

AN EYE TRACKING STUDY ASSESSING CODE READABILITY

by

Nishitha Yedla

Submitted in Partial Fulfilment of the Requirements

for the Degree of

Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

May, 2017

Nishitha Yedla

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Centre and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

---

*Nishitha Yedla*, Student Date

Approvals:

---

*Bonita Sharif*, Thesis Advisor Date

---

*Alina Lazar*, Committee Member Date

---

*Feng Yu*, Committee Member Date

---

Dr. Salvatore A. Sanders, Associate Dean of Graduate Studies Date

## **Abstract**

Software developers spend considerable time on a wide variety of tasks such as fixing bugs and implementing new features. All these tasks require reading and understanding code. One of the key ideas for improving maintenance processes is that code should be highly readable so that it is easy to understand. The hypothesis behind this is that the degree of understandability of the source code has a crucial impact on cost and effort. To test this theory, we conducted an eye tracking study to determine how two types of code readability rules affect performance. We focused specifically on two coding practices: minimize-nesting rule and avoid do/while loops. We also gave participants two ranking tasks, one for each rule. The results show a higher accuracy in solving tasks in the minimize-nesting rule programs and avoid do/while rule programs but only for the correct methods that followed the rule. No significant difference is found in the amount of time spent to analyze the methods that followed the minimize-nesting rule and avoid do/while rule. For minimize-nesting rule, we found significant difference in ease of readability and level of confidence when the method followed the rule. For minimize nesting rule, the visual effort was less to analyze conditional statements and the overall method when the method followed the rule. The ranking for readability was higher for methods that followed the rule. In the method comparison task, the visual effort (in terms of fixations and their durations) was less to analyze the conditional statements of the methods that followed the avoid do/while rule. The results of this study can be used by developers and practitioners to create coding style guides based on these rules.

## **Acknowledgements**

Firstly, I would like to express my true thankfulness to my advisor, Dr. Bonita Sharif. I feel very honoured to work with her whose motivation, understanding, and patience helped me to complete my research. I appreciate her vast knowledge and skills in many areas and her assistance in writing my thesis report. I could not have imagined having a better advisor and mentor for my Masters. I will forever be thankful.

I would like to thank Dr. Aponte and his student Sergio Luis Lubo Argumedo from National University of Colombia who worked with us on a similar online study. Further, I would like to thank the esteemed members of my committee, Dr. Alina Lazar and Dr. Feng Yu for the assistance they provided in this research project.

A special thanks to my family for all the motivation, support given by them and the reason being for my graduate career. I would also thank the Department of Computer Science and the STEM College for the financial assistance during my graduate studies.

## TABLE OF CONTENTS

<b>LIST OF FIGURES .....</b>	<b>VIII</b>
<b>LIST OF TABLES .....</b>	<b>XI</b>
<b>CHAPTER 1 INTRODUCTION.....</b>	<b>1</b>
1.1 Motivation.....	2
1.2 Contributions.....	3
1.3 Research Questions.....	3
1.4 Organization.....	4
<b>CHAPTER 2 BACKGROUND AND RELATED WORK.....</b>	<b>6</b>
2.1 Code Readability Studies.....	6
2.2 Eye Tracking Overview .....	12
2.2.1 Eye-tracking Studies in Software Engineering.....	12
2.2.2 Program Comprehension .....	15
2.2.3 Debugging.....	19
2.3 Eye tracking Studies in other domains.....	20
<b>CHAPTER 3 THE EYE TRACKING STUDY .....</b>	<b>28</b>
3.1 Experiment Design.....	28
3.2 Hypotheses.....	30
3.3 Participants.....	32
3.4 Tasks .....	34

3.5	Data collection .....	37
3.6	Eye-Tracking Apparatus .....	38
3.7	Conducting the Study.....	38
<b>CHAPTER 4 RESULTS AND ANALYSES .....</b>		<b>41</b>
4.1	Accuracy .....	41
4.1.1	Rule 1 .....	41
4.1.2	Rule 2 .....	43
4.2	Time .....	44
4.2.1	Rule 1 .....	44
4.2.2	Rule 2 .....	46
4.3	Visual Effort.....	47
4.3.1	Creating Areas of Interest .....	47
4.3.2	Fixation Counts .....	48
4.3.3	Fixation Durations .....	55
4.4	Ease of readability.....	62
4.4.1	Rule 1 .....	63
4.4.2	Rule 2 .....	63
4.5	Level of confidence.....	64
4.5.1	Rule 1 .....	64
4.5.2	Rule 2 .....	65
4.6	Method Comparison.....	66
4.6.1	Rule 1 .....	66

4.6.2 Rule 2.....	70
4.7 Post Questionnaire Results .....	73
4.8 Observations and Discussion .....	74
4.9 Threats to Validity .....	76
<b>CHAPTER 5 CONCLUSIONS AND FUTURE WORK.....</b>	<b>78</b>
APPENDIX Study Material.....	80
A.1. Study Instructions .....	80
A.2. Pre-Questionnaire.....	81
A.3. Tasks and Comprehension .....	83
A.4. Post Questionnaire .....	116
References.....	118

## LIST OF FIGURES

Figure 1. Proficiency in English and Knowledge in Java.....	32
Figure 2. Mother Tongue Distribution.....	33
Figure 3. Number of years actively programming in Java and other languages.....	33
Figure 4. Accuracy based on logical correctness for Rule 1 (minimize-nesting).....	42
Figure 5. Accuracy based on certain input for Rule 1 (minimize-nesting).....	42
Figure 6. Accuracy based on logical correctness for Rule 2 (avoid do/while).....	43
Figure 7. Accuracy based on certain input for Rule 2 (avoid do/while).....	44
Figure 8. Time taken to analyze Rule 1 (minimize-nesting) problems.....	45
Figure 9. Time taken to analyze Rule 2 (avoid do/while) problems.....	46
<b>Figure 10. Areas of Interest.....</b>	<b>47</b>
Figure 11. Gaze plot.....	48
Figure 12. Overall fixation count for Rule 1 (minimize-nesting).....	50
<b>Figure 13. Fixation count for problem statements for Rule 1 (minimize-nesting)....</b>	<b>51</b>
Figure 14. Fixation count for conditional statements for Rule 1 (minimize-nesting) .....	52
Figure 15. Overall fixation count for Rule 2 (avoid do/while).....	53
Figure 16. Fixation count for Rule 2 (avoid do/while) problem statement .....	54
Figure 17. Fixation count for Rule 2 (avoid do/while) conditional statements .....	55
Figure 18. Overall fixation duration for Rule 1 (minimize-nesting) .....	57
<b>Figure 19. Fixation duration for Rule 1 (minimize-nesting) problem statement.....</b>	<b>58</b>
Figure 20. Fixation duration for Rule 1 (minimize-nesting) conditional statements.....	59



Figure 21. Overall fixation duration for Rule 2 (avoid do/while).....	60
Figure 22. Fixation duration for Rule 2 (avoid do/while) problem statement .....	61
Figure 23. Fixation duration for Rule 2 (avoid do/while) conditional statements .....	62
Figure 24. Ease of readability for Rule 1 (minimize-nesting rule).....	63
Figure 25. Ease of readability for Rule 2 (avoid do/while) .....	64
Figure 26. Level of confidence for Rule 1 (minimize-nesting) .....	65
Figure 27. Level of confidence for Rule 2 (avoid do/while) .....	66
Figure 28. Readability ranking for Rule 1 (minimize-nesting).....	67
Figure 29. Overall Fixation Count for the two methods in Rule 1 (minimize-nesting) comparison task .....	67
Figure 30. Overall fixation duration for the two methods in Rule 1 (minimize-nesting) comparison task .....	68
Figure 31. Conditional statements fixation count for the two methods in Rule 1 (minimize-nesting) comparison task.....	69
Figure 32. Conditional statements fixation duration for the two methods in Rule 1 (minimize-nesting) comparison task.....	69
Figure 33. Readability ranking for Rule 2 (avoid do/while).....	70
Figure 34. Overall fixation count for the two methods in Rule 2 (avoid do/while) comparison task .....	71
Figure 35. Overall fixation duration for the two methods in Rule 2 (avoid do/while) comparison task .....	71

Figure 36. Conditional statements fixation count in Rule 2 (avoid do/while) comparison task .....	72
Figure 37. Conditional statements fixation duration in Rule 2 (avoid do/while) comparison task .....	73
Figure 38. Importance of minimize-nesting rule .....	73
Figure 39. Importance of avoid do/while loop rule .....	74

## LIST OF TABLES

Table 1. Experiment overview .....	29
Table 2. Treatment combination used in the study for each problem.....	30
Table 3. Possible trials used in the study. The columns in grey indicate comparison tasks for ranking two code snippets .....	35
Table 4. Overview of problem statements used in R1 (Minimize-nesting rule).....	36
Table 5. Overview of problem statements used in R2 (Avoid do/while rule) .....	37
Table 6. Hypotheses results .....	76

# CHAPTER 1

## INTRODUCTION

Program reading is an important programmer activity, as code that is readable is often considered more maintainable. Readability is a key factor in overall software quality as it is related to maintenance (Buse and Weimer 2010). Maintenance of existing software consumes 70% of the total software lifecycle. Programmers take a lot of time in maintaining the software than writing code from scratch. Maintenance of software involves tasks such as modifying the code or adding new features to the code written by someone else. Source code can be considered readable to the programmer who has written it, but it might not be understood easily by others. The original author of the program is usually unavailable, and the programmer who maintains the existing software has no other choice than to read and understand the code he is required to fix (Deimel 1985).

In general, reading plays a crucial role in tasks such as debugging, analysis, maintenance, comprehension, and most importantly learning. Though modern IDEs come with lot of features to help developers read the code more comfortably and quickly, readability depends largely on how the programmer writes the source code. Many textbooks have described programming practices to write a readable code to help developers in writing a code that is more readable and easily understandable by others. In this project, we conducted an empirical study using eye tracking equipment to understand how two programming practices impact developers on efficiency, visual effort, time, ease

of readability, level of confidence and accuracy in understanding the code.

## **1.1 Motivation**

Developers are likely to introduce new bugs while trying to fix the old ones or while adding new features to the existing software if the code is difficult to read. This makes it difficult for developers to understand the flow and side effects of the code. Several studies have shown that software maintenance consumes a large percentage of the overall lifecycle costs. The changing software often demands a lot of time and effort. Therefore, researchers have created several theories, techniques, practices, and tools with the aim of improving the wide variety of maintenance tasks.

A list of parameters have been investigated to predict the readability of a program into the following categories: Reader characteristics, intrinsic factors, representational factors, typographic factors, and environmental factors (Lionel and J.Fernando 1990). One of the key ideas for improving maintenance processes is that code should be highly readable so that it is easily understood. The hypothesis behind this is that the degree of understandability of source code has a crucial impact on the cost and effort required for evolving a software system. In this study, we mainly focused on two programming practices aiming at making the control flow in the source code easy to read. Hence, we are doing this research to provide empirical evidence of the impact of good programming practices so that it helps the programmers to design better coding style guides based on the results provided from this study.

## 1.2 Contributions

The main contribution of this thesis is an empirical study that assesses how good programming practices impacts developers in readability of source code. Data collection was done using an eye tracker. Subjects were students at Youngstown State University. The goal of this thesis is to develop better teaching strategies that helps the programmers to create coding style guides so that they can write code that is more readable. However, to do this, we first need to conduct several studies to determine individual behavior by using an eye tracker to determine if any differences exist before we can generalize this process. The contributions are listed as follows:

- First eye-tracking study analyzing code readability rules.
- Two code readability rules were tested: minimize-nesting and avoid do/while loops
- The effect of correctness of the code in combination with the two rules was tested.
- A qualitative ranking was obtained for two code readability rules with respect to two programs.

## 1.3 Research Questions

The research questions we seek to answer are given below.

### **Research Question 1 based on the minimize-nesting rule**

**RQ1a:** Does the minimize-nesting rule reduce the *time* a developer spends to understand source code?

**RQ1b:** Does the minimize-nesting rule increase the *accuracy* of a developer in understanding source code?

**RQ1c:** Does the minimize-nesting rule increase the *level of confidence* a developer has after reading source code?

**RQ1d:** Does the minimize-nesting rule improve the *ease of readability* a developer faces when reading source code?

**RQ1e:** Does the minimize-nesting rule reduce the *visual effort* of a developer when reading source code?

### **Research Question 2 based on the avoid do/while rule**

**RQ2a:** Does the avoid do/while rule reduce the *time* a developer spends to understand source code?

**RQ2b:** Does the avoid do/while rule increase the *accuracy* of a developer in understanding source code?

**RQ2c:** Does the avoid do/while rule increase the *level of confidence* a developer has after reading source code?

**RQ2d:** Does the avoid do/while rule improve the *ease of readability* a developer faces when reading source code?

**RQ2e:** Does the avoid do/while rule reduce the *visual effort* of a developer does when reading source code?

## **1.4 Organization**

This thesis is organized as follows. The next chapter gives a brief introduction to eye tracking and related work. Chapter 3 presents the details of the experimental design and process used for the study. Chapter 4 discusses observations and results. Chapter 5

concludes the thesis and presents future work. Parts of this thesis will be published in peer-reviewed conferences and journals.



## CHAPTER 2

### BACKGROUND AND RELATED WORK

This chapter first presents an overview of empirical studies in program readability in various tasks and settings. Next, the chapter presents related work in eye tracking and its use in the software engineering domain.

#### 2.1 Code Readability Studies

A simple readability measure for software, Software Readability Ease Score (SRES) is proposed and applied to object-oriented textbook examples based on the idea of Flesch. The Flesch Reading Score (FRES) is a readability measure of natural language (Dubay 2004; Rudolph 1948). The FRES is computed using average sentence length (ASL-words/sentences) and average word length (AWL-syllables/words) on a scale 0...100. Texts with  $FRES < 30$  are considered very difficult and  $FRES > 90$  are considered very easy.

$$FRES = 206.835 - 1.015ASL - 84.6AWL$$

In a similar way, SRES is calculated by interpreting the lexemes (identifiers, keywords and symbols) of a programming language as ASL and AWL as average number of words per statement or block (delimited by curly brackets and semicolons) (Abbas 2009).

$$SRES = ASL - 0.1AWL$$

Buse and Weimer (Buse and Weimer 2010) proposed a readability measure based on 25 code features like number and length of identifiers, counts of various syntactical elements and structures, line length, and indentation which are judged by human annotators for

small code snippets extracted from production code. Maintainability Index (MI) is a measure used for the measurement of maintainability of software systems. The quality of object-oriented software is characterized by using simple composite measures like Number of Java statements (NOS) and Cyclomatic Complexity (CC). Halstead's effort calculated the effort required to construct a program based on number of operands and operators. SRES was compared with the readability measures including B&W (Buse & Weimer-measure), NOS (Number of Java statements), MI (Maintainability Index), CC (Cyclomatic Complexity), E (Halstead's Effort) and found to be a useful tool for helping educators in the selection and development of suitable programs.

