 0 (1)
        - CJB4Final.main(String[]) uses generates a random value

CJB3Final.java | CJB4Final.java

```
216         } else {
217             boolean foundit = false;
218             int candidate=0;
219             while ((candidate<9)&&(!foundit))
220             {
221                 if (board[candidate]==0) // spot is empty
222                 {
223                     board[candidate]=2;
224                     foundit = (checkBoard(2)>=0);
225                     board[candidate]=0;
226                 }
227                 if (!foundit) {
228                     candidate++;
229                 }
230             }
231             if (foundit==true) {
232                 System.out.println("My move is " + candidate);
233                 board[candidate] = 2;
234             } else {
235                 // do a random move
236                 while (moveIsNotLegal(candidate)) {
237                     candidate = (int) Math.random()*9;
238                 }
239                 System.out.println("My move is (random) " + candidate);
240                 board[candidate] = 2;
241
242             }
243         }
244
245         while (moveIsNotLegal(num)) {
246             printBoard();
247             num = getMove();
248         }
249
250         board[num]=1;
```

Bug Info | Console | Bug Reviews

CJB4Final.java: 237

Navigation

CJB4Final.main(String[]) uses generates a random value from 0 to 1 and then coerces that value to the integer 0

Bug: CJB4Final.main(String[]) uses generates a random value from 0 to 1 and then coerces that value to the integer 0

A random value from 0 to 1 is being coerced to the integer value 0. You probably want to multiple the random value by something else before coercing it to an integer, or use the `Random.nextInt(n)` method.

Rank: Scariest (1), **confidence**: High
Pattern: RV_01_TO_INT
Type: RV, **Category**: CORRECTNESS (Correctness)

XML output:

CJB4Final.main(String[]) uses generates a random value from 0 to 1 and then coerces that value to the integer 0 [Scariest(1), High confidence]   Writable   Smart Insert   237 : 73

Figure 3.1: FindBugs used in the Eclipse environment

to possible be a bug. This is shown by the red bug on line 237.

Chapter 4

PMD

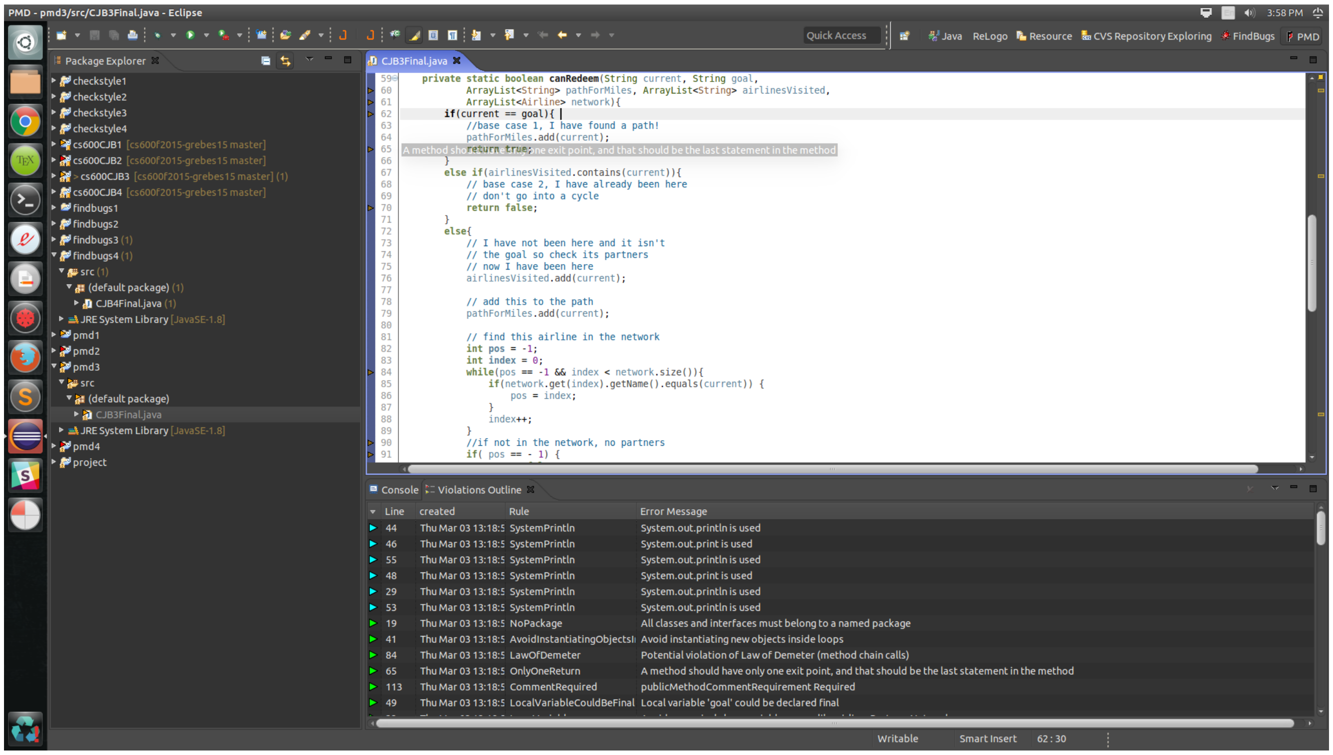
PMD is a static source code analyzer. This tool looks at Java source code and identifies bugs or potential anomalies. These anomalies include dead code, duplicated code or overcomplicated expressions [43].

The first version of PMD (version 0.1) was introduced in 2002. The version of PMD that was used in this project is 5.4.1.

PMD can check for more that plain java program issues. Its rules are divided into 5 sections: jsp, xsl, java, ecmascript, and xml. In the java section there are 276 different checks divided into 16 categories: Design (52), Coupling (10), Jakarta Commons Logging (3), Basic (23), Strict Exceptions (12), Security Code Guidelines (2), Java Logging (4), Android (3), Controversial (23), Comments (3), Type Resolution(4), Empty Code(11), String and StringBuffer (15), Code Size (11), Braces (4), Unused Code (5), Unnecessary (7), J2EE (9), JavaBeans (2), Migration (14), Import Statements (6), JUnit (12), Naming (20), Finalizer (6), Optimization(12), and Clone Implementation(3). The number in parenthesis marks the number of checks in that category. The rules range from styling checks like "ShortMethodName: Method names that are very short are not helpful to the reader" in the Naming category to rules for checking code design like "UseSingleton: For classes that only have static methods, consider making them Singletons. Note that this doesn't apply to abstract classes, since their subclasses may well include non-static methods. Also, this class is a Singleton, a private constructor should be added to prevent instantiation."

From the rule sets PMD uses, it is able to assist programmers in finding logic based bugs. Some of the categories that may be able to identity potential bugs include Controversial, Empty Code, and String and StringBuffer. One of the rules in the category, Controversial that may cause a logic based bugs is the Unnecessary-Parantheses. UnnecessaryParantheses is using the order of operation to calculate a certain math equation, and parantheses that are used may lead to an incorrect answer. In the category Empty Code, one ruleset that may lead to logic based bugs is EmptyCatchBlock. If the catch block is empty, then nothing will be done which could cause an incorrect calculation. In the category of String and StringBuffer, one ruleset that caused a logic based bug was the UseEqualsToCompareStrings. In this ruleset, if two string are compared in an if statement by using ==, this is not going to work correctly. This is because the == operator looks at what the string point to rather than the content of the strings. To fix this, the .equals() method should be used to compare two strings.

