DOKUZ EYLÜL UNIVERSITY

GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

A TOOL DEVELOPMENT FOR TEST CASE BASED CODE OPTIMIZATION IN JAVA

by

Turgay TAYMAZ

October, 2019

İZMİR

A TOOL DEVELOPMENT FOR TEST CASE BASED CODE OPTIMIZATION IN JAVA

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Degree of Master of Science
In Computer Engineering Program

by
Turgay TAYMAZ

October, 2019
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled "**A TOOL DEVELOPMENT FOR TEST CASE BASED CODE OPTIMIZATION IN JAVA**" completed by **TURGAY TAYMAZ** under supervision of **ASST. PROF. DR. KÖKTEN ULAŞ BİRANT** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

.....

Asst. Prof. Dr. Kökten Ulaş BİRANT

Supervisor

.....

Doç. Dr. Şemistan UTKU

(Jury Member)

.....

Doç. Dr. Ömer DEDEROĞLU

(Jury Member)

Prof. Dr. Kadriye ERTEKİN

Director

Graduate School of Natural and Applied Sciences

ACKNOWLEDGMENTS

I would like to thank my supervisor Asst. Prof. Dr. Kökten Ulaş BİRANT for his advises, support and endless help to complete my thesis.

I would like to thank my co-workers, especially Kadir ÇIRAY.

I'm also thankful to my wife Hacer ARIOL TAYMAZ who has been extremely supportive of me throughout this entire process and has made countless sacrifices to help me get to this point.

My parents, Niyazi and Şehriye TAYMAZ, and my brother, Mahmut TAYMAZ, deserve special thanks for their continued support and encouragement.

Turgay TAYMAZ

A TOOL DEVELOPMENT FOR TEST CASE BASED CODE OPTIMIZATION IN JAVA

ABSTRACT

Java has been a popular programming language since its first stable release in 1996 because of its platform independence. Platform independence provides flexibility and simplicity for programmers. Along with its popularity Java has been a focus of performance studies since its debut. Software performance has lost popularity as advances in hardware have improved the performance of devices running Java. With the release of Android OS and rapid increase in mobile device ownership Java language usage has increased once again. Mobile devices having far less system resources compared to personal computers has re-brought software performance studies into the spotlight, and reinvigorated code optimization studies. However mobile devices have gone into a fast-paced development like all other information technologies and this brought down the need for software performance studies, again.

Also, worth mentioning that development of new Java Virtual Machine (JVM) versions has made the specialized compiler studies, which may threaten the platform independency, obsolete except for specific situations.

Today it is not enough to consider code optimization solely in terms of performance improvement. Much broader vision is needed like software development processes such as maintainability, code readability, improving cooperation in multi-programmer projects, software quality assurance.

In this study, white box testing approach is adopted as the software testing technique and static code analysis method is selected to ensure line coverage. A new software (JPA) has been developed based on a currently used testing tool (PMD) to improve the user experience. Because PMD runs through the command line, it is necessary to know the commands to use. JPA is designed with a graphical user interface to make it easier to use. If there is a previously created report with the same name as the violation report

in the command entered to run PMD, this report is overwritten. In JPA, there is no overwriting because reports are stored in database. In addition, the generated reports can be compared. Also, JPA can still run on any operating system like PMD.

Keywords: Java, code optimization, software testing, PMD, static code analysis

JAVA DİLİNDE, TEST TEMELLİ KOD ENİYİLEME İÇİN ARAÇ GELİŞTİRİLMESİ

ÖZ

Java, çalıştırıldığı platformdan bağımsız olması nedeniyle ilk stabil versiyonunun çıktığı tarih olan 1996 yılından günümüze dek oldukça popüler bir programlama dili olmuştur. Platform bağımsızlığı, programcılar için esneklik ve basitlik sağlar. Java'nın sahip olduğu popülerliğinin paralelinde de ilk versiyonundan itibaren performans çalışmalarına konu olmuştur Donanımdaki gelişmeler, Java'yı çalıştıran cihazların performansını geliştirmesi ile yazılım performansı popülaritesini kaybetmiştir. 2008 yılında Android işletim sisteminin ortaya çıkışıyla ve mobil cihazların kullanımının hızla yayılmasıyla birlikte Java dilinin kullanımı yeniden artmıştır. Mobil cihazların kişisel bilgisayarlara kıyasla çok daha düşük sistem kaynaklarına sahip olması yazılım performansını tekrar gündeme getirmiş ve kod optimizasyon çalışmaları yeniden hız kazanmıştır. Mobil cihazlar da diğer bilişim cihazları gibi hızlı bir gelişim sürecine girmiş ve yazılım performansı çalışmalarına olan ihtiyacı azaltmıştır.

Şunu da belirtmekte fayda var, Java kodlarının üzerinde çalıştığı Java Sanal Makinesi (JVM) versiyonlarının gelişimi platform bağımsızlığını ortadan kaldırabilen özelleştirilmiş derleyici çalışmalarını, özel durumlar dışında gereksiz kılmıştır.

Ancak günümüzde kod optimizasyonuna yalnızca performans artırma açısından bakmak yetersiz kalacaktır. Bakım kolaylığı, kod okunabilirliği, çok programcılı çalışmalarda aradaki uyumun sağlanması, yazılım kalite güvencesi gibi yazılım geliştirme süreçlerinde kod optimizasyon yöntemlerinin kullanımı da dahil edilerek daha geniş bir pencereden bakılmalıdır.

Bu tez çalışmasında yazılım test tekniklerinden beyaz kutu testi yaklaşımı benimsenmiş ve satır kapsamayı sağlamak için statik kod analizi yöntemi seçilmiştir. Bu yöntemi otomatik gerçekleştirmek için halihazırda kullanılan bir test aracı olan PMD baz alınmış ve kullanıcı deneyimini geliştirmeye yönelik bir yazılım (JPA)

geliştirilmiştir. PMD komut satırından çalıştığı için, kullanılacak komutları bilmek gereklidir. JPA kullanımı kolaylaştırmak için bir grafik kullanıcı arayüzü ile tasarlanmıştır. PMD'yi çalıştırmak için girilen komuttaki ihlal raporuyla aynı ada sahip daha önce oluşturulmuş bir rapor varsa, bu raporun üzerine yazılır. JPA'da, raporlar veri tabanında saklandığından üzerine yazma yoktur. Buna ek olarak, oluşturulan raporlar karşılaştırılabilir. Ayrıca, JPA hala PMD gibi herhangi bir işletim sisteminde çalışabilir.

Anahtar kelimeler: Java, kod optimizasyonu, yazılım testi, PMD, statik kod analizi

CONTENTS

Page

LIST OF FIGURES

Page

LIST OF TABLES

Page

CHAPTER ONE
INTRODUCTION

1.1 General Information and Purpose

Java programming language is a general-purpose, concurrent, class based, object-oriented language which allows application developers to write programs that can run on any platform; either online or on different types of devices (Gosling et al., 2018). Java compiler converts the source code to bytecode, which then can be executed on any operating system using JVM (Java Virtual Machine). Independence from operating systems provides flexibility and simplicity, which has considerably increased the popularity to this programming language since the release of version 1.0 in 1996. However, being flexible can also lead the developers to a non-focused implementation and in turn might make some applications to be less efficient than other platform dependent programming languages, which brings out a need for performance improvement applications for Java.

While performance studies have been conducted, starting with the first version of Java programming language with projects such as High Performance Java (HPJAVA) and The Ninja Project, these studies have discontinued due to Java Development Kit (JDK) updates and due to the fact that developments in computer hardware have rendered these studies redundant. (Carpenter et al., 1997; Moreira et al., 2001) Nevertheless, performance studies for Java came back into focus again with the announcement of Android operating system in 2008.

Android operating system is based on an open-source distribution of Linux operating system. Programmers can develop Java-based applications and deploy them on Android devices (Hall & Anderson, 2009). There has been a steady growth from the start in the numbers of the applications that were developed for Android OS because Java was already a popular and well known programming language when Android OS was commercially released in 2008. Today, Android OS is the most

popular operating system for all devices surpassing even Microsoft Windows given in Figure 1.1.

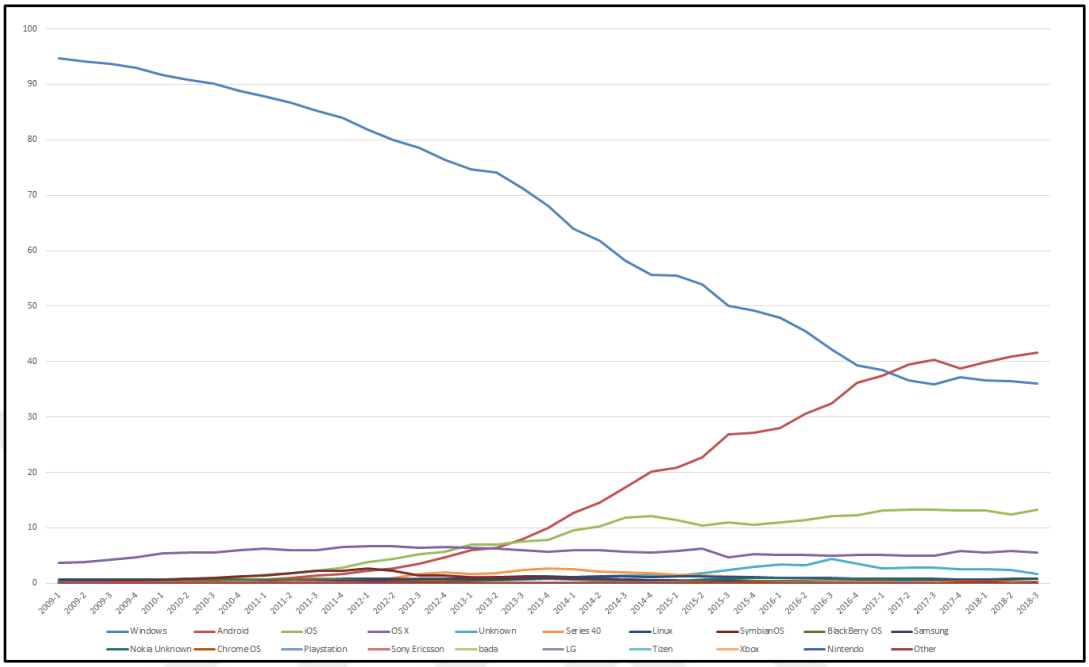

Figure 1.1 Operating system market share worldwide (Statcounter, n.d.)

With the birth of Android OS optimization of Java programming came into agenda mainly because the mobile devices that run on Android at the time were mostly low on performance. Over time, the components in devices has improved and so has the overall performance of Android using devices. This improvement in hardware has reduced the effects of software improvements to an insignificant level. Of course, waiving performance improvements does not mean forsaking optimization altogether.

In this thesis, the code optimization is focused on and limited to Source Code level. In other words, the purpose of this program is to detect the weak points of Java code and report them to the programmer with suggestions, so that he/she can decide whether or not to apply the said suggestions. Applying the changes without confirmation has been purposefully avoided since it may cause other issues as discussed in Chapter 2.3.

1.2 Organization of the Thesis

This thesis consists of five chapters and the remainder of this thesis is organized as follows.

In chapter two, code optimization and optimization levels are explained. In this chapter, when Java code optimization should be done is answered.

In chapter three, software testing is briefly described. It is explained that white box testing which is the one of box approaches for software testing can be performed by static code analysis in Java.

In chapter four, design and implementation of this thesis are explained.

Finally, in chapter five, conclusion and future works are explained.