Several software quality metrics have been proposed and are validated empirically proving that readability is one of the important factors affecting maintainability. A study was conducted to analyse the scan patterns of eye movements for subjects of two experience level, when they view a short complex algorithm written in Pascal and (Crosby and Stelovsky 1990). The experiment was conducted with 19 volunteers which include 10 subjects with low-experience, eight graduate students with high-experience and one PhD faculty member. The binary search algorithm written in Pascal code was used in the study. The results show that experience influences the viewing strategies of subjects with respect to reading time and spend more time focusing on meaningful areas. With increase in experience, subjects learn to discover and focus on the key areas of information.

A book by Boswell and Foucher was written in 2011 with a key idea to make programs highly readable and easily understood (Boswell and Foucher 2011). Many

example programs from different languages including C++, Python, JavaScript, and Java are illustrated and analysed to show what made the code bad in terms of readability. Several principles and techniques like surface-level improvements (naming, commenting, and aesthetics), simplifying loops and logic, reorganizing the code were discussed based on a fundamental theorem of readability-code. It would be easy to read a code if it has conditions, loops, or any other control flow statements. These jumps and branches makes the code confusing quickly. Some key points that helps the learners to avoid reread the code were presented in this book.

When a do/while loop is used, a block of code may or may not be executed based on the condition underneath it. One reads the code from top to bottom, which makes the programmers ending up twice in reading the code in case of do/while loops. When while loops are used, the reader knows the condition for all the iterations inside the block which makes while loops easier to read. A reader requires more concentration to understand deeply nested code, since at each level of nesting an extra condition needs to be pushed onto to the reader's mind. Linear code is better choice to avoid deep nesting. These different aspects help the programmers to write easy code. These two programming practices are mainly focused in this study.

Learning object-oriented programming depends on the individual experience, interest and cognitive capabilities. A study was conducted focusing mainly in determining the quality of example programs taken from different Java textbooks to help students in learning object-oriented programming (Abbas 2009). A set of quality attributes were proposed in this study to distinguish between good and bad examples.

Different readability metrics were calculated and were compared. It was observed that comments in a program have a fair impact on readability and quality. The results show that SRES scores differently on a set of example programs evaluated and it can also add some weights based on comments, beacons, coding standards, indentation, and other readability factors as SRES does not consider comments, white-spaces, and indentation style at present.

Examples play an important role in teaching and learning programming. A novice programmer can learn from good examples and will be able to know the desirable properties on their own from the programs. A study was conducted in which the experienced educators evaluated the quality of object-oriented example programs for novices taken from popular Java textbooks (Börstler, Nordström, and Paterson 2011). The examples are categorized as First User Defined Class, Multiple User Defined Class, and Control Structures. These examples were rated on a 7-point Likert-type scale based on 10 quality factors, which are classified into three categories: technical, object-oriented, and didactic qualities. Results show that the example quality was not as good as one might expect from common textbooks. Educators should be careful when taking examples directly from the textbooks.

Human cognition is reflected in the text of computer programs. A study was conducted to explore different ways in which programmers select and use names in code (Liblit, Begel, and Sweetser 2006). Morphological and Metaphorical regularities, grammatical sensibility in name use, containers (objects) and paths (pointers), and overloading phenomena are mainly focused. Lexical and morphological conventions

convey basic. The results show that programmers leverage fundamental aspects of cognition and natural language comprehension to make code easier to read and understand.

A study has been conducted to assess the impact of 25 proposed readability features on overall code readability in positive or negative way (Tashtoush, Zeinab, and Yatim 2013). It was observed that meaningful names, comments, and consistency has high positive impacting improving readability. Whereas, recursive functions, nested loops and arithmetic formulas were found to be a negative impact on the general readability attribute. Others, such as, short scopes, identifier name length, identifier frequency, inheritance, overriding, if-else statement, switch statement, loops (for, while, do-while), and array were found to have no major effect on readability.

A recent study presented a new code readability testing technique to determine whether the technique increases programmers' ability in writing readable code (Sedano 2016). Twenty-one programmers followed code readability testing in four sessions, where the experienced programmers were instructed to read code samples and think out loud expressing the reader's thought in understanding the code. The results indicated that most of the programmers could write readable code from unreadable code after the four sessions. The study identified several fixes to unreadable code which includes improvements to variable names, improvements to method names, the creation of new methods to reduce code duplication, simplifying if conditions and structures, and simplifying loop conditions.

Refactoring, which means to improve the program's structure without changing its behavior makes a program more readable. A study was conducted to refactor 2823 lines of C# code across 111 source files of an educational video game (Dibble II and Gestwicki 2014). The Refactoring in this study is divided into two phases: First, a popular add-on for Microsoft Visual studio developed by JetBrains (ReSharper) was used to determine the effect of refactoring and the second phase of refactoring consisted of six-week manual inspection of 111 source files. ReSharper identifies the issues in the code and applies automated refactorings on each file as recommended. The refactored code is found to be more readable and understandable. Unity3D's built in profiler was used to measure the performance of the system before and after refactoring. It was observed that the performance of the system after refactoring the code has increased. A reduction of 5.46% was observed in the total lines of code. ReSharper identified 970 potential issues within the code which was decreased to 53 by the end of the refactoring process. Though ReSharper could find possible ways to reduce nesting, it was not able to offer any suggestions on how to improve the structure of the method, which was focused in phase two. Identifying and fixing these kinds of problems was entirely done by manual operation.

A study was conducted using an eye tracker to investigate the influence of syntax highlighting on reading source code snippets (Beelders and du Plessis 2016). Participants are given code snippets of either black-and-white or using standard syntax highlighting. It was observed that participant's behavior did not differ and thus there is no effect on the reading behavior.

## **2.2 Eye Tracking Overview**

An eye tracker helps to detect where one is looking on the screen or their movement of eye relative to the head. Eye tracker equipment captures various types of eye movements. When a person looks at a scene, read, or search for an object the eye movements made by them are called *Saccades*. The eye remains still between the eye movements during *fixation* for about 200-300 Ms. *Saccades* and *fixation* are the two common types of eye movements (Busjahn et al. 2015). Only a blur would be perceived during a saccade, because the eyes would be moving so fast across the stable visual stimulus and hence no new information can be obtained. Eye tracking is a source of valuable information, which cannot be obtained by other methods. For example, a programming educator will ask students to report answers after debugging or tracing a program. The outcome after the specific task has ended will be recorded neglecting the information that help understand how and why a student chose an answer (Busjahn et al. 2014). We will now discuss about the studies done in some fields starting with software engineering and program comprehension.

### **2.2.1 Eye-tracking Studies in Software Engineering**

The modern information system is formed by software systems, and many of those things are most complex things ever created. Software engineering concerns the development of new, or modification of existing, technologies (which includes process models, methods, techniques, tools or languages) to support SE activities. The SE research studies also includes the evaluation of the effect of using such technology in

complex interaction of individuals, teams, projects and organizations, and various types of task and software system. Let us now discuss some studies related to this research.

A study was conducted using *Tobii* eye tracker to obtain an understanding of how human subjects use different UML class diagrams in performing their tasks (Sharif and Maletic 2010b). The eye tracker captured activities which includes fixation, saccades, audio, and video. Questions related to the basics of UML class diagram and other related to software design were presented to the participants. They observed that experts analyzed from the centre of the diagram to the edges whereas novices focused from top-to-bottom and left to right.

A workshop has been conducted bringing together educators as well as practitioners to analyze how eye tracking data benefits programming education (Busjahn et al. 2014). Recording sessions were conducted with natural language texts and comprehension questions related to the text so that the subjects would be familiarized with the instruments and tasks. The recording was done with an SMI REDm 120 Hz eye tracker using the OGAMA tracking software. The professional software developers were asked to analyze a Java code and answer few questions related to that method. Subject 1 was told to expect an answer about the return value of the program and subject 2 was told to expect answer for a multiple-choice question about the algorithm in the code. The analysis showed that subject 1, whose gaze was considered erratic, answered correctly whereas, subject 2 whose gaze seemed to be more methodical, chose the wrong answer to a multiple-choice question about the code. It is unlikely that the difference in the accuracy of their answers are directly related to the difference in their gaze patterns as



both the subjects are expert programmers. These eye gaze patterns help the programmers to identify differences in the strategies adopted with various levels of expertise, domain knowledge, and skills. A novice can track themselves and build self-awareness while solving a task using an eye tracker and understand how he was thinking by where he was looking. These eye gaze data can be used to explore expert's strategies and develop teaching materials. This study shows that experts can read the same code using different strategies and the one that is offered to the learners should be used from the consistent ways used by many expert programmers. Individual students can also adapt a strategy that fits well with their own level of approach.

A study was conducted to investigate the benefit of iTrace on large software systems (Sharif, Shaffer, et al. 2015). The design of iTrace is flexible to record eye movements on software artifacts which includes Java code, text/html/xml documents, diagrams, and IDE user interface elements. The study was conducted with 22 software developers using an Eclipse plugin. The study was compared with mylyn interaction history data, which were gathered simultaneously and the results showed that iTrace captures more contextual data on source code elements from different aspects of developer's activity compared to interaction data.

A systematic literature review (SLR) evaluated the current state of eye-trackers in software engineering to provide evidence on the uses and contribution of eye-trackers in software engineering (Sharafi, Soh, and Guéhéneuc 2015). To analyze the articles that have been published using eye trackers in software engineering research from 1990 to 2014, an automated search was performed using Engineering Village instead of

performing manual search. Engineering visualization is an information discovery platform that is connected to several trusted engineering literature databases. Out of 649 publications found, 35 relevant papers were identified to the uses of eye-tracking in software engineering. The results of eye-tracking studies determine how participants perform different software engineering tasks and how they use different models along with the source code to understand software systems. The eye-tracking studies were analyzed using different categories which includes model comprehension, debugging, code comprehension, collaborative interactions, and traceability. Recommendations were also suggested for new researchers and who wanted to perform an eye-tracking study in software engineering community.

Behavior of the developers' while performing a realistic change task was investigated in a study using both eye tracking and interaction data (Sharif, Kevic, et al. 2015). The study was conducted with 22 developers, were the developers were asked to work in three change tasks of JabRef open source system. The analysis show that the gaze data collected by the eye tracker contains more data than the interaction data. The behavior of the developers indicates that very few lines of code are observed by developers when working on a realistic change task. Developers chase variables flows within the methods, rarely follow call graph links and mostly only switch to the elements close to the method within in the class when it comes to switches within methods.

### **2.2.2 Program Comprehension**

A study was conducted using an eye tracker to investigate if developers read error messages within the Eclipse IDE (Barik et al. 2017). Developers often get compile time

error messages through a variety of textual and visual mechanisms in integrated development environments. Errors are represented as textual and visual mechanisms, such as popups and wavy red underlines. To understand how developers' use error messages, an eye tracking study was conducted with 56 participants. The participants were asked to resolve problematic, common defects in Java code within in the Eclipse development environment and provide a reasonable solution for the defect.

A Gaze Point GP3 eye tracking instrument was used in this study with GP3 software and drivers installed on the computer to collect both the screen recording of the desktop environment and to synchronize the time of recordings with the eye tracking instrument. If all the unit tests in the Apache Common Collections library are passed after the compile errors are removed, then the solution was considered correct. A comparison was made between source code reading, error messages, and prior work on silent reading. The results showed that developers were reading the error messages and the difficulty of reading these messages was comparable to the difficulty of reading source code. The fixations indicated that developers spend a considerable amount of time on error message area of interest, even though most tasks had a single error message.

Inspired by the linearity that people exhibit while reading natural language text, a recent study designed local and global gaze-based measures to characterize linearity in reading source code (Busjahn et al. 2015). A source code is executable and requires a specific approach to read, unlike natural language text. Eye movements of novices and experts were compared in this study by instructing the participants to read and comprehend short snippets of natural language text and Java programs. The study was

conducted with 14 novices who attended a Java beginner's course at the university and the study with the expert participants was conducted at their office location. Each module given to the novice participant consisted of three programs ranging from few lines of Java code to an entire screen full of text. Experts were given six programs in total, of which two were the same one given to the novices in the study. The SMI RED-m remote eye tracker set to sample at 120 Hz was used in the study and all the eye tracking data was recorded using the open-source tool Ogama (OpenGaze-AndMouseAnalyser). After reading each program the participants were asked to write a summary of the code, write the value of a variable after program execution, or answer a multiple-choice question about the algorithmic idea. The results show that novices read the source code less linearly than natural language text. Experts read code less linearly than novices, which indicates that non-linear reading skills increase with expertise.

The effect of programming language on student in comprehending source code was assessed in a study, where C++ and python languages are compared in two categories: overview and finding bug in a task (Turner et al. 2014). The study was conducted with thirty-eight students, where the participants were instructed to read a complete task from C++ or python language and answer questions related to the program. The eye gazes of the participants were tracked while the participants assess the source code. The results showed no statistical difference between C++ and Python code with respect to accuracy and time, however significant difference was reported on the rate at which students look at buggy lines of code.

A study was conducted to investigate the impact of gender on subjects' visual effort, time and accuracy to recall Camel Case and Underscore identifiers in understanding the code and thus, ultimately, on program comprehension (Sharafi et al. 2012). Female subjects seem to spend more time to figure out wrong answers by carefully looking at all options whereas, male subjects seem to quickly set their minds on some answers. Though female subjects spent more effort to analyze wrong answers, they had a higher percentage of correct answers when compared to male subjects. Several software quality metrics have been proposed and are validated empirically proving that readability is one of the important factors affecting maintainability.

Understanding a program plays an important role for the learners. A study has been conducted using eye-tracker which shows how the identifier style in a program impacts the code reading and comprehension (Sharif and Maletic 2010a). As the identifiers represent major part of the program text, they play a key role in program understanding. Camel-case and underscore are the two main identifier styles focused in this study. A variety of data collection methods are used which includes online questionnaires, verbal, and eye-tracking methods focusing on low-level readability issues and mainly on semantic implications of identifier style. The results from these studies indicate that camel-case is the best choice specially for beginning programmers.

An experiment was conducted to discover if there is any development in the way programmers visually attend the representations provided by a program visualization tool: Jeliot 3 during program comprehension using a remote eye-tracker (Bednarik and Tukiainen 2006). The execution of Java programs is automatically visualized by Jeliot 3,

using an object-oriented approach. Participants were instructed to comprehend the Java program and use Jeliot as they found it necessary. Some of the experienced programmers did not use the tool but instead studied the code only. Ratio of fixation counts, attention switching, and fixation durations between the main representations of a program were analyzed. Experienced programmers exhibited different behavior, they spent more time focusing on visualization than on the code. The results indicate that eye-movement based analysis can contribute to our understanding of cognitive process involved in program comprehension, debugging, and visualization.