Comparing FindBugs and PMD, PMD has found to produce a larger amount of warnings FindBugs. Despite that more warnings in a piece of source code will be generated with PMD compared to FindBugs, this does not mean that FindBugs or PMD is the better tool. These two tools focus on two different aspects of software quality [25]. PMD focus on looking at a coding style. FindBugs focuses on helping to uncover errors while ignoring style issues. Since these two tools focus on two different aspects of software quality, these two tools are complements of each other and not substitutes.


PMD - pmd3/src/CJB3Final.java - Eclipse

Package Explorer

- checkstyle1
- checkstyle2
- checkstyle3
- checkstyle4
- cs600CJB1 [cs600f2015-grebes15 master]
- cs600CJB2 [cs600f2015-grebes15 master]
- > cs600CJB3 [cs600f2015-grebes15 master] (1)
- cs600CJB4 [cs600f2015-grebes15 master]
- findbugs1
- findbugs2
- findbugs3 (1)
- findbugs4 (1)
  - src (1)
    - (default package) (1)
      - CJB4Final.java (1)
  - JRE System Library [JavaSE-1.8]
- pmd1
- pmd2
- pmd3
  - src
    - (default package)
      - CJB3Final.java
  - JRE System Library [JavaSE-1.8]
- pmd4
- project

CJB3Final.java

```java
    private static boolean canRedeem(String current, String goal,
            ArrayList<String> pathForMiles, ArrayList<String> airlinesVisited,
            ArrayList<Airline> network){
        if(current == goal){
            //base case 1, I have found a path!
            pathForMiles.add(current);
            return true;
        }
        else if(airlinesVisited.contains(current)){
            // base case 2, I have already been here
            // don't go into a cycle
            return false;
        }
        else{
            // I have not been here and it isn't
            // the goal so check its partners
            // now I have been here
            airlinesVisited.add(current);

            // add this to the path
            pathForMiles.add(current);

            // find this airline in the network
            int pos = -1;
            int index = 0;
            while(pos == -1 && index < network.size()){
                if(network.get(index).getName().equals(current)) {
                    pos = index;
                }
                index++;
            }
            //if not in the network, no partners
            if( pos == - 1) {
```

A method should have only one exit point, and that should be the last statement in the method

Console | Violations Outline

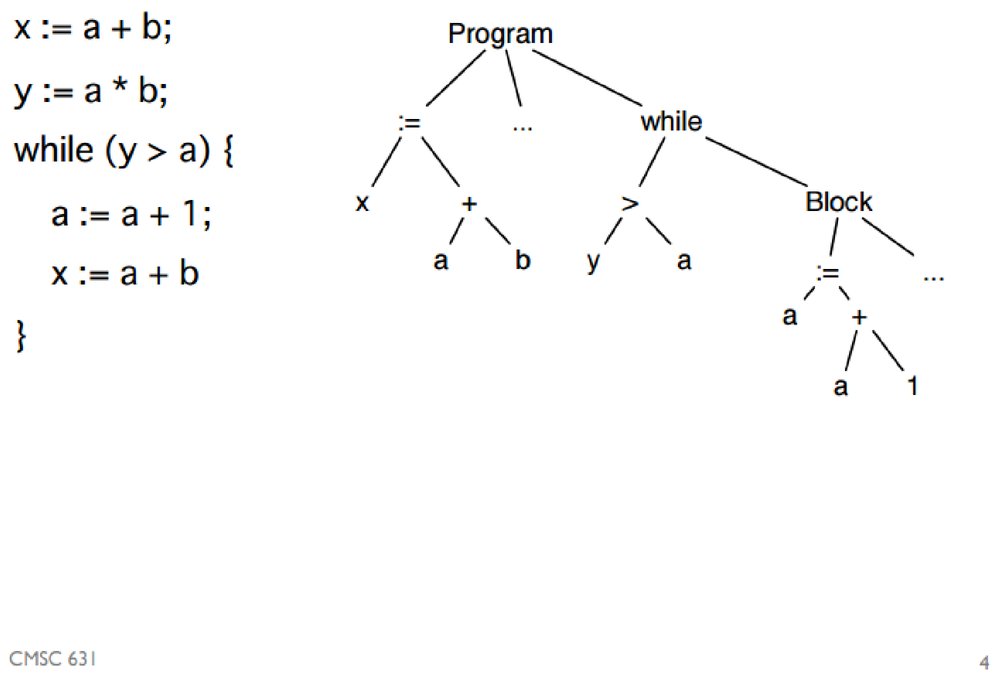
Line	created	Rule	Error Message
44	Thu Mar 03 13:18:5	SystemPrintln	System.out.println is used
46	Thu Mar 03 13:18:5	SystemPrintln	System.out.print is used
55	Thu Mar 03 13:18:5	SystemPrintln	System.out.println is used
48	Thu Mar 03 13:18:5	SystemPrintln	System.out.print is used
29	Thu Mar 03 13:18:5	SystemPrintln	System.out.println is used
53	Thu Mar 03 13:18:5	SystemPrintln	System.out.println is used
19	Thu Mar 03 13:18:5	NoPackage	All classes and interfaces must belong to a named package
41	Thu Mar 03 13:18:5	AvoidInstantiatingObjectsI	Avoid instantiating new objects inside loops
84	Thu Mar 03 13:18:5	LawOfDemeter	Potential violation of Law of Demeter (method chain calls)
65	Thu Mar 03 13:18:5	OnlyOneReturn	A method should have only one exit point, and that should be the last statement in the method
113	Thu Mar 03 13:18:5	CommentRequired	publicMethodCommentRequirement Required
49	Thu Mar 03 13:18:5	LocalVariableCouldBeFinal	Local variable 'goal' could be declared final

Figure 4.1: Example of PMD used in the Eclipse Integrated Development Environment

Figure 4.1 is a screenshot taken that shows the PMD tool used in the empirical study. Inside of the Eclipse Integrated Development Environment, PMD highlights multiple lines of source code and displays a warning. All of the highlighted lines can be seen on the left side of the line numbers.

4.1 AST - Abstract Syntax Tree

An Abstract Syntax Tree is an tree representation of the source code. This tree can be viewed as a structured document - just like XML. Since it is conceptually similar to XML, it can be queried with XPath to find a pattern [3]. By having the source code represented as a tree means the checks can be made at isolated nodes in the tree, irrespective of branches higher up and below. The Abstract Syntax Tree shows the structure of the code - the blocks and statements that are contained within each element.



```
x := a + b;
y := a * b;
while (y > a) {
   a := a + 1;
   x := a + b
}
```

Figure 4.2: Example of an Abstract Syntax Tree that both PMD and Checkstyle use. [22].

Chapter 5

Checkstyle

Checkstyle is a static analysis tool used for checking Java source code with sets of coding rules. It was originally released in 2001.