CHAPTER TWO
CODE OPTIMIZATION

2.1 What is Optimization?

Engineering is a profession where science and mathematics are applied in a manner to benefit people. A mechanical engineer designs or evaluates an engine or other mechanical devices for human use. A civil engineer designs a bridge, a dam or other structures and takes precautions to make sure the structure is intact and safe for everyone involved. An electrical engineer designs an integrated circuit, controls and improves the said circuit for specific purposes, again for the betterment of humanity. A computer engineer designs, develops, maintains, tests and evaluates computer software and hardware. A brutally competitive market or other reasons may force the engineer surpass the nominal design and achieve superiority against the competitors in some way. The process of determining the best approach and improving the design is called optimization (Parkinson et al., 2013).

The definition of the word optimization according to Merriam-Webster is an act, process, or methodology of making something (such as a design, system, or decision) as fully perfect, functional, or effective as possible (Merriam-Webster, n.d.).

2.2 What is Code Optimization?

A definition that most programmers will agree is that the main purpose of code optimization is to increase the quality of the code in terms of time and space without affecting the output result of the code (Bajwa et al., 2016).

The terms time and space lead to a common perception of code optimization as only the performance increase; in a wider perspective, optimization may also mean an improvement in other aspects as defined below.

Code optimization is the process of modifying source code to improve code quality and efficiency. For instance, a program can be optimized so that it runs faster, or works with less memory storage or other resources, or consumes less power, or can be more readable to facilitate maintenance and updating (Bajwa et al., 2016; Johnson 2008; Lins, 2017; Palaniappan, 2016).

Code optimization can be performed at levels such as design level, algorithms and data structures, source code level, build level, compile level, assembly level and run time (Lins, 2017).

2.2.1 Levels of Code Optimization

In general, higher optimization levels have a greater impact and are more difficult to change later in a project, so they require a complete rewrite if they need to be changed. For this reason, optimization generally proceeds from higher to lower levels. Larger gains can be achieved at higher levels with less effort, then gains get smaller and require more effort as levels go lower. However, this is not always the case, in some cases the performance of the program may show tremendous increases in performance with small changes made at lower levels. Therefore, it is not possible to foresee whether the time and effort required are worth the performance increase, not to mention the unforeseen errors that may occur. Because of this unpredictability, the changes made for optimization can be abandoned, partially abandoned or postponed to a later date, depending on the size and complexity of the project.

Assuming the code optimization only as performance enhancements would compromise the stability of the program. Because such an approach would be ignoring the concept of code quality as in debugging, maintenance efforts and design process of later versions.

2.2.1.1 Design Level

This level is the highest level of optimization. Design can be optimized to best use targets, constraints and available resources according to the expected use / load.

2.2.1.2 Algorithms and Data Structures

A good design must be implemented using efficient algorithms and data structures. After design level, proper selection of the algorithm and data structures has a great influence over the efficiency of the program as in performance and in other aspects. In general, changing data structures and algorithms is more difficult than changing source code.

2.2.1.3 Source Code Level

Focusing on the concrete source code level instead of the application of general algorithms on an abstract machine can make a great difference. For instance, in the early years of Java programming language it was a recommended optimization technique to clear unnecessary variable definitions or to change appropriate types of variables according to the need, to increase the performance of the program and reduce the amount of ram required. However, some optimizations of this type can be performed automatically by compilers today. Nevertheless, the benefits that can be achieved by working on the source code, such as writing high quality code and consequently increasing readability and visibility of the code, cannot be ignored.

2.2.1.4 Build Level

Directives and build flags can be used to deactivate the non-required software features and/or build the software specifically for the processor model or hardware capacity between source code and compile levels.

2.2.1.5 Compile Level

Use of an optimized compiler i.e. compilers for specific platforms can optimize the software significantly depending on the compiler's prediction.

2.2.1.6 Assembly Level

With the use of assembly language designed for a specific hardware as the lowest level of optimization, is without a doubt the most efficient way. Therefore, most operating systems used in embedded systems are written with assembler code. However, this approach requires great skill and tremendous amount of effort and time.

2.2.1.7 Run Time

To reduce the compilation overhead i.e. the excess resources used, Just-in-time (JIT) compilers can produce customized machine code based on run time data and thus creating better performance compared to static compilers. For instance, Java JIT compiler comes in two different ways. Which of these two compilers to use depends on the compiler flag to be selected when running the application. The two compilers are known as client and server then compiles and optimizes the code accordingly. These names come from the command-line argument used to select the compiler (e.g., either -client or -server) (Oaks, 2014).

2.3 When to Optimize?

Choosing the right programming language and platform in the design phase will be the most basic start for optimization. The right architecture selection is also made at this beginning stage since it can be very challenging to change later. In general, it will be even more difficult to change the data structure than the algorithm because the data structure assumption and its performance assumption will be used throughout the entire program. For the sake of improving performance, adding new codes and changing source code may reduce readability. This can result in serious complications

in maintaining and debugging the program. Therefore, optimization for performance improvement is better to be left to the end of the development phase. As expressed in the famous quote from Sir Tony Hoare, popularized by Donald Knuth; *"Premature optimization is root of evil."*.

2.3.1 What is Premature Optimization?

Premature optimization is the (so-called) improvement effort in an immature system. The following quote is about premature optimization from Donald Knuth: "The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming." (Knuth, 1974, p. 671).

Indisputably that was a different time when mainframes and punch cards were common and CPU processing cycles were also scarce. With advancing technology, innovations such as much higher CPU processing cycles have emerged, still premature optimization has become a controversial issue.

Nowadays, programs can be quickly distributed over the internet and the codes are updated if necessary, afterwards. A classic example of this would be a start-up that spends a lot of time trying to figure out how to scale their software to handle millions of users. This may seem like a valid concern to be considered. But it makes more sense to worry about processing millions of users, after making sure that at least 100 users like this product and want to use it. If the product is coded in an easy to maintain, the necessary optimizations are easily taken care of (Watson, 2017).

Developments in compilers made some optimizing operations unnecessary, such as bitwise shift and mask used to divide or multiply a positive integer expression by two, because the compiler performs these operations automatically when compiling the code. As a result, ease of maintenance by writing readable code has come to the fore once again.

Before starting optimization, it would be more useful to prepare a report of the program code including suggestions in source code level and then apply the selected suggestions and re-report including a comparison between the multiple versions previously tested by the user.

2.4 Optimizing Java Source Code

In order to understand Java code optimization, it is necessary to explain the technology behind Java programming language and the development of this technology.

In other programming languages the compiler generates machine code for a particular system. But in Java programming language, the compiler generates its own alternative format, which is called bytecode, for Java Virtual Machine (JVM) instead of the machine language.

JVM is an abstract computing machine and provides Java programming language hardware and platform independence.

Java Runtime Environment (JRE) is a software package which bundles the libraries and JVM, and other components to run applications written in Java. JVM is just a part of JRE distributions.

Java Development Kit (JDK) includes several development tools, such as Java libraries, Java source compilers, Java debuggers, building and deployment tools, with JRE.

Just-In-Time (JIT) compiler improves the performance of Java applications at run time and it is a component of JVM.

The compilation and execution steps of Java source code are shown in Figure 2.1.

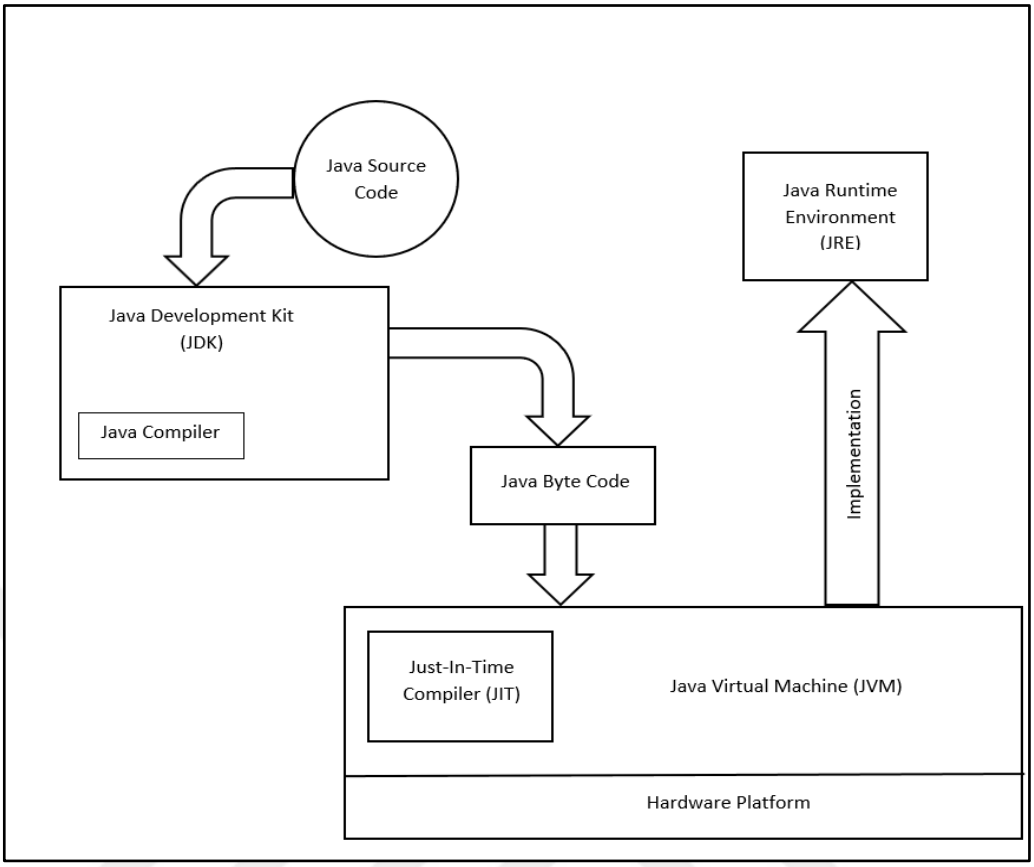

Figure 2.1 Java source code compilation and operation diagram (Javapapers, 2016)

JVM loads the class files at program runs time. The class files determine the meaning of each bytecode and make the appropriate calculation. For comparing to a native application, additional processor and memory usage during interpretation may cost Java application extra time. However, as JIT completes the compilation, the application reaches its peak performance approaches the performance of a native application.

JIT compiler comes in two flavours in Java, and the choice of which compiler to use is often the only compiler tuning that needs to be made when running an application. The two compilers are known as client and server (Oaks, 2014). Choosing the right JIT compiler will directly affect the performance of the program. Additionally, there are also flags available to improve JIT compiler performance in Java programming language.

In Java programming language, to make performance improvements, codes can be refactored or settings can be adjusted on the compiler, and even a new compiler can be designed to generate bytecode for JVM. However, in this thesis, instead of these options, it was aimed to create a report by examining the source code with static code analysis and to increase the optimization by increasing the code quality depending on this report. The main reasons for this are listed below.

Java is a language licensed by General Public License developed by Oracle and is regularly updated, shown in Table 2.1. With these updates, there is also an increase in performance.

Table 2.1 History of Java

1	First stable Java version codename Oak 1.0 was released on January 23, 1996 for Linux, Solaris, Mac and Windows.
2	Java version 1.1 was released on February 19, 1997.
2	Java version 1.2 was released on December 8, 1998.
3	Java Version 1.3 codename Kestrel was released on May 8, 2000.
4	Java Version 1.4 codename Merlin was released on February 6, 2002.
5	Java Version 1.5/Java SE 5 codename Tiger was released September 30, 2004.
6	Java Version 1.6/Java SE 6 codename ‘Mustang’ was released on December 11, 2006.
7	Java Version 1.7/Java SE 7 codename ‘Dolphin’ was released on July 28, 2011.
8	Java Version 1.8/Java SE 8 codename ‘Spider’ was released on March 18, 2014.
9	Java Version 1.9/Java SE 9 was released on September 21, 2017.
10	Java SE 10 was released on March, 20, 2018.
11	Java SE 11 was released on September 25, 2018 is the current stable release (2018).
12	Java 12 and 13 are still developing and available for early-Access. (as of June 2019)

The speed of execution of Java code is highly dynamic and fundamentally depends on Java Virtual Machine. An old piece of Java code may well run faster on a more recent JVM, even without recompiling Java source code. An old piece of Java code may well execute faster on a more recent JVM, even without recompiling Java source

code (Evans et al., 2018). Combining this fact with the possibility that refactoring may not sufficiently improve the software performance, refactoring approach was not pursued in this research. Although there may be optimizations that can still be applied for special circumstances and unforeseen cases in the future, they cannot be generalized and may not be expected for the same performance increase in the upcoming Java versions.