### **2.2.3 Debugging**

A study was conducted to show whether method name is likely to be meaningful and appropriate to the implementation of the method (Høst and Østvold 2009). A natural language processing technique called part-of-speech tagging is performed on the method names which decomposes the name into individual fragments to identify their grammatical structure. On the other hand, signature and Java bytecode of the method implementation are analysed, deriving a semantic profile for each implementation. Name-specific implementation rules are used to identify naming bugs and suggest a replacement phrase to use for the method. The results from applying the extracted implementation rules on the corpus are presented in this study. These rules can also be applied to any Java application or a library.

Software developers need to coordinate various information sources to perform maintenance tasks effectively. The role of visual attention in the coordination of representations during maintenance tasks was presented in a study using eye-tracking to

capture the visual attention strategies during debugging (Bednarik 2012). Two distinct experience level programmers were instructed to debug a program with the help of multiple representations. The time spent at looking each representation, the frequency of switching attention between visual representations and the type of switch were investigated during consecutive phases of debugging. The results determined that repetitive patterns in visual strategies were associated with less expertise. Both the code and graphical representations were used by the novice developers while switching between them. Experienced developers spent more effort integrating the available information and systematically related the code to the output.

### **2.3 Eye tracking Studies in other domains**

Eye trackers have also been used in other disciplines such as physics. This section briefly reports on some representative studies in other domains. A study has been conducted as a part of physics education research to deal with the difference between novices and expert in physics problem solving (Rosengrant 2010). Problem solving is not always finding a numerical answer, whereas it also includes qualitative solutions. In this study the problem-solving gap is mainly focused to see how both the groups understand and comprehend the new material and solve problems. The scan paths of the subjects are monitored while solving problems, instead of simply relying on verbal responses and written work from experts and novices. This study is focused on one type of problem solving scenario i.e., solving problems with electrical circuits to find a quantitative solution. Eleven subjects participated in the case study of which nine subjects were considered novices. Each subject is given a series of questions based on the circuits and

each of them received four subjects one at a time. The circuits were given in a Microsoft paint document, so that they can write their work next to the circuit since the monitor was a graphics display monitor. Calculator is also provided to subjects on the computer screen to assist them with any quantitative analysis if needed. The questions given to the subjects required auditory responses and numeric responses. Each subject wore a head mounted eye-tracker while answering the questions. The eye tracker used is an applied science laboratories model 6000 mobile control unit with scene camera. In addition to the camera from the eye-tracker a video camera is also used to record the entire interview.

Experts evaluate their work while solving problems whereas, novices do not. Both the experts and the novices looked back at the circuit to double check the value of the resistors in their mathematical formulas. However, the novices only focused on their mathematical work until they arrived at the solution by looking back and forth in their work, but not to the circuit. On the other hand, the experts gazed back and forth between the circuit, their work and circuits they may have redrawn to help them solve the problem. Another difference between experts and novices is how they initially looked at the circuit.

The subjects analyzed the circuit by simply going from one resistor to other resistor by following the shortest path between the resistors which was common among both the experts and the novices. Another interesting information among both experts and novices is that they would group resistors together when they first analyzed the circuit either in series or parallel. This method of gaze scribing provides a unique opportunity to



analyze problem solving behaviors. Using the scan path algorithm path difference between subjects who answer physics problems correctly and incorrectly is investigated.

A scan path analysis is performed using an algorithm called ScanMatch, which is used to compare DNA sequence. ScanMatch takes a saccade sequence and records this information to create a sequence of letters which represents the location, duration, and order of the fixations. The letter sequence of two sets of eye movements are then compared to each other to calculate a similarity score. A similar score representing two sets of eye movements are compared in this study (Madsen et al. 2012). The average ScanMatch scores are produced by comparing the correct solvers to one another (C-C comparison), the incorrect solvers to one another (I-I comparison), and the correct solvers to the incorrect solvers (C-I comparison). There were 24 participants, with two different levels of experience in physics out of which 10 participants were PhD students and 13 participants were introductory psychology students.

Although PhD students were expected to answer correctly when compared with the psychology students, this may not always be the same as a wide distribution of expertise among introductory physics students and physics graduate students was seen (Mason and Singh 2011). The study consisted of ten multiple-choice conceptual physics problems covering various topics in introductory physics. The participants' eye movements were recorded and each participant was asked to provide a verbal retrospective report for which they were shown replay of their eye movements to explain their thought processes. This method has been found to provide more depth of explanation than a retrospective report. Differences in reading the problem statement and

answer choices may have overcome small changes in diagram viewing, resulting in no difference in the ScanMatch scores of the C-C and I-I comparisons compared to C-I comparisons. The results of this study suggest that wrong answers have roots in incorrect ways student think about how the world works, not a how a problem diagram looks. These findings implicate the educational interventions aimed at helping novices learn to answer such conceptual questions correctly.

The distribution of expertise among introductory physics students was assessed by asking the students to categorize the mechanics problems based on the similarity of solution (Mason and Singh 2011). The categorization of physics graduate students and faculty members was compared to evaluate the effect of problem context on students. A large overlap was observed between the categorizations. The results showed that it is not appropriate to classify all graduate physics students as experts and all introductory physics students as novices.

The use of eye tracking in gaming changes the experience of players with or without the use of manual input devices and are widely used to assist people with decreased limb mobility by preventing manual operation of computers (Isokoski et al. 2009). An eye-mouse mode of operation is offered by most eye trackers. The mouse cursor will be placed at the point where a player is looking by using the tracker software commands. The mouse button press is also sent to the operating system whenever the player's gaze remains within a small area for specific amount of time by most of the eye trackers. These mouse emulation techniques are generally sufficient to play some games. An additional software is also used outside both the eye tracker and the game itself. The

source code of the game can also be modified to allow eye control if it is available. This method is applied very rarely since most of the commercial games source code is not available and this method has more labor intensive than the previous ones. Building a new game from the scratch also allows the game design to maximize the potential of eye tracker input. This method is most labor intensive method and differs from other game genres and can be achieved only by creating new games (Isokoski et al. 2009). When a person is using eye tracker for the first time, the experience may feel as if things happen merely by thinking about them and tends to evoke positive reaction. Using the gaze attention of the player's, the behavior of the players can be estimated. Distinct gaze patterns of players with varying skill and gender results in new considerations for future game design.

An eye tracking study was conducted to supplement usability tests in both commercial and academic practice (Ehmke and Wilson 2007). An empirical study was conducted to find a range of possible correlations between usability problems and eye-tracking patterns. The users were asked to test two websites to extract the usability problems from the data and correlate with eye-tracking patterns. Usability companies are offering many eye-tracking services like website evaluation reports with session images and heat maps (Ehmke and Wilson 2007). Two different websites were used in this study with one specific task each. These website-task combinations included different kinds of interactions like scanning text, reading, text input, usage of navigation elements, scanning of pictures, searching etc. A wide range of usability problems were expected from these tasks. The study was performed using a Tobii X50 eye-tracker which is a free standing,

non-invasive device. The eye tracker return the tracking status every 20ms. Verbal protocols and observational data of the participants was collected to identify usability problems. Two distinct approaches were used for correlating with eye-tracking patterns: page problem, where the usability problem and the eye-tracking pattern are connected to the same element on a page and site problem, where the usability problem and the eye-tracking pattern are connected to the same element that is present on different pages of the site. With the help of eye tracking patterns the user behaviour was explored to specific usability problems. A series of different metrics were observed from the sequences of eye-tracking patterns which includes high number of fixations across the page and navigation, followed by fixations on one element. (Ehmke and Wilson 2007).

An eye tracking study was conducted investigating the use of eye movements in evaluating the usability of World Wide Web-Page designs (Cowen, Ball, and Delin 2002). The performance measure of four different websites was compared based on different eye movement metrics. Participants were asked to complete two tasks on each of the four website homepages to find information about either using a mobile phone abroad or buy a new mobile phone handset. Total Fixation Duration, Number of Fixations, Average Fixation Duration, and Fixation Spatial Density were used in analysing the data. The time-based eye-tracking metrics which are Total Fixation Duration and Average Fixation Duration showed the same significant difference as shown in the performance measures (Cowen, Ball, and Delin 2002).

A study was conducted to support synchronous collaboration in remote settings with a shared workspace and reduce the effort required to communicate about specific

locations in code. Two programmers can work together on a single project using pair programming. With the use of novel gaze visualization, remote pair programmers can identify where their partner is looking currently in the code. The color of the code changes when they are looking at the same thing (D'Angelo and Begel 2017). Software developers from technology company were asked to participate in this survey. Each of the two participants had their own display, mouse, keyboard and an attached remote eye tracker. Tobii TX300, Tobii EyeX was used at monitor 1 and monitor 2 respectively. A visual studio 2015 extension was created which helps to share the screen and support gaze visualization. Two code editor windows displayed the project code in the extension whose text and user interface are accessible to one another.

Participants were asked to work in two refactoring tasks to clean up the C# code for a game like Flappy Bird. The participants were asked to fill out a 13-question Likert scale questionnaire about their feelings in performing the task. Whenever a participant made a reference or verbally acknowledged a reference, the type was recorded, the time was logged, and it was given a description. The total amount of time spent looking at same thing was calculated for each pair of participants, which was divided by the total time spent on each task. The programmers were faster and more successful at communicating using implicit, deictic references (such as this, that) as the programmers were looking at the same thing at the same time with the help of visualization. The design served as a helpful aid for coordinating remote work and could easily be integrated into many other forms of collaborative document editing.

A computational cognitive modelling of the perceptual, strategic, and oculomotor processes that people use in a search task was presented in a study (Halverson and Hornof 2007). Visual search is an important part of human-computer interaction. A minimal visual search model was proposed that helps in predicting and understanding user behavior in HCI. The research proposed three characteristics of a minimal model of visual search: eye movements tend to go to nearby objects, fixated objects are not always identified, and eye movements start after the fixated objects are identified. It was proposed that any applied model of visual search should include at least these three characteristics. The cognitive models used in this study were built using the EPIC (Executive Process-Interactive Control) which captures human perceptual, cognitive, and motor processing constraints in a computational framework. A variety of eye movement data observed in the visual search behavior were explained when compared to the previous models of the same task without using eye movement data.

The next chapter will cover the details of the eye tracking study.

## CHAPTER 3

### The Eye Tracking Study

This chapter presents the details of the empirical study conducted as part of this thesis. It gives details on the experiment design, hypotheses, participants, data collection, tasks, and how the study was instrumented.

#### 3.1 Experiment Design

To understand the impact of good programming practices on software developers, we decided to conduct this experiment on two code readability rules taken from (Boswell and Foucher 2011). We focused on the programming practices aimed at making the control flow in the source code easy to read and the avoidance of the do/while loop. Specifically, we considered two coding practices that we present as rules:

- R1. Minimize-nesting rule
- R2. Avoid do/while loop rule

An overview of the experiment is given in Table 1. The experiment seeks to analyze methods with these two coding rules for the purpose of evaluating their impact on the readability of the source code. The participants are instructed to read and understand the code snippet shown to them and answer few questions related to the code snippet that will give a measure of their actual level of understanding. They were also asked to answer their level of difficulty and confidence level in their own degree of understanding the code snippets.

**Table 1. Experiment overview**

<b>Goal</b>	Impact of two programming practices on developers
<b>Independent variables</b>	Correctness, readability
<b>Dependent variables</b>	Time, confidence level, Level of understanding

Since we are going to manipulate two Boolean variables (correctness, readability), our study will have four treatments namely, T1, T2, T3, and T4. The table below summarizes the correctness and readability values of the four treatments designed for testing one readability rule. Correctness refers to whether a Java method satisfies the problem specification (Boolean variable) and readability refers to whether a Java method follows the readability rule (Boolean variable). We have designed 4 problem statements for each rule and each problem will have 4 different solutions (All solutions are expressed as Java methods). Thus, the  $j$ th solution (or version) for the  $P_k$  problem is denoted as  $P_kV_j$ , where  $k \in 1..4$ , and  $j \in 1..4$ . With respect to the factors, such solutions or versions will have the following characteristics:

- $P_kV_1$  is a correct solution to the  $P_k$  problem that follows the readability rule we are interested in.
- $P_kV_2$  is another correct solution to the  $P_k$  problem that does not follow the readability rule.
- $P_kV_3$  is an incorrect solution to the problem  $P_k$  that follows the readability rule.
- $P_kV_4$  is an incorrect solution to the problem  $P_k$  that does not follow the readability rule.



**Table 2. Treatment combination used in the study for each problem**

Treatment Combination		Readability	
		Follows the rule	Does not follow the rule
<b>Correctness</b>	correct	<p style="text-align: center;"><b>T1</b></p> <p>This is a correct solution to a problem that follows the readability rule (The <b>version1</b> method)</p>	<p style="text-align: center;"><b>T2</b></p> <p>This is a correct solution to a problem that does not follow the readability rule (The <b>version2</b> method)</p>
	incorrect	<p style="text-align: center;"><b>T3</b></p> <p>This is a buggy solution to a problem that follows the readability rule (The <b>version3</b> method)</p>	<p style="text-align: center;"><b>T4</b></p> <p>This is a buggy solution to a problem that does not follow the readability rule (The <b>version4</b> method)</p>

Therefore, we used 16 methods to test one single readability rule. The methods do not call any other methods, or use objects of other classes. Thus, the participants do not require to study additional code to understand the method or scroll down as the method fits on a single screen. To avoid carryover effect, each participant is given four methods in a randomly selected order, and each one of such solutions corresponds to a different problem.

### 3.2 Hypotheses

Based on the research questions presented above four detailed null hypotheses based on each of the dependent variables are given below.

H1<sub>0</sub>: There is no difference in the *time* a developer spends in understanding the source code with or without the minimize-nesting rule.

H2<sub>0</sub>: There is no difference in *accuracy* a developer obtains after understanding the source code with or without the minimize-nesting rule.

H3<sub>0</sub>: There is no difference in the *level of confidence* a developer has after reading the source code with or without the minimize-nesting rule.

H4<sub>0</sub>: There is no difference in the *ease of readability* a developer faces when reading source code with or without the minimize-nesting rule.

H5<sub>0</sub>: There is no difference in the *visual effort* a developer faces when reading source code with or without the minimize-nesting rule.

H6<sub>0</sub>: There is no difference in the *time* a developer spends in understanding the source code with or without the avoid do/while rule.

H7<sub>0</sub>: There is no difference in *accuracy* when a developer obtains after understanding the source code with or without the avoid do/while rule.

H8<sub>0</sub>: There is no difference in the *level of confidence* a developer reach after reading the source code with or without the avoid do/while rule.