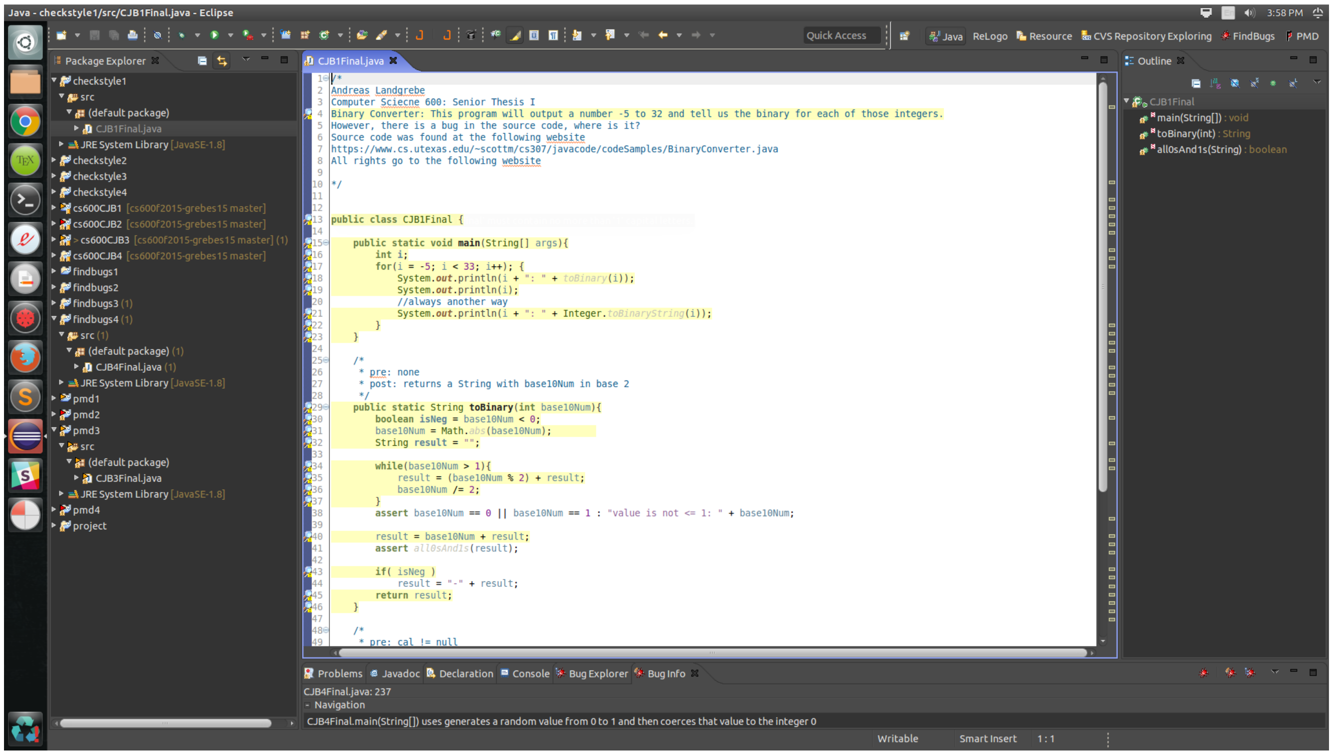
Checkstyle is a development tools to help programmer write Java code that adheres to a coding standard [32]. This is also a static code analysis tool used in software development such as FindBugs and PMD. Checkstyle checks at the Java source code level to see if it complies with coding rules set upon.

The version of Checkstyle that is used in the current experiment is 6.17.

Checkstyle contains a number of individual checks that can be performed on the source code. The current version has 153 checks grouped into 14 Categories: Annotations (7), Block Checks(6), Class Design(9), Coding(43), Headers(2), Imports(8), Javadoc Comments(12), Metrics(6), Miscellaneous(15), Modifiers(2), Naming Conventions(15), Regex(5), Size Violations(8), and Whitespace(15). The number in the parenthesis is the number of checks for each category. Many of the checks in Checkstyle relates to program style issues. In the current study the focus is on the coding checks.

In the category, Coding, the checks that identifies bad programming style often results in errors. Examples of these checks are:

- DefaultComesLast: Check that the default is after all the cases in a switch



```
/*
Andreas Landgrebe
Computer Sciecne 600: Senior Thesis I
Binary Converter: This program will output a number -5 to 32 and tell us the binary for each of those integers.
However, there is a bug in the source code, where is it?
Source code was found at the following website
https://www.cs.utexas.edu/~scottm/cs307/javacode/codeSamples/BinaryConverter.java
All rights go to the following website

*/


public class CJB1Final {

    public static void main(String[] args){
        int i;
        for(i = -5; i < 33; i++); {
            System.out.println(i + ": " + toBinary(i));
            System.out.println(i);
            //always another way
            System.out.println(i + ": " + Integer.toBinaryString(i));
        }
    }

    /*
     * pre: none
     * post: returns a String with base10Num in base 2
     */
    public static String toBinary(int base10Num){
        boolean isNeg = base10Num < 0;
        base10Num = Math.abs(base10Num);
        String result = "";

        while(base10Num > 1){
            result = (base10Num % 2) + result;
            base10Num /= 2;
        }
        assert base10Num == 0 || base10Num == 1 : "value is not <= 1: " + base10Num;

        result = base10Num + result;
        assert all0sAnd1s(result);

        if( isNeg )
            result = "-" + result;
        return result;
    }

    /*
     * pre: cal != null
```

```
CJB4Final.java: 237
- Navigation
CJB4Final.main(String[]) uses generates a random value from 0 to 1 and then coerces that value to the integer 0
```

Figure 5.1: Example of Checkstyle being used in the Eclipse Integrated Development Environment

statement.

- EmptyStatement: Detects empty statements (standalone “;” semicolon).

- FallThrough: Checks for missing break, return, throw or continue in a switch statement.

- InnerAssignment: Checks for assignments in a subexpression like String s = Integer.toString(i = 2);. This is considered bad coding style and sometimes is because the programmers wants to do a comparison (==).

- MissingSwitchDefault: Checks that there is always a default - line in a switch block.

- ModifiedControlVariable: Checks if a for loop variable is modified inside the block that is being executed - like for(int i = 0; i <= 5; i++) { ... i = 3; }

A check is a separate Java Class that is being executed when Checkstyle goes through the source file. Just as PMD, Checkstyle uses internally an Abstract Syntax Tree representation of the source code as shown in Chapter 4.1. This module approach makes it easy to implement new and custom designed checks. This feature has not been examined in this study.

Checkstyle can either be run as a plugin in an IDE or generate reports that summarized the “rules violated” on a project basis. In this experiment I will utilize the integration with the Eclipse IDE.

Chapter 6

Methodology

The next step for this senior project was to conduct an empirical study. Participants in this study have been selected among students who have had the class of Computer Science 112 or more advanced classes at Allegheny College. These students will in this paper be referred to as participants.

Conducting the empirical study help me to determine how effective and useful the tools are. The results will tell me if participants were able to use the tools to assist them in finding logic based bugs in their code or if they were better off reading and analyzing the source code line by line to find the bugs.

Ideally, the sample size should be at least 20 participants in this study. However, for practical reasons it has only been possible for me to get six students to participate and respond to my study. Therefore the results from my study cannot form the basis for a strong conclusion but rather give indications of the usefulness of the tools.

I have developed four Java programs and each program has a bug in its code. These four programs are used in this study. All participants used each of the three tools FindBugs, PMD, and Checkstyle to see if the tools could assist them in finding the bug in each of the four Java programs. In order to get the broadest feed back for the tools, it is essential that each participant get the chance to use each of the tools at least once. A participant may have to use one of the tools consecutively, but as long as each participant has used each of the tools at the end of the study, then the study has been properly completed. I have ensured that each participant used each of the tools at least once.

There were preparations I needed to make for the empirical study to be conducted smoothly. Participants used the Eclipse Integrated Development Environment (Eclipse IDE) for the study. The Eclipse IDE that was provided to the participants had already the three tools FindBugs, PMD and Checkstyle installed. The rule sets in the tools are highly configurable and I had configured the rule sets to the default settings in all three tools - ready for the participants to use.

I also provided instructions to participants on how to use the tools.

In order to successfully perform this empirical study, I needed to get the approval of the Institutional Review Board (IRB). I was successful in getting the approval for my study.