JDK compilers can be customized to create a more efficient bytecode for JVM. Additionally, new programming languages such as Scala and Kotlin have been developed and are available as alternatives, which can also work with JVM. Scala combines object-oriented and functional programming in one concise, high-level language. Scala's static types help avoid bugs in complex applications, and its JVM and JavaScript runtimes let you build high-performance systems with easy access to huge ecosystems of libraries (Scala n.d.). Kotlin is a statically typed programming language that runs on Java Virtual Machine and can also be compiled to JavaScript source code or use LLVM compiler infrastructure (Kotlin, n.d.).

Considering the above-mentioned information, working to optimize the software at the source code level appears much more promising than just increasing the performance for current runtime version. Additionally, leaving the final decision to the programmer by producing reports provides flexibility for the programmer who may have specific needs or specific requirements. For this purpose, statement coverage, a White-Box Test technique, is implemented in this thesis.

CHAPTER THREE
STATIC CODE ANALYSIS IN JAVA

3.1 Software Testing

Software testing is used to expose defects and errors in the software. Principal benefits that can be gained by testing are software quality assurance, reliability estimation of software, validation and verification. Software testing is a key component of software quality assurance and represents a refinement of specification, design and coding (M. E. Khan & F. Khan, 2012).

The primary purpose of the software test is to verify the quality of the software system, another purpose is to determine the integrity and accuracy of the software and ultimately reveal undiscovered errors. Software testing ensures an effective performance of the application (Singh & Kazi, 2016).

Software can be tested with box approach. There are essentially two types of the "box" approach; black box testing and white box testing. The combination of said approaches is called grey box testing. Figure 3.1 shows the “box” approaches in software testing.

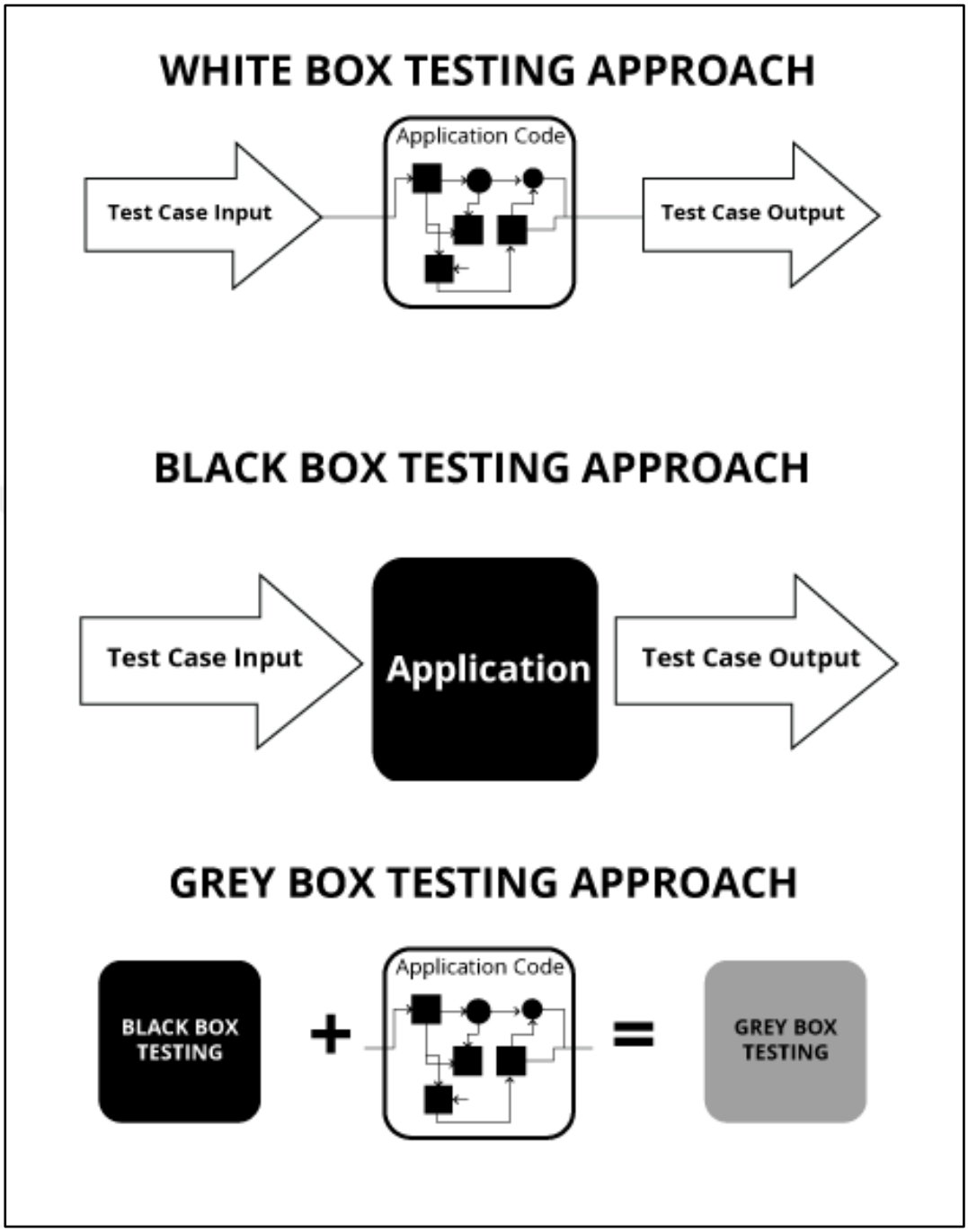

Figure 3.1 Software Testing "Box" Approaches (Invensis, n.d.)

White box testing deals entirely with the code structure. Both the source code and the compiled code of the project are tested. Such tests require inside knowledge of the internal logic of the application code as well as a mastery in the used programming languages. Statements, flow controls, conditions are tested. To emphasize again; it is an approach based entirely on code (Sawant et al., 2012).

Unlike white box testing, black box approach, allows testing to be performed without any requirement of inside knowledge of the code structure or design of the project. The comparison of the input and output can obviously only test if the functional requirements of the system are met or not. In black box test strategy, the user only needs to know the requirements of the system, i.e. the type and range of the input data that the application should be able to process, and what the appropriate responses to these requirements are (Sawant et al., 2012).

Grey box testing, as previously stated, is a combination of white-box testing and black-box testing approaches. Although Grey box testing depend on some inside knowledge of the code structure, it is actually platform and language independent. The reason behind this is; it is essentially a black box test modified according to the main data structures and algorithm of the application but not the details of the code. This approach is particularly useful in integration tests because it is focused on interfaces and boundary values rather than the entire code (Jovanović, 2009; Sawant et al., 2012).

Black box testing doesn't need any knowledge of the internal structure or coding in the program, and more importantly doesn’t give any conclusions or suggestions about it either. Similarly, grey box testing doesn’t require that the tester have access to the entire source code and again limited to boundary values and interfaces between program modules. White box testing is preferred in this study because it is focused on the code and would give more meaningful results pertaining to the code structures (Jovanović, 2009).

3.1.1 White Box Testing

White box testing is sometimes referred to as clear box testing, glass box testing, transparent box testing due to its access to the codes and algorithms or structural testing because of its focus on internal structure or working of a software, rather than its functionality (Karnavel & Santhoshkumar, 2013).

Performing white box testing technique follows a step by step approach and tries to verify each program statement even the comments. White box testing enables performance of data processing and calculations correctness tests, software qualification tests, maintainability tests and reusability tests. The implementation of white box testing is based on controlling the data processing for each test case which in turn raises the issue of coverage of a huge number of possible processing paths and the numerous lines of code. This adversity has given rise to two approaches called "Path coverage" and "Statement coverage". Path coverage is to check whether all possible routes are applied along a certain part of the code. Statement coverage, also known as line coverage, is verifying whether each statement in the program is executed or not (Galin, 2004).

In a software structure, different paths are created by conditional expressions such as IF - ELSE, DO - WHILE. Path testing attempts to provide the full scope analysis of a program by testing all possible paths. Therefore, "path test's completeness" is defined as the percentage of program paths carried out during the test. The concept of path testing is not practical in most cases because of the vast resources needed for its performance. Because of this predicament, statement coverage has been developed as an alternative. In statement coverage, test cases leave most of the possible paths untested however requires far fewer test cases to cover all paths compared to path coverage. As an alternative to creating multiple test cases to cover all paths, static code analysis is a viable solution, or even better, using a tool to automate static code analysis (Galin, 2004).

3.2 Static Code Analysis

Static code analysis, also called static analysis, is the method of examining the program codes without the actual execution. Static analysis can be considered a code review process. Code reviewing is one of the oldest and safest methods for detecting errors in the source. It suggests to read the source code carefully and make suggestions on how to improve it. This process is used to locate existing errors and pieces of code that may cause future errors. Code review process is useful, because programmers are

more easily to notice others' mistakes than their own. The most important problem in this process is the need for periodical meetings of programmers to review each new code, or re-review a code after the proposed changes are applied. When programmers review large pieces of code at a time, they lose their attention quickly, so they need to rest regularly. Otherwise, code review will not help. This is a serious problem because of its immense cost in man hours. Automation of static analysis, i.e. static code analysis tools would be a good solution to reduce this cost (Ayewah et al., 2008).

3.2.1 Static Code Analysis Tools

Static code analysis tools examine the source code of programs and give suggestions to the programmer as to which parts of the code to reconsider. These tools may not replace a code review by a team of programmers, but the benefit/cost ratio makes the use of static analysis a very good option. Static code analysis tools are very successful in detecting errors in programs and providing code formatting suggestions. One of the main advantages of static analysis is that it allows the cost of eliminating errors in the software to be greatly reduced. This is mainly because this analysis can be performed at the coding stage. Fixing an error at the testing stage costs about ten times more compared to development stages, as shown in Table 3.1. Static analysis can be performed in construction, system test and post-release phases.

Table 3.1 Average cost of fixing defects based on when they're introduced and detected (McConnell, 2014)

	Time Detected				
Time Introduced	Requirements	Architecture	Construction	System Test	Post-Release
Requirements	1	3	5-10	10	10-100
Architecture	-	1	10	15	25-100
Construction	**-**	**-**	**1**	**10**	**10-25**

Static analysis does not depend on the compiler used and the platform in which the compiled program is executed, thus making it possible to find hidden errors, such as

undefined behavioural errors, that may occur even a few years after it was created or errors that may occur in different compilers and platforms. In addition, typos and other errors caused by Copy-Paste usage can be easily and quickly detected.

Static code analysis tools perform these operations according to the rules and standards of the programming language. There are a lot of commercial and free static code analysers. In a research, PMD was deemed to be the best static analysis tool for Java programming language (Abdallah & Al-Rifaee, 2017). In Figure 3.2, the standards of Java programming language set by several companies and institutions are compared under six headings. These titles are: Name of author(s), date issued, document number of pages, structure of standards, number of standard rules' categories, number of rules and whether there are examples for each rule in the document or not.

For most Java standards, the standards are very vague and can be misinterpreted by different developers. This may hinder or reverse the objectives of the standards.

