

---

MASTER THESIS

---

# EFFECTIVENESS OF VERIFICATION TOOLS

---

conducted at the  
Institute of Software Technology  
Graz University of Technology, Austria

in co-operation with  
SSI Schäfer Automation GmbH  
Graz, Austria

by  
Marcel Hannes Ablasser, 01230278

Supervisors:  
Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Assessors/Examiners:  
Univ.-Prof. Dipl.-Ing. Dr.techn. Frank Kappe  
Univ.-Prof. Dipl.-Inf. Univ. Dr.rer.nat. Marcel Carsten Baunach

Graz, February 26, 2019



## Abstract

Software faults, also referred to as bugs, can occur on each line of code; therefore, verification and validation (V&V) is crucial before software products are delivered to customers. The process of V&V is mainly performed by testing and reviewing requirements, design and source code. Software developers can perform both tasks: reviewing and testing. However, one, two or even ten developers might not find all bugs in a software project. A reason why bugs survive this V&V process is the size and complexity of software projects. Another reason why bugs survive is because code reviewing and testing is not performed thoroughly or at all by each team member. Why not add an additional verification mechanism with an automated pair of eyes in order to compensate these drawbacks by checking each line of code again for faults, inappropriate programming habits and coding violations? Verification tools are created to perform this additional check and are designed to work as an automated second pair of eyes. Therefore, these tools help to further improve the code quality and might even detect problems overlooked by entry-level as well as senior software developers.

The aim of this master thesis is to analyze and define the effectiveness of selected verification tools – PMD, SpotBugs, SourceMeter, Infer, SonarQube, RATS and Cppcheck – for software written in **Java** and **C++**. A case study is performed in order to measure the effectiveness by means of criteria, for example, the number of detected errors (true positives), how many false positives are reported and how many bugs are not detected by a verification tool (false negatives). These criteria are captured for a **Java** program with 82 lines of code (LOC) called *Argument Printer*. Furthermore, self-created programs are analyzed which contain common fault(s), for instance, null pointer exceptions and index out of bounds bugs. In addition to this **Java** program and these self-created programs, software projects from SSI Schaefer Automation GmbH, an international company, are taken into account. Associates from SSI Schaefer Automation GmbH took part in this case study using the selected verification tools. Hence, experience from employees at SSI Schaefer Automation GmbH is considered and has a direct influence on the outcome regarding the effectiveness of the selected verification tools.

The evaluation of these selected verification tools showed that more bugs are detected for software written in **Java** than **C++**. Furthermore, performing an analysis for **Java** programs is easier and more straightforward. Nevertheless, the best verification tool for both programming languages (PLs) – **C++** and **Java** – is Infer. It detects a good amount of bugs, does not detect many false positives and supports both PLs.

To sum up, this master thesis analyzes the effectiveness of selected verifications tools and addresses the following research question:

How effective are verification tools, nowadays, for **C++** and **Java** source code?



## Kurzfassung (German)

Softwarefehler, auch Bugs genannt, können in jeder Codezeile von Softwareprojekten auftreten. Eine Verifizierung und Validierung (V&V) ist somit essentiell, bevor Softwareprodukte an Kunden ausgeliefert werden. Der Prozess V&V wird hauptsächlich durch Testen und Überprüfen von Anforderungen, Design und Quellcode durchgeführt. Softwareentwicklerinnen und Softwareentwickler können diese beiden Aufgaben ausführen. Jedoch, einer, zwei oder sogar zehn Personen finden möglicherweise nicht alle Fehler in einem Softwareprojekt. Ein Grund, warum nicht alle Bugs durch diesen V&V Prozess gefunden werden, ist die Größe und Komplexität von Softwareprojekten. Ein weiterer Grund ist die nicht ordnungsgemäße Durchführung des V&V Prozesses aller Teammitglieder. Warum nicht einen zusätzlichen Überprüfungsmechanismus mit einem automatisierten Augenpaar hinzufügen, um diese Nachteile zu kompensieren, indem jede Codezeile erneut auf Fehler, schlechte Programmiergewohnheiten und Programmierstandards geprüft wird? Verifikationstools wurden erstellt, um diese zusätzlichen Überprüfungen durchzuführen. Somit helfen diese Tools die Codequalität weiter zu verbessern und finden möglicherweise Probleme, die von Softwareentwicklerinnen und Softwareentwicklern übersehen werden.

Das Ziel dieser Masterarbeit ist es, die Wirksamkeit ausgewählter Verifikationstools – PMD, SpotBugs, SourceMeter, Infer, SonarQube, RATS und Cppcheck – für Software geschrieben in C++ und Java zu analysieren und zu definieren. Eine Fallstudie wird durchgeführt, um die Wirksamkeit zu messen. Anhand von Kriterien kann beispielsweise die Anzahl der richtig erkannten Fehler (True Positives) und die Anzahl der falsch erkannten Fehler (False Positives) ermittelt werden. Diese Kriterien werden für ein Java-Programm mit 82 Codezeilen, genannt *Argument Printer*, erfasst. Darüber hinaus werden selbst erstellte Programme analysiert, die allgemeine Fehler enthalten. Zum Beispiel, NullPointerExceptions. Neben diesem Java-Programm und diesen selbst erstellten Programmen werden Softwareprojekte von SSI Schaefer Automation GmbH, einem internationalen Unternehmen, berücksichtigt. Mitarbeiter von SSI Schaefer Automation GmbH haben an dieser Fallstudie mit den ausgewählten Verifikationstools teilgenommen. Daher fließt die Erfahrung und Beurteilung der Mitarbeiter von SSI Schäfer Automation GmbH ein und hat einen direkten Einfluss auf das Ergebnis hinsichtlich der Wirksamkeit dieser ausgewählten Verifikationstools.

Die Auswertung der Tools hat gezeigt, dass mehrere Software Fehler in Java als in C++ entdeckt wurden. Außerdem ist es einfacher eine Analyse für Java Programme als für C++ durchzuführen. Das beste Verifikationstool für beide Programmiersprachen – C++ und Java – ist Infer. Es erkennt viele richtige Fehler, meldet wenige falsche Fehler und unterstützt beide Programmiersprachen.

Zusammenfassend kann gesagt werden, dass diese Masterarbeit die Wirksamkeit ausgewählter Verifikationstools analysiert und die folgende Forschungsfrage beantwortet:

Wie effektiv sind Verifikationstools für C++ und Java Software heutzutage?



## **AFFIDAVIT**

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

---

Date

---

Signature

## **Eidesstattliche Erklärung**

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

---

Datum

---

Unterschrift





# Contents

Acronyms . . . . .	IX
Glossary . . . . .	X
<b>1 Introduction</b>	<b>13</b>
1.1 Motivation . . . . .	13
1.2 Problem and Thesis Statement . . . . .	17
1.3 Organization . . . . .	18
<b>2 Software Verification and Validation</b>	<b>19</b>
2.1 Definition Software Bug . . . . .	20
2.2 Verification Methods . . . . .	21
2.2.1 Testing . . . . .	21
2.2.2 Code Review . . . . .	22
2.2.3 Verification Tools . . . . .	22
<b>3 Defining Effectiveness</b>	<b>25</b>
3.1 Captured Criteria . . . . .	25
3.1.1 Source Code Criteria . . . . .	25
3.1.2 Verification Tool Criteria . . . . .	25
3.2 Calculated Parameter . . . . .	26
3.2.1 Source Code Parameter . . . . .	26
3.2.2 Verification Tool Parameter . . . . .	26
3.3 Effectiveness Metrics . . . . .	29
<b>4 Selected Verification Tools</b>	<b>33</b>
4.1 PMD . . . . .	34
4.2 SpotBugs (FindBugs) . . . . .	36
4.3 Infer . . . . .	38
4.4 SonarQube . . . . .	40
4.5 SourceMeter . . . . .	42
4.6 RATS . . . . .	43
4.7 Cppcheck . . . . .	44
<b>5 Case Study</b>	<b>45</b>
5.1 Detect Bug Challenge . . . . .	45
5.2 Programs with Common Errors in C++ . . . . .	46
5.2.1 Null Pointer Bugs in C++ . . . . .	50
5.2.2 Index Out of Bounds Bugs in C++ . . . . .	56
5.2.3 Resource Bugs in C++ . . . . .	59
5.3 Programs with Common Errors in Java . . . . .	60
5.3.1 Null Pointer Bugs in Java . . . . .	65
5.3.2 Index Out of Bounds Bugs in Java . . . . .	71
5.3.3 Resource Bugs in Java . . . . .	74
5.4 Software Projects from SSI Schaefer Automation GmbH . . . . .	75

5.4.1	Project-CA in C++ . . . . .	75
5.4.2	Project-JA in Java . . . . .	76
5.4.3	Project-JB in Java . . . . .	77
5.4.4	Beta Releases . . . . .	78
<b>6</b>	<b>Outcome</b>	<b>79</b>
6.1	Detect Bug Challenge . . . . .	79
6.2	Programs with Common Errors . . . . .	81
6.3	Software Projects from SSI Schaefer Automation GmbH . . . . .	83
<b>7</b>	<b>Conclusion</b>	<b>85</b>
	<b>Appendices</b>	<b>87</b>
<b>A</b>	<b>Appendix Chapter 1 Introduction</b>	<b>88</b>
A.1	Section 1.1 Motivation . . . . .	88
<b>B</b>	<b>Appendix Chapter 4 Selected Verification Tools</b>	<b>90</b>
B.1	Section 4.1 PMD . . . . .	90
	B.1.1 Section 4.1 Get Started . . . . .	90
B.2	Section 4.2 SpotBugs (FindBugs) . . . . .	94
	B.2.1 Section 4.2 Get Started . . . . .	94
B.3	Section 4.6 RATS . . . . .	98
<b>C</b>	<b>Appendix Chapter 5 Case Study</b>	<b>99</b>
<b>D</b>	<b>Appendix Chapter 6 Outcome</b>	<b>101</b>
D.1	Section 6.1 Detect Bug Challenge . . . . .	101
	<b>List of Figures</b>	<b>103</b>
	<b>List of Tables</b>	<b>104</b>
	<b>Bibliography</b>	<b>106</b>

# Acronyms

- API** application programming interface. 38
- ch** chapter. 21–23
- CPD** copy-paste-detector. 34, 40
- CWE** common weakness enumeration. 77
- FDR** false discovery rate. 27
- GUI** graphical user interface. 28
- IDE** integrated development environment. 34, 36, X
- KLOC** thousand lines of code. 26, X, *Glossary*: KLOC
- LOC** lines of code. 13, 14, 16, 18, 21, 25, 26, 40, 45, 46, 50–60, 65–77, 79, 85, I, X, *Glossary*: LOC
- OS** operating system. 33, 34, 36, 38, 40, 42–44
- PL** programming language. 16, 17, 33–37, 40–43, 46, 50–60, 65–77, 79, 80, 82, 84, 85, I, X
- QA** quality assurance. 78
- RATS** Rough Auditing Tool for Security. 33, 43, 98, VII, VIII
- sec** section. 29
- SOA** service-oriented architecture. 76
- TLLOC** total logical lines of code. 45, 46, 50–60, 65–77
- TLOC** total lines of code. 42, 45, 46, 50–60, 65–77
- TNOS** total number of statements. 42, 45, 46, 60, 65–74, 76, 77
- V&V** verification and validation. 13, 18, 19, 78, I, III

## Glossary

**#** This symbol indicates the *number of something*. For example, #Files indicates the number of files. 16, 18, 30, 45–48, 50–63, 65–77, 83

**Bug** The term software bug, or short bug, describes in a colloquial way an error, failure or fault in the source code of a software program[3, p. 358]. 13, 14, 16, 17, 19–23, 25–27, 29–31, 33, 36, 42, 44, 45, 75–81, 83–86, 101, I, III

**Eclipse** Eclipse is an integrated development environment (IDE) that is free and open-source software. This IDE supports many programming languages (PLs) and is extendable by a huge number of various plug-ins written by the Eclipse community, other developers or even by oneself. 34–37, 90–92, 94–96, 103

**False Negative** A bug that exists in the analyzed source code but is not detected and reported by a verification tool. 16, 23, 25, 26, 46–48, 50–63, 65–77, I

**False Positive** A bug that does not exist in the analyzed source code but is detected and reported by a verification tool. False positives are reported due to missing information during the analysis. 16, 23, 25–27, 29–31, 44, 46–48, 50–63, 65–77, 79, 83, 84, 86, I, III

**KLOC** A thousand lines of code (KLOC) from a software project. 26, 75–77, X

**LOC** All lines of code (LOC) from a software project. 13, 18, 21, 25, 40, 45, 46, 60, 76, 79, 85, I, X

**SonarQube** SonarQube is a web platform that focuses on continuous code quality. This web platform supports more than 20 PLs including these major languages: C, C++, C#, Java, Javascript, Php and Python. Furthermore, various plug-ins are available in order to enhance the code analysis and to detect even more faults, inappropriate programming habits and coding style violations (see more about SonarQube in section 4.4). 16, 28, 33–37, 40–42, 60, 65–74, 76, 77, 93, 97, 103

**True Negative** A bug that does not exist in the analyzed source code and is not reported by a verification tool. 23

**True Positive** A bug that exists in the analyzed source code and is detected and reported by a verification tool. 16, 23, 25–27, 29, 30, 46–48, 50–63, 65–77, 79–81, I, III

## 1

# Introduction

## 1.1 Motivation

Verifying software is crucial before new products or releases are shipped to users. This process of verification and validation (V&V) is mainly performed by testing and reviewing every single line of code. Performing these two steps properly and thoroughly is going to reduce the number of faults alias bugs<sup>1</sup> and lead to more confidence in the correctness and reliability of software product(s). Unfortunately, each of these tasks (testing and reviewing of source code) is time-consuming[32, p. 87] – and the more lines of code (LOC) and the more complex a software project is the more work hours and machine hours are needed to test and review each line of code[10, p. 9]. Due to this drawback, one or even both tasks (testing and reviewing of source code) might not be performed appropriately or at all. Another drawback is that both methods do not guarantee the correctness of software product(s) nor that all bugs are eliminated.

In order to compensate or rather diminish these drawbacks, verification tools can be added in software development processes – in addition to basic code reviews – to further reduce the number of faults and, therefore, to foster the confidence in the correctness and reliability of software product(s). Verification tools are code analyzers that scan source code and report a list of issues (real and potential errors & warnings like a compiler). By means of this list, developers have another closer look to parts of source code where real or potential faults are. Even with the use of verification tools, one can still not guarantee the correctness of software product(s) nor that all bugs are eliminated. However, these verification tools enhance the V&V process by detecting overlooked real and potential bugs of entry-level as well as senior software developers.

In addition to diminishing testing and reviewing drawbacks, verification tools can be used to teach all and, in particular, entry-level software developers to avoid or rather reduce the number of real and potential bugs. Furthermore, some verification tools report, in addition to real and potential bugs, inappropriate programming habits and coding violations. For instance, having an empty catch block of a try-catch statement in `Java` is defined as an inappropriate programming habit. Coding violations depend on the defined coding standard by the team of software developers. To illustrate, a constant named `errorMessage` is a coding violation if the coding standard defines that all constants should be written only with capital letters and underscores (`errorMessage` → `ERROR_MESSAGE`). By reviewing these issues (real and potential bugs, inappropriate programming habits and coding violations), a software developer performs an own additional code review by means of verification tools. Each issue is listed with a description that explains why a statement or a combination of statements is marked as an error or warning by a verification tool. Due to this explanation, a software developer is taught and reminded by a verification tool to avoid well known critical and potential software problems as well as bad coding habits.

---

<sup>1</sup> The term software bug, or short bug, describes in a colloquial way an error, failure or fault in the source code of a software program[3, p. 358].

Statement of a 55 to 64-year-old man with 20 to 30 years of professional experience as a software developer who participated in this case study (original language, German):

*PS: SonarQube ist eine echte Hilfe! Ich habe 92 Fehler von 92 behoben; bei einigen hätte ich ohne SonarQube nicht einmal gewußt, dass es ein Fehler ist.*

Same statement translated into English:

*PS: SonarQube is a real help! I have fixed 92 errors of 92; for some, without SonarQube, I would not even have known it was a mistake.*

Another advantage of using verification tools is that the workload of a software developer who performs code reviews of other developers decreases. If a software developer performs another additional code review by means of verification tools, the code quality increases because real and potential bugs, inappropriate programming habits and coding violations are reduced by performing this additional review. In other words, a developer who uses verification tools submits source code with fewer software issues than a developer who performs a submission without additional check. As a consequence, the workload of a software developer who performs code reviews of other developers decreases.

To demonstrate the strength of verification tools, a short Java program called *Argument Printer* with 82 LOC – see more statistics in Table 5.1 – is created and analyzed by software developers and by verification tools. This program contains several bugs which in the worst case – crash the program. The following excerpts shown on the next page illustrate the main functions of the program *Argument Printer* in Listing 1.1 and Listing 1.2 (the whole source code is depicted in Listing A.1 and Listing A.2).

```

15     private static final Random RANDOM = new Random();
16
17     /**
18      * Entry point of program
19      *
20      * @param args
21      * @throws InterruptedException
22      */
23     public static void main(String[] args) {
24         LOGGER.info("Start program...");
25         final ArgumentPrinter argPrinter = new ArgumentPrinter();
26         for (int i = 0; i < 5; i++) {
27             if (generateRandomInteger() > 0)
28                 LOGGER.log(Level.INFO, "Log file path: '{0}'", argPrinter.initLogFile().toString());
29             Object[] testArguments = i == 0 ? args : generateRandomArgs();
30             argPrinter.setArguments(testArguments, 5);
31             argPrinter.logAll();
32             LOGGER.log(Level.INFO, "{0}: iteration..", i);
33         }
34     }
35
36     private static Object[] generateRandomArgs() {
37         int numArgs = RANDOM.nextInt(10);
38         if (numArgs > 0) { return new Object[numArgs]; }
39         return null;
40     }
41
42     private static Integer generateRandomInteger() {
43         Integer randomValue = Integer.valueOf(RANDOM.nextInt(2));
44         if (randomValue > 0)
45             return randomValue;
46         return null;
47     }

```

Listing 1.1: Excerpt of the program: Argument Printer written in Java (file Main.java).

```

13 public class ArgumentPrinter {
14
15     /** Default Java logger */
16     private static final Logger LOGGER = Logger.getLogger(ArgumentPrinter.class.getName());
17
18     private Object[] args;
19     private Integer maxArgs;
20
21     public void setArguments(final Object[] args, Integer maxArgs) {
22         this.args = new String[maxArgs]; this.maxArgs = maxArgs;
23         initArgs(args, maxArgs);
24     }
25
26     private synchronized void initArgs(final Object[] args, final Integer maxNumArgs) {
27         for (int i = 0; i < maxNumArgs; i++)
28             this.args[i] = args[i];
29     }
30
31     public File initLogFile() {
32         File logFile = new File("logfile.log");
33         try {
34             FileOutputStream fos = new FileOutputStream(logFile);
35             fos.write("LogFile: ArgumentPrinter".getBytes(Charset.defaultCharset()));
36             fos.close();
37         } catch (IOException e) {
38             // Ignore
39             return null;
40         }
41         return logFile;
42     }
43
44     /**
45      * Logs the number of arguments and the content of each argument.
46      */
47     public void logAll() {
48         logNumberOfArguments(); logAllArguments();
49     }
50
51     private void logNumberOfArguments() {
52         String formatString = createFormatStringNumArgs();
53         if (!formatString.isEmpty())
54             LOGGER.log(Level.INFO, formatString, args.length);
55     }
56
57     private void logArgumentContent(Integer index) {
58         LOGGER.log(Level.INFO, "Arg[" + index + "]: '{0}'", args[index].toString());
59     }
60
61     private void logAllArguments() {
62         for (int i = 0; i <= maxArgs; i++)
63             logArgumentContent(i);
64     }
65
66     private String createFormatStringNumArgs() {
67         if (args.length == 1) return "{0} argument is entered!";
68         else if (args.length > 1) return "{0} argument(s) are entered!";
69         return null;
70     }
71 }

```

Listing 1.2: Excerpt of the program: Argument Printer written in Java (file ArgumentPrinter.java).

Software developers and selected verification tools performed a code review of the `Java` program *Argument Printer* (see the whole source code in Listing A.1 and Listing A.2). The best software developer correctly detected six out of six bugs (see Table 1.1 – column C and row #True Positives). On the contrary, the best-selected verification tool is able to correctly detect four out of six bugs (see Table 1.2 – column Infer and row #True Positives). Furthermore, a comparison of the reported false positives – best software developer nine and best-selected verification tool one (see Table 1.1 and 1.2 – row #False Positives) – enhances the benefits of verification tools. In addition to true positives and false positives, execution time is another important criterion. The best and fastest software developer who found all six bugs needed 1510 seconds which are 25 minutes and ten seconds. By contrast, the best-selected verification tool required less than three seconds to correctly detect four out of six bugs in 82 LOC.

Table 1.1: Captured criteria of all software developer (software developer A, software developer B, ..., D) who participated in the detect bugs challenge of the `Java` program *ArgumentPrinter*.

Criteria \ Developer	A	B	C	D
#Reported Issues	5	2	15	7
#True Positive	2	1	6	5
#False Positive	3	1	9	2
#False Negative	4	5	0	1
Execution Time [s]	1666	2143	1510	1997

Table 1.2: Captured criteria of the program *ArgumentPrinter* – written in `Java`. All criteria are listed for each verification tool which supports the programming language (PL) of the program *ArgumentPrinter* (`Java`).

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	53	0	13	3	5
#True Positive	2	0	0	3	4
#False Positive	51	0	13	0	1
#False Negative	4	6	6	3	2

To sum up, from the result of this challenge – software developer vs verification tools – one can see that verification tools do not detect every bug but analyze the source code of the `Java` program *Argument Printer* faster than software developers. In other words, verification tools detect a good amount of bugs and perform an analysis quite fast, therefore, why not performing an additional check of source code with an automated pair of eyes?

**#MakeCodeVerificationEasier**



## 1.2 Problem and Thesis Statement

**Problem Statement.** The source code of software products is developed in several statements, functions, classes, files and directories. Due to the complexity and size of software projects, it is impossible to check every single line of code again by a software developer after each committed change from another co-worker. Even if every single line of code could be checked by one, three or ten software developers, some issues could be overlooked.

Why not add an additional verification mechanism with an automated pair of eyes in order to perform this tedious task? Verification tools are created to process this additional check by scanning every single line of code for bugs, inappropriate programming habits and coding violations. Therefore, this master thesis investigates how effective verification tools are nowadays.

**Thesis Statement.** This master thesis analyzes the effectiveness of selected verifications tools and addresses the following research question:

How effective are verification tools, nowadays, for **C++** and **Java** source code?

In order to address this research question, a prerequisite has to be resolved:

How can one define the effectiveness of verification tools in terms of performance, usability and reported issues?

The following statements are defined in order to define the scope of this master thesis:

- All verification tools which are analyzed during this master thesis are static code analyzer.
- Source code in the following programming languages (PLs) is analyzed: **C++** and **Java**.
- In order to determine the effectiveness of selected verification tools, a case study is performed. Programs with common errors are created and analyzed as part of this case study as well as software projects from an international company: SSI Schaefer Automation GmbH.
- Verification tools that perform a dynamic code analysis are not investigated and not part of this master thesis.
- Not part of this master thesis is, adding one or a number of verification tools to the software development process of SSI Schaefer Automation GmbH. Only an evaluation of all selected verification tools is performed.

## 1.3 Organization

This master thesis is divided into seven chapters.

- The first chapter – 1 Introduction – initiates the topic of this master thesis by addressing the following questions in section 1.1 Motivation: What is this master thesis about? What are verification tools? Why should the source code of software products be checked by verification tools?
- Second, a closer look at the process of verification and validation (V&V) of software projects is addressed – including the V&V methods: testing, code reviewing and verification tools. Additionally, the reason why V&V should be performed is described in chapter two – 2 Software Verification and Validation.
- Chapter three – 3 Defining Effectiveness – enumerates and defines all criteria that are captured and calculated in order to derive the effectiveness of verification tools. In this chapter, an answer to the following question is stated: Which terms have an influence on the effectiveness of a verification tool?
- The next chapter – 4 Selected Verification Tools – contains an overview of all selected verification tools that are investigated in this master thesis. Each verification tool is described in its own section and is divided into three subsections: Description, Get Started and Demonstration.
- Fifth, all about the case study which is divided into four main groups: (i) Detect Bug Challenge, (ii) Programs with Common Errors in C++, (iii) Programs with Common Errors in Java and, last but not least, (iv) Software Projects from SSI Schaefer Automation GmbH. For each software project of all groups statistics about the source code are provided, for example, lines of code (LOC), the number of (#) Files and #Classes. Furthermore, all captured and calculated criteria are listed in tables for each software project. In addition to statistics and these criteria tables, the source code is depicted for all software projects of the (i) first, (ii) second and (iii) third group.
- The sixth chapter – 6 Outcome – presents the results of this master thesis.
- Finally, chapter seven – 7 Conclusion – sums up the concept behind this master thesis and the insights that are accomplished and obtained by performing this case study.

In addition to these seven chapters, appendices are attached.

## 2

## Software Verification and Validation

The process of verification and validation (V&V) is an engineering practice – according to Fisher[15, p. 3] – and is used to foster the confidence in the correctness and reliability of software product(s). Nowadays, software is part of almost everyone’s life. To illustrate, banks make use of software to manage thousands of bank accounts, elementary school teachers work with software to prepare the material and to submit the grades of all pupils, and cities use software to control the traffic light on streets which are used on a daily basis. Furthermore, other devices that are used several times on a daily basis are smartphones and computers. Consequently, humans rely on these software products and, therefore, developers should ensure that these products are reliable and behave as expected. In particular, software which is deployed in critical systems like spaceships, airplanes, cars or medical devices has to run correctly because a fault could cause enormous damage and even life-threatening situations. For example, an excerpt of lists about cost-intensive and severe bugs (see bug definition in section 2.1):

- **July 28, 1962 – Mariner I space probe**[34, p. 1][1][17][27]  
A rocket had to be destroyed because of a software bug that caused the rocket to divert from its intended launch path.
- **June, 1982 – Soviet Gas Pipeline Explosion**[34, p. 1][1][17]  
A software sabotage by an operation of the Central Intelligence Agency (CIA) from the United States of America (USA) led to a gas pipeline explosion and was reported as the largest non-nuclear explosion in the planet’s history.
- **1985-1987 – Therac-25 medical accelerator**[34, p. 1][18, p. 2][17][25]  
A race condition bug caused a machine called Therac-25 to overdose the radiation. This machine was used to deliver radiation therapy for cancer patients. Due to this bug, at least three patients died because of the overdose of radiation and many were seriously injured.

These historical bugs should remind each software developer, in particular, the ones who develop critical source code about the importance and responsibility of deploying correct and error-free code. Hence, the V&V of software is necessary before releases are delivered to customers.