Figure 6.1 below displays how each buggy Java program was analyzed. In this study each participant was provided with one of the four buggy Java programs. The order of the Java programs in which the participants performed the analysis was determined by me to prevent any learning effect - meaning that the participants got experience and got better using the tools as they worked through this study. Then the participants were provided with one of the three tools FindBugs, PMD or Checkstyle - again the tool was determined by me. After the participant had used the first tool to analyze the source code, there were two possibly outcomes. The tool could either have assisted the participant in finding the bug or the bug was not found. If the tool assisted in locating the bug, then the analysis of that Java program was completed and the participant continued by moving on to analyze the next buggy Java program. If the tool was not able to assist the programmer to locate the bug, I determined the next tool for the participant to use. If the second tool was successful in assisting

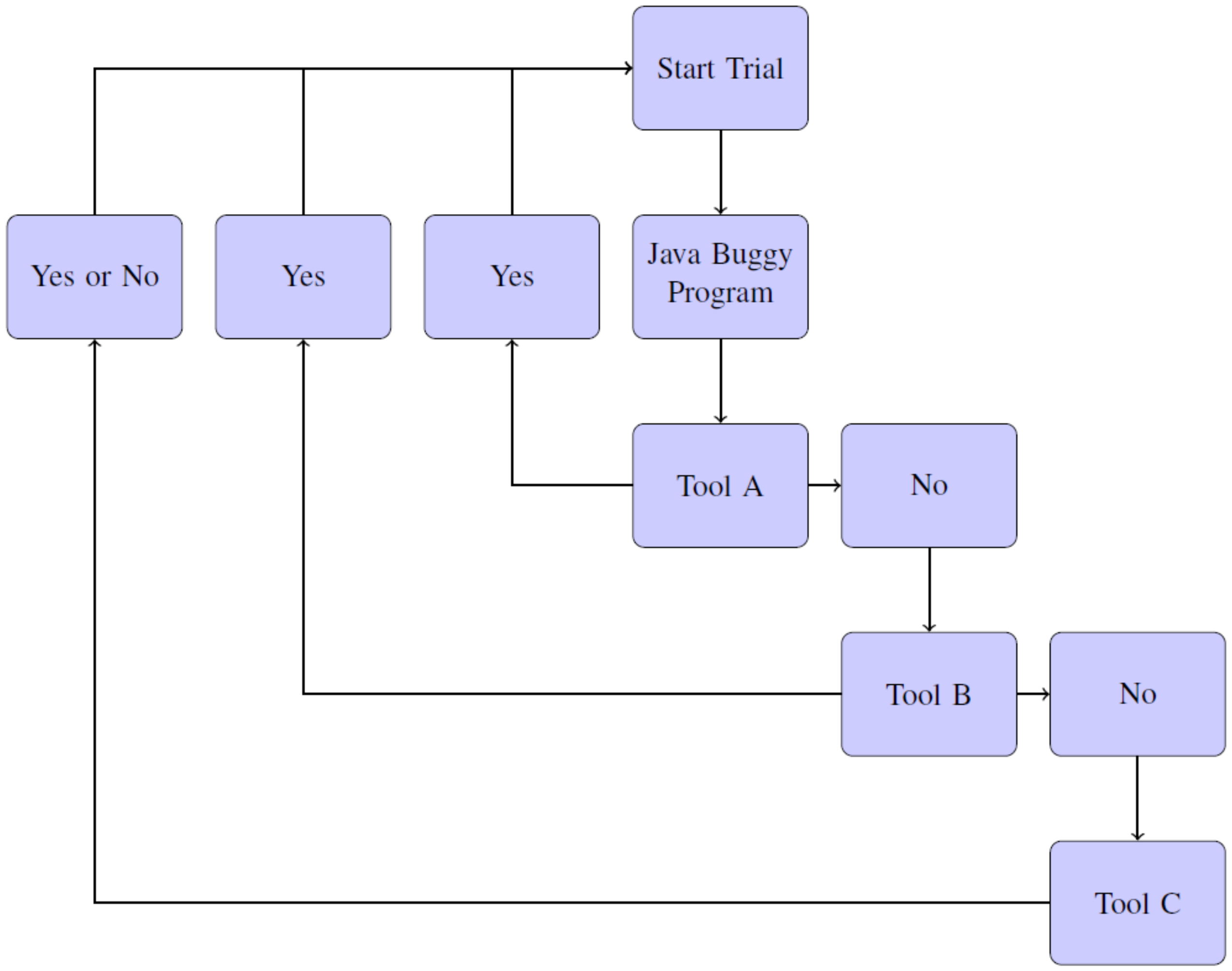


Figure 6.1: The process of how this empirical study was conducted.

locating the bug, the participant had completed analyzing the program and if not the third tool was provided to the participant.

This process was repeated for each of the four buggy programs. Each participant had one hour to complete their tasks of analyzing the four programs.

After the participants had completed the study, a survey was given to them. The survey asked the participants to answer the six questions listed below for each program:

- Did you find the bugs?
- If yes, did FindBugs assist you in finding these bugs?
- If yes, did PMD assist you in finding these bugs?
- If yes, did Checkstyle assist you in finding these bugs
- Would you use any of these tools in the future for finding logic based bugs?
- Suggestion to further improve these tools.

The responses from the survey told me results from the study and were used to draw the conclusion.

I have developed four Java programs each with a bug in its code. The four programs are intended to represent some of the most common mistakes made by Java programmers. I have chosen the bugs based on informal conversations with common students and professors and by searching the Internet.

The first Java program that was used in this study should convert the numbers ranging from -5 to 32 into the binary representation. The program has a 'for loop' that contains an iteration of the numbers from -5 to 32. However, when participants ran the program, they learned that the program would output the binary representation of the number 33. It is because there is a bug with the 'for loop' in the code. Having a semi-colon right before the program block, will cause the 'for loop' to continue to iterate through the loop without going to the code inside the block. The semicolon was misplaced, as it should not be there at all.

The second Java program calculated the velocity and velocity squared by getting the kinetic and mass as input. As long as mass is not equal to zero the program should run properly, since division with zero is not possible. The program will print out 4 lines. The first and second line will display the kinetic and mass given as input. The third line will display the calculated velocity squared and the last line will display the calculated velocity. The output for velocity squared is incorrect because the equation used in the calculation is incorrect. In the source code the equation is as follows

```
velocity_squared = 3 * (kinetic / mass);
```

This equation is incorrect. The equation in the Java source code to calculate velocity squared correctly should have been the following:

```
velocity_squared = 2 * (kinetic / mass);
```

The bug in this program is the incorrect number used to calculate the velocity squared.

The third Java program takes a text file as input. The file includes a list of airlines that are collaborating so miles can be transferred from airline to another. For example, miles can be transferred from Air Canada to Ocean Air. However, that is not possible when running the program. The bug is in the canRedeem() method. In the very first 'if' statement, when two strings of current and goal are compared, the '==' operator is used. This is incorrect and the .equals() method should be used instead. The '==' operator checks whether the references to the objects are equal, and the .equals() method checks for the actual contents of the string.

The last Java program was a program of the game tic-tac-toe. The game was set to a difficult level so the only outcome of the game was either the computer was winning or the game ended in a tie. When the program ran, the output did not display correctly. The bug was in the printBoard() method. In the printBoard() method a switch statement with 3 cases is used. Both Case 0 and Case 1 need a break statement. However, as can be seen in the source code, there is only a break statement after case 1.

Chapter 7

Discussion and Results

7.1 Discussion

The first Java program that was used for this study calculated the binary of a particular number. FindBugs did not assist any of the participants in finding this bug. Checkstyle was able to detect several warnings in the source code regarding indentation levels. However, it did not further assist programmers in finding this bug. PMD highlighted several lines of the code to give warnings. One of these warnings included the 'for loop' that did not have brackets around it. The semi-colon, may have caused PMD to give this warning and assisted participants to locate logic based bugs in the program.