More than half the standards, namely the ones that are specified as in text format in Figure 3.2, has no equations, no notes, or no symbolic forms. This may lead to different interpretations in the implementation of the standards. In standard documents that use text, most rules are between paragraphs, unlike documents that use numerical or pointed structures. As a result, rules have not been organized and in some cases it has become difficult to remove.

In most documents, there is no weight for rules. that is, all rules are equally important.

Some standards are even made by individuals and not by companies or standard associations.

While most static analysis tools use SUN or Google standards, some tools such as PMD use developers approved rules in addition to SUN and Google standards. This

makes PMD a multi-standard based tool to implement. In addition, PMD enables the development of tools through the APIs it provides. Therefore, PMD is used in this study.

Specifications / Standards	Author	Date issued	Document size (in pages)	Rules structure (numbered or text)	Number of Rule Categories	Number of Rules	Examples explain the rules
Java Code convention	SUN microsystems	1996	20	Text	8	Around 81	Yes
Java coding standards	Philip Johnson	1996	2	Text	8	Around 16	Yes
CHiMu	ChiMu corporation	1997	54	Text	15	Around 105	No
Netscape	Netscape	1998	13	Text	6	Around 44	Yes
Coding convention for C++ and java	Macadamian	1998	10	Text	5	Around 76	No
Infospheres	CalTech	1999	24	Text	4	Around 91	No
Ambysoft	Scott W. Ambler	2000	76	Text	9	Around 76	Yes
Java coding standards	CS department in RIT	2000	4	Text	10	Around 54	No
Java coding style guide	Achut Reddy	2000	23	Text	7	Around 188	Yes
Java coding standards	Exolab Arnaud Blandin et.al.	2002	39	Pointed 90 rules 15 recommendations	4	105	Yes
Coding standards for java	New England Java Users group	2002	62	Numbered 9 standards, 29 style 22 conventions	3	60	Yes
BSSC	European Space Agency	2005	113	Numbered 146 rules 77 recommendations	11	223	No
Java coding standards	Department of Veterans Affairs	2009	72	Text	13	Around 64	Yes
Java standards	Bahraini e-government Sharmila Naveen	2010	31	Text + points	8	Around 135	Yes
The CERT® Oracle® Secure Coding Standard for Java	Fred Long, Dhruv Mohindra, Robert C. Seacord, Dean F. Sutherland and David Svoboda	2011	699	Text	17	N/A	Yes
JPL java coding	Jet Propulsion Laboratory (NASA)	2014	330	Points	3 (Critical, Important, Advisory)	218	Yes
The java language specifications SE 8ed	James Gosling, Bill Joy Guy Steele, Gilad Bracha and Alex Buckley	2014	644	Text	16	N/A	Yes
Java programming style guide lines	Geotechnical Software Service	2015	16	Numbered	5	86	Yes
Code convention for java	Steve Yohanan	2016	67	Points	9	Around 112	Yes
Google Java Style	Google	N/A	18	Numbered	7	27	Yes

Figure 3.2 Java standards list (Abdallah & Al-Rifaee, 2017)

CHAPTER FOUR
DESIGN AND IMPLEMENTATION

4.1 Implementation Environments

4.1.1 PMD Source Code Analyzer

PMD is an open source, static code analysis tool with comprehensive configurable rule sets (Thomas et al., 2003). PMD supports Java, JavaScript, Salesforce.com Apex and Visualforce, PLSQL, Apache Velocity, XML, XSL languages (Nembhard et al., 2017).

In PMD, multiple rules or rulesets can be used together, or a custom ruleset can be created. For Java Programming Language, there are more than 280 rules which are defined in eight rulesets: Best Practices, Code Style, Design, Documentation, Error Prone, Multithreading, Performance and Security (PMD n.d.). Additionally, PMD users can execute custom analyses by developing new evaluative rules.

Instead of the source code itself, PMD uses abstract syntax trees (ASTs) created by a JavaCC[1] parser from Java sources. The main loop of PMD then examines AST, visiting all registered rules related to specific AST structures. The rule scan then checks AST and report violations (Aderhold & Kochtchi, 2013). The following example code, which is given in Figure 4.1 and generated AST for this code by PMD Rule Designer in Figure 4.2, illustrates rule creating process for PMD and also illustrates inner workings of PMD.

```
class Example {
    void bar() {
        while (baz)
            buz.doSomething();
        }
}
```

Figure 4.1 Sample Java code

[1] Java Compiler Compile (JavaCC) is a parser generator. A parser generator is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar.

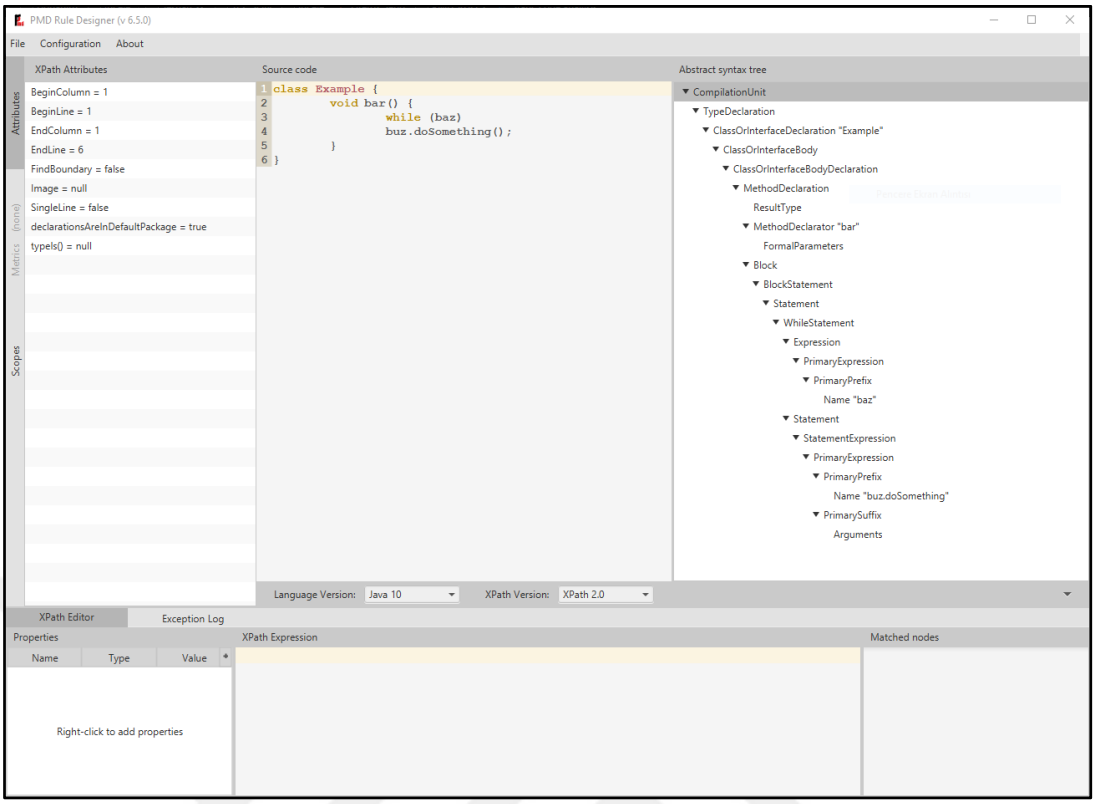

Figure 4.2 Generated AST for sample code by PMD Rule Designer

In PMD, rules are written as Java classes or XPath expression. It actually gets quite difficult to follow source code especially as it gets longer, shown in Figure 4.1. This is mainly because it is difficult to tell where the curly braces belong. In Figure 4.2 AST is used to apply “While loops must use braces” rule which can analyse this difficulty. To be able to do this, it is necessary to determine the changes that happen in AST if “buz.doSomething()” clause had braces inserted as shown in Figure 4.3.

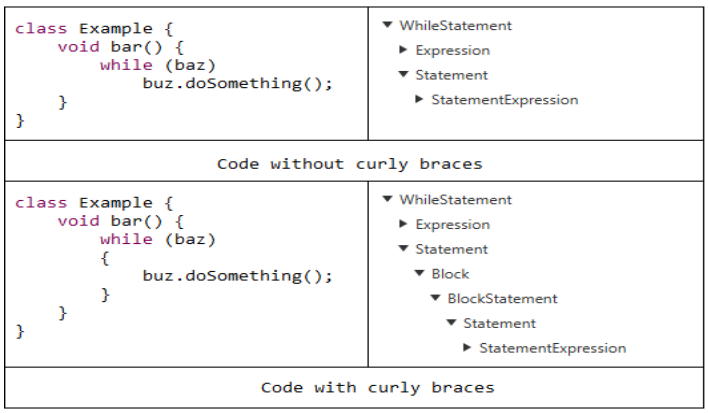

Figure 4.3 Different AST for sample code with-without curly braces

When the curly braces are added, AST nodes are formed with the names “Block” and “BlockStatement”. A rule violation can be detected by writing a rule that detects a "WhileStatement" declaration that is not followed by “Block”. To write a custom rule, a new java class needs to be created that is inherited from net.sourceforge.pmd.lang.java.rule.AbstractJavaRule. PMD works by creating AST and then traverses it recursively. By doing this, a rule can get a call back for any type it’s interested in. The rule that gets called whenever AST traversal finds a “WhileStatement” can be seen in Figure 4.4.

```
import net.sourceforge.pmd.lang.java.rule.*;
import net.sourceforge.pmd.lang.java.ast.*;
public class WhileLoopsMustUseBracesRule extends AbstractJavaRule {
    public Object visit(ASTWhileStatement node, Object data) {
        System.out.println("Avoid using 'while' statements without using curly braces");
        return data;
    }
}
```

Figure 4.4 WhileLoopsMustUseBracesRule Java Code

Once the rule class is written, PMD must be told of this. PMD needs a ruleset XML file to recognize the rule. “sampleCustomRule.xml” file can be seen in Figure 4.5. The elements and attributes of the file are explained below.

- name - The rule’s name.
- message - Message for report.
- class - Location of the rule class.
- description - A description of what this rule looks for.
- priority - There are five levels of priority in PMD for the rules:[2]
    1. **High priority.** Code revision absolutely required.
    2. **Medium high priority.** Code revision highly recommended.
    3. **Medium priority.** Code revision recommended.
    4. **Medium low priority.** Code revision optional.
    5. **Low priority.** Code revision highly optional.
- example - A code fragment between CDATA tags to explain the rule violation.

[2] Rules priority: https://pmd.github.io/latest/pmd_userdocs_extending_rule_guidelines.html#how-to-define-rules-priority

```
<?xml version="1.0"?>
<ruleset name="Sample custom rules"
    xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 https://pmd.sourceforge.io/ruleset_2_0_0.xsd">
  <rule name="WhileLoopsMustUseBracesRule"
      message="Avoid using 'while' statements without curly braces"
      class="WhileLoopsMustUseBracesRule">
    <description>
      Avoid using 'while' statements without using curly braces
    </description>
    <priority>3</priority>

    <example>
      <![CDATA[
        public void doSomething() {
          while (true)
          x++;
        }
      ]]>
    </example>
  </rule>
</ruleset>
```

Figure 4.5 sampleCustomRule.xml

To test the rule run PMD in command line by giving command:

pmd.bat -d “Path to Sample Code” -R “Path to sampleCustomRule.xml”

After running the command PMD prints “Avoid using ‘while’ statements without using curly braces”, as shown Figure 4.6

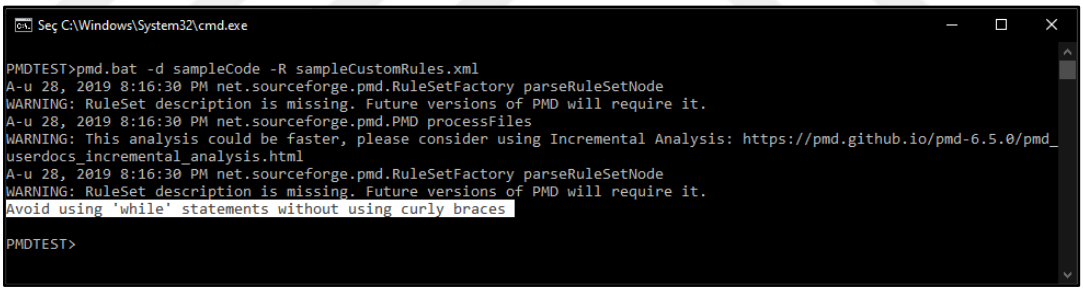

Figure 4.6 Output of running PMD with sample custom rule

After testing the rule, it is necessary to make changes to the rule class in order to include the rule in the reports to be prepared by PMD. However, since the aim is to explain how PMD works instead of preparing a custom rule for PMD, the continuation of the subject and/or preparing the rule with XPATH expressions[3] will not be included here.