What is the difference between verification and validation? An American software engineering Bohem (1981) put it in a nutshell[9, p. vi]:

*You are doing validation when you are answering the question: "Are we building the right product?" You are doing verification when you are answering the question: "Are we building the product right?"*

**Software Verification:** A process to review the source code for faults which could lead to wrong behavior, an unexpected result or in the worst case to shutdown and out of service situations. To put it differently, is the software product reliable and does it always deliver the expected output? For instance, a customer wants the following use case: As a user it is possible to see images on the company's website via any browser (Firefox, Internet Explorer, Google Chrome, ...). After the implementation, the verification process of the developed software is performed. It is possible to see the first image but when a user clicks "Next" to see another one, the program does not respond. Therefore, the developers did not build the product right (the software contains a bug).

**Software Validation:** A process to review the specified requirements with the behavior of the developed software product. In other words, does the software product deliver what the customer wants? The same use case: As a user I want to see images on the company's website via any browser (Firefox, Internet Explorer, Google Chrome, ...). However, there was a miscommunication which led to a misinterpretation and the team of developers implemented the function for videos instead of images. During the verification process, no problem is detected because the program executes without faults – the project was built right. However, during the validation process with the customer the image/video problem is appeared – the developers did not build the right product.

Validation is as important as verification in order to ensure customer satisfaction. However, this master thesis focuses on verifying source code of software, thus, at this point, there is no further explanation of validating software. Instead of targeting validation, a closer look at methods and techniques of software verification is outlined in the following section 2.2. In addition to these methods, the definition of a software bug is explained in the next section 2.1.

## 2.1 Definition Software Bug

The term software bug, or short bug, describes in a colloquial way an error, failure or fault in the source code of a software program[3, p. 358] Due to such an error, a software program can produce wrong behavior, unexpected results or in the worst case scenario – it crashes or stops responding by being in a deadlock.

According to T. Anderson and B. Randell, a software bug is defined as follows[3, p. 358]:

*One or more software bugs exist in a system if a software change is required to correct a single major error or minor error so as to meet specified or implied system performance requirements.*

*Thus, a bug is simply the colloquial and highly descriptive term for a software error.*

## 2.2 Verification Methods

In order to verify software, state-of-the-art methods should be used to ensure and deliver software products with high confidence about correctness and reliability. An introduction about three state-of-the-art methods[15, p. 3] is outlined in the following sections:

- **2.2.1 Testing**
- **2.2.2 Code Review**
- **2.2.3 Verification Tools**

### 2.2.1 Testing

In addition to productive code, software developers write test code which is executed and performs a verification of productive code. To put it differently, test code checks if a productive code behaves as expected. For instance, a software developer implemented the add function of two integers – `public Integer add(Integer a, Integer b) { return a + b; }`. The following test cases are used to verify this function:

- input: a = 5, b = 2 → expected output: 7
- input: a = -5, b = 2 → expected output: -3
- input: a = 5, b = -2 → expected output: 3
- input: a = -5, b = -2 → expected output: -7

A simple test which takes the input (a & b) and compares the actual output which is obtained from the function `add` with the expected output of a test case is called unit test (or module test). Hence, a unit test checks one line of code, a statement, or several individual lines of code (LOC) of a software project[24, ch. 5]. In addition to unit tests, there are various types of tests like integration testing, system testing, object-oriented testing and more[19]. All tests regardless of which type focus on detecting bugs in productive source code and enhancing the confidence in the correctness and reliability of software products. In other words from Homès[18, p. 7]:

*Testing is a set of activities with the objective of identifying failures in a software or system and to evaluate its level of quality, to obtain user satisfaction. It is a set of tasks with clearly defined goals.*

More about specific testing types and the topic testing, in general, is readable in the following recommended references:

#### References for section 2.2.1

- [18] B. Homès, *Fundamentals of Software Testing*. London: ISTE [u.a.], 2012, 342 pp., OCLC: 796194421, ISBN: 978-1-84821-324-1.
- [19] P. C. Jorgensen, *Software Testing, 4th Edition*, 4th ed. Auerbach Publications, Apr. 8, 2016, 494 pp., ISBN: 978-1-4987-8578-5.
- [24] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing, 3rd Edition*, 3rd ed. John Wiley & Sons, Nov. 8, 2011, 240 pp., ISBN: 978-1-118-13315-6.

## 2.2.2 Code Review

Another technique to verify the source code of software products is code review (also known as peer review[22, ch. 1]). This process of reviewing source code is performed by at least two or more software developers and can basically be performed in three steps. First, a software developer makes changes by implementing a task – for example, a new feature or an improvement. Second, after the changes are finalized another co-worker reviews the developed source code by comparing the old version with the new adaptation (to see only the changes). The software developer who performs this review and did not implement the task determines if the new changes are acceptable or not. Finally, depending on the review either the changes are integrated or rejected. This process of a four-eye-principle increases the code quality and reduces the number of bugs in a software product.

A well known open-source tool that assists software developers in performing a code review is the open-source tool – Gerrit<sup>2</sup>. This tool automates the comparison for a developer who performs the code review. Furthermore, the integration and rejection of changes after a review are automated by Gerrit. A step-by-step guide about this tool – Gerrit – and a closer look at the verification technique – Code Review – is given by the book *Learning Gerrit Code Review* by Luca Milanese[22].

## 2.2.3 Verification Tools

There are two different types of verification tools: (i) static code analyzer and (ii) dynamic code analyzer[15, p. 3]. A static code analyzer scans source files, performs an analysis and reports a list of issues (see an overview about the process of a static code analyzer in Figure 2.1). On the other hand, dynamic code analyzers run with the program simultaneously in order to capture criteria during the execution of a program. These criteria are needed to perform an analysis and to create a report of issues. Regardless of which type of verification tool is used, a list of issues is the outcome.

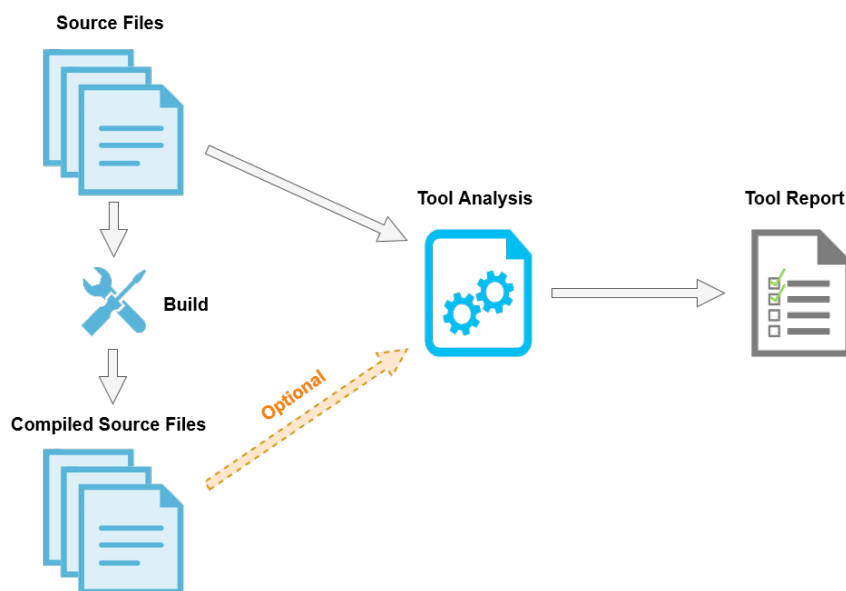


Figure 2.1: Overview of a verification tool that performs a static code analysis. As you can see, a static code analyzer needs source code which is written in several source files. Furthermore, some static code analyzers need compiled source files as well in order to perform an analysis.

<sup>2</sup> Gerrit homepage: <https://www.gerritcodereview.com/>

An issue indicates a problem in the source code of a program that should be reviewed by a software developer. For example, an issue can be a null pointer bug – one of the most common faults in Java[30][35]. Furthermore, memory leaks, buffer overflows/underflows (also known as index out of bounds bugs), resource leaks, synchronization errors (deadlocks, race conditions and livelocks) are known as common bugs and problems in software[33, ch. 2]. These common problems are the reason why verification tools exist. To put it differently, verification tools are created to detect these common problems and to help software developers finding and fixing these issues.

### Definitions

In this master thesis, the following terms are used in order to classify the reported issues by verification tools. By means of this classification criteria are defined which are needed to calculate the effectiveness of verification tools.

<b>True Positive</b>	A bug that exists in the analyzed source code and is detected and reported by a verification tool.
<b>False Positive</b>	A bug that does not exist in the analyzed source code but is detected and reported by a verification tool. False positives are reported due to missing information during the analysis.
<b>False Negative</b>	A bug that exists in the analyzed source code but is not detected and reported by a verification tool.
<b>True Negative</b>	A bug that does not exist in the analyzed source code and is not reported by a verification tool.

This master thesis focuses on selected static code analyzers which are outlined in chapter 4 Selected Verification Tools.





## 3

**Defining Effectiveness**

In order to define the effectiveness of verification tools, three steps have to be performed. First, criteria have to be captured – before, during and after an execution of a verification tool. All captured criteria, for example, the number of reported issues, are listed in section 3.1 and categorized in two groups: source code and verification tool. Second, by means of these criteria, equations are defined in section 3.2 to derive parameters. These parameters are categorized in the same two groups: source code and verification tool. Finally, these parameters are combined in equations to obtain metrics that represent the effectiveness of verification tools (see section 3.3). In addition to well-known equations in literature, self-defined parameters and metrics are listed which were independently declared from the ones in literature.

**3.1 Captured Criteria****3.1.1 Source Code Criteria**

<i>LOC</i>	All LOC from a software project.
<i>numBugs</i>	Defines the number of bugs in a software project which are known by the time of analysis.
<i>numStmt</i>	The number of all code statements of the analyzed software project.

**3.1.2 Verification Tool Criteria**

<i>numFalseNeg</i>	Represents the number of false negatives from a report by a verification tool.
<i>numFalsePos</i>	Represents the number of false positives from a report by a verification tool.
<i>numReportedIssues</i>	The number of reported issues of an analysis by a verification tool.
<i>numTruePos</i>	Represents the number of true positives from a report by a verification tool.

## 3.2 Calculated Parameter

### 3.2.1 Source Code Parameter

#### KLOC

A thousand lines of code (KLOC) from a software project.

$$KLOC = \frac{LOC}{1000} \quad (3.1)$$

#### Ratio Bugs per KLOC:

A well-known parameter as well as metrics of how many bugs are included per KLOC.

$$ratioBugsKLOC = \frac{numBugs}{KLOC} = \frac{numBugs}{\frac{LOC}{1000}} \quad (3.2)$$

### 3.2.2 Verification Tool Parameter

#### Precision:

The parameter *precision* stands for the ratio of true positives out of the number of true positives and false positives[23][11][28]. If the summation ( $numTruePos + numFalsePos$ ) is defined as zero, *precision* is undefined (division by zero; marked with: na – not applicable).

$$precision = \frac{numTruePos}{numTruePos + numFalsePos} \quad (3.3)$$

#### Recall:

Recall (also known as sensitivity[23][11][28]) stands for the ratio of true positives out of the number of all true positives and false negatives. The summation of these two criteria is equal to the amount of bugs that are known by the time of analysis. If  $numBugs$  is defined as zero, *recall* is undefined (division by zero; marked with: na – not applicable).

$$recall = \frac{numTruePos}{numTruePos + numFalseNeg} = \frac{numTruePos}{numBugs} \quad (3.4)$$

**Ratio of False Positives:**

The self-defined parameter  $ratioFalsePos$  stands for the percentage of false positives out of all reported true positives and false positives by a verification tool. If the number of true positives and false positives is zero then as a consequence the ratio of false positives cannot be determined and leads to an undefined result (zero divided by zero).

$$ratioFalsePos = \frac{numFalsePos}{numTruePos + numFalsePos} \quad (3.5)$$

This self-defined  $ratioFalsePos$  is also known as false discovery rate (FDR)[11] and can be calculated by the well-known parameter  $precision$  as well (see Equation 3.6).

$$\begin{aligned} ratioFalsePos &= \frac{numTruePos + numFalsePos - numTruePos}{numTruePos + numFalsePos} \\ &= \frac{\cancel{numTruePos} + numFalsePos}{\cancel{numTruePos} + numFalsePos} - \frac{numTruePos}{numTruePos + numFalsePos} \quad (3.6) \\ &= \underline{\underline{1 - precision}} \end{aligned}$$

**Ratio of True Positives:**

The ratio of true positives out of all bugs (known by the time of analysis) in the source code of a program is declared by the self-defined parameter  $ratioTruePos$ . The number of bugs is defined by software developers. As mentioned before, this number represents only the amount of bugs which are known by the time of analysis. If  $numBugs$  is defined as zero, the ratio of true positives is undefined (division by zero).

This self-defined parameter  $ratioTruePos$  is equal to  $recall$  (see Equation 3.4) but was defined independently. However,  $recall$  is used on the next pages instead of the self-defined parameter  $ratioTruePos$  because  $recall$  is a well-known parameter as well as metric in literature.

$$ratioTruePos = \frac{numTruePos}{numBugs} = recall \quad (3.7)$$

### Rating of Usability:

Questionnaires by means of SurveyMonkey<sup>3</sup> are used in order to rate the usability of graphical user interfaces (GUIs) which are combinable with the selected verification tools of this master thesis. The following question is asked each software developer who participated in this case study.

How user-friendly is the GUI of X?

Each participant could choose between five ratings: Extremely user-friendly, Very user-friendly, Somewhat user-friendly, Not very user-friendly and Not at all user-friendly (see example question with answers in Figure 3.1).

\* 3. How user-friendly is the graphical user interface (GUI) of SonarQube?

<input type="radio"/> Extremely user-friendly	<input type="radio"/> Not very user-friendly
<input type="radio"/> Very user-friendly	<input type="radio"/> Not at all user-friendly
<input type="radio"/> Somewhat user-friendly	

Figure 3.1: Excerpt of the questionnaire to determine the usability of the GUI of SonarQube.

---

<sup>3</sup> The homepage of SurveyMonkey: <https://www.surveymonkey.com/>

### 3.3 Effectiveness Metrics

#### Effectiveness defined by F-measure: $F_\beta$

This metric combines two well-known parameters – *recall* and *precision* – to obtain a single value. Related works calculated this metric to quantify the effectiveness as well as the performance of verification tools (see references [5, sec. VI][23, p. 7]). The F-measure is based on Van Rijsbergen’s effectiveness measure [6, p. 25][16, sec. 1.4][11][29].

$$F_\beta = (1 + \beta^2) \cdot \frac{\textit{precision} \cdot \textit{recall}}{\beta^2 \cdot \textit{precision} + \textit{recall}} \quad (3.8)$$

#### Effectiveness defined by M-measure: $M_\alpha$

In addition to the F-measure, another metric – the M-measure  $M_\alpha$  – is defined to determine the effectiveness of reported issues from verification tools. Criteria of section 3.1 and parameter of section 3.2 are used for this equation  $M_\alpha$  (see Equation 3.10). A comparison between the F-measure and this metric is illustrated in Table 3.3.

**Idea behind this equation:** First statement: the more true positives are detected out of all bugs the higher should the effectiveness of a verification tool be. On the contrary, the second statement: the effectiveness of a verification tool should be decreased if false positives are reported. Last but not least, third statement: one should be able to distinguish between verification tools that do not detect any bug and tools that find bugs.

The first statement is defined by the first term in Equation 3.10 case (iii) where *recall* is directly proportional to the result  $M_\alpha$ . The penalty which decreases the effectiveness of a verification tool is defined by the second term of Equation 3.10 case (iii) and declared in Equation 3.9. This penalty of detected false positives is rated by the weight  $\alpha$  and zero if no false positives are reported. In order to satisfy the third statement, case (ii) of Equation 3.10 is defined and case (iii) is limited higher than case (ii).

In a special case (i) of Equation 3.10 where a software project does not have any bugs (known by the time of analysis) the first term is set to one. In other words, a verification tool should not report issues if there are no bugs in the source code of a software project (known by the time of analysis). The penalty of the second term is in both cases – (i) & (iii) – the same.

$$\textit{penalty} = \begin{cases} 0 & (i) \text{ if } \textit{numFalsePos} = 0 \\ \frac{1}{100} \cdot e^{10 \cdot \alpha \cdot (1 - \textit{precision})} & (ii) \text{ otherwise} \end{cases} \quad (3.9)$$

$$M_\alpha = \begin{cases} \max \left[ 0; 1 - \textit{penalty} \right] & (i) \text{ if } \textit{numBugs} = 0 \\ 0 & (ii) \text{ else if } \textit{numTruePos} = 0 \\ \max \left[ 0.01; \textit{recall} \cdot (1 - \textit{penalty}) \right] & (iii) \text{ otherwise} \end{cases} \quad (3.10)$$

**Choosing weights for F-measure and M-measure:** Selecting a weight for an equation is difficult if one does not know the purpose of this weight, therefore, a closer look at the definition of these weights is provided below. Additionally, Table 3.3 illustrates example values and results of both measures with different weights.

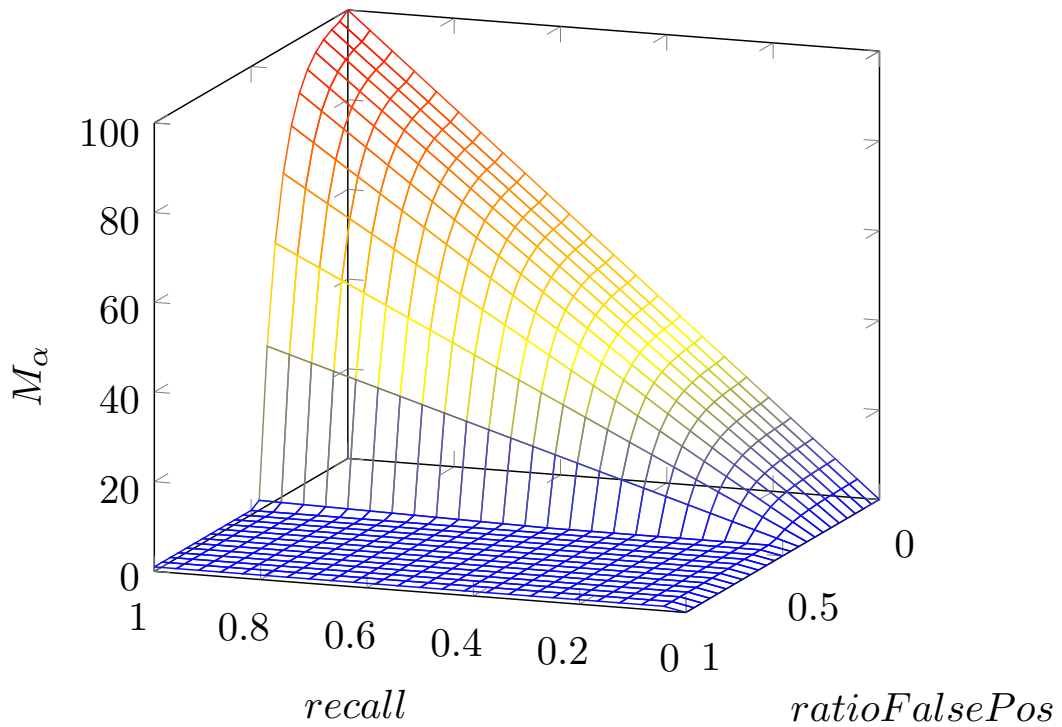
**F-measure:** One needs to decide whether detecting bugs or reporting a small number of false positives is more important. After this decision, there are three cases:

1.  $\beta > 1$  Detecting bugs (high recall) is more important and there is no problem if a verification tool reports many false positives (low precision).
2.  $\beta < 1$  Reporting a small amount or no false positives (high precision) is more important than detecting bugs.
3.  $\beta = 1$  Otherwise, recall and precision are rated equally.

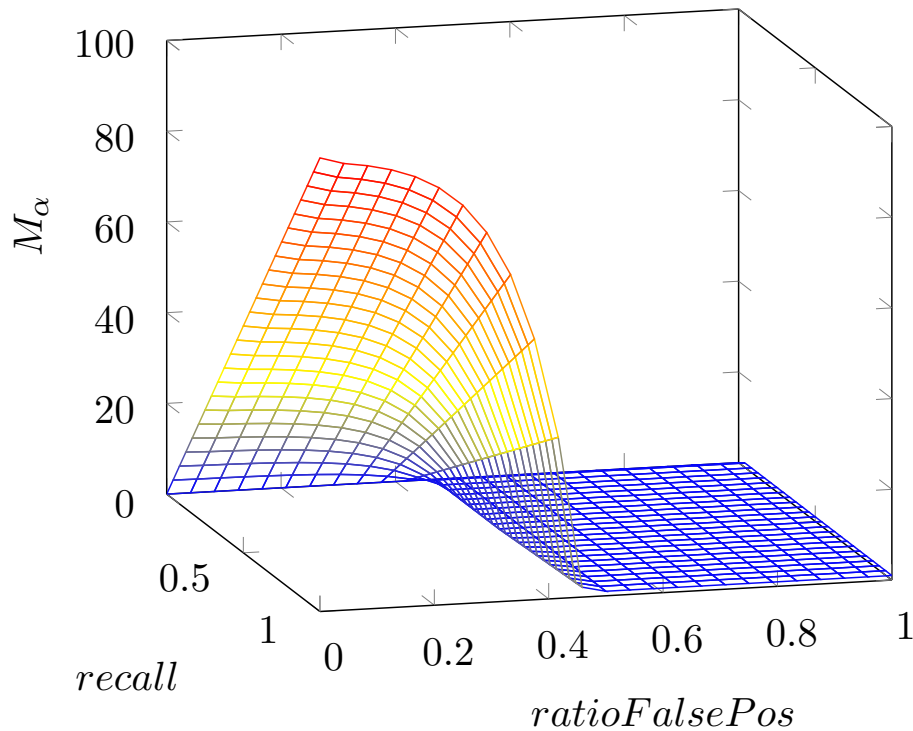
**M-measure:** How high should the penalty for reported false positives be? The higher the weight  $\alpha$  the lower is the result of this metric for the same amount of false positives. Example values and results are demonstrated in Table 3.3. Additionally, a plot of the M-measure is displayed in Figure 3.2.

Table 3.3: Example values and results of effectiveness metrics – M-measure and F-measure – with different weights. To illustrate, if the number of bugs ( $\#Bugs$ ) is 50,  $\#True$  Positives is 25 (25 bugs are detected by a verification tool) and  $\#False$  Positives is zero then according to the M-measure the result is 50% because 25 out of 50 bugs are correctly detected. On the other hand, the result for the F-measure depends on the weight and varies from around 56 to 83%.

Num.	#Bugs	#True Positives	#False Positives	M-measure in [%]			F-measure in [%]		
				$\alpha = 0.5$	$\alpha = 1$	$\alpha = 2$	$\beta = 0.5$	$\beta = 1$	$\beta = 2$
1	10	10	0	100	100	100	100	100	100
2	10	10	1	98.425	97.518	93.839	92.593	95.238	98.039
3	10	10	2	97.699	94.706	71.968	86.207	90.909	96.154
4	10	10	3	96.83	89.949	1	80.645	86.957	94.34
5	10	10	4	95.827	82.588	1	75.758	83.333	92.593
6	10	10	5	94.706	71.968	1	71.429	80	90.909
7	10	10	8	90.772	14.847	1	60.976	71.429	86.207
8	10	10	9	89.319	1	1	58.14	68.966	84.746
9	10	10	10	87.818	1	1	55.556	66.667	83.333
10	10	10	15	79.914	1	1	45.455	57.143	76.923
11	50	25	0	50	50	50	83.333	66.667	55.556
12	50	25	1	49.394	49.265	48.921	81.169	65.789	55.31
13	50	25	2	49.276	48.951	47.8	79.114	64.935	55.066
14	50	25	3	49.146	48.54	45.738	77.16	64.103	54.825
15	50	25	4	49.003	48.014	42.111	75.301	63.291	54.585
16	50	25	5	48.85	47.353	35.984	73.529	62.5	54.348
17	50	25	10	47.914	41.294	1	65.789	58.824	53.191
18	50	25	15	46.74	28.739	1	59.524	55.556	52.083
19	50	25	20	45.386	7.424	1	54.348	52.632	51.02
20	50	25	25	43.909	1	1	50	50	50



(a) As you can see in the back of this plot, the effectiveness rises linearly due to the ratio of detected bugs out of all known ones (definition of *recall*).



(b) Another perspective of the same equation shows the effectiveness of a verification tool decreasing – the more false positives are reported.

Figure 3.2: Visualization of the Equation 3.10 case (ii) and (iii) with the weight  $\alpha$  set to 1. As you can see, the domain for  $M_\alpha$  is defined in percentage from zero to 100% and *recall* and *ratioFalsePos* from zero to one. To demonstrate, a verification tool is to 100% effective in terms of reported issues if no false positives are reported and all bugs which are known by the time of analysis are detected and reported correctly ( $\text{numTruePos} = \text{numBugs} \rightarrow \text{recall} = 1$  and  $\text{numFalsePos} = 0 \rightarrow \text{ratioFalsePos} = 0$  therefore the effectiveness of reported issues  $M_\alpha = 100\%$ ).





## 4

## Selected Verification Tools

All verification tools that are investigated during this master thesis are described in this chapter. These verification tools are selected because of two reasons. First, all of them are free to use and an open-source version of all selected tools, except for SourceMeter, is available. Second, each of the selected tools supports either the programming language (PL) C++, Java, both or even more PLs.

An overview of all selected verification tools is provided in Table 4.1. In addition to this enumeration of all tools, Table 4.1 lists characteristics of each tool, for example, on which operating system (OS) is an execution of the selected verification tools supported. This and more characteristics are outlined in the following sections of this chapter as well: 4.1 PMD, 4.2 SpotBugs (FindBugs), . . . , 4.7 Cppcheck. Each section contains a description, get started and demonstration subsection.

Table 4.1: Overview of all selected verification tools with the used version of each tool. Additionally, the following characteristics for each tool are listed: OS, PL, Variant and History. In the first block, one can see which OS supports the execution of each tool. The second block illustrates which PL is supported by each verification tool. Next, the available variants are listed – either open source, commercial or both. The last block (History) contains information about the first and latest release date. The references of this information are provided in the upcoming sections 4.1 PMD, 4.2 SpotBugs (FindBugs), . . . , 4.7 Cppcheck.