The second program calculated the velocity squared incorrectly. FindBugs did not assist any of the participants in finding this bug. Checkstyle presented several warnings in the source code regarding indentation level. However, it did not further assist programmers in finding this bug. PMD highlighted several lines of code to give warnings. Unfortunately, it did not assist any of the participants in this study to find the logic based bugs.

The third program used the operator '==' instead of the method .equal(). FindBugs was not able to assist any of the participants in finding this bug. FindBugs detected an incorrect line of source code where an absolute path was declared, however it was not the issue and it was therefore a false-positive. Checkstyle detected several warnings in the source code regarding indentation levels. However, it did not further assist programmers in finding this bug. PMD highlighted several lines of code to give warnings. Fortunately, it highlighted the specific line of source code that contained the bug. On this specific line, PMD highlights and says to use the .equals() method to compare object references.

In the fourth program, a break statement was missing. FindBugs was not able to assist any of the participants in finding this bug. FindBugs detected an incorrect line in the source code where an absolute path was declared, however that was not the issue and was a false-positive. Checkstyle detected several warnings in the source code regarding indentation level. However, it did not further assist programmers in finding this bug. PMD highlighted several lines of code to give warnings. Fortunately, it highlighted the specific line of source code that contained the bug. On this specific line, PMD highlights and says to use a break statement in a switch case statement. While it appears that PMD seems to be the most efficient bug finding tool in my research, it should be stated that the other tools might prove their strengths in other situations, such as checking the style of how the code is written.

7.2 Results

Each program was ran by six participants. The tables in this chapter show the result from each participant when using the three tool, including if the three tools were able to assist the participant in finding the bug.

Participant Number	FindBugs	PMD	Checkstyle
1	No	Yes	No
2	N/A	N/A	No
3	No	Yes	No
4	N/A	No	No
5	No	N/A	No
6	N/A	Yes	N/A

Table 7.1: This is output from all of the participants from Program 1.

Participant Number	FindBugs	PMD	Checkstyle
1	No	No	No
2	No	N/A	N/A
3	N/A	N/A	No
4	No	No	N/A
5	No	No	No
6	No	No	No

Table 7.2: This is output from all of the participants from Program 2.

Participant Number	FindBugs	PMD	Checkstyle
1	N/A	Yes	N/A
2	N/A	Yes	No
3	No	Yes	N/A
4	No	Yes	No
5	No	Yes	N/A
6	N/A	Yes	N/A

Table 7.3: This is output from all of the participants from Program 3.

Participant Number	FindBugs	PMD	Checkstyle
1	N/A	N/A	Yes
2	N/A	Yes	N/A
3	N/A	N/A	No
4	N/A	Yes	N.A
5	N/A	N/A	No
6	N/A	Yes	No

Table 7.4: This is output from all of the participants from Program 4.

7.3 Summary

FindBugs did not assist the participants in finding any of the bugs in the programs, but highlighted several false positives.

Checkstyle only assisted one of the participants in finding the bug in one of the programs. When Checkstyle was ran for each program, almost every line of source code was highlighted. One of the reasons was the default naming convention. The default configuration of Checkstyle was that certain method names must contain no more than one capital letter. Another contributing factor was using the indentation level. Unlike a programming language such as Python, the source code in Java does not differ if the code is indented or not. If the Checkstyle tool is used for a Python program it would be beneficial in finding bugs because of incorrect indentation.

PMD was the most useful tool according to the surveys from the participants. The only program where PMD did not assist any of the participants was the second program. In fact, none of the tools were able to assist the programmers in finding the logic based bug in the second program.

According to the surveys, the end result indicated that the participants would use PMD in the future for finding logic based bugs.

There were multiple suggestions to improve PMD. Several of the participants also suggested improvements to Checkstyle. In the default configuration of Checkstyle, several of the highlighted lines were false positives. Participants suggested to remove most of these warnings since it was difficult to look at the source code when so many false positives were shown.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

From the empirical study performed in this senior project, there are indications that the PMD tool best assisted participants in finding logic based bugs. But in all fairness, this study has only scratched the surface of these tools functionality and each of the tools have at least 100 rules/checks that they can perform. FindBugs and Checkstyle do not find as many false positives as PMD and since all tools seemed to work pretty fast, the cost (time wise) of running all three together is small. All three tools are free, so there is neither a financial cost involved in using them. This suggests that all three tools should be used together. I assume it is easier to go through an eventual false positive rather than finding a bug going through the code line by line.

8.2 Future Work

To make the work done in this senior project more conclusive, the project can be extended. First, more participants could be added to get more reliable results.

Secondly, the number of programs with bugs could be increased to explore additional logic based bugs. In this project, we have investigated misplaced semi-colon in a for loop, incorrect formulas, incorrect object comparisons, and incorrect placement of break statements in a switch statement. Another logic based bug that could be examined is the order of operation or also known as a operator precedence. For example, if a Java program has two primitive data types of int a = 5 and int b = 8, and should calculate the average by using the following formula: (a + b) / 2. If the parenthesis are not included, the formula will be: a + b / 2. The two formulas calculate different results because of operator precedence, where the first formula is correct for calculating the average of a and b. In the last formula, division is evaluated before addition.

The three tools examined each has more than 100 different checks and they can be configured to perform the different checks. Different configurations of the tools could be explored including adding programmers own checks.

The project could also run over an extensive period of time, so participant's long term experience with the tools in an everyday situation could be examined.

Another consideration is to evaluate additional tools to explore if there are other tools that are more suited to detect logic based bugs than the tools included in this study.

Lastly, it could be considered to develop a new tool by combining the benefits of the three tools and at the same time minimize the amount of flaws in the tools.

Appendix A

Java Code

A.0.1 Common Java Bug 1

```
/*
Andreas Landgrebe
Computer Sciecne 600: Senior Thesis I
Binary Converter: This program will output a number -5 to 32 and
   tell us the binary for each of those integers.
However, there is a bug in the source code, where is it?
Source code was found at the following website
https://www.cs.utexas.edu/~scottm/cs307/javacode/codeSamples/
   BinaryConverter.java
All rights go to the following website

*/



public class CJB1Final {

    public static void main(String[] args){
      int i;
        for(i = -5; i < 33; i++); {
            System.out.println(i + ": " + toBinary(i));
            System.out.println(i);
            //always another way
            System.out.println(i + ": " + Integer.toBinaryString(i))
                ;
        }
    }

    /*
     * pre: none
     * post: returns a String with base10Num in base 2
     */
    public static String toBinary(int base10Num){
        boolean isNeg = base10Num < 0;
        base10Num = Math.abs(base10Num);
        String result = "";

        while(base10Num > 1){
            result = (base10Num % 2) + result;
            base10Num /= 2;
        }
        assert base10Num == 0 || base10Num == 1 : "value is not <=
            1: " + base10Num;

        result = base10Num + result;
        assert all0sAnd1s(result);

        if( isNeg )
            result = "-" + result;
        return result;
    }

    /*
```

```
     * pre: cal != null
     * post: return true if val consists only of characters 1 and 0,
          false otherwise
     */
    public static boolean all0sAnd1s(String val){
        assert val != null : "Failed precondition all0sAnd1s.
           parameter cannot be null";
        boolean all = true;
        int i = 0;
        char c;

        while(all && i < val.length()){
            c = val.charAt(i);
            all = c == '0' || c == '1';
            i++;
        }
        return all;
    }
}
```