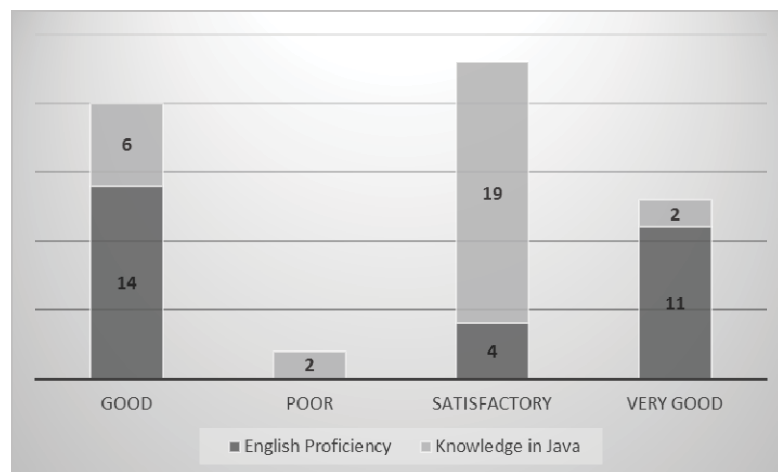
H9<sub>0</sub>: There is no difference in the *ease of readability* a developer face when reading source code with or without avoid the do/while rule.

H10<sub>0</sub>: There is no difference in the *visual effort* a developer does when reading source code with or without avoid the do/while rule.

Alternative Hypothesis: Our alternative hypothesis is that the source code that follows the mentioned rules is more readable, that is, the readers may require less time to understand it, their level of understanding is higher, their confidence level in their own level of understanding is higher, and the visual effort required is lower.

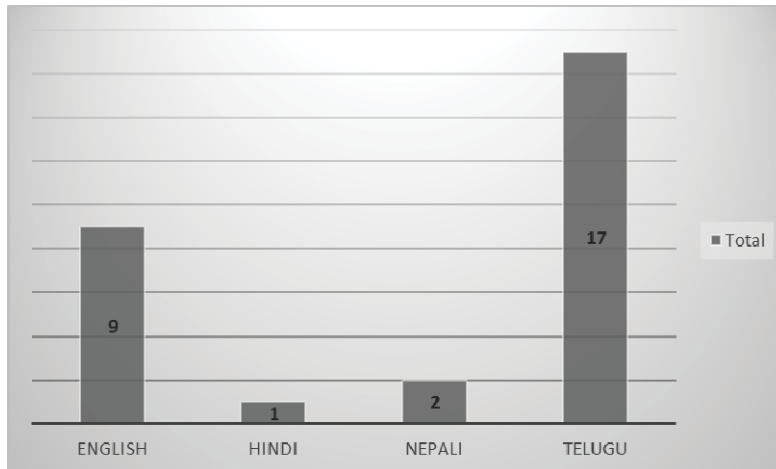
### 3.3 Participants

We recruited graduate and undergraduate students from three classes at YSU. The first class was principles of programming language; the second class was the server-side class on web development and the third class was software engineering. There were 29 students from three of these classes, out of which 20 were graduate students and 9 were undergraduate students. All the methods are created using Java programming language and we asked our participants to self-assess their programming skills in the pre-questionnaire. We asked our participants to self-assess their skills in the pre-questionnaire. Figure 1 shows demographics of participants.



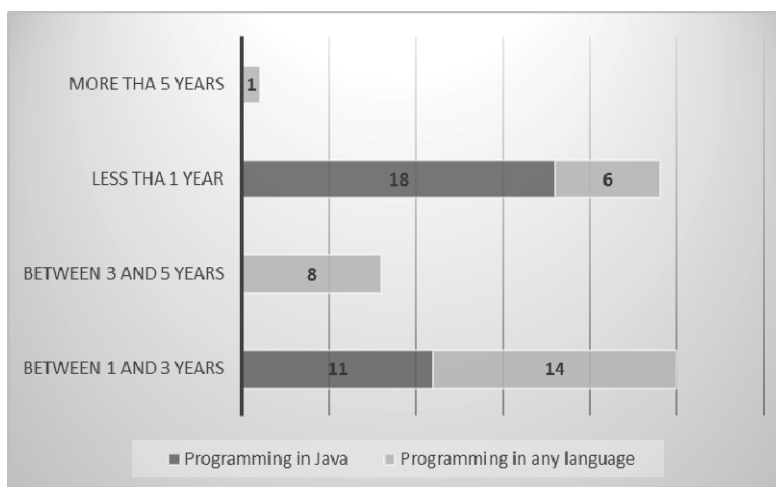
**Figure 1. Proficiency in English and Knowledge in Java**

We can see that most of the participants have very good proficiency in English. Nineteen participants have satisfactory skill whereas, a very few have poor skill in Java language. Figure 2 shows the mother tongue of the participants.



**Figure 2. Mother Tongue Distribution**

Figure 3 shows the number of years' participants have been working in any programming language and specifically in Java. Most of the participants have been actively programming between 1 and 3 years and only one participant have been programming from more than 5 years. Two of the participants are currently working as a developer in software companies.



**Figure 3. Number of years actively programming in Java and other languages**

### 3.4 Tasks

We designed four methods for both the rules of the study with four different solutions for each method. The experiment is divided into two parts: Part A is a *method analysis* task which includes four problems from Rule 1 and four problems from Rule 2, Part B is a *method comparison* task which includes two versions we designed for problem  $P_k$ . Assuming that, there will be 24 subjects (denoted by  $S_{b_j}$ ) and all subjects will be exposed to all treatments, each participant will be given the four solutions in a randomly selected order, and each one of such solutions will correspond to a different problem.

Table 3 shows all the possible trials used in the study. The grey shaded areas correspond to part B of the experiment. Note that  $P_kV_j$  for R1 is different from  $P_kV_j$  for R2. Subject 1 analyze solution V1 for problem P1, the solution V2 for problem P2, the solution V3 for problem P3, and solution V4 for problem P4 with a similar arrangement of tasks for another readability rule as shown below.

**Table 3. Possible trials used in the study. The columns in grey indicate comparison tasks for ranking two code snippets**

	Rule 1				Rule 2							
Sb1	P1V1	P2V2	P3V3	P4V4	P1V1	P1V2	P1V3	P2V4	P3V1	P4V2	P1V1	P1V2
Sb2	P1V1	P2V2	P3V4	P4V3	P2V1	P2V2	P1V3	P2V4	P3V2	P4V1	P2V1	P2V2
Sb3	P1V1	P2V3	P3V2	P4V4	P3V1	P3V2	P1V4	P2V1	P3V2	P4V3	P3V1	P3V2
Sb4	P1V1	P2V3	P3V4	P4V2	P4V1	P4V2	P1V4	P2V1	P3V3	P4V2	P4V1	P4V2
Sb5	P1V1	P2V4	P3V2	P4V3	P1V1	P1V2	P1V3	P2V4	P3V1	P4V2	P1V1	P1V2
Sb6	P1V1	P2V4	P3V3	P4V2	P2V1	P2V2	P1V3	P2V4	P3V2	P4V1	P2V1	P2V2
Sb7	P1V2	P2V1	P3V3	P4V4	P3V1	P3V2	P1V4	P2V1	P3V2	P4V3	P3V1	P3V2
Sb8	P1V2	P2V1	P3V4	P4V3	P4V1	P4V2	P1V4	P2V1	P3V3	P4V2	P4V1	P4V2
Sb9	P1V2	P2V3	P3V1	P4V4	P1V1	P1V2	P1V3	P2V1	P3V2	P4V4	P1V1	P1V2
Sb10	P1V2	P2V3	P3V4	P4V1	P2V1	P2V2	P1V3	P2V1	P3V4	P4V2	P2V1	P2V2
Sb11	P1V2	P2V4	P3V1	P4V3	P3V1	P3V2	P1V3	P2V2	P3V1	P4V4	P3V1	P3V2
Sb12	P1V2	P2V4	P3V3	P4V1	P4V1	P4V2	P1V3	P2V2	P3V4	P4V1	P4V1	P4V2
Sb13	P1V3	P2V1	P3V2	P4V4	P1V1	P1V2	P1V2	P2V3	P3V1	P4V4	P1V1	P1V2
Sb14	P1V3	P2V1	P3V4	P4V2	P2V1	P2V2	P1V2	P2V3	P3V4	P4V1	P2V1	P2V2
Sb15	P1V3	P2V2	P3V1	P4V4	P3V1	P3V2	P1V2	P2V4	P3V1	P4V3	P3V1	P3V2
Sb16	P1V3	P2V2	P3V4	P4V1	P4V1	P4V2	P1V2	P2V4	P3V3	P4V1	P4V1	P4V2
Sb17	P1V3	P2V4	P3V1	P4V2	P1V1	P1V2	P1V1	P2V4	P3V2	P4V3	P1V1	P1V2
Sb18	P1V3	P2V4	P3V2	P4V1	P2V1	P2V2	P1V1	P2V4	P3V3	P4V2	P2V1	P2V2
Sb19	P1V4	P2V1	P3V2	P4V3	P3V1	P3V2	P1V2	P2V1	P3V3	P4V4	P3V1	P3V2
Sb20	P1V4	P2V1	P3V3	P4V2	P4V1	P4V2	P1V2	P2V1	P3V4	P4V3	P4V1	P4V2
Sb21	P1V4	P2V2	P3V1	P4V3	P1V1	P1V2	P1V1	P2V2	P3V3	P4V4	P1V1	P1V2
Sb22	P1V4	P2V2	P3V3	P4V1	P2V1	P2V2	P1V1	P2V2	P3V4	P4V3	P2V1	P2V2
Sb23	P1V4	P2V3	P3V1	P4V2	P3V1	P3V2	P1V1	P2V3	P3V2	P4V4	P3V1	P3V2
Sb24	P1V4	P2V3	P3V2	P4V1	P4V1	P4V2	P1V1	P2V3	P3V4	P4V2	P4V1	P4V2

For Rule 1, the Java methods involved are FindTheBiggest, FindGrade, BodyMassIndex, and CountPosNumbers. For Rule 2, the methods involved are WhoIsTheAuthor, ChangePassword, SumPositiveNum, and CountLetter. The problem statements for R1 and R2 rules are shown in Table 4 and Table 5 respectively. The complete set of study questions including all background questions and post questionnaires can be found in the Appendix.

**Table 4. Overview of problem statements used in R1 (Minimize-nesting rule)**

<b>Problem</b>	<b>Problem statement</b>
1	Finding the largest of three numbers. Given three integer numbers, the method must return the greatest.
2	Determining the grade using the marks obtained. The given mark is an integer between 0 and 100. The method must return a letter. Letter A if mark $\geq 90$ ; B if mark $\in [80; 90)$ ; C if mark $\in [70; 80)$ ; D if mark $\in [60; 70)$ ; and F if mark $< 60$ .
3	Determining the Body Mass Index Category. The method receives the body mass index (bmi), and must return the category in which the index is located. The category is "very severely underweight" if $bmi \leq 15$ ;" severely underweight" if bmi in $[15; 16)$ ;" underweight" if $bmi \in [16; 18.5)$ ;" healthy weight" if $bmi \in [18.5; 25)$ ;" overweight" if $bmi \geq 25$ .
4	Counting the positive numbers. Given three integers, the method must count how many of them are positive numbers.

**Table 5. Overview of problem statements used in R2 (Avoid do/while rule)**

<b>Problem</b>	<b>Problem statement</b>
1	Prompting the user to answer a multiple-choice question. The method must show the question, get the user response, and end when the user chooses the correct answer or when she decides not to try more (typing 'q' or 'e').
2	Let's assume that we want to force a user to change her password. The user must give a new password that has 4 different characters. Besides, the given password must be different to the old one. The method receives the old password as a parameter and asks the user to give the new one.
3	Summing the positive numbers in an array. The method receives an integer array as a parameter and must return the sum of the positive numbers in the array.
4	Counting the occurrences of a character. This method receives a string and a character as parameters. It must count and return the number of occurrences of the character in the string.

### **3.5 Data collection**

The participants were asked to answer questions related to the method after analyzing each task through an online questionnaire. In the method analysis task, the participants were instructed to read the problem statement, analyze one of the four solution methods proposed for a problem, rate the readability of the method, answer a multiple-choice comprehension question, rate the level of confidence about their own level of comprehension, and answer about the correctness of the method. On the other hand, in the method comparison task the participants were instructed to read the problem statement, and rank the order of readability of the methods from best to worst. There was



no time constraint in reading the method. Each question was timed and the subjects' eyes were tracked using eye tracker. We used Tobii Studio to collect data for this study.

In addition to the online questionnaires, we collected eye tracking data and audio/video recordings of subjects at Youngstown State University because we have access to an eye tracker at this location. We obtained IRB approval (153-2017) and training before we began this study.

### **3.6 Eye-Tracking Apparatus**

The Tobii X60 eye tracker was used in this study in the Software Engineering and Empirical Studies Lab at the Computer Science Department at YSU. This eye tracker generates 60 samples of eye data per second and the user does not require to wear any head gear. Tobii X60 is a 60Hz video-based binocular remote eye tracker, whose average accuracy is 0.5 degrees which averages to about 15 pixels. A dual monitor extended desktop setting was used in the study. The monitor used was a 24 inch screen with a resolution set at 1920\*1080. The first monitor was used by the experimenter to setup and initiate the study. The eye gaze data and audio/video recordings of the entire study are recorded by the eye tracker on the second monitor. The eye gaze data includes timestamps, gaze positions, fixations count, fixation duration, pupil size, and validity codes.

### **3.7 Conducting the Study**

The test was conducted in the Software Engineering Research and Empirical Studies Lab (SERESL). The participants are provided with the consent forms as well as

an instruction sheet. Once they read the instructions, they will decide whether to participate in the study or not. They can withdraw themselves if they have any concerns. The participants were then asked to fill out the pre-questionnaire that include their background details, programming experience, knowledge of Java, and the reading skills in English. The test can be attempted by one student at a time as the lab can have one student in at a time. Before starting the study, calibrations will be done to make sure that the subjects eyes are in sync with the eye tracker.

The participants are asked to maintain a position in the chair during the study so that we do not lose the tracking of their eyes. Subjects will see a red circle with a black dot in the middle on the screen when the calibration is started. Once we get good calibration results, researcher can start the session. A good calibration appears with the green vector in the circle and not too far away from the circle. If the researcher does not find a good calibration, the re-calibration must be done. Subjects are encouraged to ask questions, so that they understand the instructions. Correct answers of the test are not given to the subjects.

The researcher is always present with the subject during the study to make sure that the eyes are always tracked which is determined by the track status in the Tobii studio keeping it on the left screen. The subjects are encouraged to think out loud while reading the methods and look at the screen all the time except when they need to type an answer keeping the head movement limited. The subjects can see the code snippet with no time limit and click the LEFT mouse button to move to the next screen. We do not

pause or end the session once the recording starts, unless they want to withdraw themselves due to emergency and start a new session if they wish to continue.

## CHAPTER 4

### Results and Analyses

This chapter presents the results from our controlled experiment. We discarded data from two problems analyzed by two participants since they clicked the left button of mouse too soon and were not able to analyze that method. We have removed participants' data from the avoid do/while loop rule where we noticed that the session was not recorded properly.