[3] https://pmd.github.io/pmd-6.5.0/pmd_userdocs_extending_writing_pmd_rules.html#writing-a-rule-as-an-xpath-expression

4.1.2 Eclipse IDE

Eclipse is a free, Java-based development platform known for its plug-ins that allow software developers to develop and test code written in other programming languages. Eclipse is open source and is managed by the Eclipse Foundation, a non-profit company. It is published under the terms of the Eclipse Public License. Eclipse has a base workspace which can be customizable with plugins. Eclipse began to be developed for Java programming language in 2001, when IBM donated three million lines of code from Java tools to develop an open source integrated development environment (IDE). Ada, C, C ++, C #, COBOL, Fortran, Groovy, Haskell, JavaScript, Perl, PHP, Python, R, Ruby, Rust, Scala are supported with addons added over time. (Eclipse, n.d.).

4.1.3 SQLite

SQLite is a simple and easy to use database library. SQLite is an open source, transactional and relational SQL database engine that does not require server configuration and configuration requirements developed with the C programming language. It is small, fast, compact, zero-configuration, serverless and embedded SQL database engine. SQLite does not have a separate server process, unlike most other SQL databases. It reads and writes directly to ordinary disk files. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file. SQLite file format is stable, cross-platform, and backwards compatible. Because the database file format is cross-platform it can be freely copied a database between 32-bit and 64-bit systems. The fact that SQLite comes bundled inside countless applications both in mobile and pc platforms that people use every day probably makes it the most used database in the world (SQLite, n.d.).

4.2 Java Project Analyser (JPA)

4.2.1 Operating PMD

PMD is distributed as a zip archive. The latest binary distribution can be downloaded from the releases page[4]. In Windows operating system, to run PMD, unzip it in any directory and run the file "pmd.bat" under the "bin" folder with the required parameters from the command line. PMD does not have a graphical user interface. pmd.bat requires two arguments:

- -d <path>: Path to files of source code to analyse.
- -R <path>: The ruleset file. PMD uses xml configuration files.

Other arguments of PMD are optional. For instance, PMD displays the report on command line by default. But user can change this by giving "-r" argument with a path to a file in which the report output will be recorded. Full list of arguments can be found on PMD's documentation page[5].

4.2.2 Shortcomings of PMD and Java Project Analyser (JPA)

Although the generated report will be displayed on the command screen or stored in the name and type specified at each run, PMD will not operate if the parameters are missing or incorrect. Moreover, it will overwrite a previous existing report if the same name is given as a parameter again without warning. Another major difficulty with PMD is keeping track of the names of the rulesets. Most importantly it will not be able to compare the report files. Java Project Analyser (JPA) was developed to prevent these hurdles and improve the user experience. Using JPA is considerably easier because it can be used via the graphical user interface after running the file named JPA.jar.

[4] https://github.com/pmd/pmd/releases
[5] https://pmd.github.io/pmd/pmd_userdocs_cli_reference.html

4.2.3 How Java Project Analyser Works

The following describes how JPA works with sample test codes. JPA is developed for this study and its main window can be seen in Figure 4.7.

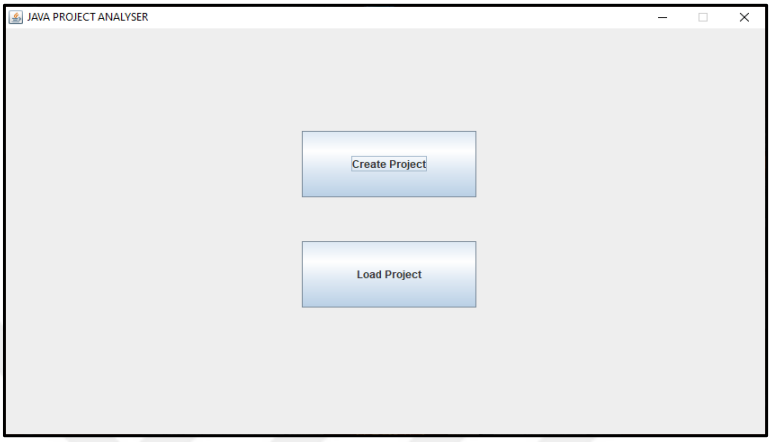

Figure 4.7 JPA main window

As shown in Figure 4.7, when JPA is first run, a window with two buttons is displayed. The first button can be used to create a new project for analysis, or the second button can be used to re-analyse an existing project or compare old analyses. To test with JPA, two source code files named "CodingHorror.java" and "StringHorror.java" are prepared in the "TestCodeFolder" folder and the codes are given in Figure 4.8.

CodingHorror.java

```
1  package testSrc;
2
3  public class CodingHorror {
4
5      public static void main(String args[]) {
6
7          //Violations for AvoidUsingShortType -Start-
8          short doNotUseShort = 1;
9          short shouldNotBeUsed = 2;
10         //Violations for AvoidUsingShortType -End-
11         doNotUseShort += shouldNotBeUsed;
12         System.out.println("Short Variable 1 : " + shouldNotBeUsed);
13         System.out.println("Short Variable 2 : " + doNotUseShort);
14
15         //Violations for BooleanInstantiation -Start-
16         Boolean bar = new Boolean("true");
17         System.out.println("Boolean Variable 1 : " + bar);
18         //Violations for BooleanInstantiation -End-
19
20         Boolean buz = Boolean.FALSE;
21         System.out.println("Boolean Variable 2 : " + buz);
22
23
24         String s = StringHorror.retS();
25         System.out.println("String Variable 1 : " + s);
26         String t = Integer.toString(456);
27         System.out.println("String Variable 2 : " + t);
28     }
29 }
```

StringHorror.java

```
1  package testSrc;
2
3  public class StringHorror {
4
5      public static String retS() {
6
7          //Violations for AddEmptyString -Start-
8          String s = "" + 123;
9          //Violations for AddEmptyString -End-
10         return s;
11     }
12 }
```

Figure 4.8 CodingHorror.java and StringHorror.java

These codes run successfully and are sent to the command screen. However, there are three violations in these codes, one high priority (1), one medium high priority (2) and one medium priority (3) in the performance ruleset[6] of PMD. Explanations for these three violations are given in Table 4.1.

Table 4.1 PMD violations in sample codes

Rule Name	Priority	Definition
AvoidUsingShortType [7]	1. High	Using "short" data type is beneficial for memory. However, since JVM can only perform arithmetic operations for data types "int" and "long", it requires to convert "short" to "int" when processing the value, and converts the result back to "short". This would result for losing of performance when trying to get more memory gains.
BooleanInstantiation[8]	2. Medium High	Avoid using “new Boolean()” object. It can be referenced “Boolean.TRUE” or “Boolean.FALSE” instead. Also, using “new Boolean()” object is deprecated since JDK 9.
AddEmptyString[9]	3. Medium	It is better to use one of the type-specific “toString()” methods to convert variable types to “string” instead of aggregating them with empty string ("").

To analyse the test code with JPA, select the folder with the "Create Project" button as in Figure 4.9.

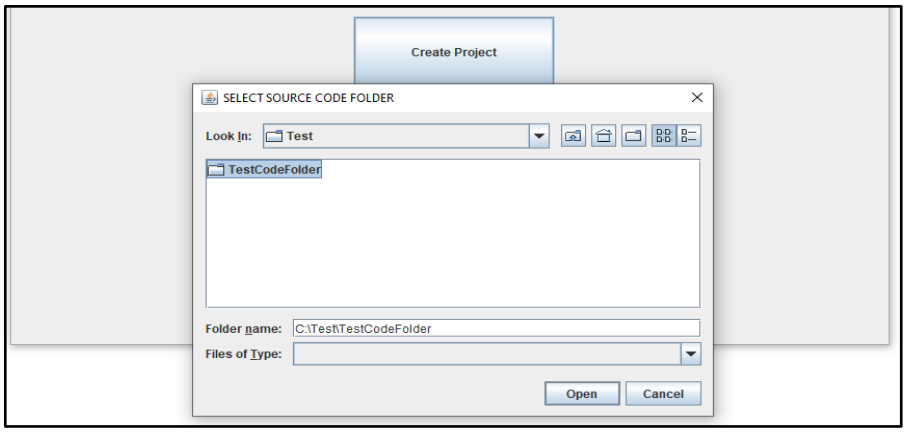

Figure 4.9 Selecting source code folder

[6] https://pmd.github.io/pmd/pmd_rules_java_performance.html
[7] https://pmd.github.io/pmd/pmd_rules_java_performance.html#avoidusingshorttype
[8] https://pmd.github.io/pmd/pmd_rules_java_performance.html#booleaninstantiation
[9] https://pmd.github.io/pmd/pmd_rules_java_performance.html#addemptystring

After selecting the folder, ruleset(s) need to be selected to run analysis otherwise program will give error as shown Figure 4.10.

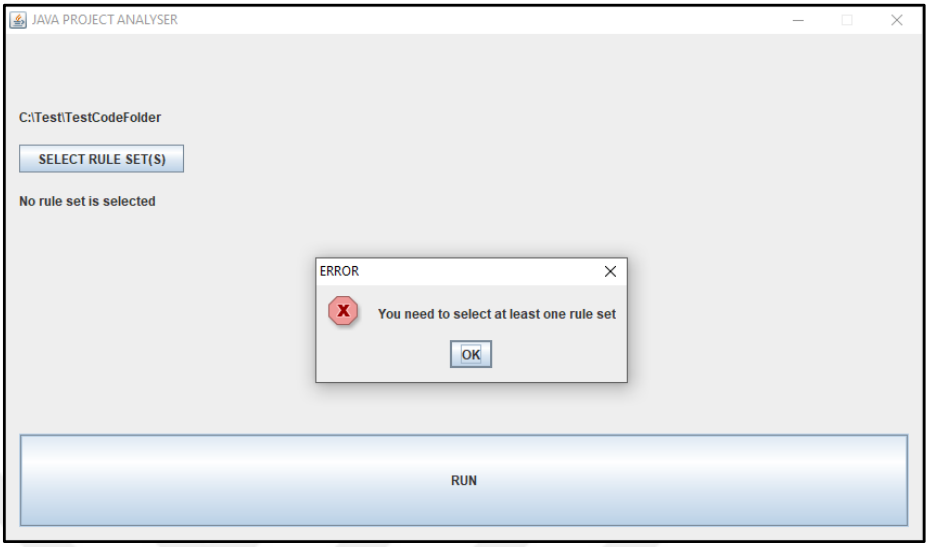

Figure 4.10 Error because ruleset(s) is not selected

"SELECT RULE SET(S)" button is used to select the ruleset(s). From the window that opens, the rulesets can be selected. For the test code "Performance" rule set was used, as in Figure 4.11.