Tool / Version		PMD	Spotbugs	Infer	SonarQube	Source Meter	RATS	Cppcheck
Characteristic		4.0.17	3.1.11	0.15.0	6.7 <sup>4</sup>	8.2.0	2.4	1.84
OS	Windows	×	×		×	×	×	×
	Linux	×	×	×	×	×	×	×
	Mac	×	×	×	×			×
PL <sup>5</sup>	C			×	×	×	×	×
	C++			×	×	×	×	×
	C#				×	×		
	Java	×	×	×	×	×		
	Javascript	×			×			
	Objective-C			×	×			
	Perl				×		×	
	Python				×	×	×	
Variant	Open-Source	×	×	×	×		×	×
	Commercial				×	×		
History	First Release	Jun. 25, 2002	Oct. 25, 2017	Jun. 11, 2015	Dec. 14, 2007	?	?	May. 8, 2007
	Latest Release	Jan. 27, 2019	Jan. 21, 2019	Jun. 5, 2018	Jan. 28, 2019	Dec. 14, 2016	?	Dec. 8, 2018

<sup>4</sup> The used version of SonarQube (web platform) was 6.7 and the version of SonarJava was 4.15.0.12310. The rules to detect bugs in Java source code are defined by the plug-in SonarJava for SonarQube.

<sup>5</sup> Not all supported programming languages are listed for PMD, SourceMeter and SonarQube.

## 4.1 PMD




PMD<sup>6</sup> is an extensible cross-language static code analyzer that mainly focuses on inappropriate programming habits and coding style violations. This analyzer supports many programming languages (PLs): Java, Javascript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity. In addition to these PLs, analyzes of XML and XLS files can be performed[26].






In addition to inappropriate programming habits and coding style violations, PMD includes a copy-paste-detector (CPD) which is based on Rabin-Karp string matching algorithm. This CPD reports code duplications and can be applied to more programming languages than the static code analyzer itself. To illustrate, the CPD finds duplicated code in Java, C, C++, C#, Groovy and further PLs[26].

The first version of PMD (version 0.1) was released on Jun. 25, 2002 and the last release (version 6.11.0) was performed on Jan. 27, 2019, according to the homepage of PMD<sup>6</sup> (visited on Feb. 10, 2019).

### Get Started

PMD can be installed and executed on all three well-known operating systems (OSs) –  Windows,  Linux and  Mac. Furthermore, an integration via a plug-in to many tools is possible – for example Maven or Ant. Moreover, a large number of integrated development environments (IDEs) like Eclipse can be extended via a plug-in with PMD.



If Eclipse or SonarQube is already installed then an easy and user-friendly way to install and run PMD on any well-known OS ( Windows,  Linux and  Mac) is to use Eclipse or SonarQube as the host platform for the PMD plug-in (a step-by-step guide on how to install this plug-in on both host platforms is shown below).

### Via Eclipse

Eclipse is an IDE that is free and open-source software. This IDE supports many PLs and is extendable by a huge number of various plug-ins written by the Eclipse community, other developers or even by oneself



### Installation of PMD via Eclipse:

A detailed step-by-step guide with screenshots is provided in the appendix – see B.1.1.

### Performing an analysis of PMD via Eclipse:

1. Open Eclipse.
2. Create or open a Java project.
3. Right-click on one or several Java projects and select *PMD* → *Check Code* (see Figure B.5).

<sup>6</sup> The homepage of PMD: <https://pmd.github.io/>

## Via SonarQube

SonarQube is a web platform that focuses on continuous code quality. This web platform supports more than 20 PLs including these major languages: C, C++, C#, Java, Javascript, Php and Python. Furthermore, various plug-ins are available in order to enhance the code analysis and to detect even more faults, inappropriate programming habits and coding style violations (see more about SonarQube in section 4.4).



## Installation of PMD via SonarQube:

A detailed step-by-step guide with a screenshot is provided in the appendix – see B.1.1.

## Performing an analysis of PMD via SonarQube:

How to perform an analysis with SonarQube is illustrated step-by-step in section 4.4.

## Demonstration

The following source code (see Listing 4.1) is created to demonstrate an analysis of PMD. Eclipse is used as host platform because it is an easy and user-friendly way to install and run PMD. After an analysis is performed, the results are shown in the violations outline view (see Figure 4.1).

```

1  package at.samplecode.main;
2
3  /**
4   * @author MaAb
5   */
6  public class DemonstrationPMD { // NOPMD by mab on 29.10.18 15:56
7
8      /**
9       * Tries to parse an integer from the transferred string.
10     *
11     * @param number
12     * @return integer value or -1 in case of an error.
13     */
14     public Integer convertString2Integer(final String number) {
15         Integer value = Integer.valueOf(-1); // NOPMD by mab on 29.10.18 15:55
16
17         try {
18             value = Integer.parseInt(number);
19         } catch(NumberFormatException e) {
20         }
21
22         return value;
23     }
24 }
25 }

```

*Listing 4.1: This code excerpt illustrates an empty catch block. It is a bad practice to ignore exceptions because no one knows how often or even if the exception is thrown during the program execution. As a consequence, if this programming habit is applied several times within a software product and many exceptions are thrown, the performance of the program is decreased.*

P	Line	Rule	Error Message
▶	5	EmptyCatchBlock	Avoid empty catch blocks

*Figure 4.1: The violations outline view in Eclipse from the PMD plug-in informs the user to avoid empty catch blocks.*

## 4.2 SpotBugs (FindBugs)

SpotBugs – previously known as FindBugs – is a free and open-source static code analyzer for Java programs. This tool checks for more than 400 bug patterns, according to the homepage of SpotBugs<sup>7</sup>. In addition to these patterns, SpotBugs can be extended by new detectors. To illustrate, one popular detector – *find-sec-bugs* – focuses on security vulnerabilities and is able to detect 128 different vulnerability types (see the homepage of detector: <http://find-sec-bugs.github.io/>).






<https://spotbugs.github.io/>, Lizenz CC BY 4.0

**Why from FindBugs to SpotBugs:** According to an official post from Andrey Loskutov (a team member of FindBugs and SpotBugs), FindBugs is dead because the project lead left the team without admin rights. Therefore, no new team members could be added and no new updates could be released (see quote below[21]).




*... FindBugs was dead because the project lead left us without admin rights and any response for more then a year now. SpotBugs is the natural successor, ...*

The first release of SpotBugs (version 3.1.0) took place on Oct. 25, 2017 and the latest release was performed on Jan. 21, 2019, according to the SpotBugs project on GitHub[31].

### Get Started

This tool can be installed and executed on all three well known operating systems (OSs) –  Windows,  Linux &  Mac.



If Eclipse or SonarQube is already installed then an easy and user-friendly way to install and run SpotBugs on any well-known OS ( Windows,  Linux and  Mac) is to use Eclipse or SonarQube as the host platform for the SpotBugs plug-in (a step-by-step guide on how to install this plug-in on both host platforms is shown below).

### Via Eclipse

Eclipse is an integrated development environment (IDE) that is free and open-source software. This IDE supports many programming languages (PLs) and is extendable by a huge number of various plug-ins written by the Eclipse community, other developers or even by oneself



<https://www.eclipse.org/artwork/>

### Installation of SpotBugs via Eclipse:

A detailed step-by-step guide with screenshots is provided in the appendix – see B.2.1.

### Performing an analysis of SpotBugs via Eclipse:

1. Open Eclipse.
2. Create or open a Java project.
3. Right-click on one or several Java projects and select *SpotBugs* → *Find Bugs* (see Figure B.11).

<sup>7</sup> The homepage of SpotBugs: <https://spotbugs.github.io/>

## Via SonarQube

SonarQube is a web platform that focuses on continuous code quality. This web platform supports more than 20 PLs including these major languages:

C, C++, C#, Java, Javascript, Php and Python. Furthermore, various plug-ins are available in order to enhance the code analysis and to detect even more faults, inappropriate programming habits and coding style violations (see more about SonarQube in section 4.4).



[www.sonarqube.org/community/logos/](http://www.sonarqube.org/community/logos/)

## Installation of SpotBugs via SonarQube:

A detailed step-by-step guide with a screenshot is provided in the appendix – see B.2.1.

## Performing an analysis of SpotBugs via SonarQube:

How to perform an analysis with SonarQube is illustrated step-by-step in section 4.4.

## Demonstration

Eclipse is used as host platform because it is an easy and user-friendly way to install and run SpotBugs. As you can see in Listing 4.2, this source code contains one null pointer bug. After an analysis is performed by SpotBugs, detailed information about a detected bug is shown in an own view of Eclipse (see Figure 4.2).

```

1  package at.faultycode.java.simple;
2
3  import java.util.logging.Logger;
4
5  /**
6   * @author MaAb
7   */
8  public final class IfNullPointerBug {
9
10     /** Default Java logger */
11     private static final Logger LOGGER = Logger
12         .getLogger(IfNullPointerBug.class.getName());
13
14     private IfNullPointerBug() {}
15
16     /**
17      * Entry point of program
18      *
19      * @param args
20      */
21     public static void main(String[] args) {
22         String message = null;
23         if (args.length == 2) {
24             message = message.concat("We have a problem here.");
25         } else {
26             message = "No bug occurred! Try another number of arguments.";
27         }
28
29         LOGGER.info(message);
30     }
31 }

```

Listing 4.2: This source code contains a simple null pointer bug.

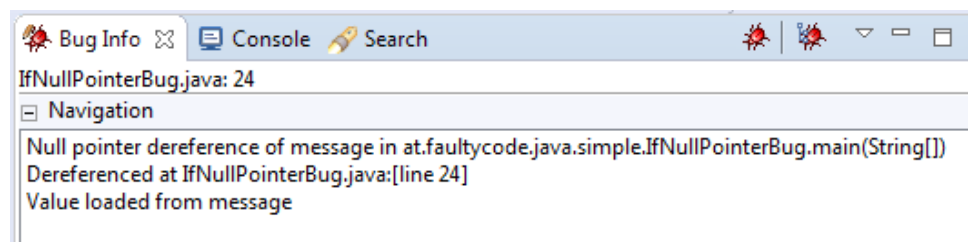


Figure 4.2: As you can see (Bug Info view of Eclipse), SpotBugs correctly detects and reports the null pointer bug of Listing 4.2.

## 4.3 Infer

Infer<sup>8</sup> is a static code analyzer that detects bugs in Java, C, C++ and Objective-C. In Java (including source code for Android applications) the following problems are checked by Infer: null pointer dereferences, resource leaks, annotation reachability, missing lock guards and concurrency race conditions[13]. On the other hand, null pointer dereferences, memory leaks, coding conventions and unavailable application programming interfaces (APIs) are checked by Infer for C, C++ and iOS/Objective-C.



<https://infer.liaohuqiu.net/>

There is one feature that is only supported by Infer: a delta analysis. In general, a static code analyzer scans the whole source code of a software project for each analysis – hence, high execution time even for small code changes. By means of a delta-analysis, static code analyzer reuse data from a previous analysis in order to perform another scan faster. In other words, instead of scanning the whole source code again only the changes have to be scanned – as a consequence, low execution time for small code changes.

Facebook develops Infer and uploaded it as an open-source project on June 9, 2015[14]. The first version of Infer (version 0.1.0) was released on Jun. 11, 2015 and the last release (version 0.15.0) was performed on Jun. 5, 2018, according to the GitHub project of Infer[2].

### Get Started

This tool can be installed via command line on platforms that run a Linux or Mac operating system (OS). Alternatively, a Docker image can be downloaded and used in order to install and execute Infer[12].

#### On Linux 64 Bit

On a 64 bit Linux OS an installation of Infer can be performed with the following commands[12]. XX and Y of the variable VERSION have to be replaced with the desired Infer version. For example, the version that is used for the case study of this master thesis: VERSION=0.15.0.

```
1 VERSION=0.XX.Y; \
2 curl -sSL "https://github.com/facebook/infer/releases/download/v$VERSION/infer-linux64-v$VERSION.tar.xz" \
3 | sudo tar -C /opt -xJ && \
4 ln -s "/opt/infer-linux64-v$VERSION/bin/infer" /usr/local/bin/infer
```

#### On Mac

On Mac, Infer can be installed with one command[12]:

```
1 brew install infer
```

<sup>8</sup> The homepage of Infer: <http://fbinfer.com/>

## Demonstration

The following two listing – Listing 4.3 and Listing 4.4 – show the input for and the output of a code analysis with Infer. After the installation of Infer, the output of Listing 4.4 can be reproduced via the following command: `infer run -- javac Main.java`

```

1 public class Main {
2
3     public static void main(String[] args) {
4
5         String message = createMessage(args.length);
6         if (message.startsWith("User")) {
7             System.out.println(message);
8         }
9     }
10
11    public static String createMessage(int num) {
12        if (num > 0) {
13            return "User typed in " + num + " argument(s).";
14        }
15        return null;
16    }
17 }

```

*Listing 4.3: A null pointer exception may be thrown at line number six. This exception occurs if no command line arguments are specified for this Java program. In this case, `args.length` is zero and is transferred as an actual parameter to the static function `createMessage(...)`. This function returns `null` if the argument `num` is negative or zero.*

```

1 ./Root/Main.java:6: error: NULL_DEREFERENCE
2 object message last assigned on line 5 could be null and is dereferenced at line 6
3
4
5 5. String message = createMessage(args.length);
6 > if (message.startsWith("User")) {
7   System.out.println(message);
8 }
9
10 Summary of the reports
11
12 NULL_DEREFERENCE: 1

```

*Listing 4.4: Infer detects the null pointer dereference from Listing 4.3 correctly. Furthermore, Infer does not report false positives and detects all bugs in this program.*

## 4.4 SonarQube



SonarQube is a web platform that focuses on continuous code quality. This web platform supports more than 20 programming languages (PLs) including these major languages: C, C++, C#, Java, Javascript, Php and Python. Furthermore, various plug-ins are available in order to enhance the code analysis and to detect even more faults, inappropriate programming habits and coding style violations.

In addition to detecting faults, inappropriate programming habits and coding style violations, SonarQube supports a copy-paste-detector (CPD) to detect code duplications and shows the number of passed and failed test cases as well as the code coverage. Furthermore, metrics are displayed like lines of code (LOC) and the number of statements.

The first release of SonarQube (version 1.0.2) took place on Dec. 14, 2007 and the latest release (7.6) was performed on Jan. 28, 2019, according to the homepage<sup>9</sup> of SonarQube (visited on Jan. 13, 2019).

### Get Started

This web platform can be installed and executed on all three well known operating systems (OSs) – Windows, Linux & Mac. Additionally, there is a cloud version (see <https://sonarcloud.io/about/sq>). To illustrate, a step-by-step guide is shown below for a Linux OS.

#### On Ubuntu 18.04.1 64 Bit

1. Download a SonarQube version from the homepage<sup>9</sup>, for example, version 7.6.
2. Extract the \*.zip file to a directory.
3. Open the following folder: <ExtractedDirectory>/sonarqube-7.6/bin/linux-x86-64/
4. Run the script as shown below:

```
1 ./sonar.sh console
```

After these steps, SonarQube should be reachable via <http://localhost:9000> in a browser.

### Demonstration

An analysis can be performed with many tools like Gradle, MSBuild, Maven, Jenkins and Ant. Additionally, the SonarQube Scanner<sup>10</sup> can be used to analyze a software project on any well known OS – Windows, Linux & Mac. To illustrate, a step-by-step guide for performing an analysis via SonarQube Scanner<sup>10</sup> is shown below.

#### On Ubuntu 18.04.1 64 Bit

1. Download SonarQube Scanner<sup>10</sup>.
2. Extract the \*.zip file to a directory.

<sup>9</sup> The homepage of SonarQube: <https://www.sonarqube.org/>

<sup>10</sup> SonarQube Scanner documentation and download: <https://docs.sonarqube.org/display/SCAN/Analyzing+with+SonarQube+Scanner>



3. Open the following folder: <ExtractedDirectory>/sonar-scanner-<version>-linux/bin/
4. Run the script as shown below:

```
1 ./sonar-scanner -Dsonar.projectKey=FirstTestProject -Dsonar.sources=<PathToSourceFiles>
```

After performing these steps, the program SonarQube Scanner displays the following message (see Listing 4.5) – if the analysis was successful. If the execution terminated successfully, a new project is created on the web platform of SonarQube (see Figure 4.3).

```
INFO: -----
INFO: EXECUTION SUCCESS
INFO: -----
INFO: Total time: 7.477s
INFO: Final Memory: 14M/499M
INFO: -----
```

Listing 4.5: Message from SonarQube Scanner if an analysis was successful.

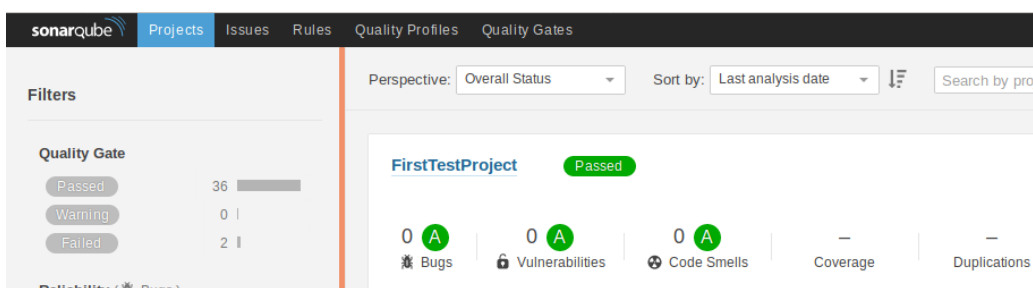


Figure 4.3: Project overview of the web platform of SonarQube.

**Combination of SonarQube with PMD, SourceMeter and SpotBugs:** As mentioned before, SonarQube is extendable with various plug-ins to enhance the code analysis and detect even more faults, inappropriate programming habits and coding style violations. A plug-in can be installed via the Marketplace of SonarQube. After the installation, one can see more quality profiles and more rules. For each PL one quality profile can be set as default and it is used for all projects that are analyzed. One can create its own quality profile and activate one, some or all rules for a specific PL. In other words, it is possible to combine one, two or several quality profiles into one profile. To illustrate, if one wants to perform an analysis of SourceMeter with SonarQube then the built-in quality profile of SourceMeter should be set as default.



The quality profile has to be set before analyzing source code.

## 4.5 SourceMeter

SourceMeter performs a static code analysis for a large number of programming languages (PLs): C, C++, Java, C#, Python, ... . In addition to software bugs, SourceMeter captures many source code metrics like the total number of statements (TNOS), total lines of code (TLOC) and many more. Moreover, it combines and executes free open-source static code analyzers like PMD, FindBugs and Cppcheck during an analysis.





**SourceMeter for Java (version 8.2.0):** Uses two well known static code analyzer: FindBugs version 3.0.0 (released on August 6, 2014; see more about FindBugs in section 4.2) and PMD version 5.2.3 (released on December 21, 2014; see more about PMD in section 4.1). The rules of PMD are carefully selected, according to the product characteristics of SourceMeter.

**SourceMeter for C and C++ (version 8.2.0):** Uses one well known static code analyzer – Cppcheck (version 1.68; see more about Cppcheck in section 4.7).

The exact version and date of the first release could not be determined – even with two tries to contact the developers of SourceMeter. However, the latest release (version 8.2) was performed on Dec. 14, 2016, according to the SourceMeter homepage<sup>11</sup> (visited on Jan. 14, 2019).

### Get Started

This tool can be installed and executed on a  Linux or  Windows operating system (OS). In both cases, a request has to be sent via the homepage of SourceMeter<sup>11</sup> in order to receive the download links. After this download, the compressed file (\*.zip on Windows and \*.tgz on Linux) needs to be extracted and SourceMeter is ready to use.

### Demonstration

An analysis can be performed via the command line or via SonarQube. Therefore, all captured issues and metrics by SourceMeter can be uploaded to SonarQube – prerequisite, the plug-in to extend SonarQube with SourceMeter<sup>12</sup> is installed.

#### On Ubuntu 18.04.1 64 Bit

SourceMeter can be started with the following command for C++ programs. The directory where SourceMeter is stored is represented by the tag <ExtractedDirectory> and <projectName> as the name implies defines the project name for this analysis of SourceMeter.

```
1 <ExtractedDirectory>/SourceMeter-8.2.0-x64-linux/CPP/SourceMeterCPP -projectName=<projectName>
2 -buildScript=build.sh -resultsDir=Results
```

### Via SonarQube

The same command can be used as shown in section 4.4 if Java source files are analyzed. If an analysis for a C++ software project is performed, an additional SonarQube parameter is needed: `-Dsm.cpp.buildfile=build.sh`



The quality profile of SonarQube has to be adjusted before an analysis.



<sup>11</sup> The homepage of SourceMeter: <https://www.sourcemeter.com/>

<sup>12</sup> The homepage of Plug-in to extend SonarQube with SourceMeter: <https://github.com/FrontEndART/SonarQube-plugin-in>

## 4.6 RATS

The Rough Auditing Tool for Security (RATS) scans various programming languages (PLs), including C, C++, Perl, Php and Python. These PLs are checked by RATS for vulnerabilities like race conditions, buffer overflows and critical function calls.

### Get Started

This tool is available for a  Linux and  Windows operating system (OS). A step-by-step guide how to install RATS on Ubuntu 18.04.1 is shown below.

#### On Ubuntu 18.04.1 64 Bit

1. Download RATS from the Google code archive<sup>13</sup>.
2. Extract the downloaded file to a directory.
3. Open the following folder: `<ExtractedDirectory>/rats-<version>/`
4. Run script as illustrated below:

```
1 ./configure && make && sudo make install
2 ./rats
```

### Demonstration

After the installation as described above, RATS can be started via the following commands. The tag `<sourceDirectory>` stands for the path that contains the source files of a software project. In addition to the default output as text (see Listing 4.6), RATS can display the result in an XML and HTML format.

- Console report: `rats -w 3 <sourceDirectory>`
- XML report: `rats -w 3 --xml <sourceDirectory> > rats-report.xml`



SonarQube can read an XML file that contains the output of RATS in XML format. In other words, all reported issues of RATS can be uploaded to SonarQube (via this sonar property: `-Dsonar.cxx.rats.reportPath=rats-report.xml`). Prerequisite: Plug-in `sonar-cxx`<sup>14</sup> is installed.

```
...
Analyzing src/DemonstrationRATS.cpp
src/DemonstrationRATS.cpp:13: High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely. They are prime targets for buffer overflow
attacks.

src/DemonstrationRATS.cpp:17: High: scanf
Check to be sure that the format string passed as argument 2 to this function
call does not come from an untrusted source that could have added formatting
characters that the code is not prepared to handle. Additionally, the format
string could contain '%s' without precision that could result in a buffer
overflow.
...
```

*Listing 4.6: Report from RATS of demonstration source code (see Listing B.1).*

<sup>13</sup> Google code archive: <https://code.google.com/archive/p/rough-auditing-tool-for-security/downloads>




<sup>14</sup> The plug-in `sonar-cxx` is open-source and free to use (see GitHub project: <https://github.com/SonarOpenCommunity/sonar-cxx>).

## 4.7 Cppcheck

C and C++ source code can be analyzed by the static code analyzer Cppcheck<sup>15</sup>. This tool is free to use, open-source and detects a large number of bugs (dead pointers, division by zero, integer overflows, null pointer dereferences, ...). The goal of the open-source community (see Cppcheck homepage<sup>15</sup>) is that the reported list of bugs contains only a few false positives.

The first version of Cppcheck (version 0.1) was released in May. 8, 2007 and the last release (version 1.86) was performed on Dec. 8, 2018, according to the homepage of Cppcheck<sup>15</sup> (visited on Jan. 15, 2019).

### Get Started

Cppcheck can be installed and executed on all three well known operating systems (OSs) –  Linux,  Windows and  Mac. A step-by-step guide on how to install Cppcheck on Ubuntu 18.04.1 is shown below.

### On Ubuntu 18.04.1 64 Bit

In order to install Cppcheck on Ubuntu only one command is necessary:

```
1 sudo apt-get install cppcheck
```

### Demonstration

Cppcheck can be executed via the following commands. The tag `<sourceDirectory>` stands for the path that contains the source files of a software project. Furthermore, the argument `-v` enables an output with more detailed error messages<sup>16</sup>. In addition to the default output as text, Cppcheck can display the result in an XML and HTML.

- Console report: `cppcheck -v --enable=all <sourceDirectory>`
- XML report: `cppcheck -v --enable=all --xml <sourceDirectory> 2> cppcheck-report.xml`



SonarQube can read an XML file that contains the output of Cppcheck in XML format. In other words, all reported issues of Cppcheck can be uploaded to SonarQube (via this sonar property: `-Dsonar.cxx.cppcheck.reportPath=cppcheck-report.xml`). Prerequisite: Plug-in `sonar-cxx`<sup>17</sup> is installed. An example of an analysis is depicted in section 5.2.

<sup>15</sup> The homepage of Cppcheck: <http://cppcheck.sourceforge.net/>

<sup>16</sup> From Cppcheck help: `-v, --verbose` Output more detailed error information.

<sup>17</sup> The plug-in `sonar-cxx` is open-source and free to use (see GitHub project: <https://github.com/SonarOpenCommunity/sonar-cxx>).

## 5

## Case Study

## 5.1 Detect Bug Challenge

As mentioned in section 1.1, a short Java program called *Argument Printer* with 82 lines of code (LOC) – see more statistics in Table 5.1 – is created and analyzed by software developers and by verification tools.

This program contains six bugs (see Table 5.1) and all bugs were detected by a software developer (see Table 1.1) who works for SSI Schaefer Automation GmbH. In addition to the number of detected bugs, 25 minutes and ten seconds were needed by this software developer to scan the source code of the Java program *Argument Printer* (see Table 1.1). Furthermore, this developer is 25 to 34 years old and has already worked with Java for ten to 20 years as well as ten to 20 years of professional experience as a software developer (see Table 5.2).

#Files	2
#Classes	2
#Functions	11
#Statements	42
#TNOS	48
#LOC	82
#TLOC	121
#TLLOC	82
#Bugs	6

Table 5.1: Statistics

In contrast, the best verification tool is able to correctly detect and report four out of six bugs in less than three seconds.

To sum up, from the result of this challenge – software developer vs verification tools – one can see that verification tools do not detect every bug but analyze the source code of the Java program *Argument Printer* faster than software developers. In other words, verification tools detect a good amount of bugs and perform an analysis quite fast.

Table 5.2: Captured criteria of all software developer (software developer A, software developer B, . . . , D) who participated in the detect bugs challenge of the Java program *ArgumentPrinter*.