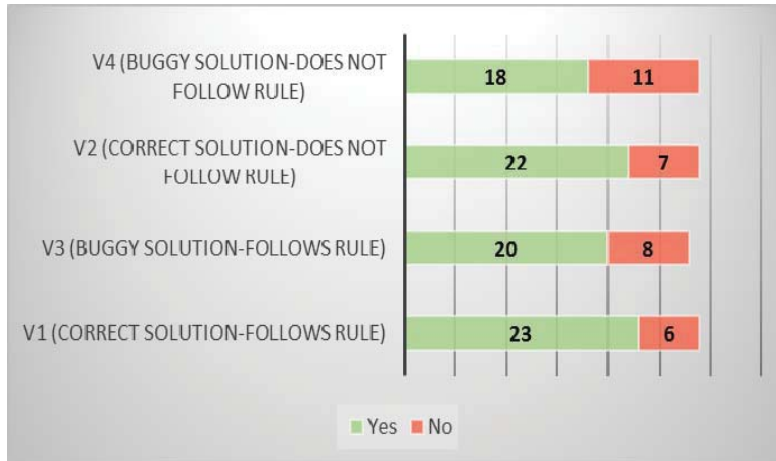
Since each participant is given a different version of the method, we used the Mann-Whitney test ( $\alpha = 0.05$ ) for all our dependent variables to find the significant differences between the methods that follow the rule and ones that do not follow the rule.

#### 4.1 Accuracy

This section presents the results of the participants based on accuracy. The participants were asked to expect whether the Java method satisfies the problem specification and expect the answer for a multiple-choice question: if the method worked properly based on certain input. For example, the method that finds the count of positive numbers is presented with the following question: The method does not work properly when the input numbers are 1, 0, and 2.

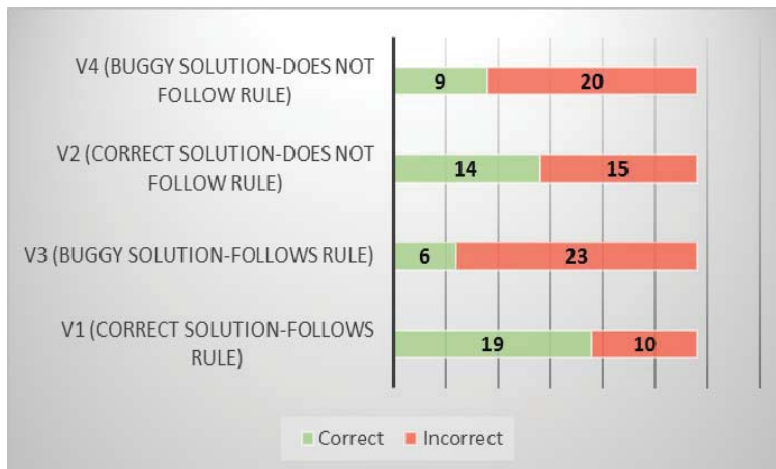
##### 4.1.1 Rule 1

With respect to accuracy in rule 1, when the participants are asked to answer whether the Java method satisfies the problem specification, Mann-Whitney test showed that there was no significant difference for accuracy ( $p=0.385$ ). Figure 4 shows the accuracy based on logical correctness of all participants grouped by problems.



**Figure 4. Accuracy based on logical correctness for Rule 1 (minimize-nesting)**

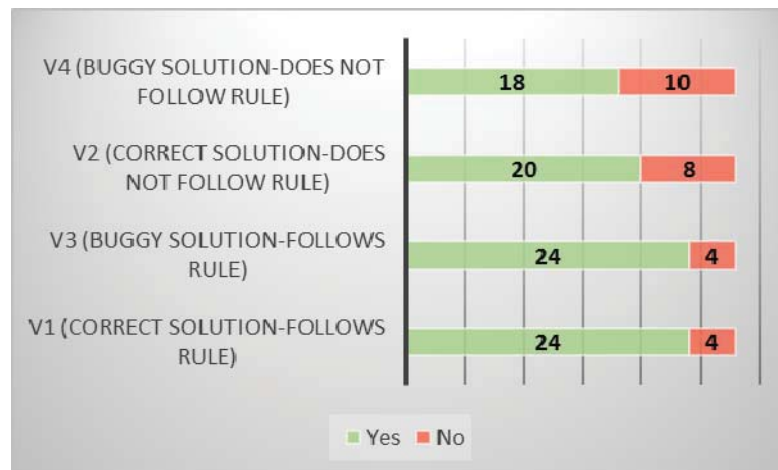
When the participants were asked to answer if the method worked properly based on certain input, we did not find any significant difference ( $p=0.376$ ) when the method followed the minimize-nesting rule. However, the participants answered more accurately when the method is correct and followed the minimize nesting rule. Figure 5 shows the accuracy based on certain input of all participants grouped by problems.



**Figure 5. Accuracy based on certain input for Rule 1 (minimize-nesting)**

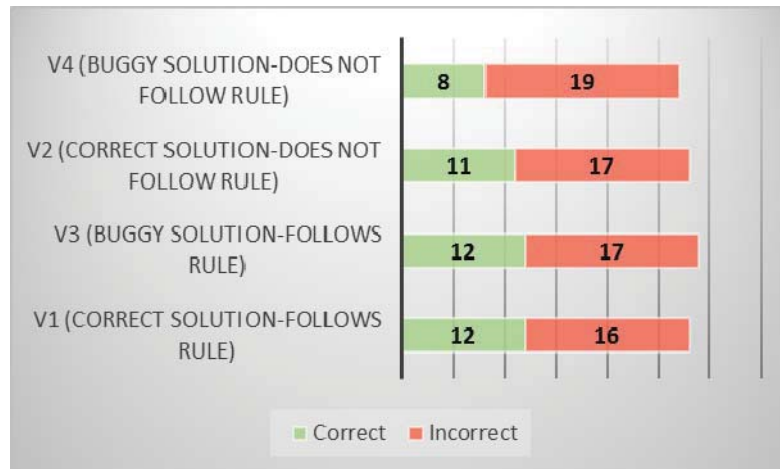
#### 4.1.2 Rule 2

In terms of accuracy with respect to logical correctness of the method, the Mann-Whitney test showed no significant difference ( $p=0.683$ ) between the methods that follow the rule and the one that does not follow the rule. We found that participants tend to answer more accurately when the method is correct and follows avoid do/while loop rule. Figure 6 shows the accuracy based on logical correctness grouped by all the methods and respective participants from avoid do/while rule.



**Figure 6. Accuracy based on logical correctness for Rule 2 (avoid do/while)**

We did not find difference in accuracy based on certain input ( $p=0.975$ ) when the method followed avoid do/while loop rule. However, the participants answered more accurately when the method is correct and follows avoid do/while rule. Figure 7 shows the accuracy based on certain input grouped by all the methods and respective participants from avoid do/while rule.



**Figure 7. Accuracy based on certain input for Rule 2 (avoid do/while)**

This shows that for both the readability rules, accuracy was higher for the correct methods that followed the rule.

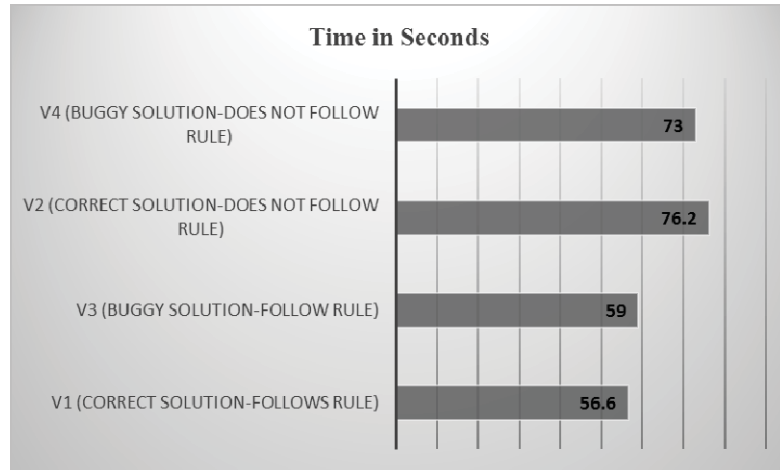
## 4.2 Time

The time taken to analyse a method by the participants is presented in this section.

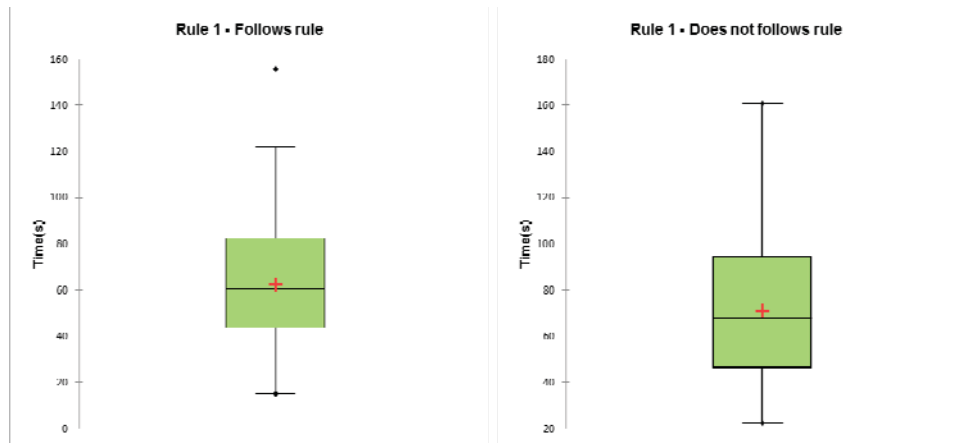
The timestamp of all the participants is recorded using Tobii Studio.

### 4.2.1 Rule 1

With respect to the time taken to analyse the method, we did not find any significant difference ( $p=0.127$ ) when the methods followed the rule. Figure 8 shows the time taken by all the participants to analyze problems from rule 1 grouped by all methods followed by the box plots for the methods that follows the rule and does not follows the rule.



a)



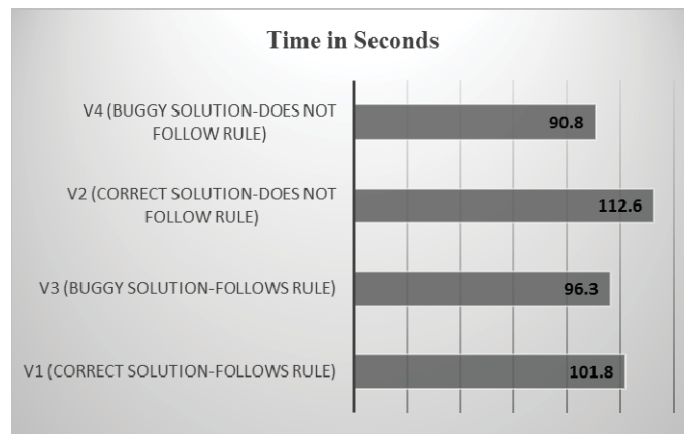
b)

**Figure 8. Time taken to analyze Rule 1 (minimize-nesting) problems**

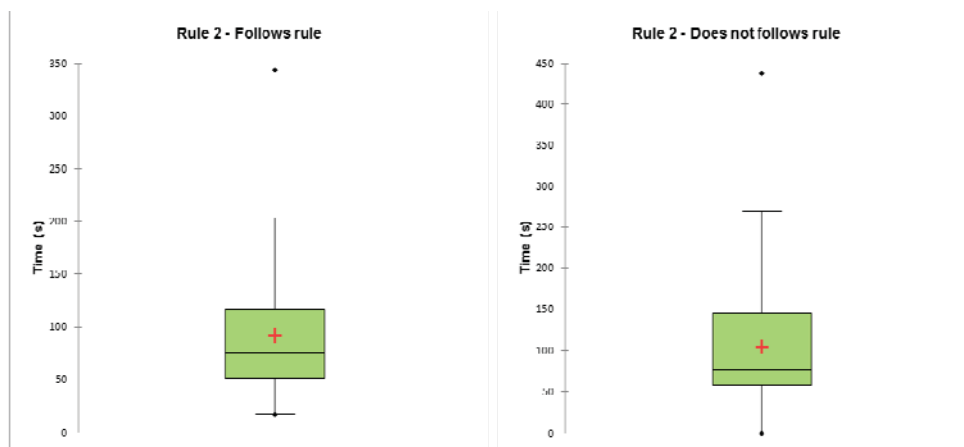


### 4.2.2 Rule 2

For the avoid do/while loop rule we did not find significant difference when the methods followed the rule ( $p=0.479$ ). Since the participant felt that some of the methods in avoid do/while loop rule are difficult, we think that the participants took more time to analyze them. Figure 9 shows the time taken by all the participants to analyze problems from rule 2 grouped by all methods followed by the box plots for the methods that follow the rule and the one that does not follow the rule.



a)



b)

**Figure 9. Time taken to analyze Rule 2 (avoid do/while) problems**

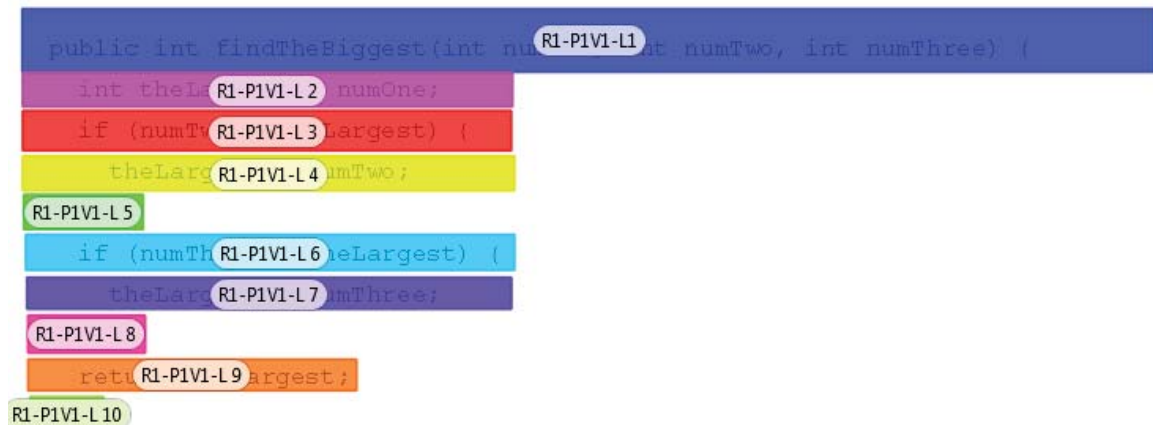
### 4.3 Visual Effort

The data from Tobii studio is exported to calculate the visual effort of the participants taken to analyze the methods. The visual effort of the participant is known by calculating fixation count and fixation duration of each participant.

#### 4.3.1 Creating Areas of Interest

Before we calculate fixation count and fixation duration, we need to create area of interest on every method used in the study. The Figure 10 shows the AOIs for find the biggest number method used in the study. The AOIs comprise of each line of source code. The number of AOIs are equivalent to the number of lines in the method. We do not count any eye gazes that fall outside these lines on blank space.