A.0.2 Common Java Bug 2

```
/*
Andreas Landgrebe
Comptuer Science 600
Common Java Mistake 2
Kinetic Bug
The following source code was from a laboratory assignment in
   Computer Science 290.
The following source code was from a repository created by Dr.
   Gregory Kapfhammer.
```

```
All credit goes to Dr. Gregory Kapfhamnmer.
*/
import static java.lang.System.out;
import java.lang.Math;
import java.util.*;
import java.util.Scanner;


public class CJB2Final{



    public static String calculateVelocity(int kinetic, int mass){

    int velocity_squared = 0;
    int velocity = 0;
    StringBuffer final_velocity = new StringBuffer();
    if (mass != 0) {
      velocity_squared = 3 * (kinetic / mass);
      velocity = (int) Math.sqrt(velocity_squared);

      out.println("This is kinetic: " + kinetic);
      out.println("This is mass: " + mass);
      out.println("This is velocity squared: " + velocity_squared);
      out.println("This is velocity: " + velocity);
    }
    else {
      final_velocity.append("Undefined");
    }
    return final_velocity.toString();
  }

      public static void main(String[] args){
        Scanner scan = new Scanner(System.in);
        System.out.println("Please put in an integer for kinetic");
        int x = scan.nextInt();
        System.out.println("Please put in an integer for mass except
            for the number");
        int y = scan.nextInt();
        calculateVelocity(x, y);
  }
}
```

A.0.3 Common Java Bug 3 and airlines.txt

```
/*
Andreas Landgrebe
Computer Science 600
Common Java Bug 3
Airline Problem
This program takes in a text file and sees if one is able to
    transfer airline miles to a different airline.
This is great and all, but unfortunately there is a bug in the
    source code, hopefully one is able to find it.
All credit goes to the following website:
https://www.cs.utexas.edu/~scottm/cs307/javacode/codeSamples/
    AirlineProblem.java
*/



import java.util.ArrayList;
import java.util.Scanner;
import java.io.File;
import java.io.IOException;
import java.util.Arrays;

public class CJB3Final{

    public static void main(String[] args){
        Scanner scannerToReadAirlines = null;
        try{
            scannerToReadAirlines = new Scanner(new File("airlines.
               txt"));
        }
        catch(IOException e){
            System.out.println("Could not connect to file airlines.
               txt.");
            System.exit(0);
        }
        if(scannerToReadAirlines != null){
            ArrayList<Airline> airlinesPartnersNetwork = new
               ArrayList<Airline>();
            Airline newAirline;
            String lineFromFile;
            String[] airlineNames;

            while( scannerToReadAirlines.hasNext() ){
                lineFromFile = scannerToReadAirlines.nextLine();
                airlineNames = lineFromFile.split(",");
                newAirline = new Airline(airlineNames);
                airlinesPartnersNetwork.add( newAirline );
            }
            System.out.println(airlinesPartnersNetwork);
            Scanner keyboard = new Scanner(System.in);
            System.out.print("Enter airline miles are on: ");
            String start = keyboard.nextLine();
            System.out.print("Enter goal airline: ");
            String goal = keyboard.nextLine();
            ArrayList<String> pathForMiles = new ArrayList<String>()
               ;
            ArrayList<String> airlinesVisited = new ArrayList<String
               >();
            if( canRedeem(start, goal, pathForMiles, airlinesVisited
               , airlinesPartnersNetwork))
                System.out.println("Path to redeem miles: " +
                    pathForMiles);
            else
                System.out.println("Cannot convert miles from " +
                    start + " to " + goal + ".");
        }
    }

    private static boolean canRedeem(String current, String goal,
            ArrayList<String> pathForMiles, ArrayList<String>
                airlinesVisited,
            ArrayList<Airline> network){
        if(current == goal){
            //base case 1, I have found a path!
            pathForMiles.add(current);
            return true;
        }
        else if(airlinesVisited.contains(current))
            // base case 2, I have already been here
            // don't go into a cycle
            return false;
        else{
            // I have not been here and it isn't
            // the goal so check its partners
            // now I have been here
            airlinesVisited.add(current);

            // add this to the path
            pathForMiles.add(current);

            // find this airline in the network
            int pos = -1;
            int index = 0;
            while(pos == -1 && index < network.size()){
                if(network.get(index).getName().equals(current))
                    pos = index;
                index++;
            }
            //if not in the network, no partners
            if( pos == - 1)
                return false;

            // loop through partners
            index = 0;
            String[] partners = network.get(pos).getPartners();
            boolean foundPath = false;
            while( !foundPath && index < partners.length){
                foundPath = canRedeem(partners[index], goal,
                    pathForMiles, airlinesVisited, network);
                index++;
            }
            if( !foundPath )
                pathForMiles.remove( pathForMiles.size() - 1);
            return foundPath;
        }
    }
```

```
    private static class Airline{
        private String name;
        private ArrayList<String> partners;

        //pre: data != null, data.length > 0
        public Airline(String[] data){
            assert data != null && data.length > 0 : "Failed
               precondition";
            name = data[0];
            partners = new ArrayList<String>();
            for(int i = 1; i < data.length; i++)
                partners.add( data[i] );
        }

        public String[] getPartners(){
            return partners.toArray(new String[partners.size()]);
        }

        public boolean isPartner(String name){
            return partners.contains(name);
        }

        public String getName(){
            return name;
        }

        public String toString(){
            return name + ", partners: " + partners;
        }
    }
}
```

```
Delta,Air Canada,Aero Mexico,Ocean Air
United,Aria,Lufthansa,Ocean Air,Quantas,British Airways
Northwest,Air Alaska,BMI,Avolar,EVA Air
Canjet,Girjet
Air Canada,Areo Mexico,Delta,Air Alaska
Aero Mexico,Delta,Air Canda,British Airways
Ocean Air,Delta,United,Quantas,Avolar
Aria,United,Lufthansa
Lufthansa,United,Aria,EVA Air
Quantas,United,Ocean Air,AlohaAir
BMI,Northwest
Maxair,Southwest,Girjet
Girjet,Southwest,Canjet,Maxair
British Airways,United,Aero Mexico
Air Alaska,Northwest,Air Canada
Avolar,Northwest,Ocean Air
EVA Air,Northwest,Luftansa
Southwest,Girjet,Maxair
AlohaAir,Quantas
```