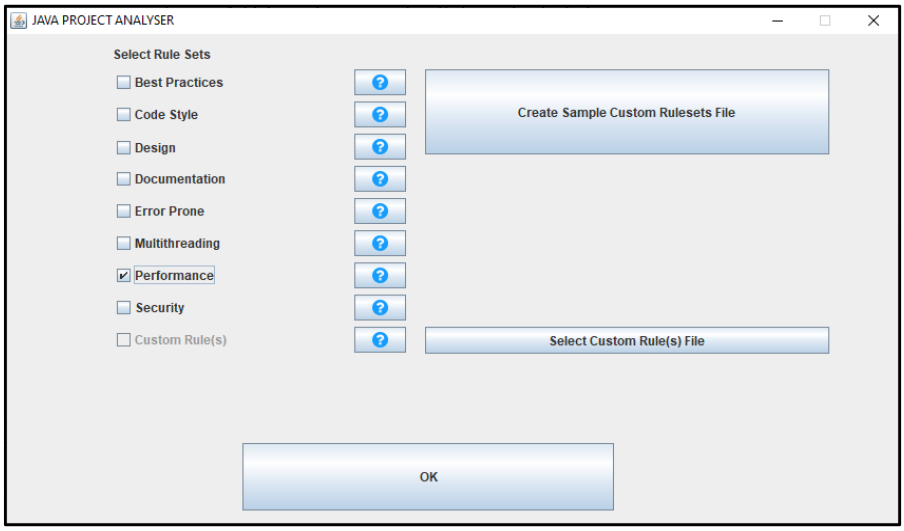

Figure 4.11 Ruleset(s) selection

Considering that this was just a demonstration and not a complete test, it was not necessary to use the entire performance ruleset for the test codes here. Since the example codes are known to contain three violations, "Custom Ruleset" file could be prepared in "xml" format as shown in Figure 4.12 and used at the rule selection shown in Figure 4.11.

```xml
<?xml version="1.0"?>

<ruleset name="Custom Rules"
    xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 http://pmd.sourceforge.net/ruleset_2_0_0.xsd">

  <!-- Add AvoidUsingShortType,AvoidUsingShortType and AddEmptyString rules from Performance Rule Set -->
  <rule ref="category/java/performance.xml" />
    <exclude name="AvoidUsingShortType"/>
    <exclude name="AvoidUsingShortType"/>
    <exclude name="AddEmptyString"/>
  </rule>

</ruleset>
```

Figure 4.12 Custom ruleset xml file

After returning with "OK" button, the selected rule sets are listed as in Figure 4.13. "RUN" button starts the analysis.

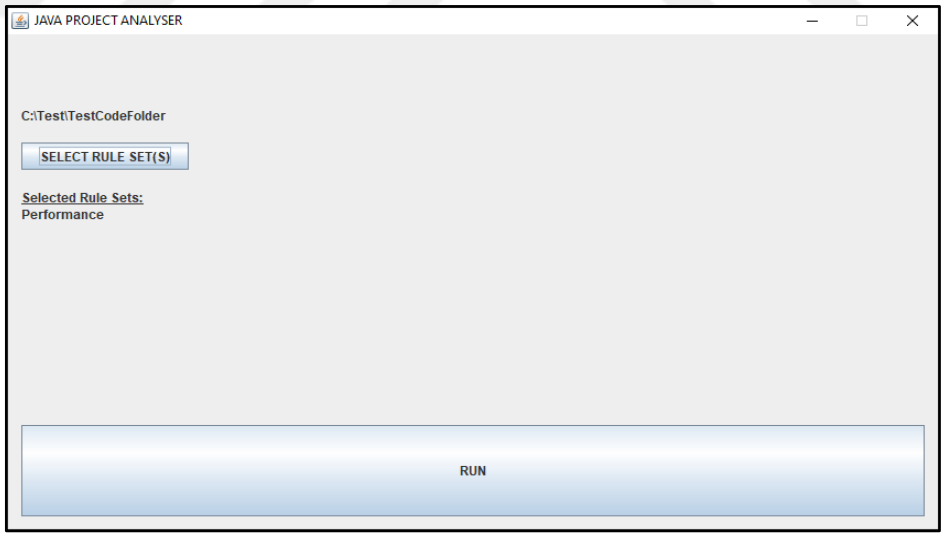

Figure 4.13 After ruleset(s) selected

JPA uses PMD APIs to analyse the project and stores the generated report in the SQLite database with the name ".AOP.db" in the same folder as test codes. To open the report, press the "OPEN REPORT" button as shown in Figure 4.14. The schema of the database is given in Figure 4.15.

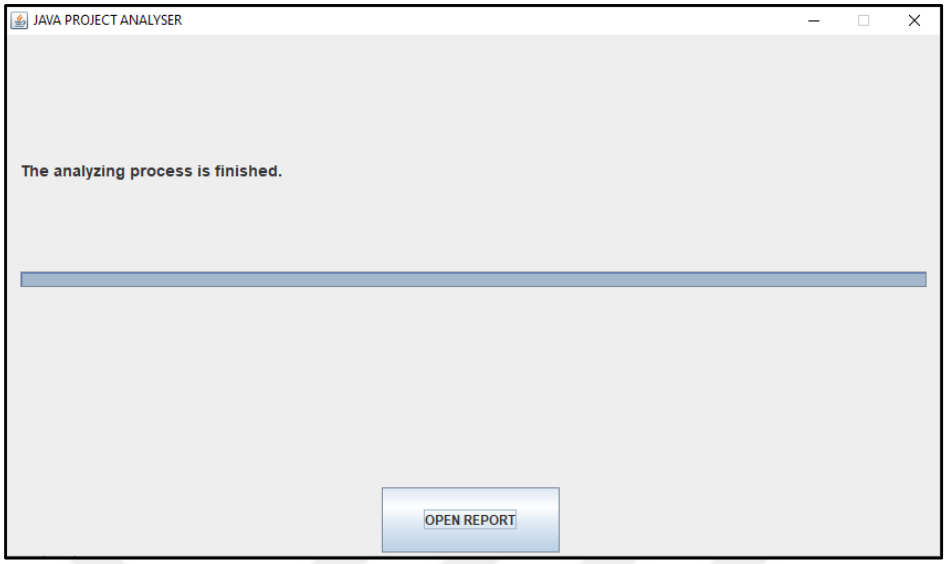

Figure 4.14 Post analysis confirmation screen

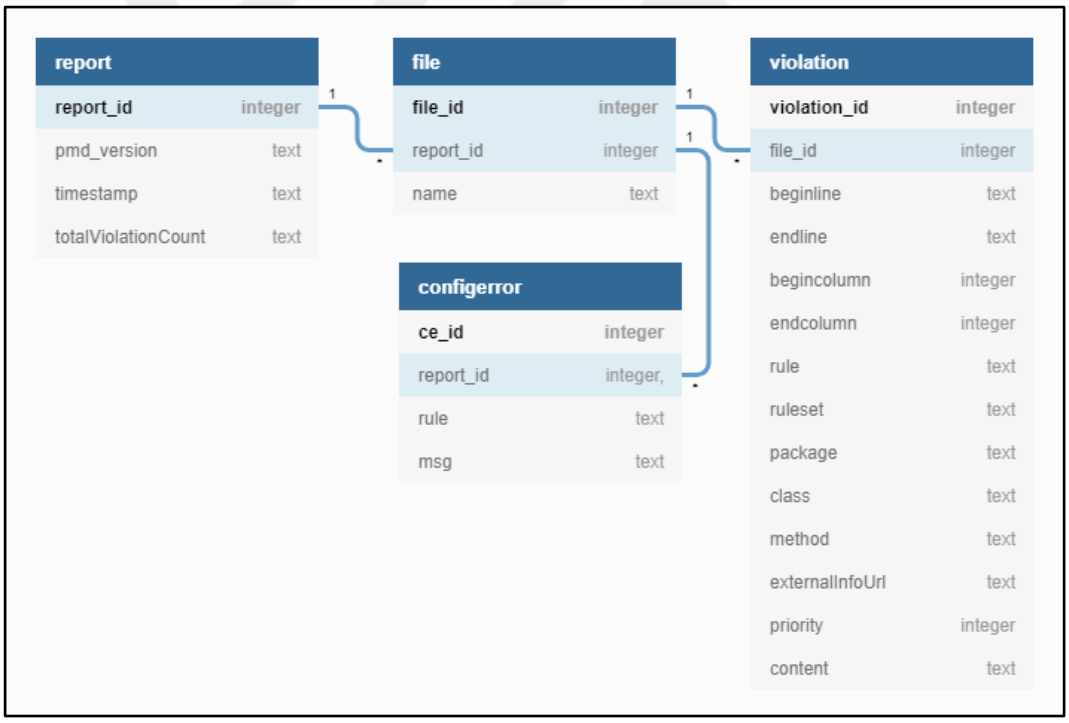

Figure 4.15 Database scheme for ".AOP.db"

As can be seen in Figure 4.16, the report can be reviewed together for all source code files or separately for each source code file.

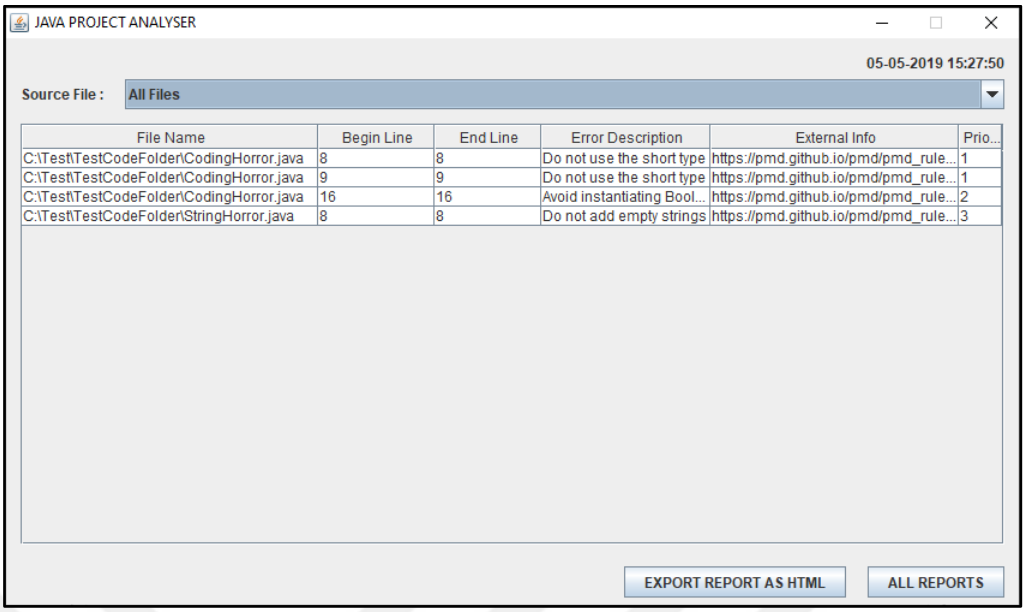

Figure 4.16 Generated report

As can be seen in Figure 4.17, the report can be exported as an "html" file named as the time stamp.

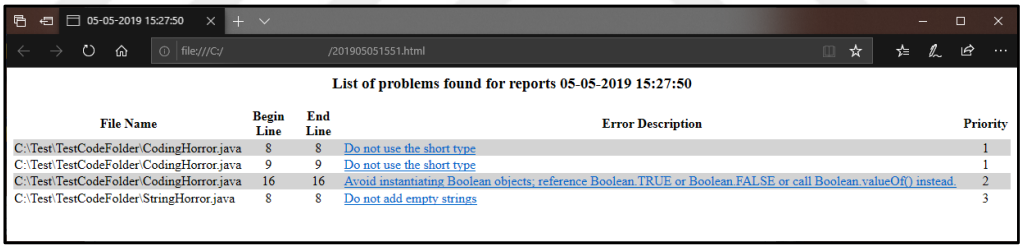

Figure 4.17 Report exported as html

“ALL REPORTS” button opens the window listing all reports. In this window, the old reports can be viewed, the reports can be deleted or compared. At least two reports are needed for comparing or program gives error as expected shown in Figure 4.18.