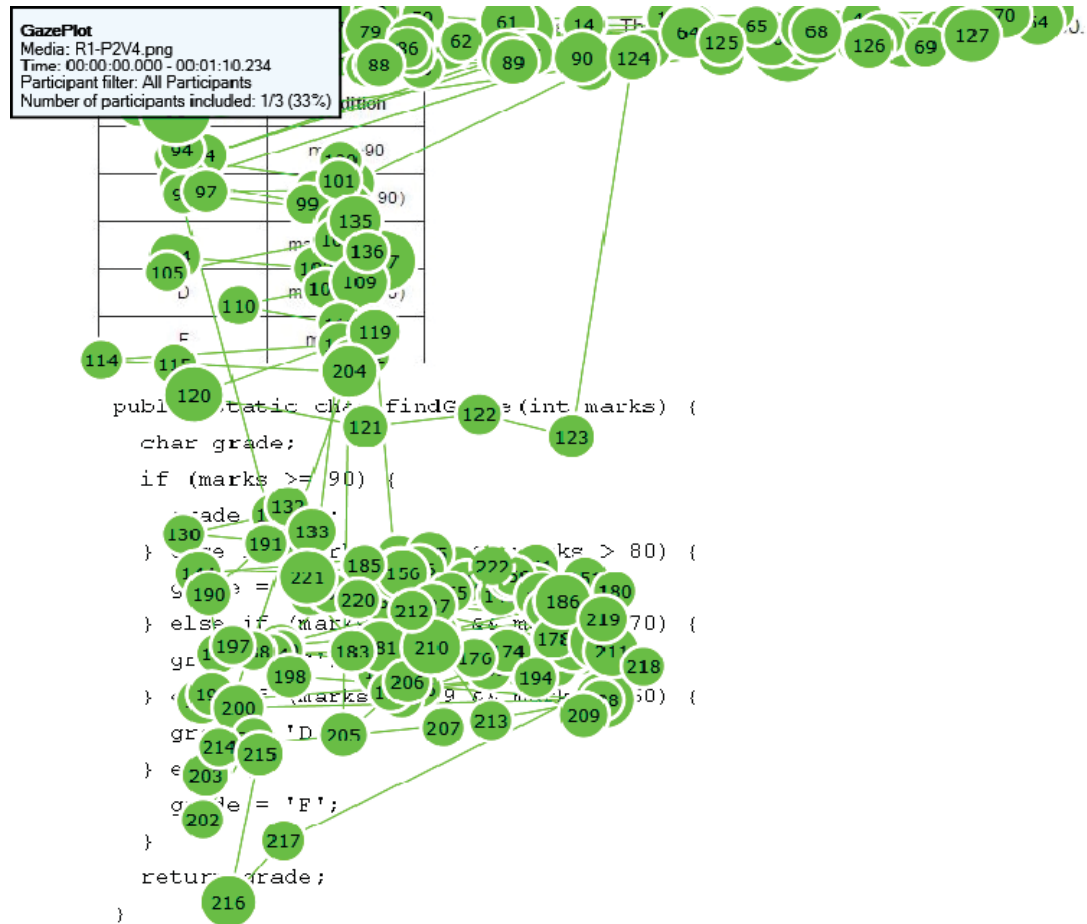
```
public int findTheBiggest(int numOne, int numTwo, int numThree) {  
    int theLargest = numOne;  
    if (numTwo > theLargest) {  
        theLargest = numTwo;  
    }  
    if (numThree > theLargest) {  
        theLargest = numThree;  
    }  
    return theLargest;  
}
```



**Figure 10. Areas of Interest**

The Figure 11 shows the gaze plot of one participants. The gaze plot visualization represents the sequence and position of the fixations on the method where the participant

has seen. Each circle indicates a fixation that the participant has made. The size of the dot indicates the fixation duration and the number in the dots represent the order of the fixations.



**Figure 11. Gaze plot**

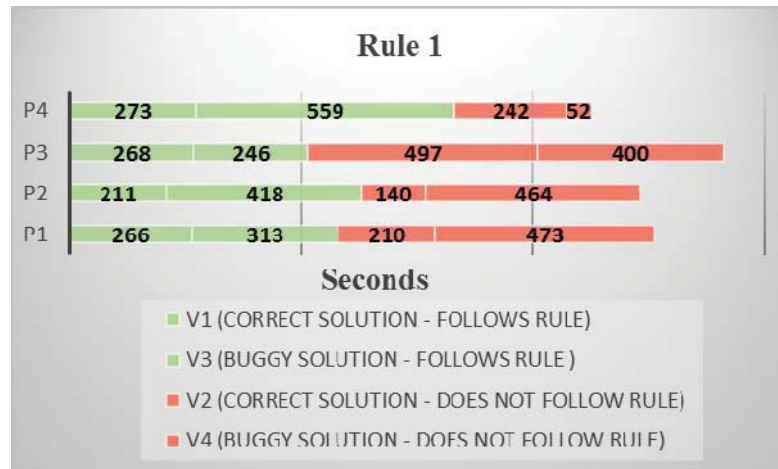
### 4.3.2 Fixation Counts

The fixation count denotes the number of fixations spent in total within the AOI for each method. For example, if a method has 10 lines, we would total the fixation counts for each of those 10 lines. This resulting number would give us the fixation count for that method. In this study, we focused on calculating the fixation count of overall method,

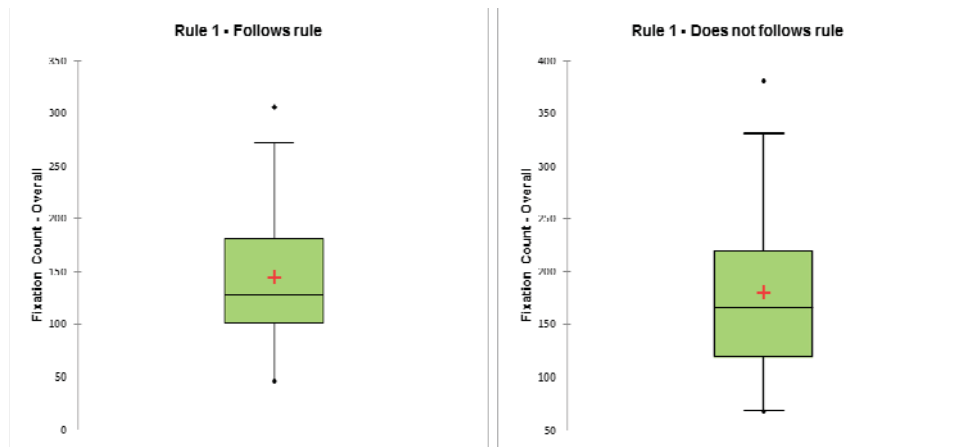
problem statement and the conditional statements used in the method. For example, in the find the biggest number we considered the “if -else” statements of the method.

#### **4.3.2.1 Rule 1**

We observed significant difference in fixation count for the correct solution that followed minimize-nesting rule ( $p=0.007$ ). This shows that people looked the method less number of times when the method was correct and it followed the rule. Figure 12 shows the fixation counts of all the participants grouped by all methods followed by box plots for the methods that follow the rule and the one that does not follow the rule.



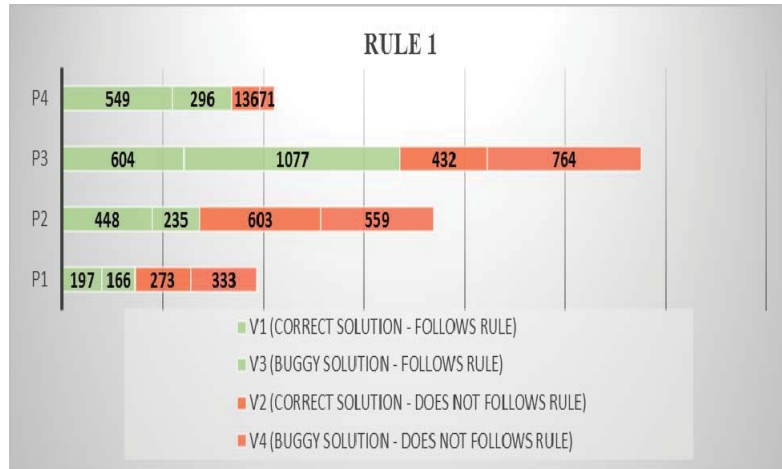
a)



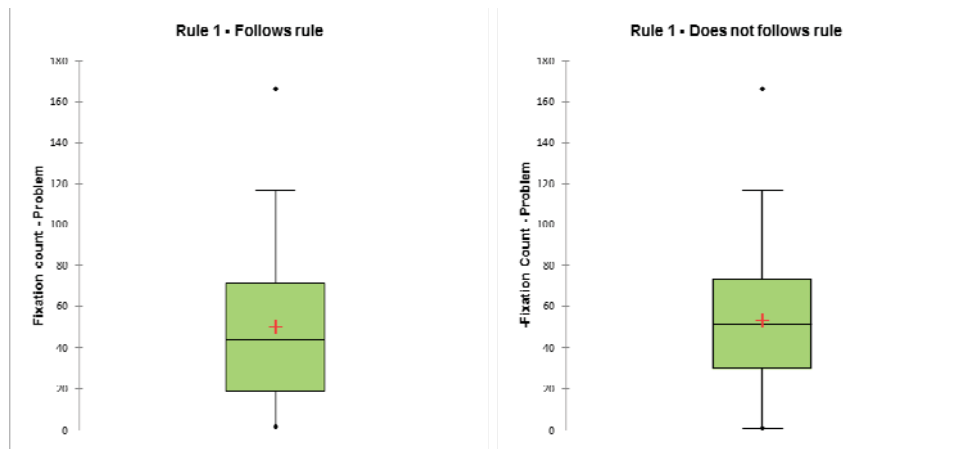
b)

**Figure 12. Overall fixation count for Rule 1 (minimize-nesting)**

We did not find significant difference in fixation count of problem statement that followed minimize-nesting rule ( $p=0.534$ ). However, for problem 1 and problem 2, we observed that the fixation count was high for the problem statement when the method followed the rule. This shows that the participants looked the problem statement less number of times when the method followed minimize-nesting rule.



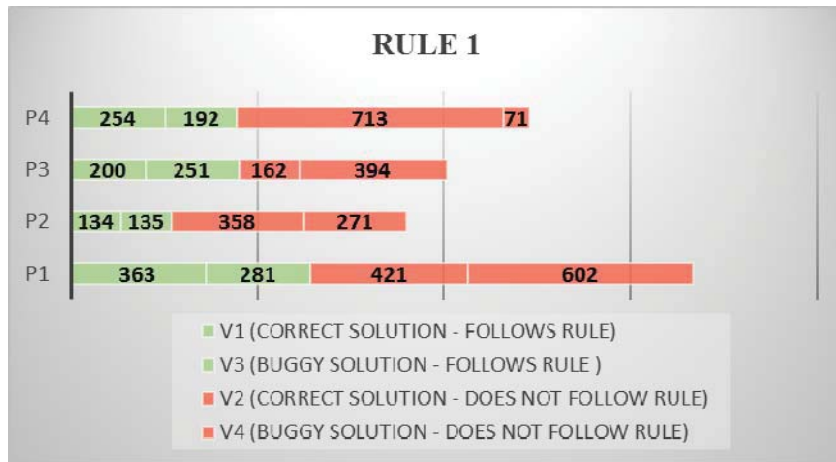
a)



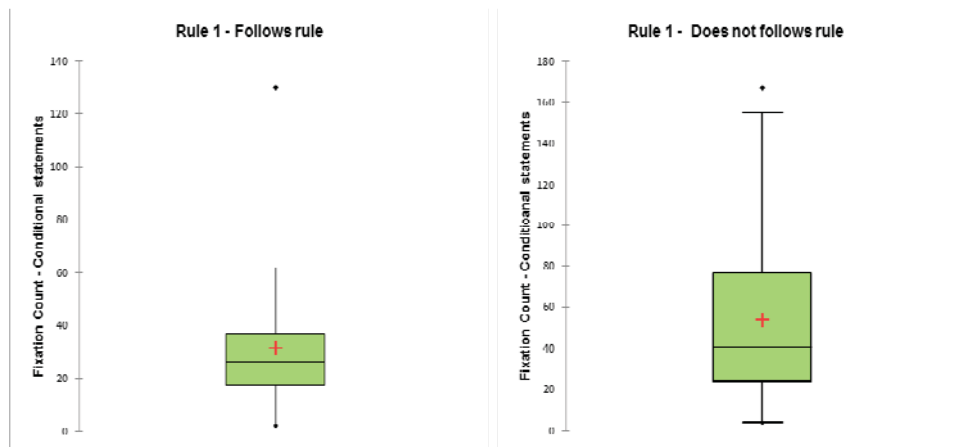
b)

**Figure 13. Fixation count for problem statements for Rule 1 (minimize-nesting)**

For conditional statements, we found significant difference in fixation count when the method followed the rule ( $p=0.003$ ). This means that the participants looked at the conditional lines more number of times when the method did not follow the rule. Figure 14 shows the fixation count of conditional statements of all the participants grouped by all methods followed by box plots for the methods that follow the rule and the one that does not follow the rule.



a)



b)

**Figure 14. Fixation count for conditional statements for Rule 1 (minimize-nesting)**

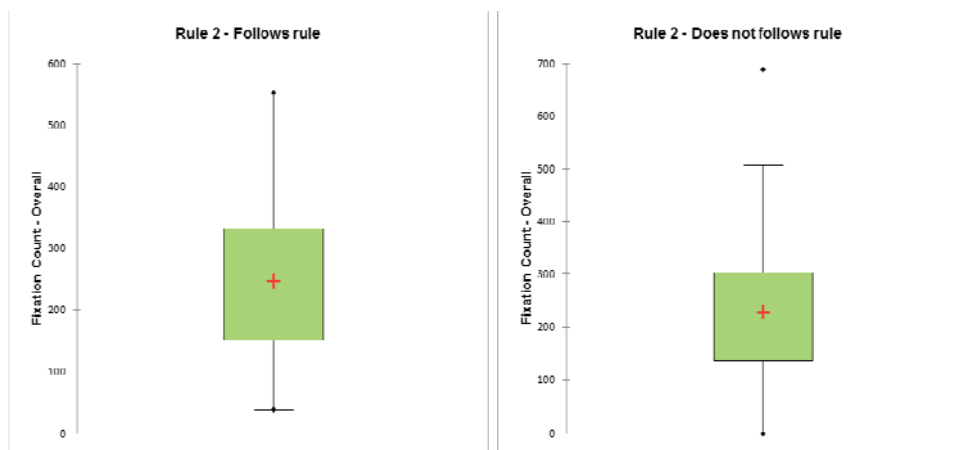
#### 4.3.2.2 Rule 2

For the avoid do/while rule, we did not find significant difference in fixation count when the method followed the rule ( $p=0.317$ ). Figure 15 shows the fixation counts

of all the participants grouped by all methods followed by box plots for the methods that follow the rule and the one that does not follow the rule.



a)