A.0.4 Common Java Bug 4

```
/*
Andreas Landgrebe
Computer Science 600
Common Java Bug 5
This is a game of Tic Tac Toe playing against the computer but wait
   there is an error
I wrote this program from scratch so I am not citing any websites.
*/
```

```
import java.util.Scanner;
import java.util.Random;
public class CJB4Final {


    static int [] board =  {0,0,0,0,0,0,0,0,0};


    public static int getMove() {
        int num=-1;
        while (moveIsNotLegal(num)){
            System.out.println();
            System.out.println("What is your move (X) 0-8?");

            Scanner scan = new Scanner(System.in);
            String str = scan.nextLine();
            num = Integer.parseInt(str);
            if (moveIsNotLegal(num)) {
                System.out.println("That move is not correct");
            }

        }

        return num;
    }

    public static void printBoard() {
        for (int row=0; row<3; row++){
            System.out.println("   |   |   ");

            for (int col=0; col<3; col++) {
```

```
            switch (board[row*3+col]) {
            case 0: System.out.print("   ");

            case 1: System.out.print(" X ");
            break;
            case 2: System.out.print(" O ");
            }
            if (col!=2) System.out.print("|");
        }
        System.out.println();
        System.out.println("   |   |   ");

        if (row!=2) System.out.println("------------");

    }

}

public static boolean moveIsNotLegal(int position) {
    if ((position<0)||(position>8)) return true;
    return (board[position]!=0);
}

public static int findWinningMove() {
    boolean foundit = false;
    int candidate=0;
    while ((candidate<9)&&(!foundit))
    {
        if (board[candidate]==0) // spot is empty
        {
            board[candidate]=2;
            foundit=(Winner()==2);
            board[candidate]=0;

        }
        if (!foundit) {
            candidate++;
        }
    }
    if (foundit) return candidate;
    return (-1);

}

public static int Winner()

{
    int winner = 0;
    for (int player = 1; player <= 2; player++) {
        if ((board[0] == player) && (board[1] == player) && (
           board[2] == player))
            winner = player;
        if ((board[0] == player) && (board[3] == player) && (
           board[6] == player))
            winner = player;
        if ((board[0] == player) && (board[4] == player) && (
           board[8] == player))
            winner = player;
        if ((board[1] == player) && (board[4] == player) && (
           board[7] == player))
            winner = player;
        if ((board[2] == player) && (board[5] == player) && (
           board[8] == player))
            winner = player;
        if ((board[3] == player) && (board[4] == player) && (
           board[5] == player))
            winner = player;
        if ((board[6] == player) && (board[7] == player) && (
           board[8] == player))
            winner = player;
        if ((board[2] == player) && (board[4] == player) && (
           board[6] == player))
            winner = player;
    }

    return winner;
}

public static int findRandomMove() {
    int candidate = -1;
    Random rg = new Random();

    while (moveIsNotLegal(candidate)) {
        candidate = rg.nextInt(9);
    }

    return candidate;

}

public static int checkBoard(int player) {
    int opponent;
    if (player==1) {
        opponent=2;
    } else {
        opponent=1;
```

```
}
int danger=-1;

if ((board[0]==player)&&(board[1]==player)&&(board[2]!=
    opponent)) danger=2;
if ((board[1]==player)&&(board[2]==player)&&(board[0]!=
    opponent)) danger=0;
if ((board[0]==player)&&(board[2]==player)&&(board[1]!=
    opponent)) danger=1;
if ((board[0]==player)&&(board[3]==player)&&(board[6]!=
    opponent)) danger=6;
if ((board[0]==player)&&(board[4]==player)&&(board[8]!=
    opponent)) danger=8;
if ((board[0]==player)&&(board[6]==player)&&(board[3]!=
    opponent)) danger=3;
if ((board[0]==player)&&(board[8]==player)&&(board[4]!=
    opponent)) danger=4;

if ((board[1]==player)&&(board[2]==player)&&(board[0]!=
    opponent)) danger=0;
if ((board[1]==player)&&(board[4]==player)&&(board[7]!=
    opponent)) danger=7;
if ((board[1]==player)&&(board[7]==player)&&(board[4]!=
    opponent)) danger=4;

if ((board[2]==player)&&(board[4]==player)&&(board[6]!=
    opponent)) danger=6;
if ((board[2]==player)&&(board[6]==player)&&(board[4]!=
    opponent)) danger=4;
if ((board[2]==player)&&(board[5]==player)&&(board[8]!=
    opponent)) danger=8;
    if ((board[2]==player)&&(board[8]==player)&&(board[5]!=
       opponent)) danger=5;

    if ((board[3]==player)&&(board[4]==player)&&(board[5]!=
       opponent)) danger=5;
    if ((board[3]==player)&&(board[5]==player)&&(board[4]!=
       opponent)) danger=4;

    if ((board[4]==player)&&(board[5]==player)&&(board[3]!=
       opponent)) danger=3;
    if ((board[4]==player)&&(board[6]==player)&&(board[2]!=
       opponent)) danger=2;
    if ((board[4]==player)&&(board[7]==player)&&(board[1]!=
       opponent)) danger=1;
    if ((board[4]==player)&&(board[8]==player)&&(board[0]!=
       opponent)) danger=0;

    if ((board[5]==player)&&(board[8]==player)&&(board[2]!=
       opponent)) danger=2;

    if ((board[6]==player)&&(board[7]==player)&&(board[8]!=
       opponent)) danger=8;
    if ((board[6]==player)&&(board[8]==player)&&(board[7]!=
       opponent)) danger=7;

    if ((board[7]==player)&&(board[8]==player)&&(board[6]!=
       opponent)) danger=6;
    return danger;



}
```

```
public static void main(String[] args) {

    String str;
    System.out.println("TicTacToe");


    int num = getMove();
    board[num]=1;

    switch (num) {
    case 4: System.out.println("My move is number 0");
            board[0]=2;
            break;
    case 0:
    case 2:
    case 6:
    case 8: System.out.println("My move is number 4");
    board[4]=2;
    break;

    case 1:
    case 3:
    case 5:
    case 7: System.out.println("My move is number 4");
    board[4]=2;
    break;
    }

    num=-1;
    while (moveIsNotLegal(num)) {
        printBoard();
        num=getMove();
    }

    board[num]=1;

    printBoard();

    // Computer move. See if opponent has two in a row
    // find if two in a row

    int nextmove =checkBoard(1);

    if (nextmove>=0) {
        System.out.println();
        System.out.println("My move is "+nextmove);
        board[nextmove]=2;
    } else {
        boolean foundit = false;
        int candidate=0;
        while ((candidate<9)&&(!foundit))
        {
            if (board[candidate]==0) // spot is empty
            {
                board[candidate]=2;
                foundit = (checkBoard(2)>=0);
                board[candidate]=0;
            }
            if (!foundit) {
                candidate++;
            }
        }
    if (foundit==true) {
        System.out.println("My move is " + candidate);
        board[candidate] = 2;
    } else {
        // do a random move
        while (moveIsNotLegal(candidate)) {
            candidate = (int) Math.random()*9;
        }
        System.out.println("My move is (random) " +
           candidate);
        board[candidate] = 2;

    }
}

while (moveIsNotLegal(num)) {
    printBoard();
    num = getMove();
}

board[num]=1;



if (Winner() != 0) {
    printBoard();
    if (Winner()==1) System.out.println("You win!!");
    else System.out.println("Guess who just won????");
    System.exit(0);
}
// can we get three in a row

if (findWinningMove() >= 0) {
    System.out.println("We have an offensive move");
    num = findWinningMove();
    board[num] = 2;
} else {

    // defensive move
    int nextmove2 = checkBoard(1);
    if (nextmove2 >= 0) {
        System.out.println("My move is " + nextmove2);
        board[nextmove2] = 2;
    } else {
        // random move
        nextmove2 = findRandomMove();
        board[nextmove2]= 2;
    }
}
if (Winner() != 0) {
    printBoard();
    if (Winner()==1) System.out.println("You win!!");
    else System.out.println("Guess who just won????");
    System.exit(0);
    }

while (moveIsNotLegal(num)) {
    printBoard();
    num = getMove();
}

board[num]=1;

if (Winner() != 0) {
    printBoard();
    if (Winner()==1) System.out.println("You win!!");
    else System.out.println("Guess who just won????");
    System.exit(0);
}
// can we get three in a row

if (findWinningMove() >= 0) {
    System.out.println("We have an offensive move");
    num = findWinningMove();
    board[num] = 2;
} else {

    // defensive move
    int nextmove3 = checkBoard(1);
    if (nextmove3 >= 0) {
        System.out.println("My move is " + nextmove3);
        board[nextmove3] = 2;
    } else {
        // random move
        nextmove3 = findRandomMove();
        board[nextmove3]= 2;
    }
}
if (Winner() != 0) {
    printBoard();
    if (Winner()==1) System.out.println("You win!!");
    else System.out.println("Guess who just won????");
    System.exit(0);
}

printBoard();
```

```
        System.out.println("Darn. It is a tie!!");


    }
}
```

Bibliography

[1] Checkstyle Standard Checks. `http://Checkstyle.sourceforge.net/checks.html`. Accessed 1 April 2016.