Developer	A	B	C	D
Criteria				
Age [Years]	25-34	18-24	55-64	18-24
Java Experience [Years]	10-20	1-3	1-3	< 1
Profession Developer [Years]	10-20	3-5	20-30	< 1
”Level of Difficulty <sup>18</sup> ”	3	5	3	4

<sup>18</sup> Level of Difficulty rated by participants: 5 Extremely difficult, 4 Very difficult, 3 Somewhat difficult, 2 not so difficult, 1 not at all difficult

## 5.2 Programs with Common Errors in C++

During this case study, simple C++ programs with common errors are created and analyzed by all verification tools which support the programming language (PL) C++: RATS, Cppcheck, SourceMeter and Infer. For each simple program with common errors an own page, called the overview page, lists the analyzed source code and provides an overview of source code statistics. Furthermore, all captured, classified and calculated criteria are depicted in tables on each overview page (see upcoming pages 50–59 in this section). This process of analyzing and capturing, classifying and calculating criteria for each verification tool which supports the PL C++ is performed step-by-step for the first C++ program: SimpleNullPointerBug (see source code in Listing 5.1).

```

1  ///
2  /// Author: MaAb
3  ///
4
5  namespace faulty {
6
7  struct A {
8      int num;
9  };
10
11 int failsForSure() {
12     A *a = nullptr;
13     return a->num;
14 }
15 }
16
17 ///
18 /// Entry point of program
19 ///
20 int main() {
21     return faulty::failsForSure();
22 }

```

Listing 5.1: SimpleNullPointerBug

#Files	1
#Classes	1
#Functions	2
#Statements	3
#TNOS	0
#LOC	12
#TLOC	11
#TLLOC	9
#Bugs	1

Table 5.3: Statistics

The statistics in Table 5.3 of the source code from Listing 5.1 are captured by means of SonarQube and SourceMeter (see the screenshot of captured metrics from SonarQube in Figure C.1 and from SourceMeter in Figure C.2). The number of files (#Files), #Classes, #Functions, #Statements and the number of lines of code (LOC) are captured by SonarQube. The criteria total number of statements (TNOS), total lines of code (TLOC) and total logical lines of code (TLLOC) are recorded by SourceMeter (all SourceMeter criteria are captured only for the namespace faulty). As you can see, TNOS is not correctly calculated by SourceMeter. Consequently, TNOS is omitted for all C++ programs. The last entry of Table 5.3, #Bugs, represents the number of bugs of the C++ program SimpleNullPointerBug. The number of bugs in this program is one – to be precise one null pointer bug is contained in the C++ program SimpleNullPointerBug because of lines 12 and 13. Which verification tool is able to detect this null pointer bug? The analysis of each selected verification tool that supports the PL C++ is illustrated below.

### RATS

RATS does not detect and report any issues, therefore, all captured criteria (#Reported Issues, #True Positive, #False Positive) are set to zero. On the other hand, the criteria #False Negative is set to one because RATS did not detect the null pointer bug (see classification in Table 5.4).

Tool \ Criteria	#Reported Issues	#True Positives	#False Positives	#False Negatives
RATS	0	0	0	1

Table 5.4: RATS classification of source code SimpleNullPointerBug (C++).

## Cppcheck

Cppcheck correctly detects and reports the null pointer bug of the C++ program SimpleNullPointerBug (see Figure 5.1). Therefore, the number of reported issues and true positives is one and the number of false positives and false negatives is set to zero (see classification in Table 5.5).

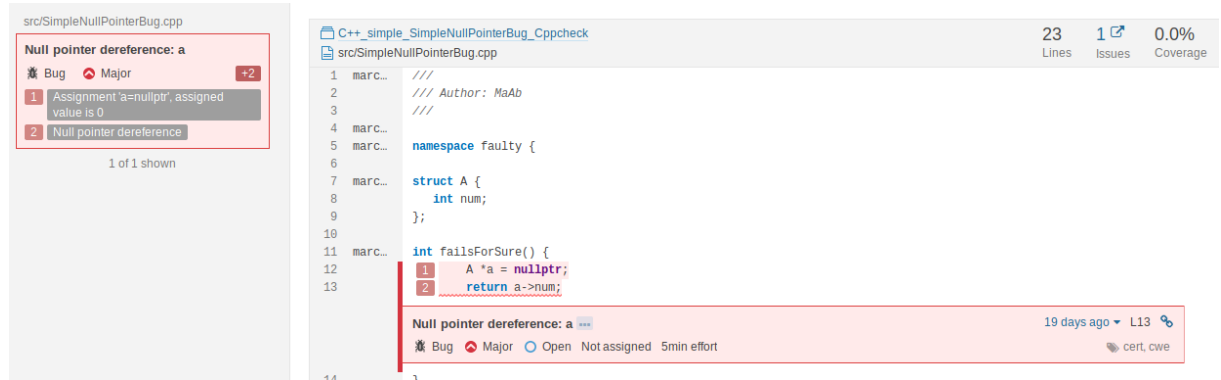


Figure 5.1: Cppcheck analysis for C++ program SimpleNullPointerBug.

Tool \ Criteria	#Reported Issues	#True Positives	#False Positives	#False Negatives
Cppcheck	1	1	0	0

Table 5.5: Cppcheck classification of source code SimpleNullPointerBug (C++).

## SourceMeter

SourceMeter does not detect and report any issues, therefore, all captured criteria (#Reported Issues, #True Positive, #False Positive) are set to zero. On the other hand, the criteria #False Negative is set to one because SourceMeter did not detect the null pointer bug (see classification in Table 5.6).

Tool \ Criteria	#Reported Issues	#True Positives	#False Positives	#False Negatives
SourceMeter	0	0	0	1

Table 5.6: SourceMeter classification of source code SimpleNullPointerBug (C++).

## Infer

Infer correctly detects and reports the null pointer bug of the C++ program `SimpleNullPointerBug` (see Figure 5.2). Therefore, the number of reported issues and true positives is one and the number of false positives and false negatives is set to zero (see classification in Table 5.7).

```

Found 1 issue

src/SimpleNullPointerBug.cpp:14: error: NULL_DEREFERENCE
pointer 'a' last assigned on line 13 could be null and is dereferenced at line 14, column 12.
12.  int main() {
13.      A *a = nullptr;
14. >   return a->num;
15.  }

Summary of the reports

NULL_DEREFERENCE: 1

```

Figure 5.2: Infer analysis for C++ program `SimpleNullPointerBug`.

Criteria \ Tool	#Reported Issues	#True Positives	#False Positives	#False Negatives
Infer	1	1	0	0

Table 5.7: Infer classification of source code `SimpleNullPointerBug` (C++).

A summary of these analyses from the C++ program `SimpleNullPointerBug` is created in section 5.2.1 on page 50. A list of created and analyzed C++ programs with common errors for this master thesis is depicted on the next page.



<b>List of analysed C++ Programs</b>
--------------------------------------

5.2.1	Null Pointer Bugs in C++ . . . . .	50
	C++ – SimpleNullPointerBug . . . . .	50
	C++ – IfNullPointerBug . . . . .	51
	C++ – SwitchNullPointerBug . . . . .	52
	C++ – ForNullPointerBug . . . . .	53
	C++ – WhileNullPointerBug . . . . .	54
	C++ – DoWhileNullPointerBug . . . . .	55
5.2.2	Index Out of Bounds Bugs in C++ . . . . .	56
	C++ – PositiveOutOfBoundsBug . . . . .	56
	C++ – NegativeOutOfBoundsBug . . . . .	57
	C++ – OfByOneBug . . . . .	58
5.2.3	Resource Bugs in C++ . . . . .	59
	C++ – ResourceLeakPartialClose . . . . .	59

## 5.2.1 Null Pointer Bugs in C++

### C++ – SimpleNullPointerBug

```

1  ///
2  /// Author: MaAb
3  ///
4
5  namespace faulty {
6
7  struct A {
8      int num;
9  };
10
11 int failsForSure() {
12     A *a = nullptr;
13     return a->num;
14 }
15 }
16
17 ///
18 /// Entry point of program
19 ///
20 int main() {
21     return faulty::failsForSure();
22 }

```

Listing 5.2: This program contains a null pointer bug.

#Files	1
#Classes	1
#Functions	2
#Statements	3
#LOC	12
#TLOC	11
#TLLOC	9
#Bugs	1

Table 5.8: Statistics

Table 5.9: Captured criteria of the program SimpleNullPointerBug – written in C++. All criteria are listed for each verification tool which supports the PL of the program SimpleNullPointerBug (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	1
#True Positive	0	1	0	1
#False Positive	0	0	0	0
#False Negative	1	0	1	0

Table 5.10: Effectiveness metrics for the program SimpleNullPointerBug – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	100	0	100
<i>precision</i>	na	100	na	100
$F_\beta$	na	100	na	100
$M_\alpha$	0	100	0	100

## C++ – IfNullPointerBug

```

1  ///
2  /// Author: MaAb
3  ///
4
5  namespace faulty {
6
7  struct A {
8      int num;
9  };
10
11 int mayFailOrNot(int argc) {
12     A *a = nullptr;
13     if (argc == 2) {
14         return a->num;
15     }
16     return argc;
17 }
18 }
19
20 ///
21 /// Entry point of program
22 ///
23 int main(int argc, char** argv) {
24     return faulty::mayFailOrNot(argc);
25 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	6
#LOC	15
#TLOC	14
#TLLOC	12
#Bugs	1

Table 5.11: Statistics

Listing 5.3: This program contains a null pointer bug.

Table 5.12: Captured criteria of the program IfNullPointerBug – written in C++. All criteria are listed for each verification tool which supports the PL of the program IfNullPointerBug (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	1
#True Positive	0	1	0	1
#False Positive	0	0	0	0
#False Negative	1	0	1	0

Table 5.13: Effectiveness metrics for the program IfNullPointerBug – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	100	0	100
<i>precision</i>	na	100	na	100
$F_{\beta}$	na	100	na	100
$M_{\alpha}$	0	100	0	100

## C++ – SwitchNullPointerBug

```

1  ///
2  /// Author: MaAb
3  ///
4
5  namespace faulty {
6
7  struct A {
8      int num;
9  };
10
11 int mayFailOrNot(int argc) {
12     A *a = nullptr;
13     switch (argc) {
14         case 1:
15             return 1;
16         case 2:
17             return 2;
18         case 3:
19             return a->num;
20     }
21     return argc;
22 }
23 }
24
25 ///
26 /// Entry point of program
27 ///
28 int main(int argc, char** argv) {
29     return faulty::mayFailOrNot(argc);
30 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	11
#LOC	20
#TLOC	19
#TLLOC	17
#Bugs	1

Table 5.14: Statistics

Listing 5.4: This program contains a null pointer bug.

Table 5.15: Captured criteria of the program SwitchNullPointerBug – written in C++. All criteria are listed for each verification tool which supports the PL of the program SwitchNullPointerBug (C++).

Criteria \ Tool	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	1
#True Positive	0	1	0	1
#False Positive	0	0	0	0
#False Negative	1	0	1	0

Table 5.16: Effectiveness metrics for the program SwitchNullPointerBug – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	RATS	Cppcheck	SourceMeter	Infer
	[%]	[%]	[%]	[%]
<i>recall</i>	0	100	0	100
<i>precision</i>	na	100	na	100
$F_{\beta}$	na	100	na	100
$M_{\alpha}$	0	100	0	100

## C++ – ForNullPointerBug

```

1  ///
2  /// Author: MaAb
3  ///
4
5  namespace faulty {
6
7  struct A {
8      int num;
9  };
10
11 int mayFailOrNot(int argc) {
12     A *a = nullptr;
13     for (int i = 1; i < argc; i++) {
14         return 0;
15     }
16     return a->num;
17 }
18 }
19
20 ///
21 /// Entry point of program
22 ///
23 int main(int argc, char** argv) {
24     return faulty::mayFailOrNot(argc);
25 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	6
#LOC	15
#TLOC	14
#TLLOC	12
#Bugs	1

Table 5.17: Statistics

Listing 5.5: This program contains a null pointer bug.

Table 5.18: Captured criteria of the program ForNullPointerBug – written in C++. All criteria are listed for each verification tool which supports the PL of the program ForNullPointerBug (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	1
#True Positive	0	1	0	1
#False Positive	0	0	0	0
#False Negative	1	0	1	0

Table 5.19: Effectiveness metrics for the program ForNullPointerBug – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	100	0	100
<i>precision</i>	na	100	na	100
$F_\beta$	na	100	na	100
$M_\alpha$	0	100	0	100

## C++ – WhileNullPointerBug

```

1  ///
2  /// Author: MaAb
3  ///
4
5  namespace faulty {
6
7  struct A {
8      int num;
9  };
10
11 int mayFailOrNot(int argc) {
12     A *a = new A();
13     a->num = argc;
14
15     int i = 2;
16     while (i < argc) {
17         a = nullptr;
18         i++;
19     }
20
21     return a->num;
22 }
23 }
24
25 ///
26 /// Entry point of program
27 ///
28 int main(int argc, char** argv) {
29     return faulty::mayFailOrNot(argc);
30 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	9
#LOC	18
#TLOC	19
#TLLOC	15
#Bugs	1

Table 5.20: Statistics

Listing 5.6: This program contains a null pointer bug.

Table 5.21: Captured criteria of the program WhileNullPointerBug – written in C++. All criteria are listed for each verification tool which supports the PL of the program WhileNullPointerBug (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	1
#True Positive	0	1	0	1
#False Positive	0	0	0	0
#False Negative	1	0	1	0

Table 5.22: Effectiveness metrics for the program WhileNullPointerBug – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	100	0	100
<i>precision</i>	na	100	na	100
$F_\beta$	na	100	na	100
$M_\alpha$	0	100	0	100

## C++ – DoWhileNullPointerBug

```

1  ///
2  /// Author: MaAb
3  ///
4
5  namespace faulty {
6
7  struct A {
8      int num;
9  };
10
11 int mayFailOrNot(int argc) {
12     A *a = new A();
13     a->num = argc;
14
15     int i = 2;
16     do {
17         a = argc == 1 ? a : nullptr;
18         i++;
19     } while (i < argc);
20
21     return a->num;
22 }
23 }
24
25 ///
26 /// Entry point of program
27 ///
28 int main(int argc, char** argv) {
29     return faulty::mayFailOrNot(argc);
30 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	9
#LOC	18
#TLOC	19
#TLLOC	15
#Bugs	1

Table 5.23: Statistics

Listing 5.7: This program contains a null pointer bug.

Table 5.24: Captured criteria of the program *DoWhileNullPointerBug* – written in C++. All criteria are listed for each verification tool which supports the PL of the program *DoWhileNullPointerBug* (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	1
#True Positive	0	1	0	1
#False Positive	0	0	0	0
#False Negative	1	0	1	0

Table 5.25: Effectiveness metrics for the program *DoWhileNullPointerBug* – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$ -measure and  $M_\alpha$ -measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	100	0	100
<i>precision</i>	na	100	na	100
$F_\beta$	na	100	na	100
$M_\alpha$	0	100	0	100

## 5.2.2 Index Out of Bounds Bugs in C++

### C++ – PositiveOutOfBoundsBug

```

1  ///
2  /// Author: MaAb
3  ///
4  #include <iostream>
5
6  namespace faulty {
7
8  void failsForSure(int argc, char** argv) {
9      std::cout << argc << " arguments are entered.\n";
10
11     std::cout << argv[argc] << "\n";
12 }
13 }
14
15 ///
16 /// Entry point of program
17 ///
18 int main(int argc, char** argv) {
19     faulty::failsForSure(argc, argv);
20     return 0;
21 }

```

Listing 5.8: This program contains a positive out of bounds bug.

#Files	1
#Classes	0
#Functions	2
#Statements	4
#LOC	10
#TLOC	8
#TLLOC	6
#Bugs	1

Table 5.26: Statistics

Table 5.27: Captured criteria of the program *PositiveOutOfBoundsBug* – written in C++. All criteria are listed for each verification tool which supports the PL of the program *PositiveOutOfBoundsBug* (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	0
#True Positive	0	0	0	0
#False Positive	0	1	0	0
#False Negative	1	1	1	1

Table 5.28: Effectiveness metrics for the program *PositiveOutOfBoundsBug* – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$ -measure and  $M_\alpha$ -measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	0	0	0
<i>precision</i>	na	0	na	na
$F_\beta$	na	na	na	na
$M_\alpha$	0	0	0	0



## C++ – NegativeOutOfBoundsBug

```

1  ///
2  /// Author: MaAb
3  ///
4  #include <iostream>
5
6  namespace faulty {
7
8  void failsForSure(int argc, char** argv) {
9      std::cout << argc << " arguments are entered.\n";
10
11     std::cout << argv[-1] << "\n";
12 }
13 }
14
15 ///
16 /// Entry point of program
17 ///
18 int main(int argc, char** argv) {
19     faulty::failsForSure(argc, argv);
20     return 0;
21 }

```

Listing 5.9: This program contains an negative out of bounds bug.

#Files	1
#Classes	0
#Functions	2
#Statements	4
#LOC	10
#TLOC	8
#TLLOC	6
#Bugs	1

Table 5.29: Statistics

Table 5.30: Captured criteria of the program NegativeOutOfBoundsBug – written in C++. All criteria are listed for each verification tool which supports the PL of the program NegativeOutOfBoundsBug (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	0
#True Positive	0	0	0	0
#False Positive	0	1	0	0
#False Negative	1	1	1	1

Table 5.31: Effectiveness metrics for the program NegativeOutOfBoundsBug – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	0	0	0
<i>precision</i>	na	0	na	na
$F_\beta$	na	na	na	na
$M_\alpha$	0	0	0	0

## C++ – OfByOneBug

```

1  ///
2  /// Author: MaAb
3  ///
4  #include <iostream>
5
6  namespace faulty {
7
8  void failsForSure(int argc, char** argv) {
9      std::cout << argc << " arguments are entered.\n";
10
11     for (int i = 0; i <= argc; ++i) {
12         std::cout << i << ": " << argv[i] << "\n";
13     }
14 }
15 }
16
17 ///
18 /// Entry point of program
19 ///
20 int main(int argc, char** argv) {
21     faulty::failsForSure(argc, argv);
22     return 0;
23 }

```

#Files	1
#Classes	0
#Functions	2
#Statements	6
#LOC	12
#TLOC	10
#TLLOC	8
#Bugs	1

Table 5.32: Statistics

Listing 5.10: This program contains an of by one bug.

Table 5.33: Captured criteria of the program OfByOneBug – written in C++. All criteria are listed for each verification tool which supports the PL of the program OfByOneBug (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	0	1	0	0
#True Positive	0	0	0	0
#False Positive	0	1	0	0
#False Negative	1	1	1	1

Table 5.34: Effectiveness metrics for the program OfByOneBug – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	0	0	0
<i>precision</i>	na	0	na	na
$F_{\beta}$	na	na	na	na
$M_{\alpha}$	0	0	0	0

### 5.2.3 Resource Bugs in C++

#### C++ – ResourceLeakPartialClose

```

1  ///
2  /// Author: MaAb
3  ///
4
5  #include <iostream>
6  #include <stdio.h>
7
8  namespace faulty {
9
10 int mayFailOrNot(int argc) {
11     try {
12         FILE* f = fopen("output.txt", "w");
13         if (f != NULL) {
14             int result = fputs("HelloWorld!", f);
15             if (result != EOF) {
16                 throw 1;
17             }
18             fclose(f);
19         }
20     } catch (...) {
21         std::cout << "Exception occurred!\n";
22         return 1;
23     }
24 }
25 return 0;
26 }
27 }
28
29 ///
30 /// Entry point of program
31 ///
32 int main(int argc, char** argv) {
33     return faulty::mayFailOrNot(argc);
34 }

```

Listing 5.11: This program does not close all resources properly.

#Files	1
#Classes	1
#Functions	2
#Statements	13
#LOC	21
#TLOC	20
#TLLOC	18
#Bugs	1

Table 5.35: Statistics

Table 5.36: Captured criteria of the program ResourceLeakPartialClose – written in C++. All criteria are listed for each verification tool which supports the PL of the program ResourceLeakPartialClose (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	1	1	0	0
#True Positive	0	0	0	0
#False Positive	1	1	0	0
#False Negative	1	1	1	1

Table 5.37: Effectiveness metrics for the program ResourceLeakPartialClose – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	0	0	0
<i>precision</i>	0	0	na	na
$F_{\beta}$	na	na	na	na
$M_{\alpha}$	0	0	0	0

### 5.3 Programs with Common Errors in Java

During this case study, simple Java programs with common errors are created and analyzed by all verification tools which support the programming language (PL) Java: PMD, SpotBugs, SourceMeter, SonarQube and Infer. For each simple program with common errors an own page, called the overview page, lists the analyzed source code and provides an overview of source code statistics. Furthermore, all captured, classified and calculated criteria are depicted in tables on each overview page (see upcoming pages 65–74 in this section). This process of analyzing and capturing, classifying and calculating criteria for each verification tool which supports the PL Java is performed step-by-step for the first Java program: SimpleNullPointerException (see source code in Listing 5.12).

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class SimpleNullPointerException {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12    .getLogger(SimpleNullPointerException.class.getName());
13
14  private SimpleNullPointerException() {}
15
16  /**
17   * Entry point of program
18   *
19   * @param args
20   */
21  public static void main(String[] args) {
22    Object obj = null;
23    final String message = obj.toString();
24
25    LOGGER.info(message);
26  }
27 }

```

Listing 5.12: SimpleNullPointerException

#Files	1
#Classes	1
#Functions	2
#Statements	3
#TNOS	3
#LOC	12
#TLOC	28
#TLLOC	12
#Bugs	1

Table 5.38: Statistics

The statistics in Table 5.38 of the source code from Listing 5.12 are captured by means of SonarQube and SourceMeter (see the screenshot of captured metrics from SonarQube in Figure C.3 and from SourceMeter in Figure C.4). The number of files (#Files), #Classes, #Functions, #Statements and the number of lines of code (LOC) are captured by SonarQube. The criteria total number of statements (TNOS), total lines of code (TLOC) and total logical lines of code (TLLOC) are recorded by SourceMeter. The last entry of Table 5.38, #Bugs, represents the number of bugs of the Java program SimpleNullPointerException. The number of bugs in this program is one – to be precise one null pointer bug is contained in the Java program SimpleNullPointerException because of lines 22 and 23. Which verification tool is able to detect this null pointer bug? The analysis of each selected verification tool that supports the PL Java is illustrated below.

#### PMD

As you can see in Figure 5.3, PMD does not detect the null pointer bug in the Java program: SimpleNullPointerException. Therefore, all three reported issues are classified as false positives. As a consequence, the number of true positives is set to zero and the number of false negatives is set to one (see classification in Table 5.39).

P	Line	created	Rule	Error Message
▶	23	Sat Dec 15 11:53:34 CET 2018	LawOfDemeter	Potential violation of Law of Demeter (object ...
▶	21	Sat Dec 15 11:53:34 CET 2018	MethodArgumentCouldBeFinal	Parameter 'args' is not assigned and could be ...
▶	22	Sat Dec 15 11:53:34 CET 2018	LocalVariableCouldBeFinal	Local variable 'obj' could be declared final

Figure 5.3: PMD analysis for Java program *SimpleNullPointerBug*.

Tool \ Criteria	#Reported Issues	#True Positives	#False Positives	#False Negatives
PMD	3	0	3	1

Table 5.39: PMD classification of source code *SimpleNullPointerBug* (Java).

## SpotBugs

SpotBugs correctly detects and reports the null pointer bug of the Java program *SimpleNullPointerBug* (see Figure 5.4). Therefore, the number of reported issues and true positives is one and the number of false positives and false negatives is set to zero (see classification in Table 5.40).

Figure 5.4 shows the SpotBugs interface. The Bug Explorer window displays a tree view of issues for *SimpleNullPointerBug* (1) [faultyCodeSamples master]. The tree includes: Scary (1), High confidence (1), and Null pointer dereference (1). The selected issue is: Null pointer dereference of obj in at.faultycode.java.simple.SimpleNullPointerBug.main(String[]) [Scary(5), High confidence]. The Bug Info window shows the details for this issue: SimpleNullPointerBug.java: 23, Navigation, Null pointer dereference of obj in at.faultycode.java.simple.SimpleNullPointerBug.main(String[]), Dereferenced at SimpleNullPointerBug.java:[line 23], Value loaded from obj.

Figure 5.4: SpotBugs analysis for Java program *SimpleNullPointerBug*.

Tool \ Criteria	#Reported Issues	#True Positives	#False Positives	#False Negatives
SpotBugs	1	1	0	0

Table 5.40: SpotBugs classification of source code *SimpleNullPointerBug* (Java).

## SourceMeter

SourceMeter correctly detects and reports the null pointer bug of the Java program *SimpleNullPointerBug* (see Figure 5.5). Therefore, the number of true positives is one and the number of false negatives is set to zero. In addition to this true positive, another issue is reported by SourceMeter and classified as false positive because the bug has already been detected by the first issue. The number of false positives is therefore set to one (see classification in Table 5.41).

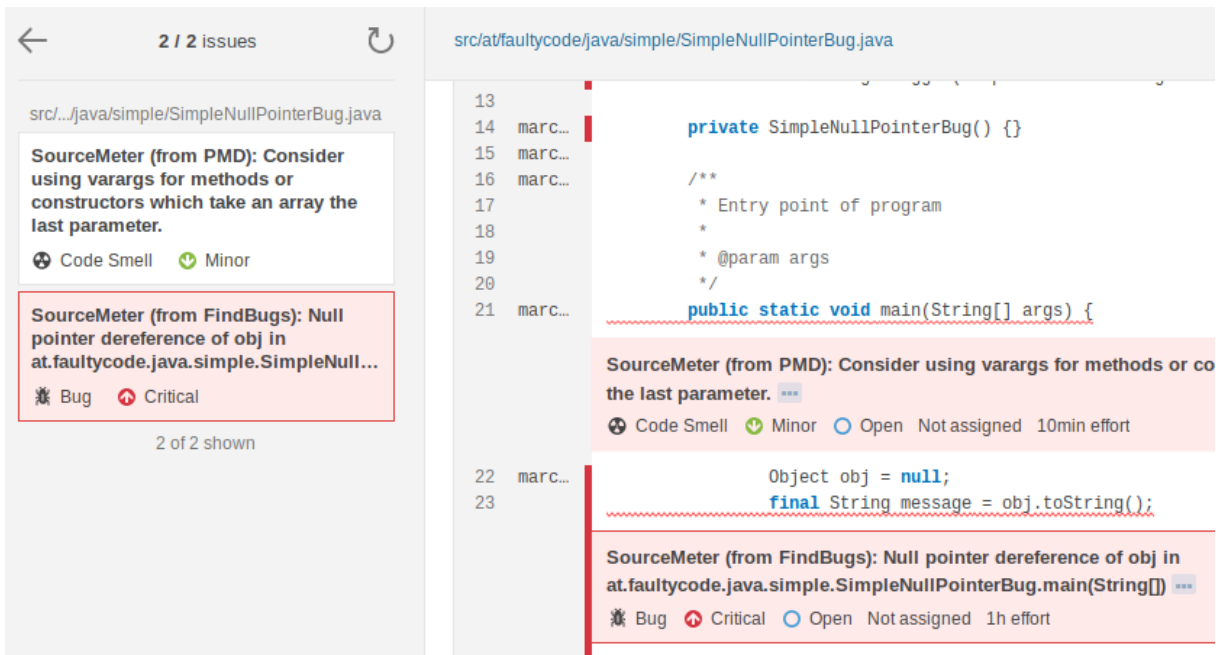


Figure 5.5: SourceMeter analysis for Java program SimpleNullPointerBug.

Criteria \ Tool	#Reported Issues	#True Positives	#False Positives	#False Negatives
SourceMeter	2	1	1	0

Table 5.41: SourceMeter classification of source code SimpleNullPointerBug (Java).