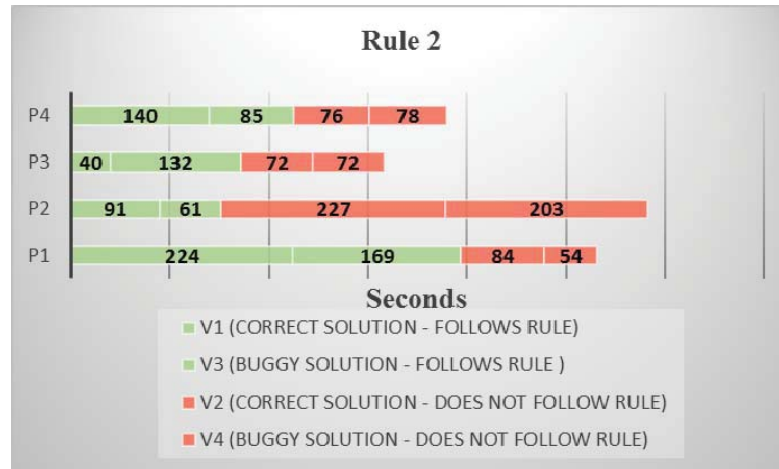
b)

**Figure 15. Overall fixation count for Rule 2 (avoid do/while)**

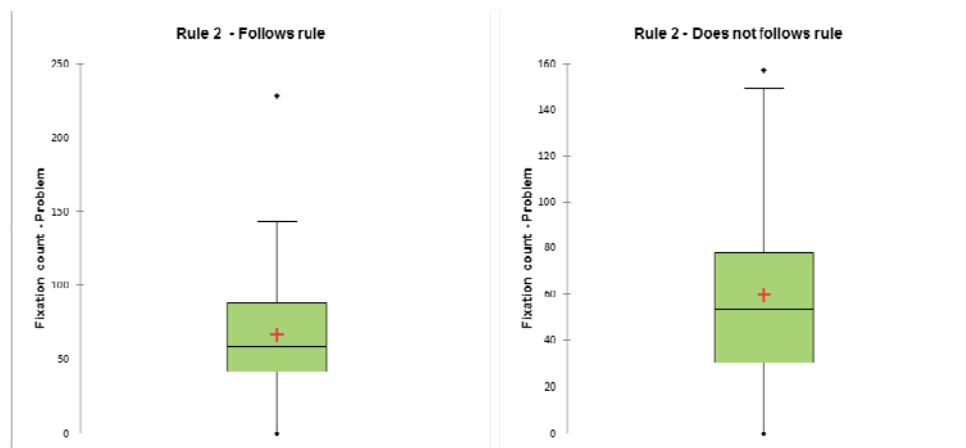
We did not find significant difference in fixation count for the problem statement ( $p=0.401$ ). Figure 16 shows the fixation counts of problem statements of all the



participants grouped by all methods followed by box plots for the methods that follow the rule and the one that does not follow the rule.



a)

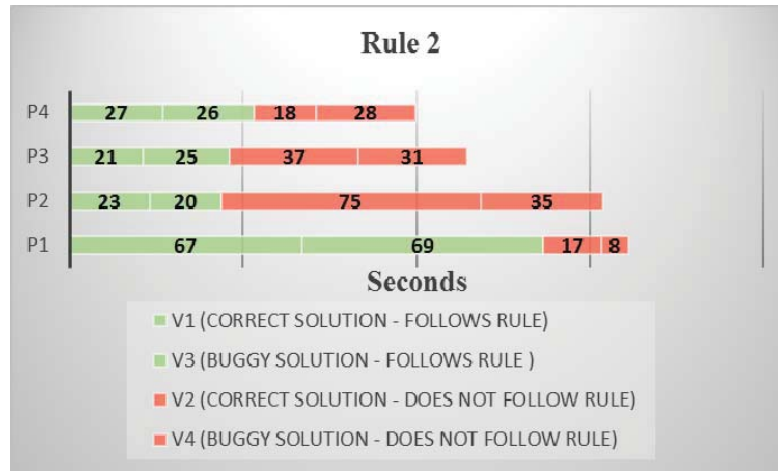


b)

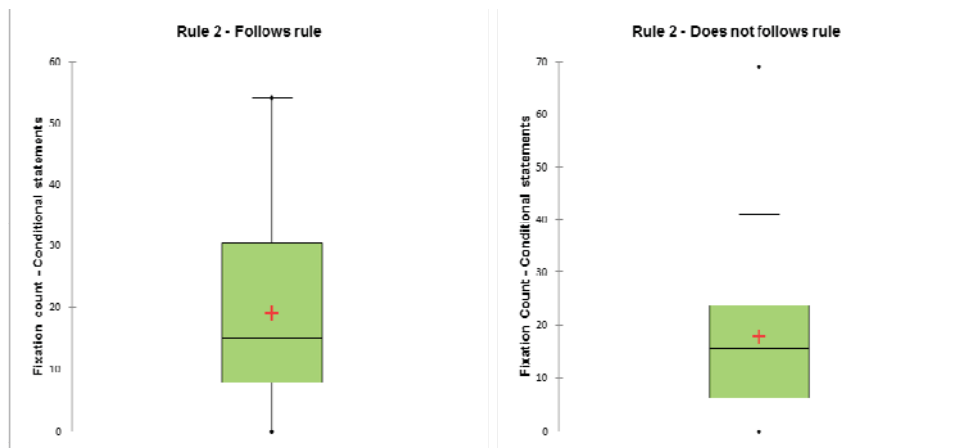
**Figure 16. Fixation count for Rule 2 (avoid do/while) problem statement**

We did not find a significant difference in fixation count when the method followed the avoid do/while loop rule ( $p=0.675$ ). However for problem 2 and problem 3,

the fixation count was less for the method that followed the rule which means that participants spent less time to analyze these conditional statements.



a)



b)

**Figure 17. Fixation count for Rule 2 (avoid do/while) conditional statements**

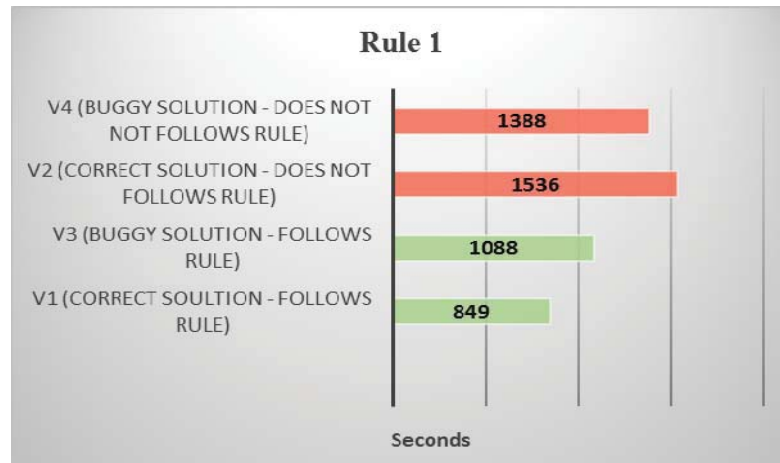
### 4.3.3 Fixation Durations

The fixation duration denotes the duration of all the fixations within the AOIs for each method. For example, if a method has 10 lines, we would total the fixation durations

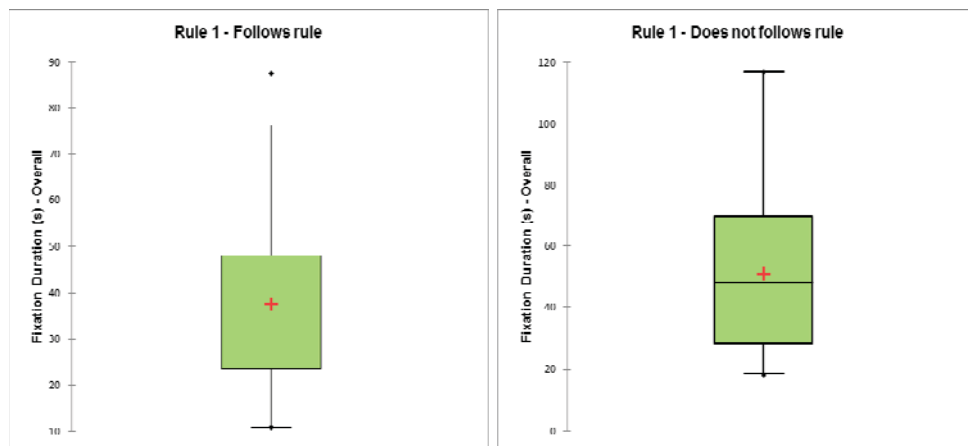
for all the fixation counts for each of those 10 lines. This resulting number would give us the fixation duration for that method. In this study, we considered to calculate the fixation count of overall method, the problem statement and the conditional statements used in the method. For example, in the “who is the author” method we considered the “do/while” statements of the method.

#### **4.3.3.1 Rule 1**

We observed that the fixation duration significantly less for the methods that followed minimize-nesting rule when compared to the methods that does not follow the rule ( $p=0.005$ ). This shows that the participants took less time to analyze the method that followed the rule. Figure 18 shows the fixation duration of all the participants grouped by all methods.



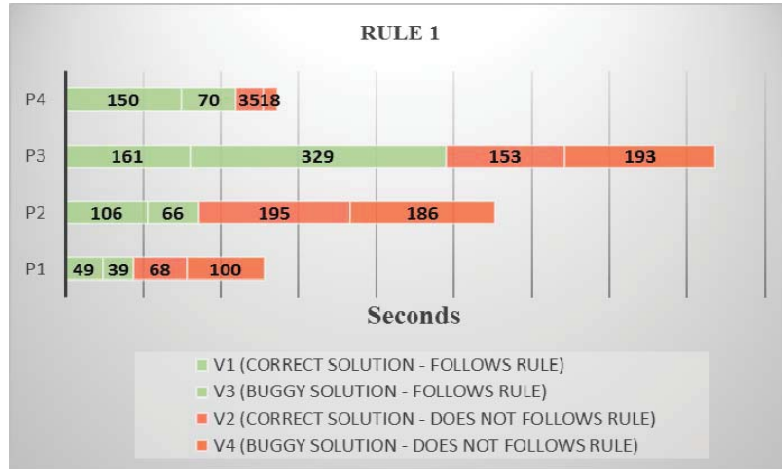
a)



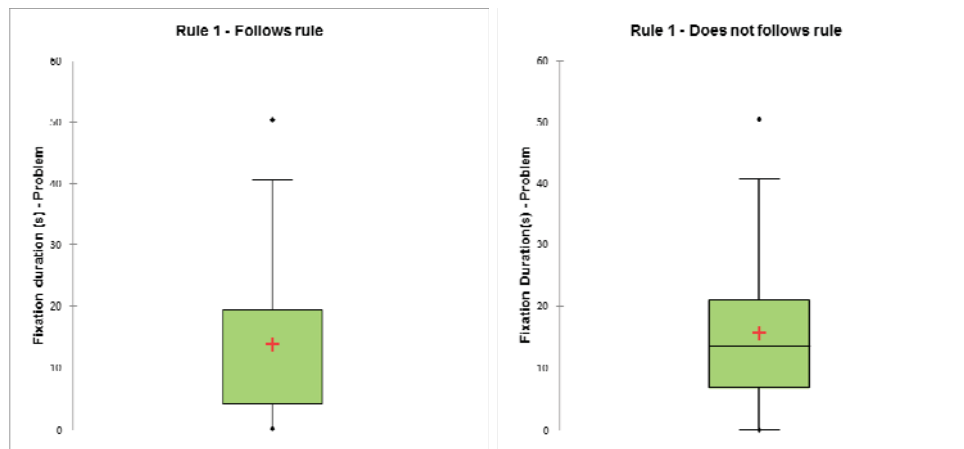
b)

**Figure 18. Overall fixation duration for Rule 1 (minimize-nesting)**

We did not find significant difference in fixation duration of problem statement for the method that followed minimize-nesting rule ( $p=0.363$ ). However for problem 1 and problem 2, we observed that the fixation duration was less for the problem statement when the method followed the rule. This means that participants took less time to analyze these problem statements when the method followed the rule.



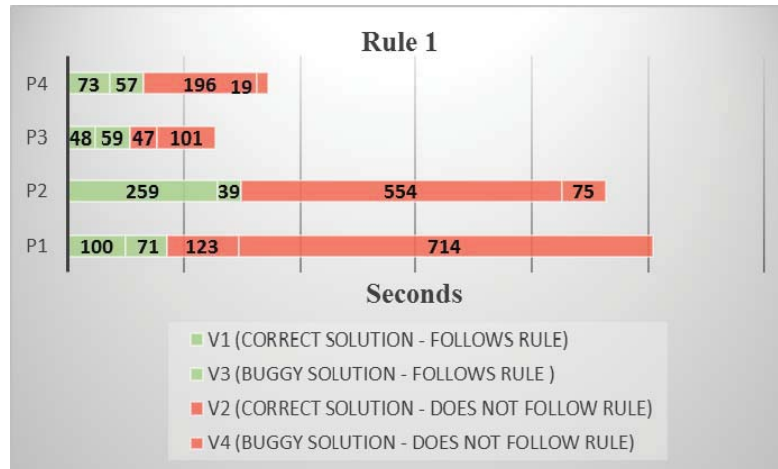
a)



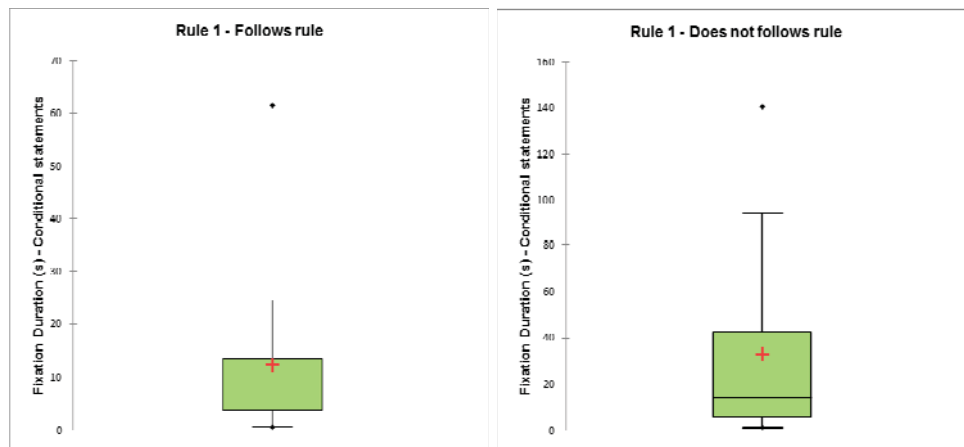
b)

**Figure 19. Fixation duration for Rule 1 (minimize-nesting) problem statement**

We found significant difference when the method followed the minimize-nesting rule ( $p=0.001$ ). This means that the participants took less time to analyze the conditional statements when the method followed the minimize-nesting rule. Figure 20 shows the fixation duration of all the participants grouped by all the methods followed by the box plots of the method that follow rule and the one that does not follow rule.



a)



b)

**Figure 20. Fixation duration for Rule 1 (minimize-nesting) conditional statements**

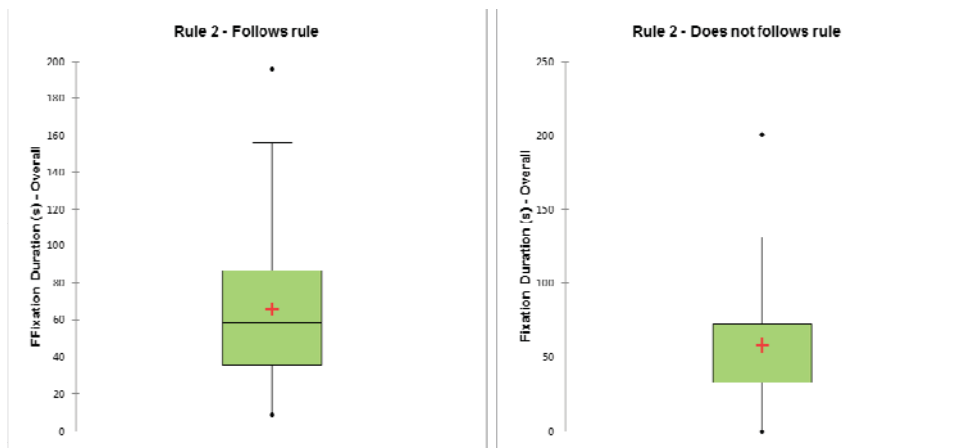
### 4.3.3.2 Rule 2

We did not find significant difference in fixation duration when the method followed avoid do/while rule ( $p=0.249$ ). However, we observed that the fixation duration was less for correct solutions that followed avoid do/while loop rule when compared to the correct method that did not follow the rule. Figure 21 shows the fixation duration of

all the participants grouped by all methods followed by the box plots of the method that follow the rule and the one that does not follow the rule.