[2] Findbugs - Find Bugs in Java Programs. `http://FindBugs.sourceforge.net/`. Access 1 April 2016.

[3] PMD. `https://pmd.github.io/`. Accessed 28 March 2016.

[4] The Byte Code Engineering Library, 2004. `http://jakarta.apache.org/bcel/`. Accessed 26 February 2016.

[5] Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proceedings of the 13th Australian Conference on Software Engineering*, ASWEC '01, pages 68–, Washington, DC, USA, 2001. IEEE Computer Society.

[6] Cyrille Artho and Klaus Havelund. Applying Jlint to Space Exploration Software. *Lecture Notes in Computer Science Verification, Model Checking, and Abstract Interpretation*, page 297308.

[7] Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. Experiences Using Static Analysis to Find Bugs. *IEEE Software*, 25:22–29, 2008. Special issue on software development tools, September/October (25:5).

[8] Nathaniel Ayewah and William Pugh. The Google Findbugs Fixit. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 241–252, New York, NY, USA, 2010. ACM.

[9] Nathaniel Ayewah and William Pugh. Null Dereference Analysis in Practice. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 65–72, New York, NY, USA, 2010. ACM.

[10] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating Static Analysis Defect Warnings on Production Software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 1–8, New York, NY, USA, 2007. ACM.

[11] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, FOSE '07, pages 104–119, Washington, DC, USA, 2007. IEEE Computer Society.

[12] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Sigplan Notices*, volume 41, pages 169–190. ACM, 2006.

[13] Simon Butler. The Effect of Identifier Naming on Source Code Readability and Quality. In *Proceedings of the Doctoral Symposium for ESEC/FSE on Doctoral Symposium*, ESEC/FSE Doctoral Symposium '09, pages 33–34, New York, NY, USA, 2009. ACM.

[14] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2. *Formal Methods for Components and Objects Lecture Notes in Computer Science*, page 342363.

[15] Brian Cole, Daniel Hakim, David Hovemeyer, Reuven Lazarus, William Pugh, and Kristin Stephens. Improving Your Software Using Static Analysis to Find Bugs. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 673–674, New York, NY, USA, 2006. ACM.

[16] Tom Copeland. PMD applied, 2005.

[17] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Puasuareanu, Robby, and Hongjun Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[18] James C. Corbett, Matthew B. Dwyer, John Hatcliff, and Robby. Bandera: A source-level interface for model checking java programs. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 762–765, New York, NY, USA, 2000. ACM.

[19] Dane Dennis. Exploit better the results of Pmd, Findbugs and CheckStyle., 2013. `https://www.javacodegeeks.com/2013/11/exploit-better-the-results-of-pmd-FindBugs-and-Checkstyle.html`. Accessed 1 April 2016.

[20] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of*

the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[21] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. *SIGPLAN Not.*, 37(5):234–245, May 2002.

[22] Jeff Foster. Cmsc 631 - program analysis and understanding - data flow analysis, 2016. `https://www.cs.umd.edu/class/fall2010/cmsc631/data-flow.pdf`. Access 4 April 2016.

[23] Ivo Gomes, Pedro Morgado, Tiago Gomes, and Rodrigo Moreira. An overview on the Static Code Analysis approach in Software Development. *Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias 4200-465, Porto, Portugal.*

[24] Michael T. Helmick. Interface-based Programming Assignments and Automatic Grading of Java Programs. In *Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, ITiCSE '07, pages 63–67, New York, NY, USA, 2007. ACM.

[25] David Hovemeyer and William Pugh. Finding Bugs is Easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 132–136, New York, NY, USA, 2004. ACM.

[26] David Hovemeyer and William Pugh. Finding Bugs is Easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[27] David Hovemeyer and William Pugh. Finding More Null Pointer Bugs, but Not Too Many. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 9–14, New York, NY, USA, 2007. ACM.

[28] Allen Hsu, Somakala Jagannathan, Sajjad Mustehsan, Session Mwamufiya, and Marc Novakouski. Analysis Tool Evaluation: PMD. *School of Computer Science Carnegie Mellon University*, Apr 2007.

[29] Sunghun Kim and Michael D. Ernst. Which warnings should i fix first? In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 45–54, New York, NY, USA, 2007. ACM.

[30] Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of Bug Fixes. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA, 2006. ACM.

[31] Guangtai Liang, Qianxiang Wang, Tao Xie, and Hong Mei. Inferring project-specific bug patterns for detecting sibling bugs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 565–575, New York, NY, USA, 2013. ACM.

[32] Paulo Merson. Ultimate Architecture Enforcement: Custom Checks Enforced at Code-commit Time. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, &#38; Applications: Software for Humanity*, SPLASH '13, pages 153–160, New York, NY, USA, 2013. ACM.

[33] Abhishek Minde, Bhanu Sistla, Jeff Salk, Nan Li, and Yuki Saito. Tool analysis report - PMD Analysis of Software Artifacts. *Carnegie Mellon University - Department of Materials Science and Engineering 2009*, 2009. `https://www.cs.cmu.edu/~aldrich/courses/654-sp07/tools/group2-pmd-2009.pdf`. Accessed 1 April 2016.

[34] Dimitris Mitropoulos, Vassilios Karakoidas, Panos Louridas, Georgios Gousios, and Diomidis Spinellis. The Bug Catalog of the Maven Ecosystem. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 372–375, New York, NY, USA, 2014. ACM.

[35] S. Panichella, V. Arnaoudova, M. Di Penta, and G. Antoniol. Would static analysis tools help developers with code reviews? In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 161–170, March 2015.

[36] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software Engineering: Theory and Practice*. Prentice Hall; 4th edition, Upper Saddle River, New Jersey, 2009.

[37] N. Rutar, C.B. Almazan, and J.S. Foster. A Comparison of Bug Finding Tools for java. In *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pages 245–256, Nov 2004.

[38] Raghvinder S. Sangwan. Taming the complexity: The need for program understanding in software engineering. In *Pennsylvania State Universit, Great Valley School of Graduate Professional Studies*, pages 1–13.

[39] Oleg Shelajev. The Wise Developers' Guide to Static Code Analysis featuring FindBugs, Checkstyle, PMD, Coverity and Sonarqube, Apr 2014. `http://zeroturnaround.com/rebellabs/developers-guide-static-code-analysis-FindBugs-Checkstyle-pmd-coverity-sonarqube/`. Accessed April 1 2016.

[40] Yannis Smaragdakis and Christoph Csallner. Combining Static and Dynamic Reasoning for Bug Detection. In *Proceedings of the 1st International Conference on Tests and Proofs*, TAP'07, pages 1–16, Berlin, Heidelberg, 2007. Springer-Verlag.

[41] Jaime Spacco, David Hovemeyer, and William Pugh. Tracking Defect Warnings Across Versions. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 133–136, New York, NY, USA, 2006. ACM.

[42] Markus Sprunck. Comparison of Static Code Analysis Tools for Java - Findbugs vs PMD vs Checkstyle - Software Engineering Candies. `https://sites.google.com/site/markussprunck/blog-1/comparison-of-FindBugs-pmd-and-Checkstyle`. Accessed 1 April 2016.

[43] Raoul-Gabriel Urma and Alan Mycroft. Programming Language Evolution via Source Code Query Languages. In *Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU '12, pages 35–38, New York, NY, USA, 2012. ACM.

[44] Steven J Zeil. Program Analysis Tools, Apr 2015. `https://www.cs.odu.edu/~zeil/cs350/s16/public/analysis/`. Accessed 1 April 2016.