## SonarQube

SonarQube correctly detects and reports the null pointer bug of the Java program SimpleNullPointerBug (see Figure 5.6). Therefore, the number of reported issues and true positives is one and the number of false positives and false negatives is set to zero (see classification in Table 5.42).

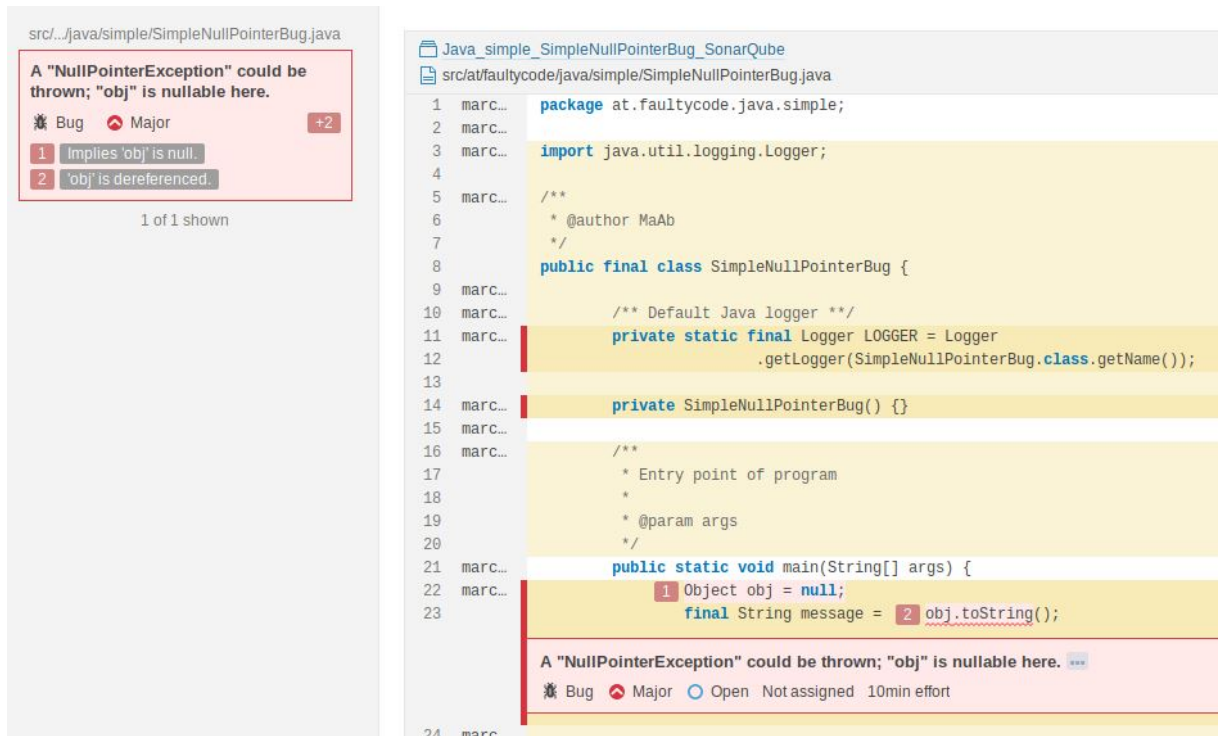
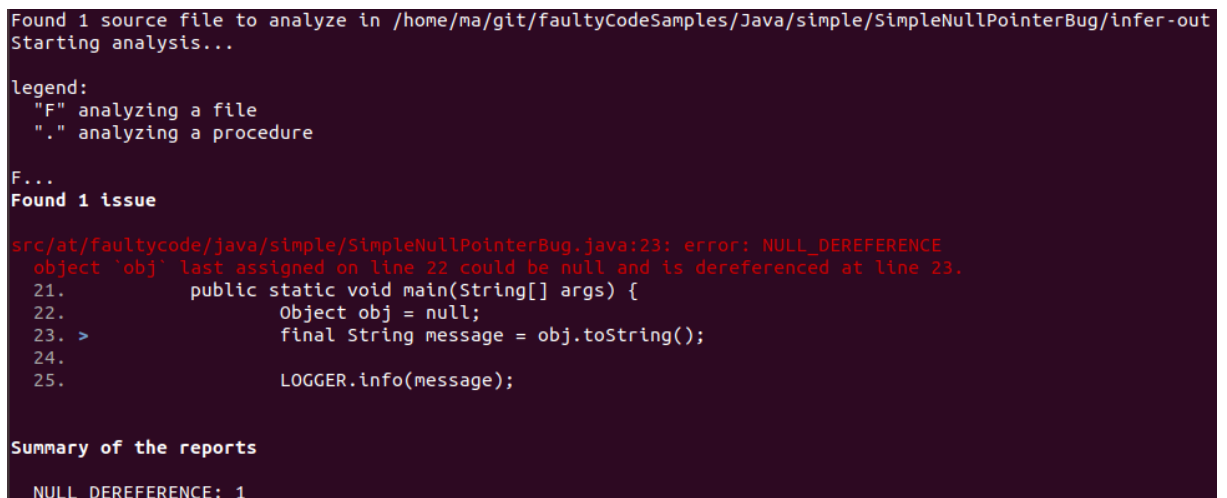
Criteria \ Tool	#Reported Issues	#True Positives	#False Positives	#False Negatives
SonarQube	1	1	0	0

Table 5.42: SonarQube classification of source code SimpleNullPointerBug (Java).

## Infer

Infer correctly detects and reports the null pointer bug of the Java program SimpleNullPointerBug (see Figure 5.7). Therefore, the number of reported issues and true positives is one and the number of false positives and false negatives is set to zero (see classification in Table 5.43).

A summary of these analyses from the Java program SimpleNullPointerBug is created in section 5.3.1 on page 65. A list of created and analyzed Java programs with common errors for this master thesis is depicted on the next page.

Figure 5.6: SonarQube analysis for Java program `SimpleNullPointerBug`.Figure 5.7: Infer analysis for Java program `SimpleNullPointerBug`.

Tool \ Criteria	#Reported Issues	#True Positives	#False Positives	#False Negatives
Infer	1	1	0	0

Table 5.43: Infer classification of source code `SimpleNullPointerBug` (Java).

<b>List of analysed Java Programs</b>
---------------------------------------

5.3.1	Null Pointer Bugs in Java . . . . .	65
	Java – SimpleNullPointerException . . . . .	65
	Java – IfNullPointerException . . . . .	66
	Java – SwitchNullPointerException . . . . .	67
	Java – ForNullPointerException . . . . .	68
	Java – WhileNullPointerException . . . . .	69
	Java – DoWhileNullPointerException . . . . .	70
5.3.2	Index Out of Bounds Bugs in Java . . . . .	71
	Java – PositiveOutOfBoundsBug . . . . .	71
	Java – NegativeOutOfBoundsBug . . . . .	72
	Java – OfByOneBug . . . . .	73
5.3.3	Resource Bugs in Java . . . . .	74
	Java – ResourceLeakPartialClose . . . . .	74



### 5.3.1 Null Pointer Bugs in Java

#### Java – SimpleNullPointerBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class SimpleNullPointerBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12  .getLogger(SimpleNullPointerBug.class.getName());
13
14  private SimpleNullPointerBug() {}
15
16  /**
17   * Entry point of program
18   *
19   * @param args
20   */
21  public static void main(String[] args) {
22  Object obj = null;
23  final String message = obj.toString();
24
25  LOGGER.info(message);
26  }
27 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	3
#TNOS	3
#LOC	12
#TLOC	28
#TLLOC	12
#Bugs	1

Table 5.44: Statistics

Listing 5.13: This program contains a null pointer bug.

Table 5.45: Captured criteria of the program *SimpleNullPointerBug* – written in *Java*. All criteria are listed for each verification tool which supports the PL of the program *SimpleNullPointerBug (Java)*.

Tool \ Criteria	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	3	1	2	1	1
#True Positive	0	1	1	1	1
#False Positive	3	0	1	0	0
#False Negative	1	0	0	0	0

Table 5.46: Effectiveness metrics for the program *SimpleNullPointerBug* – written in *Java*. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$ -measure and  $M$ -measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	PMD [%]	SpotBugs [%]	SourceMeter [%]	SonarQube [%]	Infer [%]
<i>recall</i>	0	100	100	100	100
<i>precision</i>	0	100	50	100	100
$F_\beta$	na	100	66.67	100	100
$M_\alpha$	0	100	1	100	100

## Java – IfNullPointerBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class IfNullPointerBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12  .getLogger(IfNullPointerBug.class.getName());
13
14  private IfNullPointerBug() {}
15
16  /**
17   * Entry point of program
18   *
19   * @param args
20   */
21  public static void main(String[] args) {
22  String message = null;
23  if (args.length == 2) {
24  message = message.concat("We have a problem here.");
25  } else {
26  message = "No bug occurred! Try another number of arguments.";
27  }
28
29  LOGGER.info(message);
30  }
31 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	5
#TNOS	5
#LOC	16
#TLOC	32
#TLLOC	16
#Bugs	1

Table 5.47: Statistics

Listing 5.14: This program contains a null pointer bug.

Table 5.48: Captured criteria of the program *IfNullPointerBug* – written in *Java*. All criteria are listed for each verification tool which supports the PL of the program *IfNullPointerBug (Java)*.

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	5	1	2	1	1
#True Positive	0	1	1	1	1
#False Positive	5	0	1	0	0
#False Negative	1	0	0	0	0

Table 5.49: Effectiveness metrics for the program *IfNullPointerBug* – written in *Java*. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$ -measure and  $M_\alpha$ -measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
	[%]	[%]	[%]	[%]	[%]
<i>recall</i>	0	100	100	100	100
<i>precision</i>	0	100	50	100	100
$F_\beta$	na	100	66.67	100	100
$M_\alpha$	0	100	1	100	100

## Java – SwitchNullPointerBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class SwitchNullPointerBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12    .getLogger(SwitchNullPointerBug.class.getName());
13
14  private SwitchNullPointerBug() {}
15
16  /**
17   * Entry point of program
18   *
19   * @param args
20   */
21  public static void main(String[] args) {
22    String message = null;
23    switch (args.length) {
24      case 1:
25        message = "1 argument is specified.";
26        break;
27      case 2:
28        message = "2 arguments are specified";
29        break;
30      case 3:
31        message = message.replace('2', '3');
32        break;
33      default:
34        message = "No bug occurred! Try another number of arguments.";
35        break;
36    }
37
38    LOGGER.info(message);
39  }
40 }

```

Listing 5.15: This program contains a null pointer bug.

#Files	1
#Classes	1
#Functions	2
#Statements	11
#TNOS	11
#LOC	25
#TLOC	41
#TLLOC	25
#Bugs	1

Table 5.50: Statistics

Table 5.51: Captured criteria of the program SwitchNullPointerBug – written in Java. All criteria are listed for each verification tool which supports the PL of the program SwitchNullPointerBug (Java).

Tool \ Criteria	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	7	1	2	1	1
#True Positive	0	1	1	1	1
#False Positive	7	0	1	0	0
#False Negative	1	0	0	0	0

Table 5.52: Effectiveness metrics for the program SwitchNullPointerBug – written in Java. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	PMD [%]	SpotBugs [%]	SourceMeter [%]	SonarQube [%]	Infer [%]
<i>recall</i>	0	100	100	100	100
<i>precision</i>	0	100	50	100	100
$F_{\beta}$	na	100	66.67	100	100
$M_{\alpha}$	0	100	1	100	100

## Java – ForNullPointerBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class ForNullPointerBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12  .getLogger(ForNullPointerBug.class.getName());
13
14  private ForNullPointerBug() {}
15
16  /**
17   * Entry point of program
18   *
19   * @param args
20   */
21  public static void main(String[] args) {
22  String message = null;
23  for (int i = 0; i < args.length; i++) {
24  message += String.format("%d: %s; ", i, args[i]);
25  }
26
27  if (!message.isEmpty()) {
28  LOGGER.info(message);
29  }
30  }
31 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	5
#TNOS	7
#LOC	16
#TLOC	32
#TLLOC	16
#Bugs	1

Table 5.53: Statistics

Listing 5.16: This program contains a null pointer bug.

Table 5.54: Captured criteria of the program ForNullPointerBug – written in Java. All criteria are listed for each verification tool which supports the PL of the program ForNullPointerBug (Java).

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	4	0	2	2	1
#True Positive	0	0	0	1	1
#False Positive	4	0	2	1	0
#False Negative	1	1	1	0	0

Table 5.55: Effectiveness metrics for the program ForNullPointerBug – written in Java. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
	[%]	[%]	[%]	[%]	[%]
<i>recall</i>	0	0	0	100	100
<i>precision</i>	0	na	0	50	100
$F_{\beta}$	na	na	na	66.67	100
$M_{\alpha}$	0	0	0	1	100

## Java – WhileNullPointerBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class WhileNullPointerBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12      .getLogger(WhileNullPointerBug.class.getName());
13
14  private WhileNullPointerBug() {}
15
16  /**
17   * Entry point of program
18   *
19   * @param args
20   */
21  public static void main(String[] args) {
22      String message = "No arguments are specified!";
23
24      int i = 0;
25      while (i < args.length) {
26          message = null;
27          i++;
28      }
29
30      if (!message.isEmpty()) {
31          LOGGER.info(message);
32      }
33  }
34 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	7
#TNOS	7
#LOC	18
#TLOC	35
#TLLOC	18
#Bugs	1

Table 5.56: Statistics

Listing 5.17: This program contains a null pointer bug.

Table 5.57: Captured criteria of the program WhileNullPointerBug – written in Java. All criteria are listed for each verification tool which supports the PL of the program WhileNullPointerBug (Java).

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	6	0	1	1	1
#True Positive	1	0	0	1	1
#False Positive	5	0	1	0	0
#False Negative	0	1	1	0	0

Table 5.58: Effectiveness metrics for the program WhileNullPointerBug – written in Java. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
	[%]	[%]	[%]	[%]	[%]
<i>recall</i>	100	0	0	100	100
<i>precision</i>	16.67	na	0	100	100
$F_{\beta}$	28.58	na	na	100	100
$M_{\alpha}$	1	0	0	100	100

## Java – DoWhileNullPointerException

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class DoWhileNullPointerException {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12  .getLogger(DoWhileNullPointerException.class.getName());
13
14  private static final String NO_ARGUMENTS = "No arguments are specified!";
15
16  private DoWhileNullPointerException() {}
17
18  /**
19   * Entry point of program
20   *
21   * @param args
22   */
23  public static void main(String[] args) {
24      String message = "";
25
26      int i = 0;
27      do {
28          message = args.length == 0 ? NO_ARGUMENTS : null;
29          i++;
30      } while (i < args.length);
31
32      if (!message.isEmpty()) {
33          LOGGER.info(message);
34      }
35  }
36 }

```

Listing 5.18: This program contains a null pointer bug.

#Files	1
#Classes	1
#Functions	2
#Statements	7
#TNOS	7
#LOC	19
#TLOC	37
#TLLOC	19
#Bugs	1

Table 5.59: Statistics

Table 5.60: Captured criteria of the program *DoWhileNullPointerException* – written in *Java*. All criteria are listed for each verification tool which supports the PL of the program *DoWhileNullPointerException* (*Java*).

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	7	0	1	1	1
#True Positive	0	0	0	1	1
#False Positive	7	0	1	0	0
#False Negative	1	1	1	0	0

Table 5.61: Effectiveness metrics for the program *DoWhileNullPointerException* – written in *Java*. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$ -measure and  $M_\alpha$ -measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
	[%]	[%]	[%]	[%]	[%]
<i>recall</i>	0	0	0	100	100
<i>precision</i>	0	na	0	100	100
$F_\beta$	na	na	na	100	100
$M_\alpha$	0	0	0	100	100

## 5.3.2 Index Out of Bounds Bugs in Java

### Java – PositiveOutOfBoundsBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class PositiveOutOfBoundsBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12    .getLogger(PositiveOutOfBoundsBug.class.getName());
13
14  private PositiveOutOfBoundsBug() {}
15
16  /**
17   * Entry point of program
18   *
19   * @param args
20   */
21  public static void main(String[] args) {
22    final String[] osNames = { "Windows", "Linux", "Mac" };
23    LOGGER.info(osNames[3]);
24  }
25 }

```

Listing 5.19: This program contains a positive index out of bounds error.

#Files	1
#Classes	1
#Functions	2
#Statements	2
#TNOS	2
#LOC	11
#TLOC	26
#TLLOC	11
#Bugs	1

Table 5.62: Statistics

Table 5.63: Captured criteria of the program *PositiveOutOfBoundsBug* – written in *Java*. All criteria are listed for each verification tool which supports the PL of the program *PositiveOutOfBoundsBug* (*Java*).

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	2	1	1	0	0
#True Positive	0	1	0	0	0
#False Positive	2	0	1	0	0
#False Negative	1	0	1	1	1

Table 5.64: Effectiveness metrics for the program *PositiveOutOfBoundsBug* – written in *Java*. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$ -measure and  $M_\alpha$ -measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
	[%]	[%]	[%]	[%]	[%]
<i>recall</i>	0	100	0	0	0
<i>precision</i>	0	100	0	na	na
$F_\beta$	na	100	na	na	na
$M_\alpha$	0	100	0	0	0

## Java – NegativeOutOfBoundsBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class NegativeOutOfBoundsBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12    .getLogger(NegativeOutOfBoundsBug.class.getName());
13
14  private NegativeOutOfBoundsBug() {}
15
16  /**
17   * Entry point of program
18   * @param args
19   */
20  public static void main(String[] args) {
21    final String invalidArg = args[-1];
22    LOGGER.info(invalidArg);
23  }
24 }
25 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	2
#TNOS	2
#LOC	11
#TLOC	26
#TLLOC	11
#Bugs	1

Table 5.65: Statistics

Listing 5.20: This program contains a negative index out of bounds error.

Table 5.66: Captured criteria of the program *NegativeOutOfBoundsBug* – written in *Java*. All criteria are listed for each verification tool which supports the PL of the program *NegativeOutOfBoundsBug* (*Java*).

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	1	1	1	0	0
#True Positive	0	1	0	0	0
#False Positive	1	0	1	0	0
#False Negative	1	0	1	1	1

Table 5.67: Effectiveness metrics for the program *NegativeOutOfBoundsBug* – written in *Java*. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$  and  $M_\alpha$  (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
	[%]	[%]	[%]	[%]	[%]
<i>recall</i>	0	100	0	0	0
<i>precision</i>	0	100	0	na	na
$F_\beta$	na	100	na	na	na
$M_\alpha$	0	100	0	0	0



## Java – OfByOneBug

```

1 package at.faultycode.java.simple;
2
3 import java.util.logging.Logger;
4
5 /**
6  * @author MaAb
7  */
8 public final class OfByOneBug {
9
10  /** Default Java logger */
11  private static final Logger LOGGER = Logger
12  .getLogger(OfByOneBug.class.getName());
13
14  private OfByOneBug() {}
15
16  /**
17  * Entry point of program
18  *
19  * @param args
20  */
21  public static void main(String[] args) {
22  for (int i = 0; i <= args.length; i++) {
23  String arg = args[i];
24  LOGGER.info(i + ": " + arg);
25  }
26  }
27 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	3
#TNOS	5
#LOC	13
#TLOC	28
#TLLOC	13
#Bugs	1

Table 5.68: Statistics

Listing 5.21: This program contains an of by one bug.

Table 5.69: Captured criteria of the program *OfByOneBug* – written in *Java*. All criteria are listed for each verification tool which supports the PL of the program *OfByOneBug* (*Java*).

Tool \ Criteria	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	3	0	3	1	0
#True Positive	0	0	0	0	0
#False Positive	3	0	3	1	0
#False Negative	1	1	1	1	1

Table 5.70: Effectiveness metrics for the program *OfByOneBug* – written in *Java*. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$ -measure and  $M_\alpha$ -measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	PMD [%]	SpotBugs [%]	SourceMeter [%]	SonarQube [%]	Infer [%]
<i>recall</i>	0	0	0	0	0
<i>precision</i>	0	na	0	0	na
$F_\beta$	na	na	na	na	na
$M_\alpha$	0	0	0	0	0

### 5.3.3 Resource Bugs in Java

#### Java – ResourceLeakPartialClose

```

1 package at.faultycode.java.simple;
2
3 import java.io.*;
4 import java.nio.charset.StandardCharsets;
5 import java.nio.file.Files;
6 import java.util.logging.Level;
7 import java.util.logging.Logger;
8
9 /**
10  * @author MaAb
11  */
12 public final class ResourceLeakPartialClose {
13
14     /** Default Java logger */
15     private static final Logger LOGGER = Logger
16         .getLogger(ResourceLeakPartialClose.class.getName());
17
18     private ResourceLeakPartialClose() {}
19
20     /**
21      * Entry point of program
22      *
23      * @param args
24      */
25     public static void main(String[] args) {
26         String fileContent = "HelloWorld!";
27         File out = new File("output.txt");
28
29         try {
30             OutputStream os = Files.newOutputStream(out.toPath());
31             os.write(fileContent.getBytes(StandardCharsets.UTF_8));
32             os.close();
33         } catch (IOException e) {
34             LOGGER.log(Level.SEVERE, "IO exception occurred!", e);
35         }
36     }
37 }

```

#Files	1
#Classes	1
#Functions	2
#Statements	7
#TNOS	7
#LOC	22
#TLOC	38
#TLLOC	22
#Bugs	1

Table 5.71: Statistics

Listing 5.22: This program does not close all resources properly.

Table 5.72: Captured criteria of the program *ResourceLeakPartialClose* – written in *Java*. All criteria are listed for each verification tool which supports the PL of the program *ResourceLeakPartialClose* (*Java*).

Tool \ Criteria	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	8	0	1	1	1
#True Positive	0	0	0	1	1
#False Positive	8	0	1	0	0
#False Negative	1	1	1	0	0

Table 5.73: Effectiveness metrics for the program *ResourceLeakPartialClose* – written in *Java*. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the  $F_\beta$  and  $M_\alpha$  (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	PMD [%]	SpotBugs [%]	SourceMeter [%]	SonarQube [%]	Infer [%]
<i>recall</i>	0	0	0	100	100
<i>precision</i>	0	na	0	100	100
$F_\beta$	na	na	na	100	100
$M_\alpha$	0	0	0	100	100

## 5.4 Software Projects from SSI Schaefer Automation GmbH

### 5.4.1 Project-CA in C++

Performing an analysis of this C++ project was hard. Two out of four selected verification tools of this master thesis could only execute the analysis with errors. First, three subtasks of SourceMeter (WrapperTask, LinkStaticLibsTask and Cppcheck2GraphTask) reported an error (see Figure C.5). Due to these errors, one can conclude that these failed subtasks are the reason why the captured criteria of SourceMeter total lines of code (TLOC) and total logical lines of code (TLLOC) in Table 5.74 are incorrect. Second, Infer could only perform an analysis with the additional command argument `--keep-going`. As the name implies, the analysis is continued even if an error occurs. Most of the errors occurred because many C++ files of this project used features of the QT library and Infer (version 0.15.0) could not analyze some source files of the QT library. On the other hand, no errors occurred during the analysis with RATS and Cppcheck.

#Files	455
#Classes	80
#Functions	2948
#Statements	24698
#LOC	45148
#TLOC	1466
#TLLOC	739
#Bugs	1
#Bugs/KLOC	1

Table 5.74: Statistics

**Bugs:** As you can see in Table 5.75, the selected verification tools – RATS, Cppcheck, SourceMeter and Infer – did not detect many bugs. As a result, the effectiveness of all tools is equal to or going against zero (see Table 5.76). Nevertheless, the employees of SSI Schaefer Automation GmbH who used these tools want to continue using verification tools as an automated second pair of eyes.

Table 5.75: Captured criteria of the program Project-CA – written in C++. All criteria are listed for each verification tool which supports the programming language (PL) of the program Project-CA (C++).

Tool \ Criteria	RATS	Cppcheck	SourceMeter	Infer
#Reported Issues	174	558	31	19
#True Positive	0	1	0	0
#False Positive	174	557	31	19
#False Negative	1	0	1	1

Table 5.76: Effectiveness metrics for the program Project-CA – written in C++. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	RATS [%]	Cppcheck [%]	SourceMeter [%]	Infer [%]
<i>recall</i>	0	100	0	0
<i>precision</i>	0	0.18	0	0
$F_\beta$	na	0.36	na	na
$M_\alpha$	0	1	0	0

### 5.4.2 Project-JA in Java

No problems and errors occurred during the analysis of this project with all verification tools which support the PL Java. PMD, SourceMeter and SonarQube reported a large number of issues compared to SpotBugs and Infer (see row #Reported Issues in Table 5.78). Almost 16000 issues are reported by PMD which means that in every second line of code, there is a problem according to the rules of PMD (see lines of code (LOC) in Table 5.77 versus reported issues from PMD in Table 5.78).

**Bugs:** Infer reported five resource leaks and 33 potential null pointer dereferences (see Table 5.78). Most of the 13 false positives that are reported by Infer are caused by a specific software design – called service-oriented architecture (SOA). SpotBugs did not detect these 33 potential null pointers instead of that, SpotBugs reported another five null pointer dereferences which are true positives. Therefore, Infer and SpotBugs did not detect the same null pointer bugs and were a good combination of tools in terms of reporting a small number of false positives (see higher *precision* of SpotBugs and Infer than PMD, SourceMeter and SonarQube in Table 5.79).

#Files	299
#Classes	310
#Functions	2173
#Statements	11778
#TNOS	11594
#LOC	27068
#TLOC	36222
#TLLOC	26382
#Bugs	141
#Bugs/KLOC	6

Table 5.77: Statistics

Table 5.78: Captured criteria of the program Project-JA – written in Java. All criteria are listed for each verification tool which supports the PL of the program Project-JA (Java).

Criteria \ Tool	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	15599	22	2227	1352	38
#True Positive	9	5	73	69	25
#False Positive	15590	17	2154	1283	13
#False Negative	132	136	68	72	116

Table 5.79: Effectiveness metrics for the program Project-JA – written in Java. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Property \ Tool	PMD [%]	SpotBugs [%]	SourceMeter [%]	SonarQube [%]	Infer [%]
<i>recall</i>	0.06	3.55	51.77	48.94	17.73
<i>precision</i>	0.06	22.73	3.28	5.1	65.79
$F_\beta$	0.06	6.14	6.17	9.24	27.93
$M_\alpha$	1	1	1	1	12.3

### 5.4.3 Project-JB in Java

Another Java program is analyzed and is compared to the first one – Project-JA – ten percent smaller in terms of LOC. Nevertheless, 109 pre-release bugs are detected (see in Table 5.80) by means of all verification tools which support the PL Java and further code review by software developers for similar issues.

**Bugs:** 100 out of these 109 bugs were resources which were not properly released. These resource leakages are tagged with the common weakness enumeration (CWE) id 459 and could cause in the worst case a denial of service problem if too many resources are kept open[8][20, p. 276]. The documentation of Java source code (JavaDoc) describes this problem as well (see Java interfaces `Closeable`<sup>19</sup> and `AutoCloseable`<sup>20</sup>).