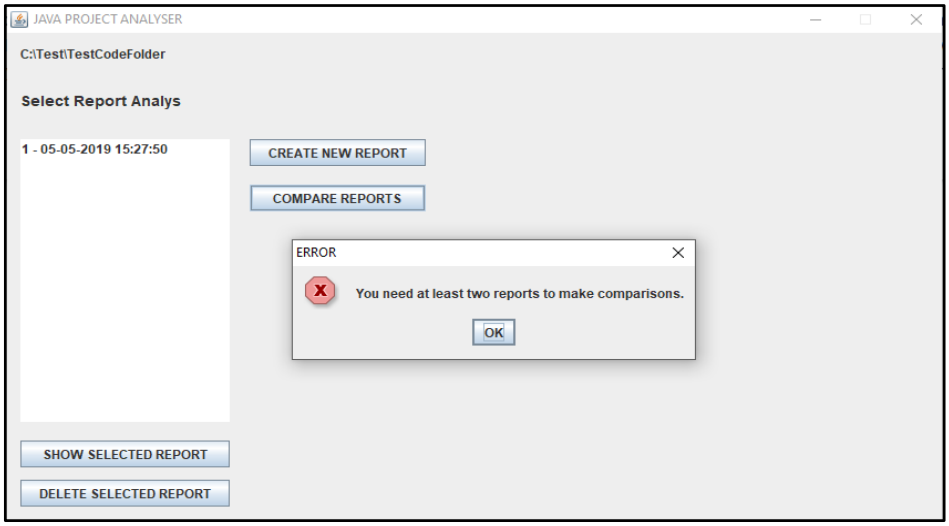

Figure 4.18 Error when try to compare reports

Before creating new report, change the codes like in Figure 4.19 to eliminate violations. In order to ignore a violation and keep the code by PMD, just type "//NOPMD" at the end of that line if the violation consists of a single line. For multi-line violations there are several ways to ignore can be found on PMD documentation under “Suppressing Warnings” title.

In this example a single line suppression was typed in line 8 of the "StringHorror.java" source file, "short" variable type in lines 8 and 9 has been replaced with "int" in the source file "CodingHorror.java". Only the violation in line 16 remains.

CodingHorror.java

```
1  package testSrc;
2
3  public class CodingHorror {
4
5      public static void main(String args[]) {
6
7          //Violations for AvoidUsingShortType -Start-
8          int doNotUseShort = 1;
9          int shouldNotBeUsed = 2;
10         //Violations for AvoidUsingShortType -End-
11         doNotUseShort += shouldNotBeUsed;
12         System.out.println("Short Variable 1 : " + shouldNotBeUsed);
13         System.out.println("Short Variable 2 : " + doNotUseShort);
14
15         //Violations for BooleanInstantiation -Start-
16         Boolean bar = new Boolean("true");
17         System.out.println("Boolean Variable 1 : " + bar);
18         //Violations for BooleanInstantiation -End-
19
20         Boolean buz = Boolean.FALSE;
21         System.out.println("Boolean Variable 2 : " + buz);
22
23
24         String s = StringHorror.retS();
25         System.out.println("String Variable 1 : " + s);
26         String t = Integer.toString(456);
27         System.out.println("String Variable 2 : " + t);
28     }
29 }
```

StringHorror.java

```
1  package testSrc;
2
3  public class StringHorror {
4
5      public static String retS() {
6
7          //Violations for AddEmptyString -Start-
8          String s = "" + 123; //NOPMD
9          //Violations for AddEmptyString -End-
10         return s;
11     }
12 }
```

Figure 4.19 Updated CodingHorror.java and StringHorror.java source code files

Figure 4.20 shows that only “BooleanInstantiation” violation is reported when the analysis is performed again.

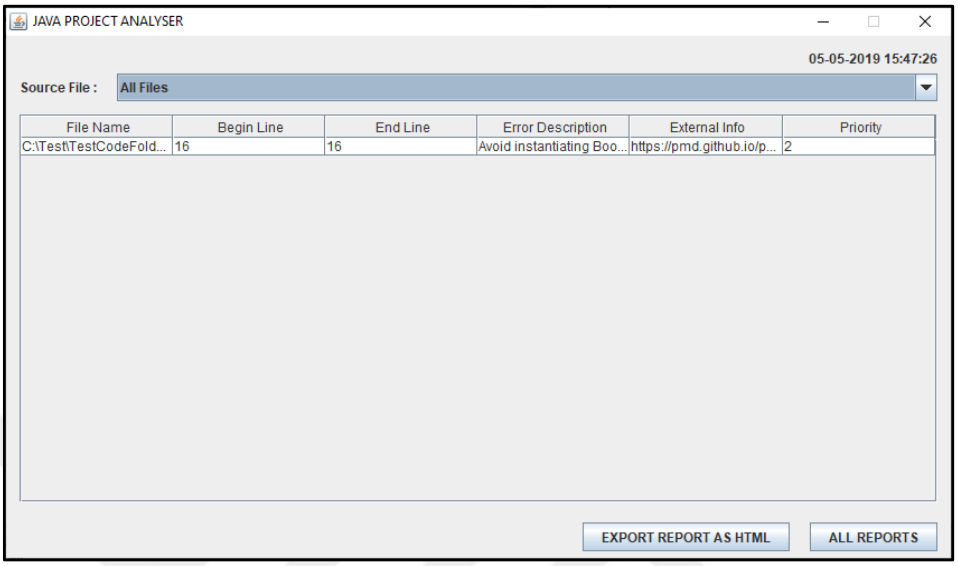

Figure 4.20 Generated report for updated codes

When comparing the two reports, as in Figure 4.21, only this violation is common.

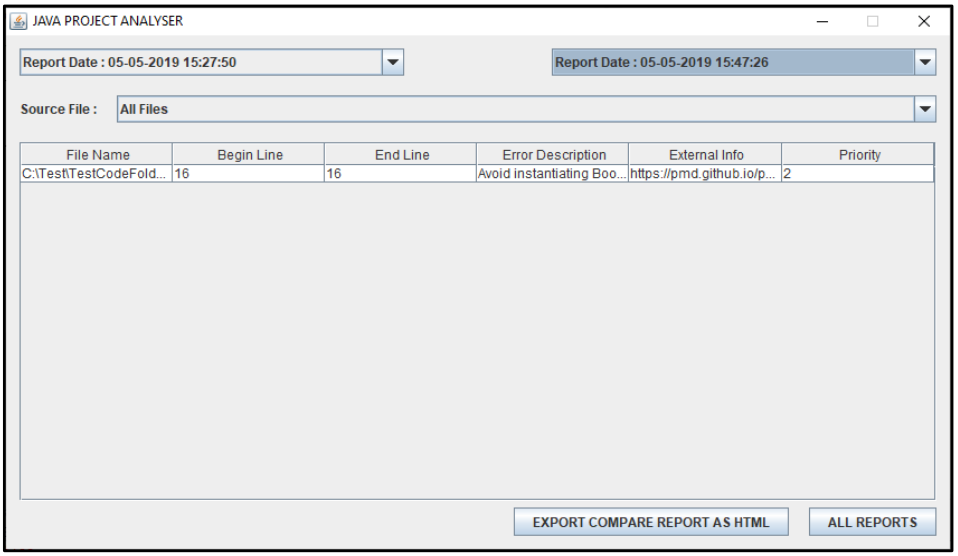

Figure 4.21 Comparison of reports

As in a single report view, the comparison of two reports can be exported as “html”. Figure 4.22 shows the comparison of reports exported as “html”.

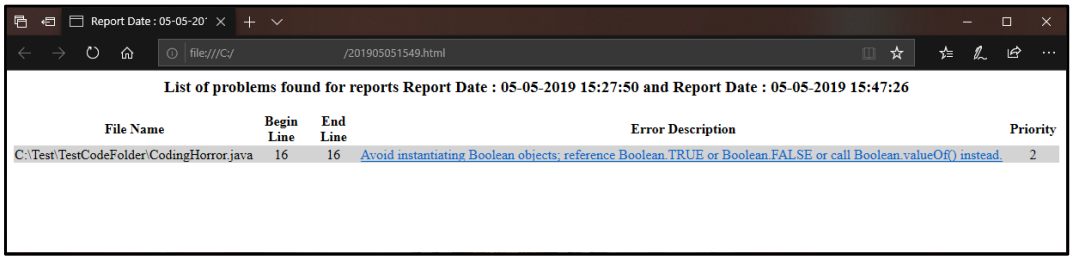

Figure 4.22 Comparison of reports exported as html

When reviewing reports in JPA, documentation page of a violation can be accessed from the column named "External Info". The same applies to "Error Description" column when exporting the report as "html". The image of a sample documentation page for “BooleanInstantiation” violation that can be accessed through the link in the report in Figure 4.22 can be seen in Figure 4.23.

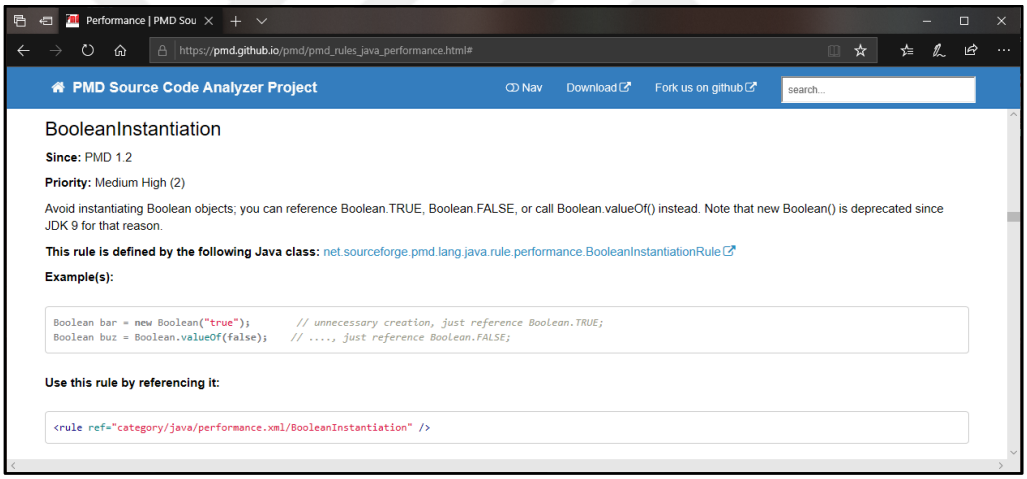

Figure 4.23 Documentation page for “BooleanInstantiation” violation

If a previously analysed folder is selected as destination while creating a new project, it will notify user of the issue and ask whether to load the existing project or not, as shown in Figure 4.24. Similarly selecting an un-analysed folder while loading a project will bring an appropriate notification, asking the user to create a new project or not as in Figure 4.25.