#Files	32
#Classes	32
#Functions	146
#Statements	1366
#TNOS	1377
#LOC	2742
#TLOC	3347
#TLLOC	2742
#Bugs	109
#Bugs/KLOC	40

Table 5.80: Statistics

Table 5.81 shows the classification of all reported issues. Infer and Spotbugs do not detect and report many problems (less or equal to five) compared to the other verification tools (more than 400): PMD, SourceMeter and SonarQube. However, SpotBugs and Infer have again a higher *precision* than the other three verification tools (see Table 5.82).

Table 5.81: Captured criteria of the program Project-JB – written in Java. All criteria are listed for each verification tool which supports the PL of the program Project-JB (Java).

Tool \ Criteria	PMD	SpotBugs	SourceMeter	SonarQube	Infer
#Reported Issues	2020	3	553	427	5
#True Positive	1	1	62	101	3
#False Positive	2019	2	491	326	2
#False Negative	108	108	47	8	106

Table 5.82: Effectiveness metrics for the program Project-JB – written in Java. The percentage of true positives out of all bugs in the whole source code of this program is declared by recall. On the contrary, precision stands for the percentage of true positives out of all reported issues. Additionally, the F-measure and M-measure (see Equation 3.8 and 3.10) are listed which combine recall and precision to obtain a single effectiveness value (na stands for not applicable).

Tool \ Property	PMD [%]	SpotBugs [%]	SourceMeter [%]	SonarQube [%]	Infer [%]
<i>recall</i>	0.92	0.92	56.88	92.66	2.75
<i>precision</i>	0.05	33.33	11.21	23.65	60
$F_\beta$	0.09	1.79	18.73	37.68	5.26
$M_\alpha$	1	1	1	1	1.25

<sup>19</sup> `Closeable` JavaDoc[7]: A `Closeable` is a source or destination of data that can be closed. The `close` method is invoked to release resources that the object is holding (such as open files).

<sup>20</sup> `AutoCloseable` JavaDoc[4] excerpt: An object that may hold resources (such as file or socket handles) until it is closed. ... it is recommended to use `try-with-resources` constructions. ...

#### 5.4.4 Beta Releases

During this case study, *Project-CA* and *Project-JA* are released as beta versions. This release was performed before the employees of SSI Schaefer Automation GmbH worked with all selected verification tools of this master thesis. After this beta release, the quality assurance (QA) team started with the verification and validation (V&V) process and reported bugs.

**Reported bugs:** The list of bugs from the QA team is investigated whether the selected verification tools of this master thesis would have detected these bugs as well or not. As mentioned in subsection 5.4.1, the analysis of the C++ *Project-CA* could not be performed without errors by all verification tools that support C++. As a consequence, zero bugs of the C++ *Project-CA* could have been detected by verification tools. However, one bug of the Java *Project-JA* could have been fixed before this beta release – if verification tools, in particular, Infer would have been used to analyze the source code for bugs.

To conclude, verification tools have the potential to detect and report bugs before the actual beta release. However, verification tools cannot replace QA teams. Nevertheless, how effective verification tools are in terms of detecting the same bugs as a QA team, cannot be illustrated with this analyzed data. In order to answer this question, more projects and bugs have to be analyzed. This analysis could be the topic of another bachelor or master thesis and, therefore, it is referred as a point of further research.



## Outcome

In order to quantify the effectiveness of verification tools, metrics which are defined in chapter 3 are calculated. The metric *precision* cannot be determined for each analyzed program because the result of Equation 3.3 is undefined if no true positives and false positives are reported (division by zero). Furthermore, *recall* is undefined as well – in one special case where a program does not contain any software bugs (highly unlikely but still possible). As a consequence of these undefined results, the outcome of another metric – F-measure ( $F_\beta$ ) – cannot be determined. In other words,  $F_\beta$  is not computable if *recall* or *precision* is undefined (see Equation 3.8). However, the M-measure ( $M_\alpha$ ) can be calculated for each program and for each case. Therefore, all four metrics – *recall*, *precision*,  $F_\beta$  and  $M_\alpha$  – are used to quantify the effectiveness of verification tools.

### 6.1 Detect Bug Challenge

As mentioned in section 1.1 and section 5.1, a short Java program called *Argument Printer* with 82 lines of code (LOC) – see more statistics in Table 5.1 – is created and analyzed by software developers and by verification tools.

Figure 6.1 depicts the results in terms of *recall* and *precision*. If a verification tool has a high *recall*, then many or all software bugs which are known by the time of analysis are detected by a verification tool. To illustrate, the metric *recall* of Infer is 66.7% because Infer detects four out of six bugs ( $\frac{4}{6} = \frac{2}{3} = 0.\bar{6}$ ) from the Java program *Argument Printer*. On the other hand, *precision* stands for the ratio of true positives out of the number of true positives and false positives (see Equation 3.3). In other words, *precision* defines how accurate a verification

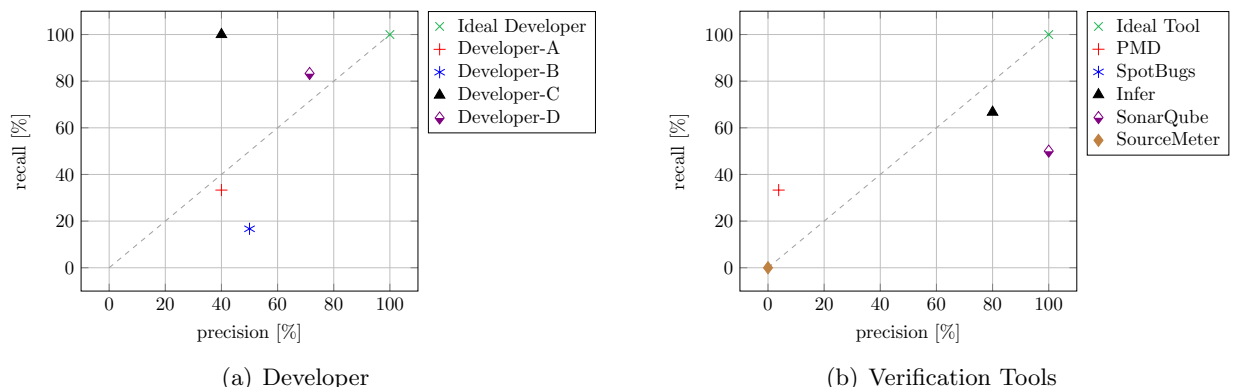


Figure 6.1: Trade-off between recall and precision of the program *Argument Printer*. As you can see, all verification tools that support the programming language (PL) Java are listed. However, the metric *precision* cannot be determined for SpotBugs because no issues are reported. Therefore, SpotBugs is not shown in (b). In addition to the outcome of verification tools, (a) shows the recall and precision trade-off for each developer.

tool can detect real software bugs (true positives). For example, the metric *precision* of SonarQube is 100% because of all reported issues (three out of three  $\rightarrow \frac{3}{3} = 1$ ) from SonarQube are real software bugs (true positives) in the Java program *Argument Printer*. In addition to all verifications tools, the performance in terms of *recall* and *precision* from all software developers of SSI Schaefer Automation GmbH who participated in this bug challenge is depicted in Figure 6.1.

On the other hand, Figure 6.2 depicts all four effectiveness metrics – *recall*, *precision*,  $F_\beta$  and  $M_\alpha$  – for all verification tools that support the PL Java. As mentioned before, the metric *precision* cannot be determined for SpotBugs and, as a consequence,  $F_\beta$  cannot be calculated for SpotBugs either. Furthermore,  $F_\beta$  for SourceMeter is undefined as well – *recall* and *precision* are zero which leads to a division by zero (see Equation 3.8). However, *recall* and  $M_\alpha$  can be calculated for each verification tool.

### Best verification tool of this bug challenge: Infer

Infer detects the most software bugs of the Java program *Argument Printer* (highest *recall* of all verification tools). Furthermore, Infer has the highest value for both effectiveness metrics –  $F_\beta$  and  $M_\alpha$  – compared to all verification tools in Figure 6.2. Moreover, a closer look at Figure 6.1(b) shows that Infer is closer than SonarQube to be an ideal verification tool.

### Best developer of this bug challenge: Developer-B

A closer look at Figure 6.1(a) shows that Developer-B is closer than Developer-A to be an ideal bug detector. Moreover,  $F_\beta$  and  $M_\alpha$  are higher than Developer-A (see Figure D.1) despite the fact that Developer-B does not detect all bugs.

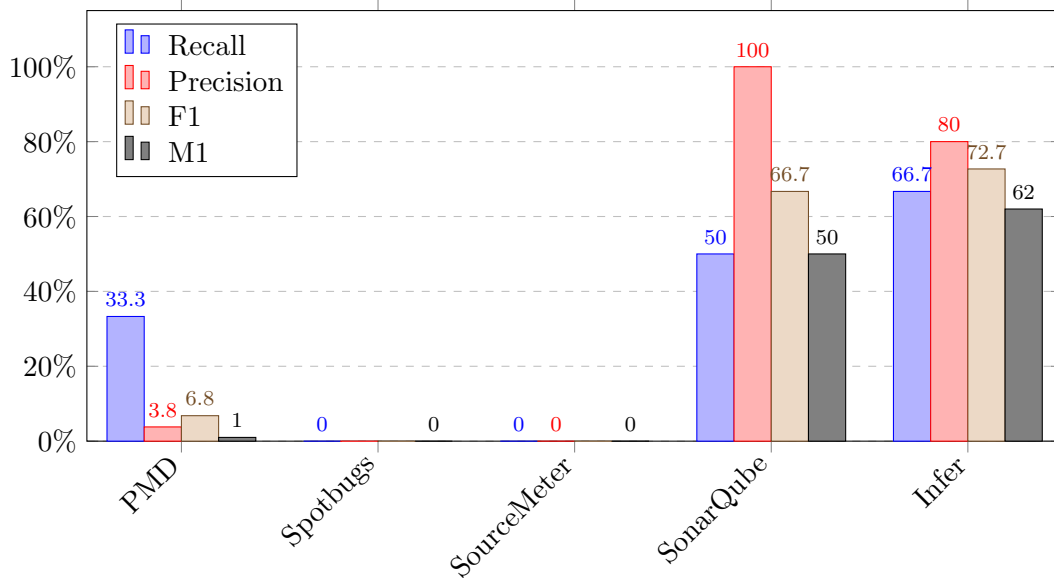


Figure 6.2: All of the selected verification tools that support the PL Java are depicted in this bar chart and recall, precision,  $F_\beta$  and  $M_\alpha$  are used to determine the most effective verification tool for the bug challenge program *Argument Printer* (Java). As you can see, not all of the selected verification tools for Java source code detect bugs. SonarQube and Infer correctly identify and report the most bugs.



## 6.2 Programs with Common Errors

As shown in chapter 5, C++ and Java programs with common errors are analyzed by the selected verification tools which are introduced in chapter 4. To be precise, ten C++ and Java programs with the same content (similar statements) are evaluated in section 5.2 and section 5.3. Each of these C++ and Java programs (in total 20) contains only one software bug.

The effectiveness of a verification tool depends on whether this one bug is correctly detected or not. To quantify the effectiveness of all verification tools, the same four metrics – *recall*, *precision*,  $F_\beta$  and  $M_\alpha$  – are used as in section 6.1. The average of ten programs (Java and C++ separated) is used to quantify the effectiveness per verification tool. As mentioned in the first paragraph of chapter 6, *recall*, *precision* and  $F_\beta$  are not determinable for specific cases. Hence, not all four metrics can be calculated for each program and verification tool (see Figures 6.3, 6.4 and 6.5).

### Best verification tools of all C++ programs with common errors: Infer & Cppcheck

Infer and Cppcheck correctly detect and report the same six out of ten bugs (all null pointer dereferences – see pages 50–59). If only all C++ programs that contain null pointer dereference bugs are averaged then Infer and Cppcheck are ideal verification tools. To put it differently, Infer and Cppcheck detect and report only one issue (one true positive) for all C++ programs that contain null pointer dereference bugs. In contrast, RATS and SourceMeter do not detect any bug of all C++ programs with common errors (zero out of ten – see Figure 6.4 and pages 50–59).

### Best verification tools of all Java programs with common errors: Infer & SonarQube

Infer and SonarQube correctly detect and report the same seven out of ten bugs (all null pointer dereferences and one resource leak bug – see pages 65–74). Both tools detect and report all null pointer dereferences like Infer and Cppcheck for all C++ programs. Therefore, if only all Java programs that contain null pointer dereferences are averaged then Infer and SonarQube are ideal verification tools. On the contrary, PMD, SpotBugs and SourceMeter do not detect as many bugs as Infer and SonarQube. However, at least one bug is detected by PMD, SpotBugs and SourceMeter.

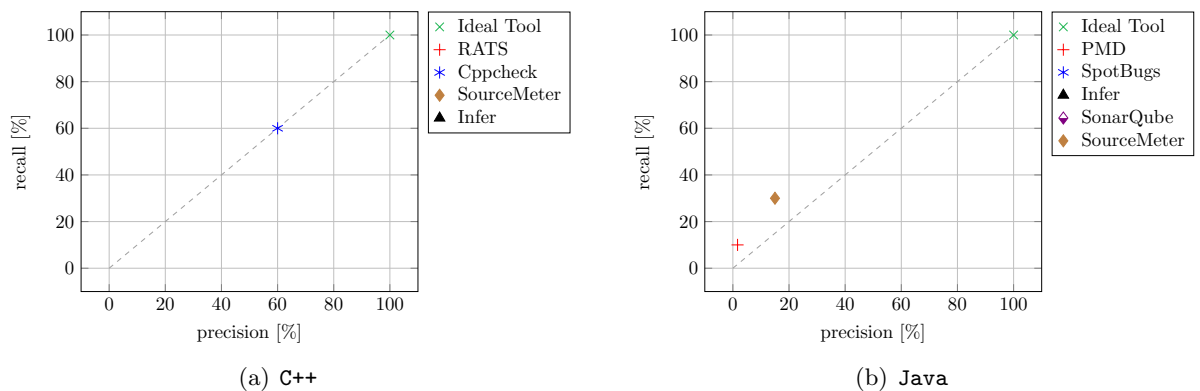


Figure 6.3: Trade-off between recall and precision of all programs with common errors. As you can see, this trade-off can only be depicted for Cppcheck (C++ verification tool) and PMD (Java verification tool).

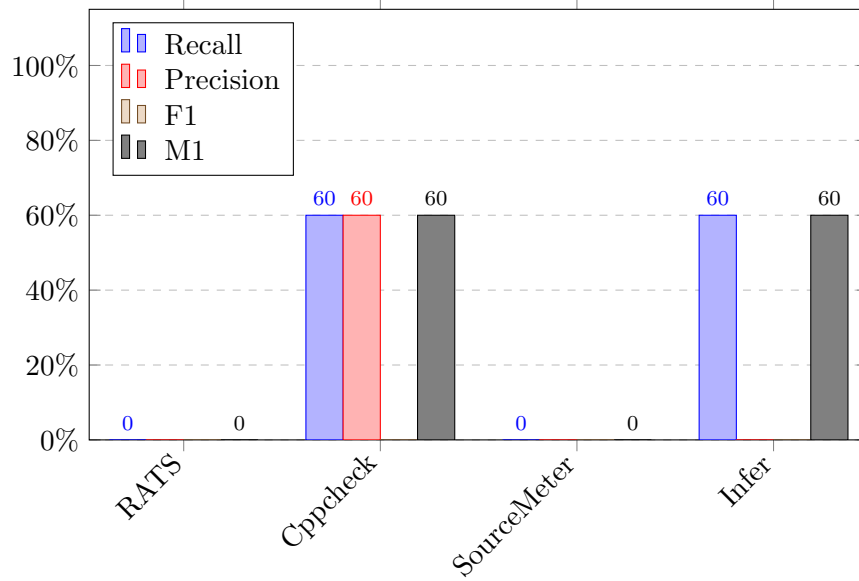


Figure 6.4: All of the selected verification tools that support the programming language (PL) C++ are depicted in this bar chart and the average of recall, precision,  $F_\beta$  and  $M_\alpha$  is used to determine the most effective verification tool for all C++ programs with common errors. As you can see, not all of the selected verification tools for C++ source code detect bugs. Infer and Cppcheck correctly identify and report the most bugs.

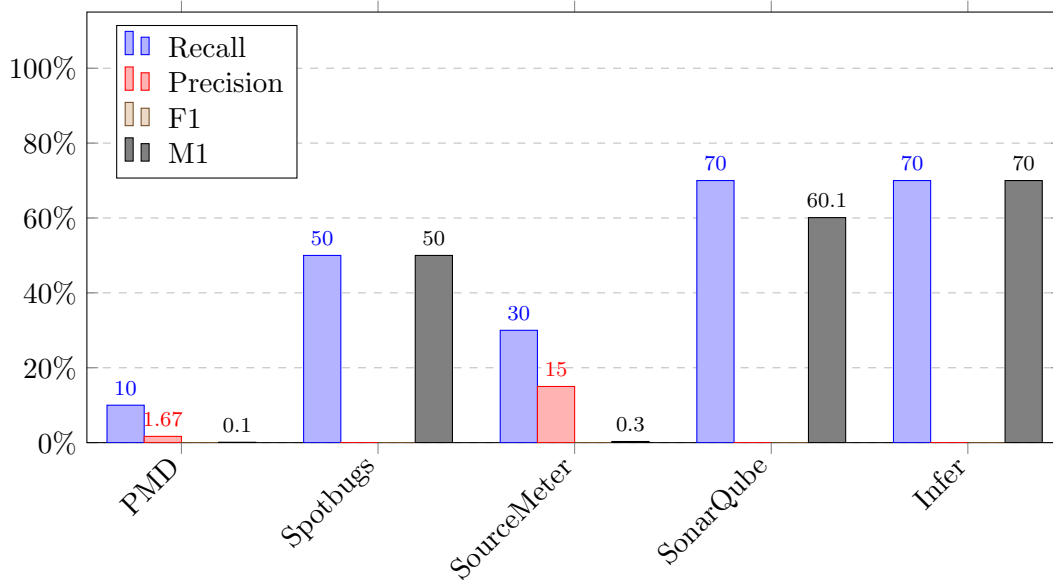


Figure 6.5: All of the selected verification tools that support the PL Java are depicted in this bar chart and the average of recall, precision,  $F_\beta$  and  $M_\alpha$  is used to determine the most effective verification tool for all Java programs with common errors. As you can see, all of the selected verification tools for Java source code detect bugs. SonarQube and Infer correctly identify and report the most bugs.

### 6.3 Software Projects from SSI Schaefer Automation GmbH

In addition to the bug challenge and all programs with common errors, software projects from SSI Schaefer Automation GmbH are analyzed with the selected verification tools which are introduced in chapter 4. One C++ software project (CA) and two Java projects (JA and JB) are investigated (see section 5.4).

The same four metrics – *recall*, *precision*,  $F_\beta$  and  $M_\alpha$  – are used to determine the effectiveness of all verification tools. Furthermore, all software developers who participated in this case study rated the best verification tool via a survey. The output of these three software projects is not 100% accurate because the exact number of bugs is unknown. The bigger and more complex a software project, the more difficult is it to determine the exact number of bugs. Nevertheless, by means of all selected verification tools and further code review by software developers for similar issues and other bugs, a number of faults for each software project is determined (see #Bugs in Tables 5.74, 5.77 and 5.80).

#### Best verification tool of CA project: Cppcheck

All other tools – RATS, SourceMeter and Infer – reported only false positives, therefore, Cppcheck is better by detecting at least one bug (see Figure 6.6(a)).

#### Best verification tool of JA and JB project: SonarQube

On the one hand, Infer is rated the best by the metrics *precision* and  $M_\alpha$ . On the other hand, SonarQube reaches the highest value for *recall* and  $F_\beta$  (see Table 5.79 and Table 5.82). In other words, SonarQube detects the most software bugs in both Java projects (see the highest *recall* in Figure 6.6(b)). Furthermore, JA and JB are not critical software projects. Hence, a tool that detects the most bugs is classified as the best.

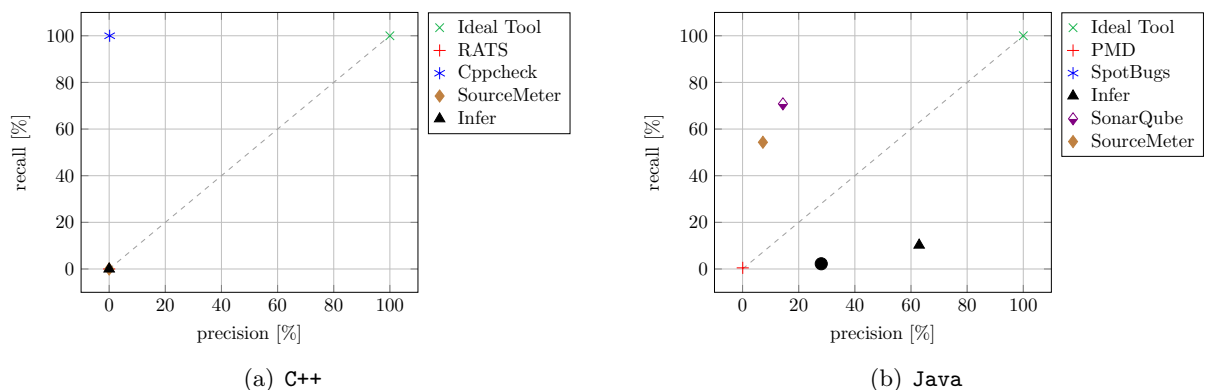


Figure 6.6: Trade-off between recall and precision of all software project of SSI Schaefer Automation GmbH. As you can see, this trade-off can only be depicted for Cppcheck (C++ verification tool) and PMD (Java verification tool).

In addition to analyzing these three software projects, associates of SSI Schaefer Automation GmbH worked with all verification tools which are investigated by means of this master thesis (see an enumeration and overview of all tools in Table 4.1). This gained experience is rated by all employees who participated in this case study via a survey.

**Best verification tool of CA project rated by developers: Cppcheck**

For the software project CA – written in C++ – Cppcheck is rated as the best verification tool out of all verification tools which support the programming language (PL) C++ – RATS, Cppcheck, SourceMeter and Infer. As mentioned in subsection 5.4.1, two out of four verification tools – SourceMeter and Infer – cannot perform the analysis of the software project CA without errors (huge impact for this rating).

**Best verification tool of JA and JB project rated by developers: SonarQube**

SonarQube was rated as the best verification tool because it detects many bugs and, in particular, a detailed description is provided by each issue. Some descriptions contain, in addition to the explanation of an issue, non-compliant and compliant code examples. Some participants stated that these code examples are the reason why SonarQube is rated as the best verification tool. Furthermore, an employee commented that processing even a large amount of false positives is easy with SonarQube because of the filter and bulk change feature of SonarQube.

## 7

**Conclusion**

The effectiveness of selected verification tools has been determined for C++ and Java programs; to be precise, for a small Java program with 82 lines of code (LOC), ten simple C++ and Java programs with common errors and three software projects (C++ and Java) from SSI Schaefer Automation GmbH – an international company. Each program and software project has been analyzed by selected verification tools which support the same programming language (PL):

- **C++:** RATS, Cppcheck, SourceMeter and Infer
- **Java:** PMD, SpotBugs, SoureMeter, SonarQube and Infer

In addition to determining the effectiveness, a new metric – the M-measure ( $M_\alpha$ ) – has been defined which is calculable for more cases than the F-measure ( $F_\beta$ ). As mentioned in chapter 6, the metrics *recall*, *precision* and  $F_\beta$  cannot be calculated in specific cases. To illustrate, if the source code of a software project does not contain any bug then *recall* is undefined (division by zero – see Equation 3.4). Furthermore, if a verification tool does not report any issues then the metric *precision* is undefined (division by zero – see Equation 3.3). As a consequence, if either one is undefined or if both (*recall* and *precision*) are zero then  $F_\beta$  is not definable either (see Equation 3.8). In contrast,  $M_\alpha$  can be calculated in such cases (see Equation 3.10).

Using verification tools or code analyzers has three main benefits. First, verifying software is crucial before new products or releases are shipped to users because software bugs can occur on each line of code. Furthermore, a bug leading to a denial of service or an unexpected result or behavior decreases the user satisfaction of software. Verification tools can be used to reduce this risk by locating bugs before software products are released. Second, in addition to detecting bugs, verification tools can be used to teach all, and in particular entry-level, software developers to avoid or rather reduce the number of bugs. Additionally, many verification tools report not only bugs but also code improvements in terms of performance and readability. Hence, software developers are taught and reminded to improve the code quality by checking the reported issues of a verification tool. Regardless of which code analyzer is used and how effective it is, by using any tool a software developer takes time to review the source code of a software project another time. This additional check might locate even more bugs which would have not been detected without this additional review. Last but not least, as simple as fixing a bug can be, it takes time because a change in the source code of a software project has to go through the whole build and release process (build a new version, run test cases and perform release). Even for a small change, these tasks may take approximately 15 to 30 minutes. Moreover, each bug fix costs resources as well as money. For these reasons, detecting and fixing bugs before releasing a software product is more efficient.

On the other hand, disadvantages of verification tools include set-up, configuration, maintenance, memory consumption, execution time and a high number of false positives. No verification tool is ideal but by addressing these disadvantages step by step it is possible to profit from the benefits of verification tools.

To sum up, verification tools do not detect every bug and it might be tedious to run the first analysis. However, these tools help to improve the code quality as well as to foster the confidence in the correctness and reliability of software products. Therefore, why not perform an additional check of source code with an automated pair of eyes?

# Appendices



# Appendix Chapter 1 Introduction

## A.1 Section 1.1 Motivation

```
1 package at.faultycode.java.advanced;
2
3 import java.util.Random;
4 import java.util.logging.Level;
5 import java.util.logging.Logger;
6
7 /**
8  * @author MaAb
9  */
10 public class Main {
11
12     /** Default Java logger */
13     private static final Logger LOGGER = Logger.getLogger(Main.class.getName());
14
15     private static final Random RANDOM = new Random();
16
17     /**
18      * Entry point of program
19      *
20      * @param args
21      * @throws InterruptedException
22      */
23     public static void main(String[] args) {
24         LOGGER.info("Start program...");
25         final ArgumentPrinter argPrinter = new ArgumentPrinter();
26         for (int i = 0; i < 5; i++) {
27             if (generateRandomInteger() > 0)
28                 LOGGER.log(Level.INFO, "Log file path: '{0}'", argPrinter.initLogFile().toString());
29             Object[] testArguments = i == 0 ? args : generateRandomArgs();
30             argPrinter.setArguments(testArguments, 5);
31             argPrinter.logAll();
32             LOGGER.log(Level.INFO, "{0}: iteration..", i);
33         }
34     }
35
36     private static Object[] generateRandomArgs() {
37         int numArgs = RANDOM.nextInt(10);
38         if (numArgs > 0) { return new Object[numArgs]; }
39         return null;
40     }
41
42     private static Integer generateRandomInteger() {
43         Integer randomValue = Integer.valueOf(RANDOM.nextInt(2));
44         if (randomValue > 0)
45             return randomValue;
46         return null;
47     }
48 }
```

Listing A.1: Source code of the program: Argument Printer written in Java (file Main.java).



```

1 package at.faultycode.java.advanced;
2
3 import java.io.File;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6 import java.nio.charset.Charset;
7 import java.util.logging.Level;
8 import java.util.logging.Logger;
9
10 /**
11  * @author MaAb
12  */
13 public class ArgumentPrinter {
14
15     /** Default Java logger */
16     private static final Logger LOGGER = Logger.getLogger(ArgumentPrinter.class.getName());
17
18     private Object[] args;
19     private Integer maxArgs;
20
21     public void setArguments(final Object[] args, Integer maxArgs) {
22         this.args = new String[maxArgs]; this.maxArgs = maxArgs;
23         initArgs(args, maxArgs);
24     }
25
26     private synchronized void initArgs(final Object[] args, final Integer maxNumArgs) {
27         for (int i = 0; i < maxNumArgs; i++)
28             this.args[i] = args[i];
29     }
30
31     public File initLogFile() {
32         File logFile = new File("logfile.log");
33         try {
34             FileOutputStream fos = new FileOutputStream(logFile);
35             fos.write("LogFile: ArgumentPrinter".getBytes(Charset.defaultCharset()));
36             fos.close();
37         } catch (IOException e) {
38             // Ignore
39             return null;
40         }
41         return logFile;
42     }
43
44     /**
45     * Logs the number of arguments and the content of each argument.
46     */
47     public void logAll() {
48         logNumberOfArguments(); logAllArguments();
49     }
50
51     private void logNumberOfArguments() {
52         String formatString = createFormatStringNumArgs();
53         if (!formatString.isEmpty())
54             LOGGER.log(Level.INFO, formatString, args.length);
55     }
56
57     private void logArgumentContent(Integer index) {
58         LOGGER.log(Level.INFO, "Arg[" + index + "]: '{0}'", args[index].toString());
59     }
60
61     private void logAllArguments() {
62         for (int i = 0; i <= maxArgs; i++)
63             logArgumentContent(i);
64     }
65
66     private String createFormatStringNumArgs() {
67         if (args.length == 1) return "{0} argument is entered!";
68         else if (args.length > 1) return "{0} argument(s) are entered!";
69         return null;
70     }
71 }

```

Listing A.2: Source code of the program: Argument Printer written in Java (file ArgumentPrinter.java).

## B

## Appendix Chapter 4 Selected Verification Tools

### B.1 Section 4.1 PMD

#### B.1.1 Section 4.1 Get Started

Installation of PMD via Eclipse:

1. Open Eclipse.
2. Select *Help* → *Install New Software...* via the top menu bar.

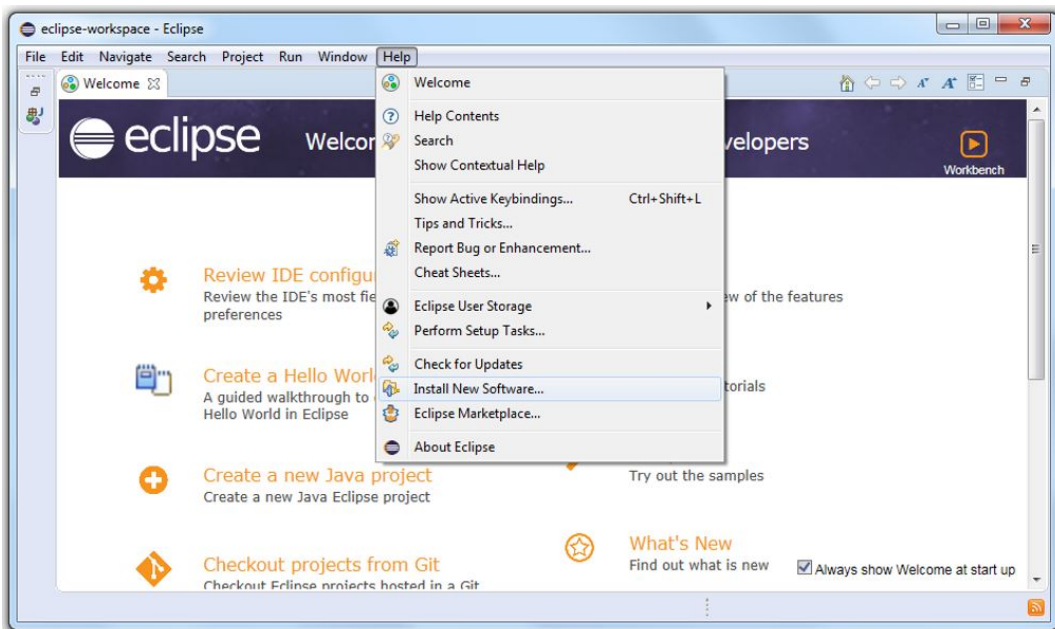


Figure B.1: Installing PMD via Eclipse: second step, open install new software dialog.

3. Click on the button *Add...* within the dialog *Install*.

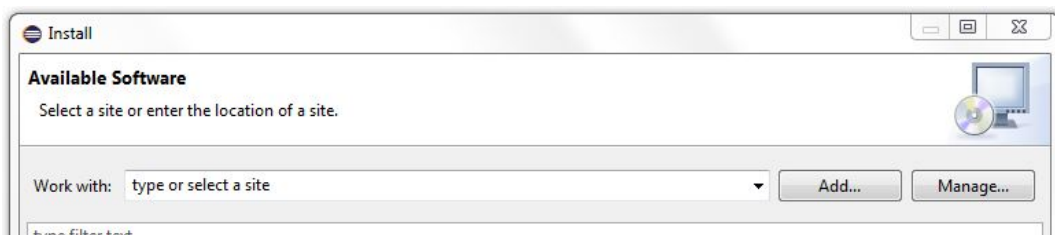


Figure B.2: Installing PMD via Eclipse: third step, open add repository dialog.

- Another dialog *Add Repository* is opened. Enter *PMD* for the *Name* field and <https://dl.bintray.com/pmd/pmd-eclipse-plugin/updates/> for the *Location* field. Submit these changes by clicking on the button *OK*.

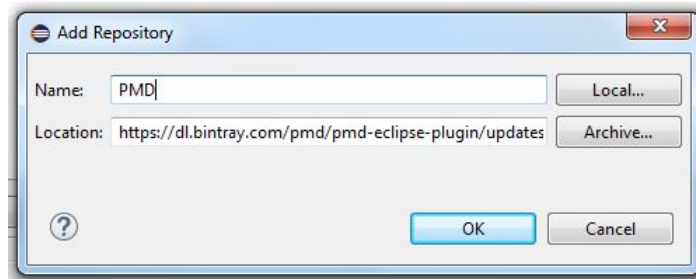


Figure B.3: Installing PMD via Eclipse: fourth step, define PMD repository.

- Select *PMD Plug-in* and follow the install wizard.

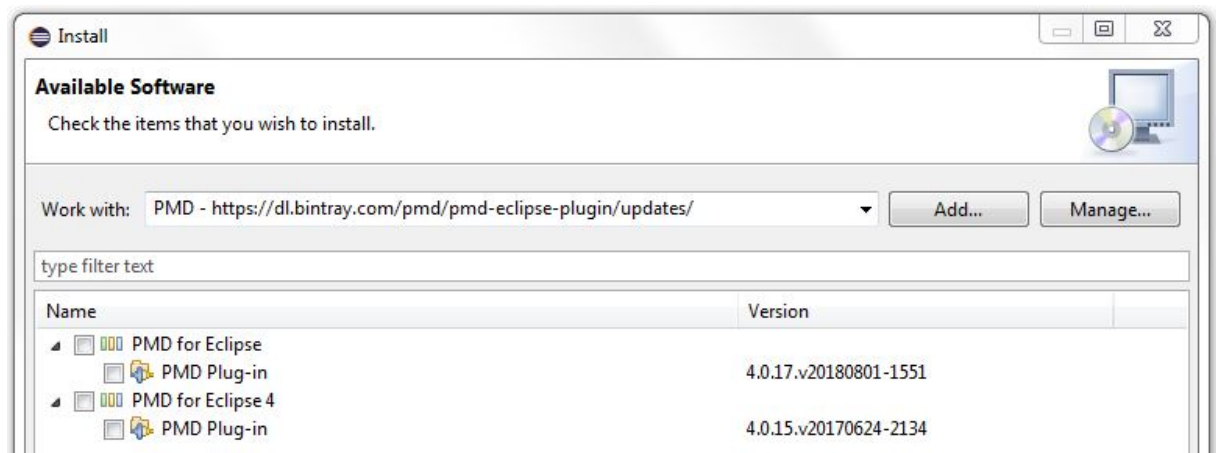


Figure B.4: Installing PMD via Eclipse: fifth step, select PMD plug-in.



After the installation, a restart of Eclipse is necessary.

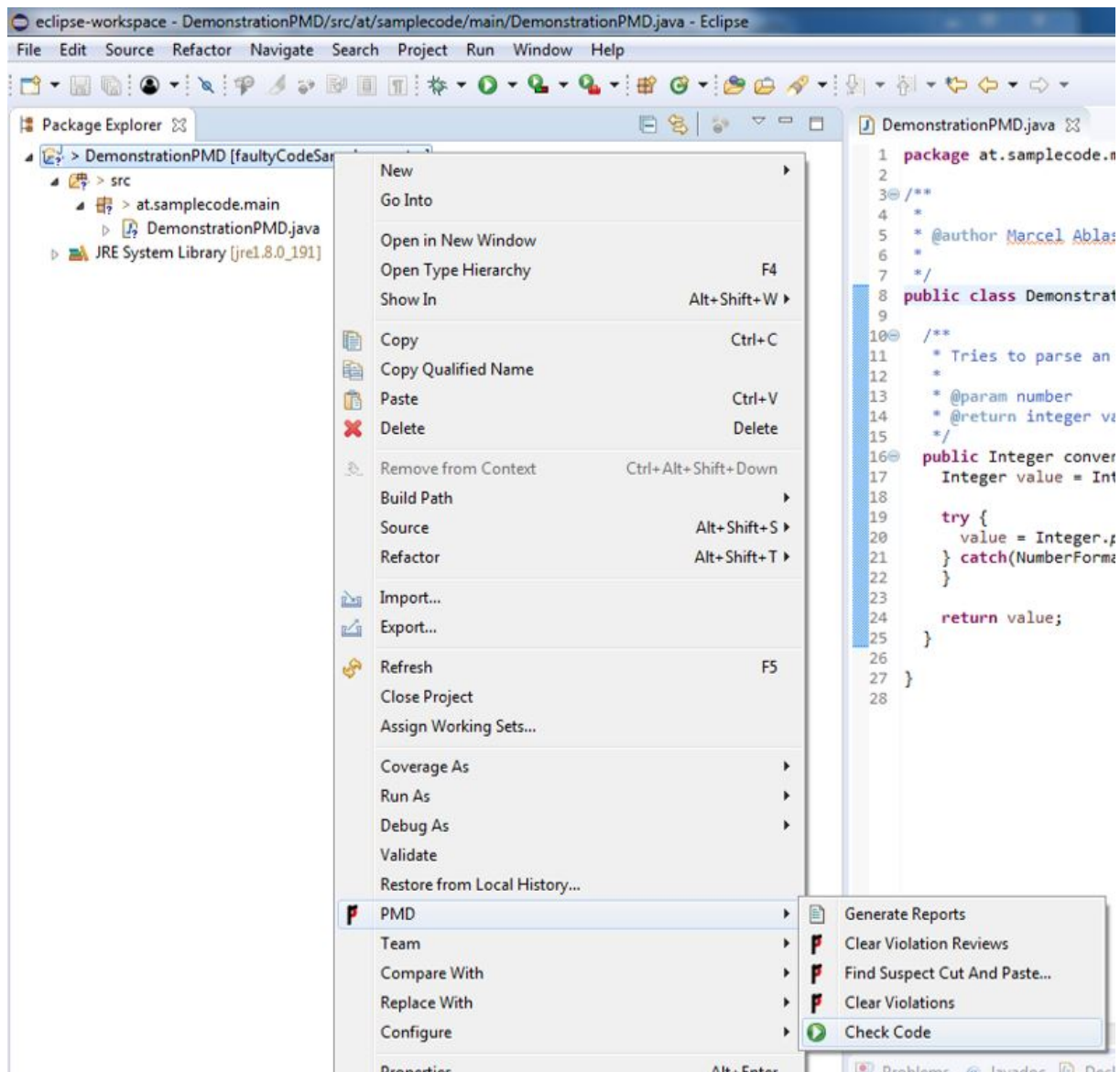



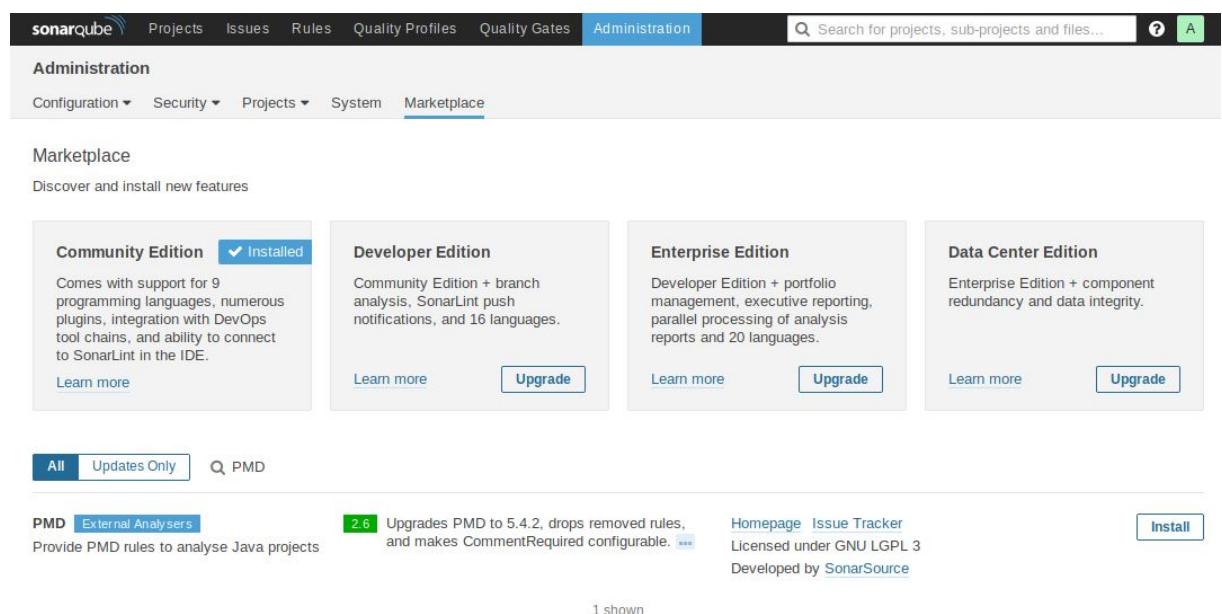


Figure B.5: Running PMD via Eclipse (PMD version: 4.0.17.v20180801-1551).

## Installation of PMD via SonarQube:

1. Open and login as administrator to SonarQube.
  2. Click on *Administration*.  This top menu tab is only visible if an administrator of SonarQube is logged in.
  3. Within this *Administration* view click on *Marketplace*.
  4. Select PMD from the list of all available plugins.  The search field can be used to filter for a specific plugin.
  5. Click on the *Install* button.
-  After the installation, a restart of SonarQube is necessary.



The screenshot shows the SonarQube Administration interface. The top navigation bar includes 'sonarqube', 'Projects', 'Issues', 'Rules', 'Quality Profiles', 'Quality Gates', and 'Administration'. A search bar is present on the right. The 'Administration' section is active, with a sub-menu showing 'Configuration', 'Security', 'Projects', 'System', and 'Marketplace'. The 'Marketplace' section is titled 'Discover and install new features' and displays four editions: 'Community Edition' (marked as 'Installed'), 'Developer Edition', 'Enterprise Edition', and 'Data Center Edition'. Each edition has a description and an 'Upgrade' button. Below this, a search bar is set to 'PMD' and shows a search result for 'PMD External Analysers' with version '2.6'. The result includes a description, a link to the 'Homepage', 'Issue Tracker', and 'Install' button. The text '1 shown' is visible below the search results.

Figure B.6: Step-by-step guide for installing PMD via SonarQube.

## B.2 Section 4.2 SpotBugs (FindBugs)

### B.2.1 Section 4.2 Get Started

#### Installation of SpotBugs via Eclipse:

1. Open Eclipse.
2. Select *Help* → *Install New Software...* via the top menu bar.

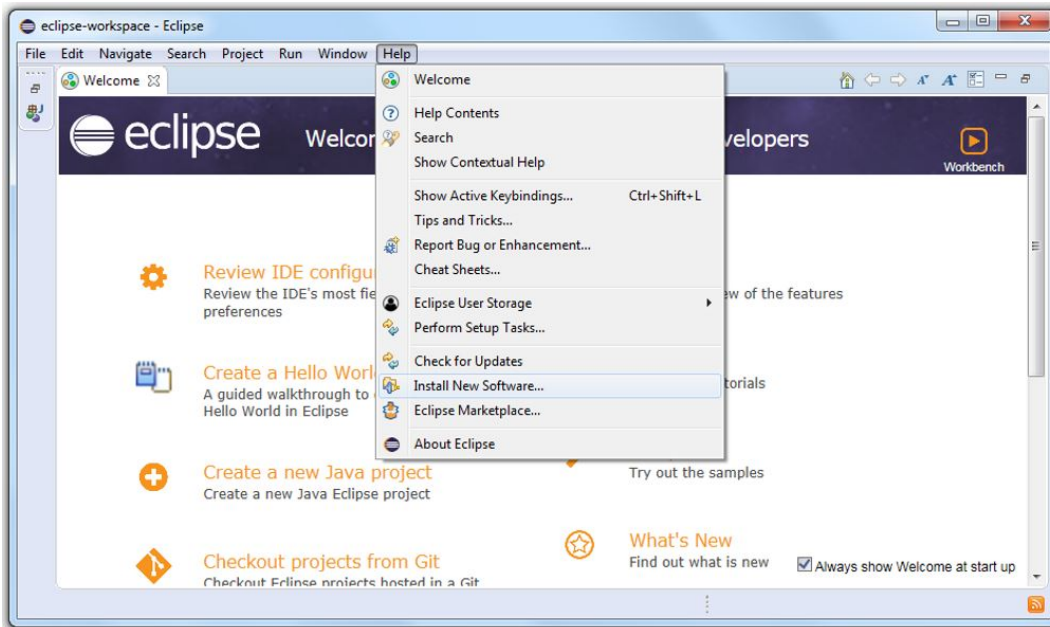


Figure B.7: Installing SpotBugs via Eclipse: second step, open install new software dialog.

3. Click on the button *Add...* within the dialog *Install*.

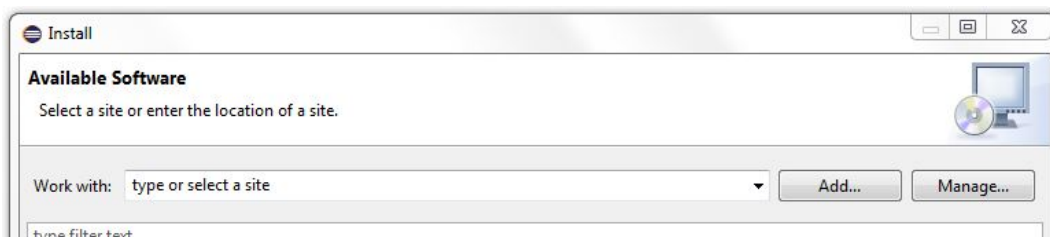


Figure B.8: Installing SpotBugs via Eclipse: third step, open add repository dialog.

4. Another dialog *Add Repository* is opened. Enter *SpotBugs* for the *Name* field and <https://spotbugs.github.io/eclipse/> for the *Location* field. Submit these changes by clicking on the button *OK*.

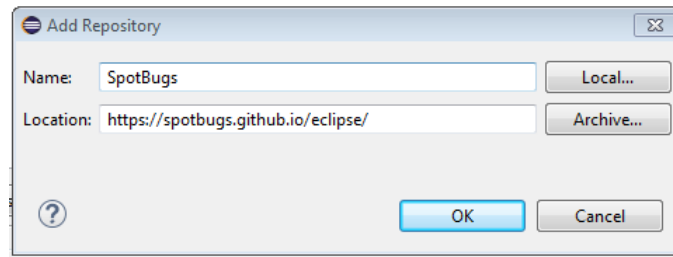


Figure B.9: Installing SpotBugs via Eclipse: fourth step, define PMD repository.

5. Select *SpotBugs Plug-in* and follow the install wizard.

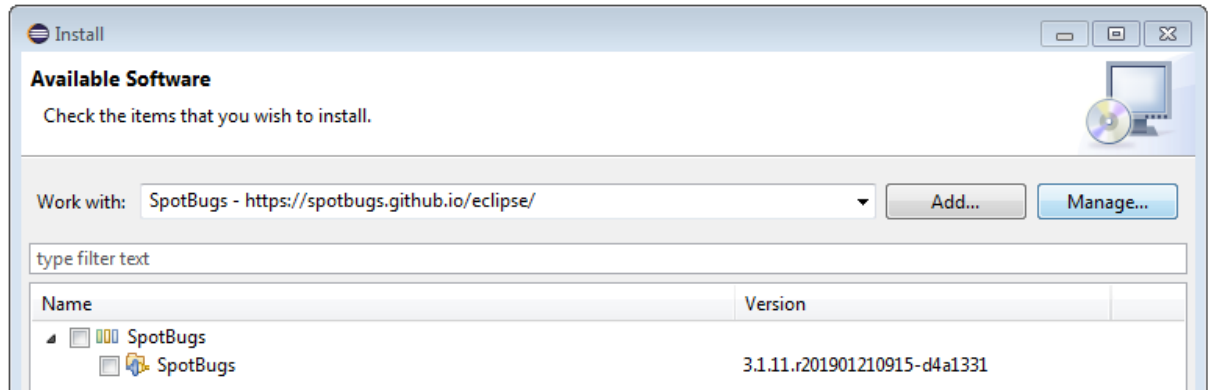


Figure B.10: Installing SpotBugs via Eclipse: fifth step, select SpotBugs plug-in.



After the installation, a restart of Eclipse is necessary.

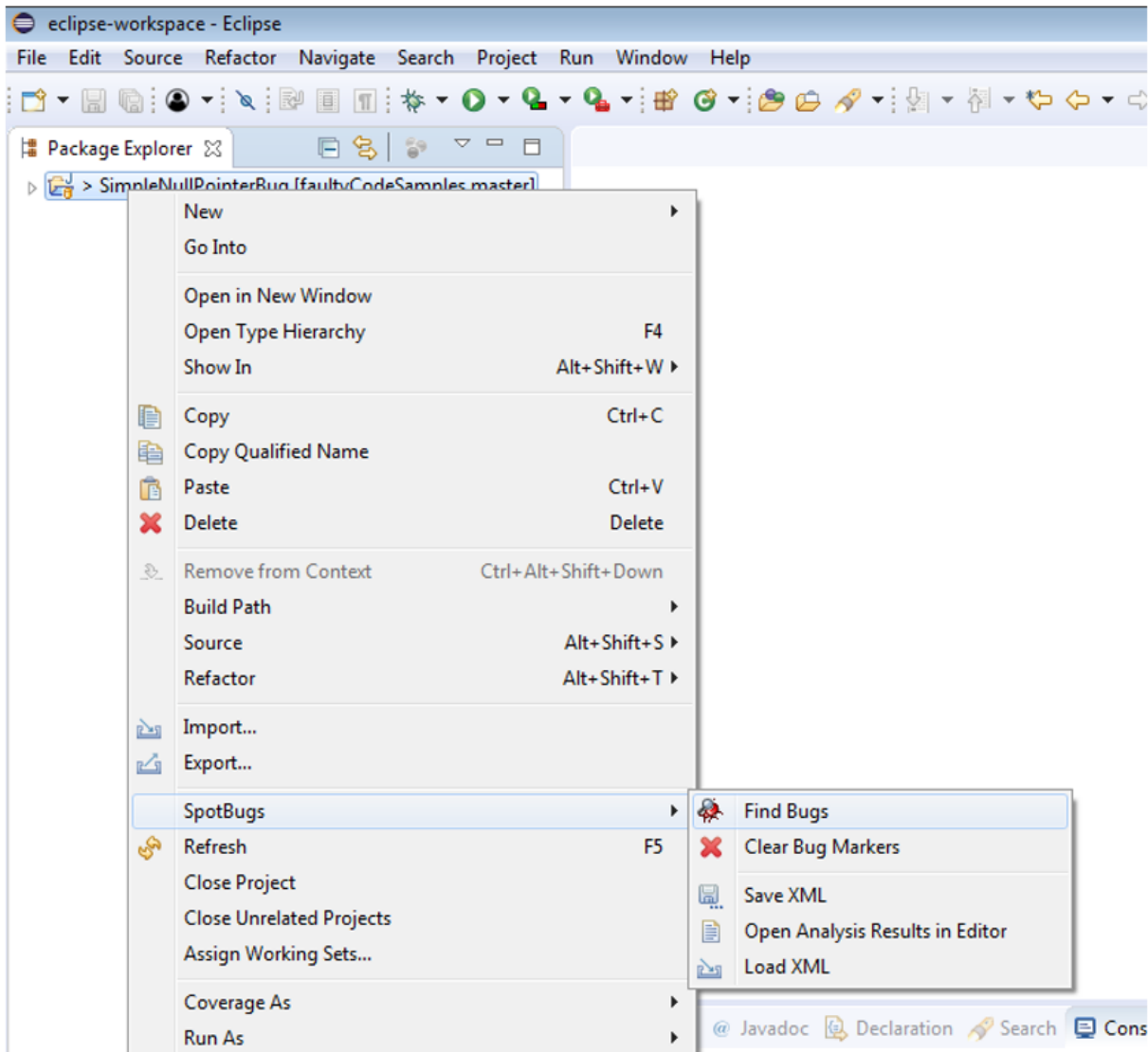



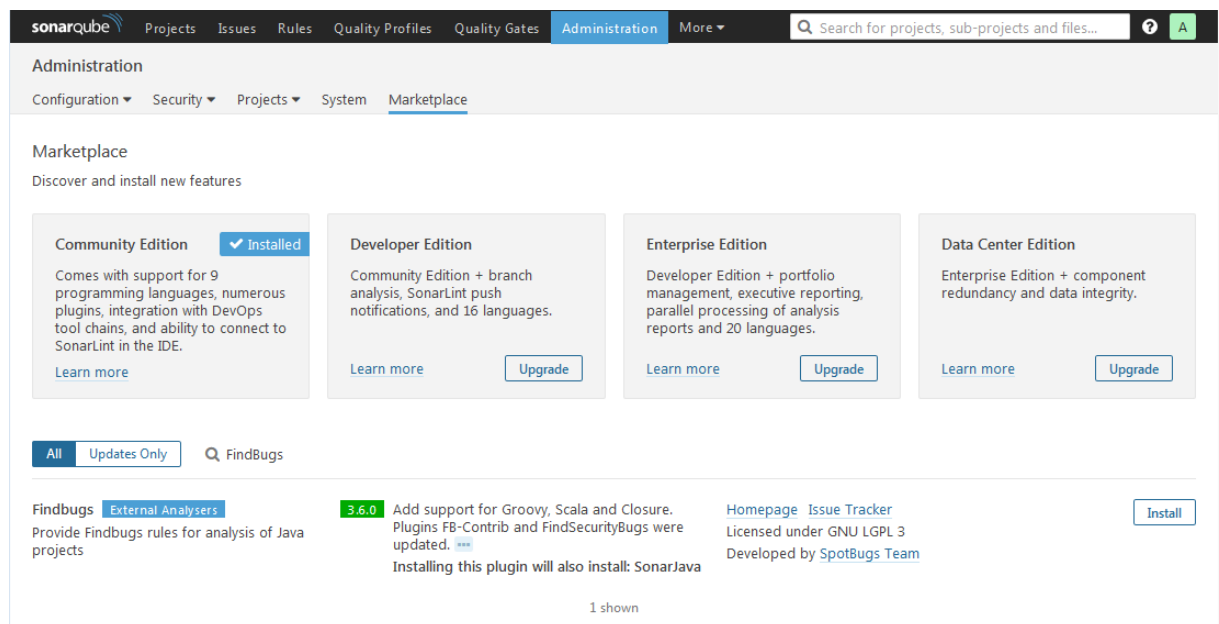


Figure B.11: Running SpotBugs via Eclipse (SpotBugs version: 3.1.11.r201901210915-d4a1331).