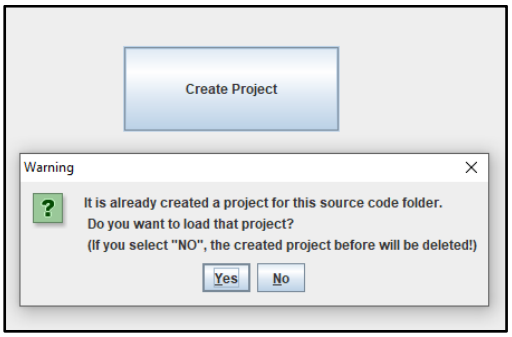

Figure 4.24 Warning for an already existing project where the new project is wanted to be created

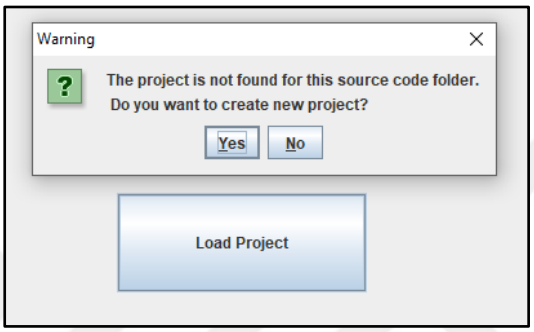

Figure 4.25 Warning for no existing project where the project is wanted to be loaded

4.2.4 Class Diagram

JPA. It is the main class. User interface is defined in this class.

Task. This class generates threads to show progress in user interface.

DBClass. This class contains the functions required for database operations.

XMLtoDB. It is the class used to transfer the report generated by PMD's APIs to the database.

CMBItem. This class defines type of the items that jcombobox's, used in the interface contain.

Figure 4.26 shows JPA’s classes relations as UML class diagram.

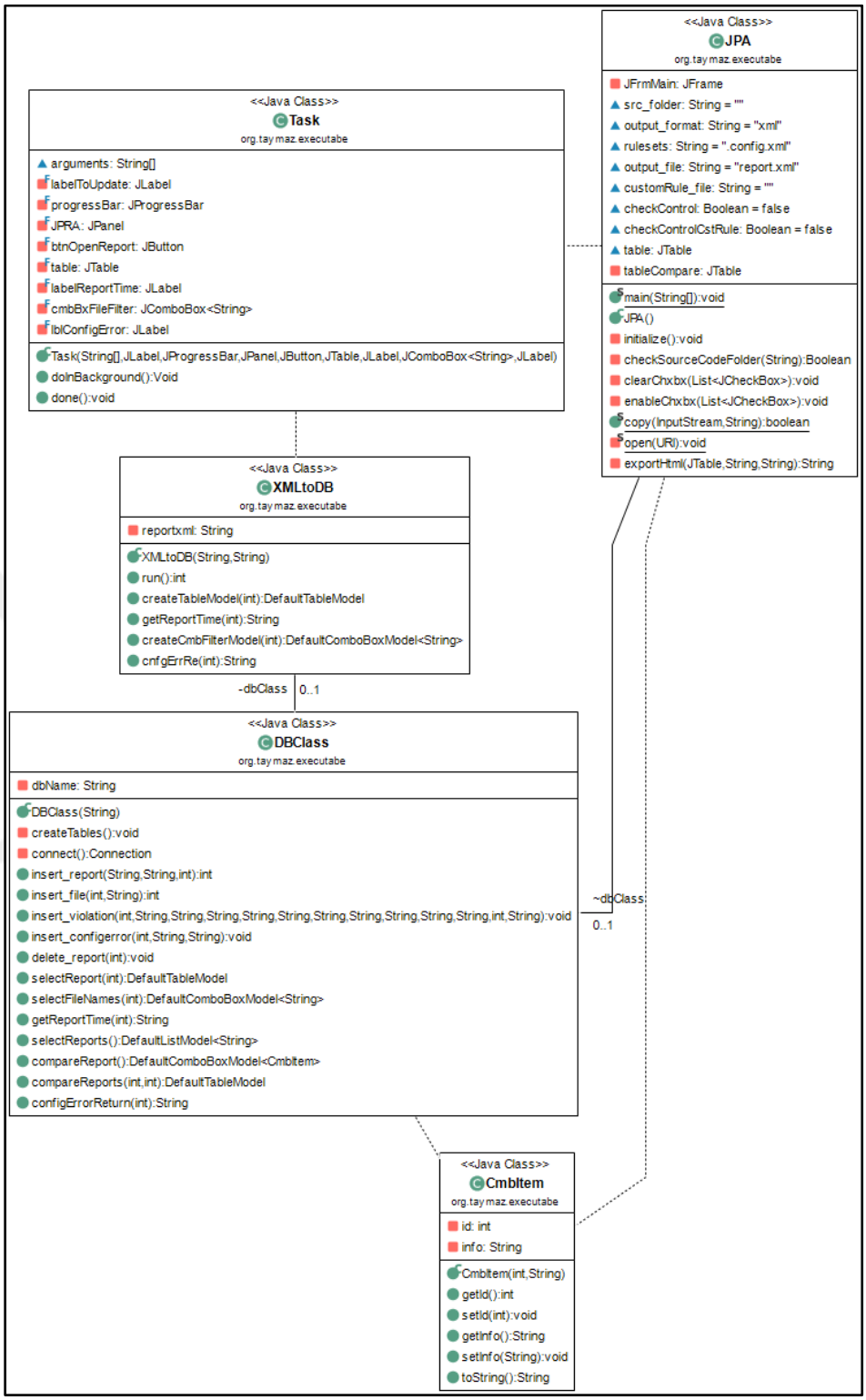

Figure 4.26 Class diagram of JPA program

CHAPTER FIVE
CONCLUSION AND FUTURE WORK

In this thesis, a platform independent tool is designed to perform static code analysis. The speed of execution of Java code is highly dynamic and fundamentally depends on Java Virtual Machine. An old piece of Java code may well run faster on a more recent JVM, even without recompiling Java source code. Combining this fact with the possibility that refactoring may not sufficiently improve the software performance, software testing approach was preferred. It was decided to implement white box approach at source code level with statement coverage to optimise code. In order to cover all code lines, a tool has been developed based on PMD source code analyzer to perform static code analysis automatically and compare the analysis outputs.

Although PMD itself was a good starting point for static analysis there were obvious shortcomings to be improved. Plugins have been developed to integrate PMD source code analyzer into IDEs, Ant and Maven build tools by third parties. JPA, the tool built in this study, on the other hand has its own graphical interface and also can store and compare reports in its own database of projects. It has fundamentally been developed to improve the user experience in automating static code analysis. As a standalone program, JPA only needs source code for code analysis and does not require any IDEs or compilers. Since JPA keeps the reports methodically in a database form it can also be a useful tool in software testing development process.

This thesis provides detailed information about PMD source code analyzer and the tool built on PMD through its API's. Future studies may be done on automatic estimation for deciding which rulesets to use in code analysis using machine learning. Additionally, custom rule creation process may also be studied for improvement in the future. Furthermore, forthcoming updates and developments in PMD may inspire similar studies, or new ideas under new conditions that prevail at the time.

REFERENCES

Abdallah, M. M., & Al-Rifaee, M. M. (2017). Java Standards: A Comparative Study. *International Journal of Computer Science and Software Engineering*, *6* (6), 146-151.

Aderhold, M., & Kochtchi, A. (2013). Tailoring pmd to secure coding. *Technical Report TUD-CS-2013-0245*

Ayewah, N., Pugh, W., Hovemeyer, D., Morgenthaler, J. D., & Penix, J. (2008). Using static analysis to find bugs. *IEEE Software*, *25* (5), 22-29.

Bajwa, M. S., Agarwal, A. P. & Gupta, N. (2016) Code optimization as a tool for testing software. *3rd International Conference on Computing for Sustainable Global Development*, 961–967.

Benjamin, J. E., Gough, J. & Newland, C. (2018) *Optimizing Java: Practical Techniques for Improving JVM Application Performance*, Retrieved January 4, 2019, from https://www.oreilly.com/library/view/optimizing-java/9781492039259/ch01.html.

Carpenter, B., Chang, Y. J., Fox, G., Leskiw, D., & Li, X. (1997). Experiments with 'HP Java'. *Concurrency: Practice and Experience*, *9*(6), 633-648.

Eclipse (n.d.). Retrieved April 27, 2019, from https://www.eclipse.org.

Galin, D. (2004). *Software quality assurance: from theory to implementation*. India: Pearson Education.

Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A. & Smith, D. (2018) *The Java(TM) Language Specification Java SE 11 Edition*, Retrieved November 14, 2018, from https://docs.oracle.com/javase/specs/.

Hall, S. P. & Anderson, E. (2009) Operating systems for mobile computing. *Journal of Computing Sciences in Colleges, 25* (2), 64-71.

Invesis (n.d.). Retrieved February 11, 2019, from https://www.invensis.net/it-outsourcing-services/outsource-software-testing-quality-assurance-qa-service.

Javapapers (2016). Retrieved December 28, 2018, from https://javapapers.com/core-java/differentiate-jvm-jre-jdk-jit/.

Jovanović, I. (2009) Software testing methods and techniques. *The IPSI BgD Transactions on Internet Research, 5* (1), 30-41.

Johnson M. (2008) Code Optimization. Handout 20.

Karnavel, K., & Santhoshkumar, J. (2013). Automated software testing for application maintenance by using bee colony optimization algorithms (BCO). *In 2013 International Conference on Information Communication and Embedded Systems (ICICES)*, 327-330.

Khan, M. E., & Khan, F. (2012). A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications, 3* (6), 12-15.

Knuth, Donald E. (1974). Computer programming as an art. *Communications of the ACM, 17* (12), 667-673.

Kotlin (n.d.). Retrieved January 5, 2019, from https://kotlinlang.org.

Lins, F. M. (2017). *The effects of the compiler optimizations in embedded processors reliability.* MSc Thesis, Universidade Federal Do Rio Grande Do Sul, Porto Alegre.

McConnell, S. (2004). *Code complete* (2nd ed.). Redmond, Washington: Microsoft Press.

Merriam-Webster (n.d.). Retrieved November 15, 2018, from https://www.merriam-webster.com/dictionary/optimization.

Moreira, J. E., Midkiff, S. P., Gupta, M., Artigas, P., Wu, P., & Almasi, G. (2001). The ninja project: Making java work for high performance numerical computing. *Commun. ACM, 44*(10), 102-109.

Nembhard, F., Carvalho, M., & Eskridge, T. (2017). A hybrid approach to improving program security. *In 2017 IEEE Symposium Series on Computational Intelligence (SSCI)* 1-8.

Oaks, S. (2014). *Java performance: The definitive guide,* Retrieved December 30, 2018, from https://www.oreilly.com/library/view/java-performance-the/9781449363512/ch04.html.

Palaniappan S. (2016). Recent trends and challenges in source code optimization. *International Journal of Trend in Research and Development, 3*(6), 603-607.

Parkinson, A. R., Balling, R. J. & Hedengren, J. D. (2013). *Optimization methods for engineering design.* Provo: Brigham Young University.

PMD (n.d.). Retrieved April 3, 2019, from https://pmd.github.io/pmd-6.5.0/pmd_rules_java.html.

Sawant, A. A., Bari, P. H., Chawan & P. M. (2012). Software testing techniques and strategies. *International Journal of Engineering Research and Applications (IJERA),* 2 (3), 980-986.

Scala (n.d.). Retrieved January 5, 2019, from https://www.scala-lang.org.

Singh, A. H., & Kazi, N. N. (2016). *Software Testing* Mumbai: Himalaya Publishing House Pvt. Ltd.

Statcounter (n.d.). Retrieved November 15, 2018, from http://gs.statcounter.com/os-market-share.

SQLite (n.d.). Retrieved May 1, 2019, from https://sqlite.org.

Tomas, P., Escalona, M. J. & Mejías, M. (2013). Open source tools for measuring the Internal Quality of Java software products. A survey. *Computer Standards & Interfaces*, *36* (1), 244-255.

Watson, M. (2017). *Why premature optimization is the root of all evil.* Retrieved January 3, 2019, from https://stackify.com/premature-optimization-evil/.