## Installation of SpotBugs via SonarQube:

1. Open and login as administrator to SonarQube.
  2. Click on *Administration*.  This top menu tab is only visible if an administrator of SonarQube is logged in.
  3. Within this *Administration* view click on *Marketplace*.
  4. Select SpotBugs from the list of all available plugins.  The search field can be used to filter for a specific plugin.
  5. Click on the *Install* button.
-  After the installation, a restart of SonarQube is necessary.



The screenshot shows the SonarQube Administration interface. The top navigation bar includes 'sonarqube', 'Projects', 'Issues', 'Rules', 'Quality Profiles', 'Quality Gates', 'Administration', and 'More'. A search bar is present on the right. The 'Administration' section is active, and the 'Marketplace' sub-tab is selected. The Marketplace page displays four editions: 'Community Edition' (marked as 'Installed'), 'Developer Edition', 'Enterprise Edition', and 'Data Center Edition'. Below these, a search filter is set to 'FindBugs'. The 'Findbugs' plugin is highlighted, showing version '3.6.0' and a description: 'Add support for Groovy, Scala and Closure. Plugins FB-Contrib and FindSecurityBugs were updated.' An 'Install' button is visible for this plugin. The page also shows '1 shown' at the bottom.

Figure B.12: Step-by-step guide for installing SpotBugs via SonarQube.

## B.3 Section 4.6 RATS

```
1  ///  
2  ///  
3  ///  
4  ///  
5  #include <stdio.h>  
6  #include <string.h>  
7  ///  
8  ///  
9  ///  
10 ///  
11 int main(int argc, char** argv) {  
12     char password[] = "1234";  
13     char buffer[80];  
14     for (int i = 0; i < 3; i++) {  
15         printf ("Please enter the password:");  
16         fflush (stdout);  
17         scanf ("%79s", buffer);  
18     }  
19     if (strcmp (password, buffer) == 0) {  
20         puts ("Logged in!");  
21         return 0;  
22     }  
23 }  
24  
25 puts ("Invalid password!");  
26 return 1;  
27 }
```

*Listing B.1: Source code of demonstration of RATS written in C++.*



## Appendix Chapter 5 Case Study

Lines of Code	12
Lines	23
Statements	3
Functions	2
Classes	1
Files	1
Directories	1
Big Functions	0
Big Functions (%)	0.0%
Big Functions Lines of Code	0
Big Functions Lines of Code (%)	0.0%
Comment Lines	2
Comments (%)	14.3%
Lines of Code in Functions	5

Figure C.1: Captured criteria from C++ program SimpleNullPointerBug by SonarQube

SourceMeter Dashboard

Namespace

Number of rows: 5 Filter: Column filter: T

	Name ↓	TAD	TCD	TCLOC	TPDA	TPUA	TLOC	TLLOC	TNA	TNCL	TNM	TNPKG	TNPA	TNPCL	TNPIN	TNPM	TNOS
1	faulty	0	0	0	0	0	11	9	0	0	0	0	0	0	0	0	0
2	global namespace	0	0	0	0	0	22	12	0	0	0	0	0	0	0	0	0

Figure C.2: Captured criteria from C++ program SimpleNullPointerBug by SourceMeter

Lines of Code	12
Lines	28
Statements	3
Functions	2
Classes	1
Files	1
Directories	1
Comment Lines	4
Comments (%)	25.0%

Figure C.3: Captured criteria from Java program SimpleNullPointerException by SonarQube

Package		TAD	TCD	TCLOC	TPDA	TPUA	TLOC	TLLOC	TNA	TNCL	TNM	TNPKG	TNPA	TNPL	TNPN	TNOS	
1	Java <root_package>	1	0.43	9	2	0	28	12	1	1	2	4	0	1	0	1	3
2	at	1	0.43	9	2	0	28	12	1	1	2	4	0	1	0	1	3

Figure C.4: Captured criteria from Java program SimpleNullPointerException by SourceMeter

```
SourceMeterCPP 8.2.0 (build 8717#45192 Linux64) - (C) 2001-2016 FrontEndART Ltd.
Executing tasks. (Multithread:2)
[2019-02-02 05:00:06] Starting task: WrapperTask
[2019-02-02 05:14:10] Task ended: WrapperTask Result:error
[2019-02-02 05:14:10] Starting task: LinkStaticLibsTask
[2019-02-02 05:14:10] Task ended: LinkStaticLibsTask Result:error
[2019-02-02 05:14:10] Starting task: Can2limTask
[2019-02-02 05:14:10] Starting task: RemoveWrapperBinsTask
[2019-02-02 05:14:10] Task ended: RemoveWrapperBinsTask Result:OK
[2019-02-02 05:14:15] Task ended: Can2limTask Result:OK
[2019-02-02 05:14:15] Starting task: ProfileTask
[2019-02-02 05:14:15] Starting task: Lim2metricsTask
[2019-02-02 05:14:16] Task ended: ProfileTask Result:OK
[2019-02-02 05:14:16] Starting task: DcfTask
[2019-02-02 05:14:16] Task ended: Lim2metricsTask Result:OK
[2019-02-02 05:14:16] Task ended: DcfTask Result:error
[2019-02-02 05:14:16] Starting task: Cppcheck2GraphTask
[2019-02-02 05:14:16] Starting task: FaultHunterCPPTask
[2019-02-02 05:14:16] Task ended: Cppcheck2GraphTask Result:error
[2019-02-02 05:14:16] Task ended: FaultHunterCPPTask Result:OK
[2019-02-02 05:14:16] Starting task: GraphMergeTask
[2019-02-02 05:14:16] Task ended: GraphMergeTask Result:OK
[2019-02-02 05:14:16] Starting task: MetricHunterTask
[2019-02-02 05:14:16] Task ended: MetricHunterTask Result:OK
[2019-02-02 05:14:16] Starting task: AddLicenceTask
[2019-02-02 05:14:16] Task ended: AddLicenceTask Result:OK
[2019-02-02 05:14:16] Starting task: GraphDumpTask
[2019-02-02 05:14:16] Task ended: GraphDumpTask Result:OK
```

Figure C.5: SourceMeter errors during analysis of SSI Project-CA.



## Appendix Chapter 6 Outcome

### D.1 Section 6.1 Detect Bug Challenge

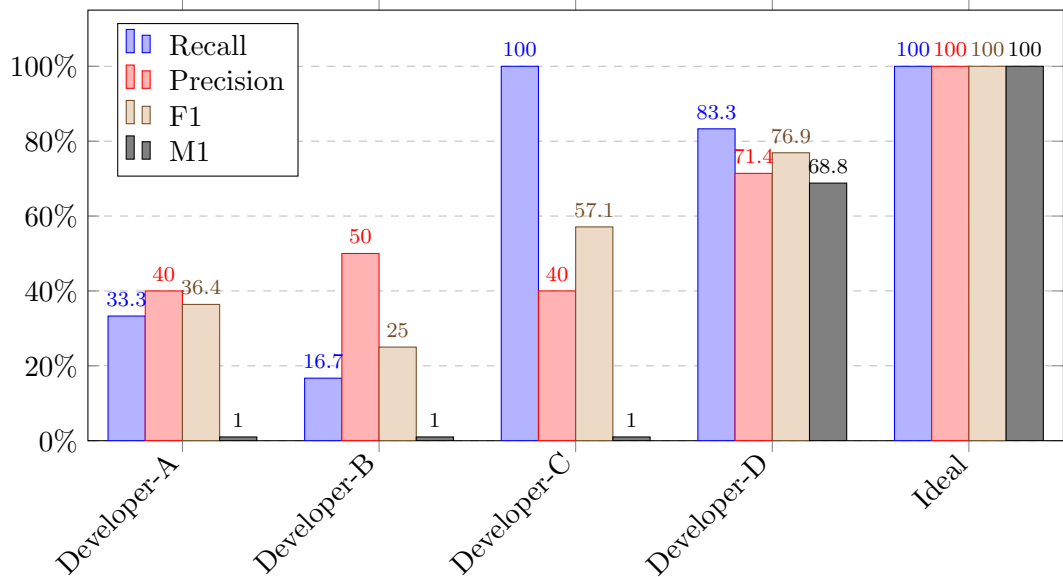


Figure D.1: The average of recall, precision,  $F_\beta$  and  $M_\alpha$  is used to determine the most effective software developer for the bug challenge program Argument Printer. As you can see, all developers detect bugs (recall is greater than zero for all developers). Developer C and D correctly identify and report the most bugs.



## List of Figures

2.1	Overview of a verification tool that performs a static code analysis. . . . .	22
3.1	Excerpt of the questionnaire to determine the usability. . . . .	28
3.2	Visualization of $M_\alpha$ (Equation 3.10; $\alpha = 1$ ). . . . .	31
4.1	Violations outline view in Eclipse from the PMD plugin. . . . .	35
4.2	Demonstration SpotBugs. . . . .	37
4.3	Project overview of the web platform of SonarQube. . . . .	41
5.1	Cplusplus analysis for C++ program SimpleNullPointerBug. . . . .	47
5.2	Infer analysis for C++ program SimpleNullPointerBug. . . . .	48
5.3	PMD analysis for Java program SimpleNullPointerBug. . . . .	61
5.4	SpotBugs analysis for Java program SimpleNullPointerBug. . . . .	61
5.5	SourceMeter analysis for Java program SimpleNullPointerBug. . . . .	62
5.6	SonarQube analysis for Java program SimpleNullPointerBug. . . . .	63
5.7	Infer analysis for Java program SimpleNullPointerBug. . . . .	63
6.1	Trade-off between <i>recall</i> and <i>precision</i> of the program <i>Argument Printer</i> . . . . .	79
6.2	Effectiveness metrics of the bug challenge program <i>Argument Printer (Java)</i> . . . . .	80
6.3	Trade-off between <i>recall</i> and <i>precision</i> of all programs with common errors. . . . .	81
6.4	Effectiveness metrics of all C++ programs with common errors. . . . .	82
6.5	Effectiveness metrics of all Java programs with common errors. . . . .	82
6.6	Trade-off between <i>recall</i> and <i>precision</i> of all software projects from SSI Schaefer Automation GmbH. . . . .	83
B.1	Installing PMD via Eclipse: second step, open install new software dialog. . . . .	90
B.2	Installing PMD via Eclipse: third step, open add repository dialog. . . . .	90
B.3	Installing PMD via Eclipse: fourth step, define PMD repository. . . . .	91
B.4	Installing PMD via Eclipse: fifth step, select PMD plug-in. . . . .	91
B.5	Running PMD via Eclipse (PMD version: 4.0.17.v20180801-1551). . . . .	92
B.6	Step-by-step guide for installing PMD via SonarQube. . . . .	93
B.7	Installing SpotBugs via Eclipse: second step, open install new software dialog. . . . .	94
B.8	Installing SpotBugs via Eclipse: third step, open add repository dialog. . . . .	94
B.9	Installing SpotBugs via Eclipse: fourth step, define PMD repository. . . . .	95
B.10	Installing SpotBugs via Eclipse: fifth step, select SpotBugs plug-in. . . . .	95
B.11	Running SpotBugs via Eclipse (SpotBugs version: 3.1.11.r201901210915-d4a1331). . . . .	96
B.12	Step-by-step guide for installing SpotBugs via SonarQube. . . . .	97
C.1	Captured criteria from C++ program SimpleNullPointerBug by SonarQube . . . . .	99
C.2	Captured criteria from C++ program SimpleNullPointerBug by SourceMeter . . . . .	99
C.3	Captured criteria from Java program SimpleNullPointerBug by SonarQube . . . . .	100
C.4	Captured criteria from Java program SimpleNullPointerBug by SourceMeter . . . . .	100
C.5	SourceMeter errors during analysis of SSI Project-CA. . . . .	100
D.1	Effectiveness metrics of the bug challenge program <i>Argument Printer</i> . . . . .	101

## List of Tables

1.1	Captured criteria for ArgumentPrinter (Java) by software developers. . . . .	16
1.2	Captured criteria for ArgumentPrinter (Java). . . . .	16
3.3	Example values and results of effectiveness metrics (M-measure and F-measure). . . . .	30
4.1	Overview of all selected verification tools. . . . .	33
5.1	Statistics of source code ArgumentPrinter (Java). . . . .	45
5.2	Captured criteria for ArgumentPrinter (Java) by software developers. . . . .	45
5.3	Statistics of source code SimpleNullPointerException (C++). . . . .	46
5.4	RATS classification of source code SimpleNullPointerException (C++). . . . .	46
5.5	Cppcheck classification of source code SimpleNullPointerException (C++). . . . .	47
5.6	SourceMeter classification of source code SimpleNullPointerException (C++). . . . .	47
5.7	Infer classification of source code SimpleNullPointerException (C++). . . . .	48
5.8	Statistics of source code SimpleNullPointerException (C++). . . . .	50
5.9	Captured criteria for SimpleNullPointerException (C++). . . . .	50
5.10	Effectiveness metrics for SimpleNullPointerException (C++). . . . .	50
5.11	Statistics of source code IfNullPointerException (C++). . . . .	51
5.12	Captured criteria for IfNullPointerException (C++). . . . .	51
5.13	Effectiveness metrics for IfNullPointerException (C++). . . . .	51
5.14	Statistics of source code SwitchNullPointerException (C++). . . . .	52
5.15	Captured criteria for SwitchNullPointerException (C++). . . . .	52
5.16	Effectiveness metrics for SwitchNullPointerException (C++). . . . .	52
5.17	Statistics of source code ForNullPointerException (C++). . . . .	53
5.18	Captured criteria for ForNullPointerException (C++). . . . .	53
5.19	Effectiveness metrics for ForNullPointerException (C++). . . . .	53
5.20	Statistics of source code WhileNullPointerException (C++). . . . .	54
5.21	Captured criteria for WhileNullPointerException (C++). . . . .	54
5.22	Effectiveness metrics for WhileNullPointerException (C++). . . . .	54
5.23	Statistics of source code DoWhileNullPointerException (C++). . . . .	55
5.24	Captured criteria for DoWhileNullPointerException (C++). . . . .	55
5.25	Effectiveness metrics for DoWhileNullPointerException (C++). . . . .	55
5.26	Statistics of source code PositiveOutOfBoundsBug (C++). . . . .	56
5.27	Captured criteria for PositiveOutOfBoundsBug (C++). . . . .	56
5.28	Effectiveness metrics for PositiveOutOfBoundsBug (C++). . . . .	56
5.29	Statistics of source code NegativeOutOfBoundsBug (C++). . . . .	57
5.30	Captured criteria for NegativeOutOfBoundsBug (C++). . . . .	57
5.31	Effectiveness metrics for NegativeOutOfBoundsBug (C++). . . . .	57
5.32	Statistics of source code OfByOneBug (C++). . . . .	58
5.33	Captured criteria for OfByOneBug (C++). . . . .	58
5.34	Effectiveness metrics for OfByOneBug (C++). . . . .	58
5.35	Statistics of source code ResourceLeakPartialClose (C++). . . . .	59
5.36	Captured criteria for ResourceLeakPartialClose (C++). . . . .	59
5.37	Effectiveness metrics for ResourceLeakPartialClose (C++). . . . .	59



---

5.38	Statistics of source code SimpleNullPointerException (Java).	60
5.39	PMD classification of source code SimpleNullPointerException (Java).	61
5.40	SpotBugs classification of source code SimpleNullPointerException (Java).	61
5.41	SourceMeter classification of source code SimpleNullPointerException (Java).	62
5.42	SonarQube classification of source code SimpleNullPointerException (Java).	62
5.43	Infer classification of source code SimpleNullPointerException (Java).	63
5.44	Statistics of source code SimpleNullPointerException (Java).	65
5.45	Captured criteria for SimpleNullPointerException (Java).	65
5.46	Effectiveness metrics for SimpleNullPointerException (Java).	65
5.47	Statistics of source code IfNullPointerException (Java).	66
5.48	Captured criteria for IfNullPointerException (Java).	66
5.49	Effectiveness metrics for IfNullPointerException (Java).	66
5.50	Statistics of source code SwitchNullPointerException (Java).	67
5.51	Captured criteria for SwitchNullPointerException (Java).	67
5.52	Effectiveness metrics for SwitchNullPointerException (Java).	67
5.53	Statistics of source code ForNullPointerException (Java).	68
5.54	Captured criteria for ForNullPointerException (Java).	68
5.55	Effectiveness metrics for ForNullPointerException (Java).	68
5.56	Statistics of source code WhileNullPointerException (Java).	69
5.57	Captured criteria for WhileNullPointerException (Java).	69
5.58	Effectiveness metrics for WhileNullPointerException (Java).	69
5.59	Statistics of source code DoWhileNullPointerException (Java).	70
5.60	Captured criteria for DoWhileNullPointerException (Java).	70
5.61	Effectiveness metrics for DoWhileNullPointerException (Java).	70
5.62	Statistics of source code PositiveOutOfBoundsBug (Java).	71
5.63	Captured criteria for PositiveOutOfBoundsBug (Java).	71
5.64	Effectiveness metrics for PositiveOutOfBoundsBug (Java).	71
5.65	Statistics of source code NegativeOutOfBoundsBug (Java).	72
5.66	Captured criteria for NegativeOutOfBoundsBug (Java).	72
5.67	Effectiveness metrics for NegativeOutOfBoundsBug (Java).	72
5.68	Statistics of source code OfByOneBug (Java).	73
5.69	Captured criteria for OfByOneBug (Java).	73
5.70	Effectiveness metrics for OfByOneBug (Java).	73
5.71	Statistics of source code ResourceLeakPartialClose (Java).	74
5.72	Captured criteria for ResourceLeakPartialClose (Java).	74
5.73	Effectiveness metrics for ResourceLeakPartialClose (Java).	74
5.74	Statistics of source code Project-CA (C++).	75
5.75	Captured criteria for Project-CA (C++).	75
5.76	Effectiveness metrics for Project-CA (C++).	75
5.77	Statistics of source code Project-JA (Java).	76
5.78	Captured criteria for Project-JA (Java).	76
5.79	Effectiveness metrics for Project-JA (Java).	76
5.80	Statistics of source code Project-JB (Java).	77
5.81	Captured criteria for Project-JB (Java).	77
5.82	Effectiveness metrics for Project-JB (Java).	77

# Bibliography

- [1] (). 11 of the most costly software errors in history [2018 update], [Online]. Available: <https://raygun.com/blog/costly-software-errors-history/> (visited on 01/15/2019).
- [2] *A static analyzer for Java, C, C++, and Objective-C: Facebook/infer*, Facebook, Feb. 13, 2019. [Online]. Available: <https://github.com/facebook/infer> (visited on 02/13/2019).
- [3] T. Anderson and B. Randell, *Computer Systems Reliability*. CUP Archive, Jul. 31, 1979, 504 pp., ISBN: 978-0-521-22767-4.
- [4] (). AutoCloseable (Java Platform SE 8 ), [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/lang/AutoCloseable.html> (visited on 01/27/2019).
- [5] G. Chatzieleftheriou and P. Katsaros, “Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities,” in *2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, Jul. 2011, pp. 96–103. DOI: [10.1109/COMPASACW.2011.26](https://doi.org/10.1109/COMPASACW.2011.26).
- [6] N. Chinchor and S. Diego, “MUC-4 EVALUATION METRICS,” p. 8,
- [7] (). Closeable (Java Platform SE 8 ), [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/index.html?java/io/Closeable.html> (visited on 01/27/2019).
- [8] (). CWE - CWE-459: Incomplete Cleanup (3.2), [Online]. Available: <https://cwe.mitre.org/data/definitions/459.html> (visited on 01/27/2019).
- [9] A. Dasso and A. Funes, *Verification, Validation and Testing in Software Engineering*. Idea Group Inc (IGI), 2007, 443 pp., ISBN: 978-1-59140-851-2.
- [10] B. D. Davia, C. D. Anderson, L. J. Merriman, and P. J. Niemeyer, “Code coverage test selection,” U.S. Patent 7603660B2, Oct. 13, 2009. [Online]. Available: <https://patents.google.com/patent/US7603660B2/en> (visited on 12/26/2018).
- [11] *F1 score*, in *Wikipedia*, Page Version ID: 874064435, Dec. 16, 2018. [Online]. Available: [https://en.wikipedia.org/w/index.php?title=F1\\_score&oldid=874064435](https://en.wikipedia.org/w/index.php?title=F1_score&oldid=874064435) (visited on 01/30/2019).
- [12] Facebook. (2018). Getting started with Infer, [Online]. Available: <https://fbinfer.com/docs/getting-started.html> (visited on 11/05/2018).
- [13] —, (2018). Infer static analyzer, [Online]. Available: <https://fbinfer.com/> (visited on 11/05/2018).
- [14] —, (). Initial synchronization · facebook/infer@b898227, [Online]. Available: <https://github.com/facebook/infer/commit/b8982270f2423864c236ff8dcdbeb5cd82aa6002> (visited on 11/05/2018).
- [15] M. S. Fisher, *Software Verification and Validation: An Engineering and Scientific Approach*. Springer Science & Business Media, Dec. 3, 2007, 178 pp., ISBN: 978-0-387-47939-2.
- [16] W. B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, 1992, 520 pp., ISBN: 978-0-13-463837-9.
- [17] S. Garfinkel, “History’s Worst Software Bugs,” *Wired*, ISSN: 1059-1028. [Online]. Available: <https://www.wired.com/2005/11/historys-worst-software-bugs/> (visited on 01/15/2019).
- [18] B. Homès, *Fundamentals of Software Testing*. London: ISTE [u.a.], 2012, 342 pp., OCLC: 796194421, ISBN: 978-1-84821-324-1.
- [19] P. C. Jorgensen, *Software Testing, 4th Edition*, 4th ed. Auerbach Publications, Apr. 8, 2016, 494 pp., ISBN: 978-1-4987-8578-5.

- 
- [20] F. Long, D. Mohindra, and R. C. Seacord, *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional, 2012, 739 pp., ISBN: 978-0-321-80395-5.
- [21] A. Loskutov, *[FB-Discuss] Announcing SpotBugs as FindBugs successor*, E-mail, Thu Sep 21 18:48:01 EDT 2017. [Online]. Available: <https://mailman.cs.umd.edu/pipermail/findbugs-discuss/2017-September/004383.html> (visited on 02/13/2019).
- [22] L. Milanese, *Learning Gerrit Code Review*. Packt Publishing, Sep. 2, 2013, 144 pp., ISBN: 978-1-78328-947-9.
- [23] J. Moerman, “Evaluating the performance of open source static analysis tools,” p. 66,
- [24] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing, 3rd Edition*, 3rd ed. John Wiley & Sons, Nov. 8, 2011, 240 pp., ISBN: 978-1-118-13315-6.
- [25] K. Pinedo. (Sep. 28, 2017). 4 of the Worst Computer Bugs in History, [Online]. Available: <https://blog.bugsnag.com/4-worst-computer-bugs-in-history/> (visited on 01/15/2019).
- [26] PMD. (2018). PMD, [Online]. Available: <https://pmd.github.io/#about> (visited on 11/05/2018).
- [27] i. QA. (May 2, 2018). History’s Most Expensive Software Bugs, [Online]. Available: <https://www.ibeta.com/historys-most-expensive-software-bugs/> (visited on 01/15/2019).
- [28] J. Renuka. (Sep. 9, 2016). Accuracy, Precision, Recall & F1 Score: Interpretation of Performance Measures, [Online]. Available: <https://blog.exsilio.com/all/accuracy-precision-recall-f1-score-interpretation-of-performance-measures/> (visited on 01/30/2019).
- [29] C. J. V. Rijsbergen, *Information Retrieval*. Butterworths, Jan. 1, 1979, 228 pp., ISBN: 978-0-408-70929-3.
- [30] M. Selivanov. (). Buggy Java Code: The Top 10 Most Common Mistakes That Java Developers Make, [Online]. Available: <https://www.toptal.com/java/top-10-most-common-java-development-mistakes> (visited on 01/16/2019).
- [31] *SpotBugs is FindBugs’ successor. A tool for static analysis to look for bugs in Java code.: Spotbugs/spotbugs*, spotbugs, Feb. 9, 2019. [Online]. Available: <https://github.com/spotbugs/spotbugs> (visited on 02/10/2019).
- [32] P. R. Srivastava and T.-h. Kim, “Application of Genetic Algorithm in Software Testing,” *International Journal of Software Engineering and Its Applications*, vol. 3, p. 10, 2009.
- [33] V. Vipindeep and P. Jalote, “List of Common Bugs and Programming Practices to avoid them,” Mar. 30, 2005. [Online]. Available: <https://pdfs.semanticscholar.org/8315/73a1011102f2360c41743c09cda6c97d5354.pdf> (visited on 01/16/2019).
- [34] M. Weiglhofer, “Automated Software Conformance Testing,” p. 253, 2009. [Online]. Available: <http://www.ist.tugraz.at/staff/weiglhofer/publications/pdf/dissertation> (visited on 01/16/2019).
- [35] J. White. (Sep. 15, 2010). Top 10 Nasty Java Bugs, [Online]. Available: <https://www.intertech.com/Blog/top-10-nasty-java-bugs/> (visited on 01/16/2019).
-