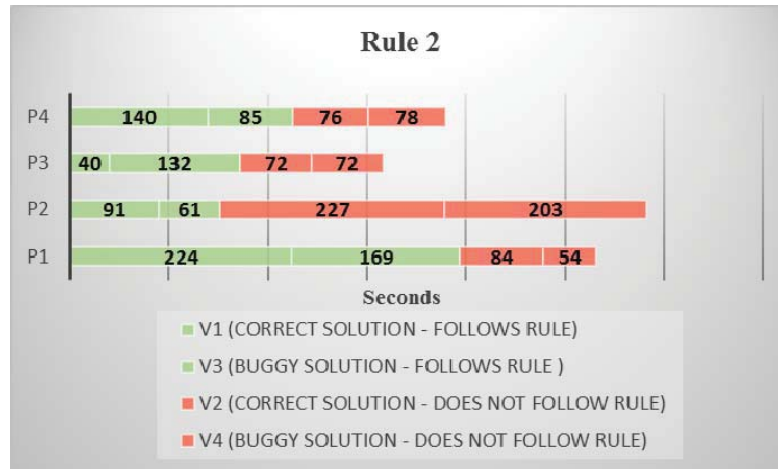
a)



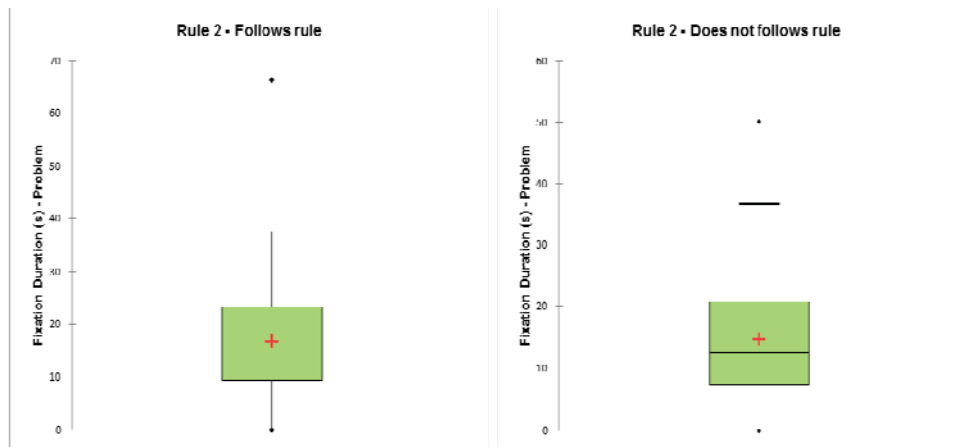
b)

**Figure 21. Overall fixation duration for Rule 2 (avoid do/while)**

We did not find significant difference in fixation duration for the problem statement ( $p=0.267$ ). However, we found that the fixation duration was less for the correct solution of problem 2 and problem 3 when the method follow avoid do/while rule which means that the participants took less time to to analyse these problem statements.



a)

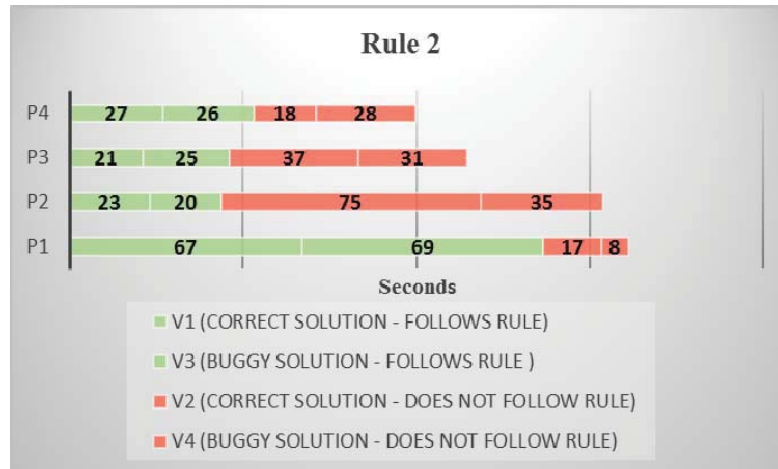


b)

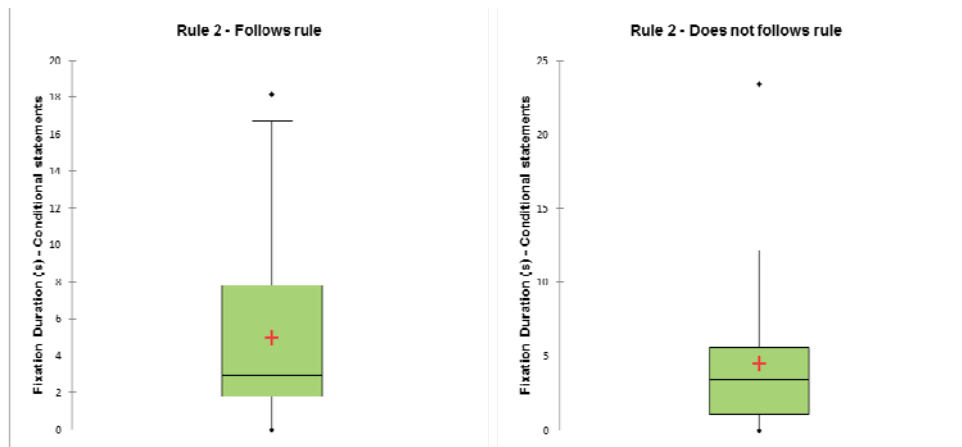
**Figure 22. Fixation duration for Rule 2 (avoid do/while) problem statement**

We did not find significant difference in fixation duration for the conditional statements ( $p=0.457$ ). However, the fixation duration was less for problem 3 and problem 2 when method followed the avoid do/while loop rule. This means that participants spent less time to analyze these conditional statements when the method followed avoid do/while rule.





a)



b)

Figure 23. Fixation duration for Rule 2 (avoid do/while) conditional statements

#### 4.4 Ease of readability

After analyzing each method, the participants were asked to answer their ease of readability about the method. The results of the participants are presented in this section.

#### 4.4.1 Rule 1

In terms of ease of readability for a method, we found a significant difference when the method followed minimize-nesting rule ( $p=0.002$ ). The participants only felt that the method is very difficult to read and difficult to read when the minimize-nesting rule is not followed by the method. Most of the participants felt that the methods are easy to read and very easy to read when the minimize-nesting rule is applied. Figure 24 shows the ease of readability of each method grouped by respective participants.



Figure 24. Ease of readability for Rule 1 (minimize-nesting rule)

#### 4.4.2 Rule 2

In terms of ease of readability when avoid do/while rule is used, we did not find significant difference ( $p=0.431$ ). However, for the method count letter, participants felt

difficult to read when the method did not follow avoid do/while loop rule and easy to read when the method followed the rule respectively. Figure 25 shows the ease of readability of each method grouped by respective participants.



Figure 25. Ease of readability for Rule 2 (avoid do/while)

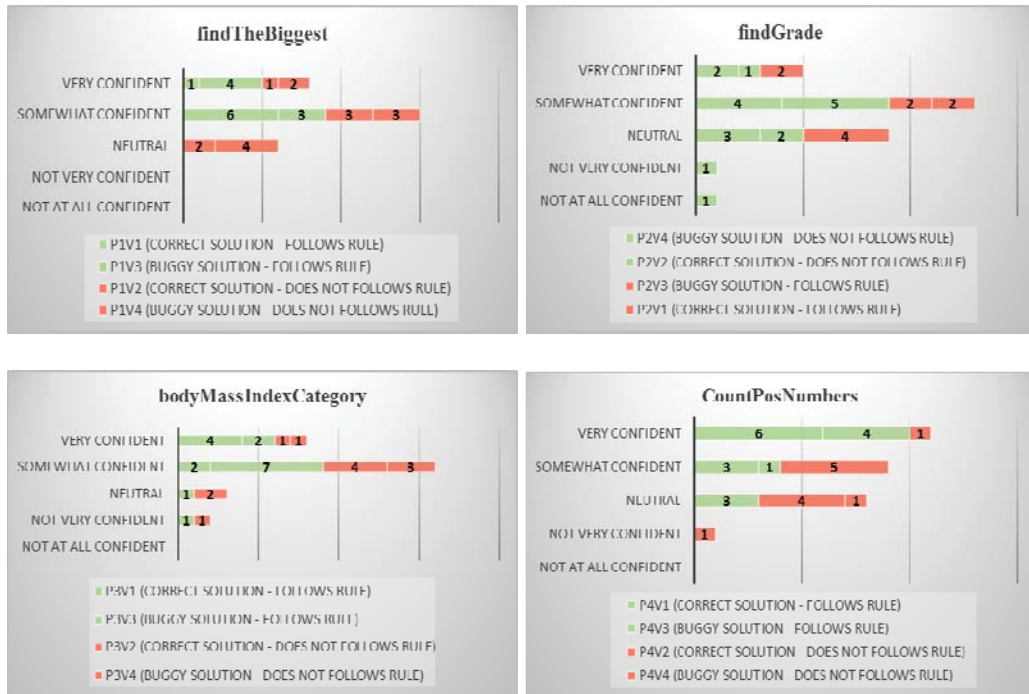
#### 4.5 Level of confidence

The participants were asked to rate their level of confidence after reading the source code. The results of the participants are presented in this section.

##### 4.5.1 Rule 1

In terms of level of confidence the participants reach after analyzing the method, we found a significant difference ( $p=0.002$ ) when the method followed minimize-nesting rule. Though few participants felt neutral and somewhat confident about the method,

most of the participants felt very confident when the minimize-nesting rule is used. Figure 26 shows the level of confidence of each method grouped by respective participants.



**Figure 26. Level of confidence for Rule 1 (minimize-nesting)**

#### 4.5.2 Rule 2

In terms of level of confidence the participants reach after analyzing the method, we did not find significant difference when the method follows avoid do/while loop rule ( $p=0.284$ ). However, for the method who is the author, the participants felt very confident and somewhat confident when the method follows avoid do/while loop rule and not very confident when the method did not follow the rule. Figure 27 shows the level of confidence of each method grouped by respective participants.



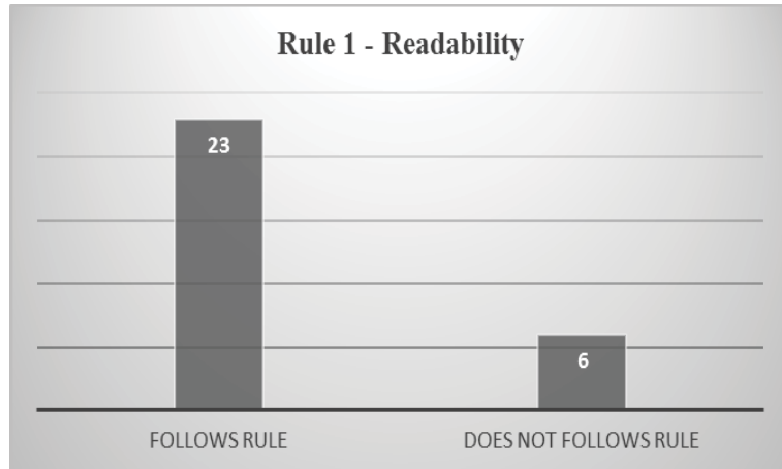
**Figure 27. Level of confidence for Rule 2 (avoid do/while)**

## 4.6 Method Comparison

The participants were asked to rank the order of readability for two correct solutions of the method that follows the rule in part B of the experiment. Results of participants are presented in this section.

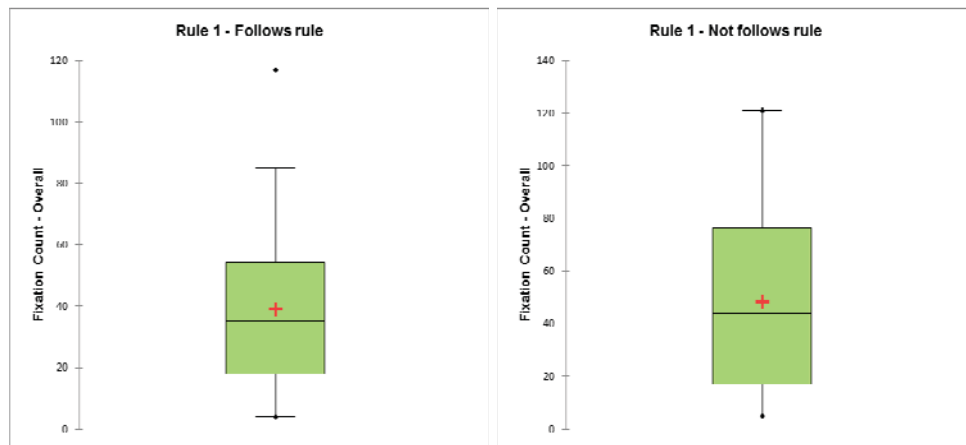
### 4.6.1 Rule 1

We observed a large variation when the method followed minimize-nesting rule. Most of the participants answered that method which follows minimize-nesting rule is more readable.



**Figure 28. Readability ranking for Rule 1 (minimize-nesting)**

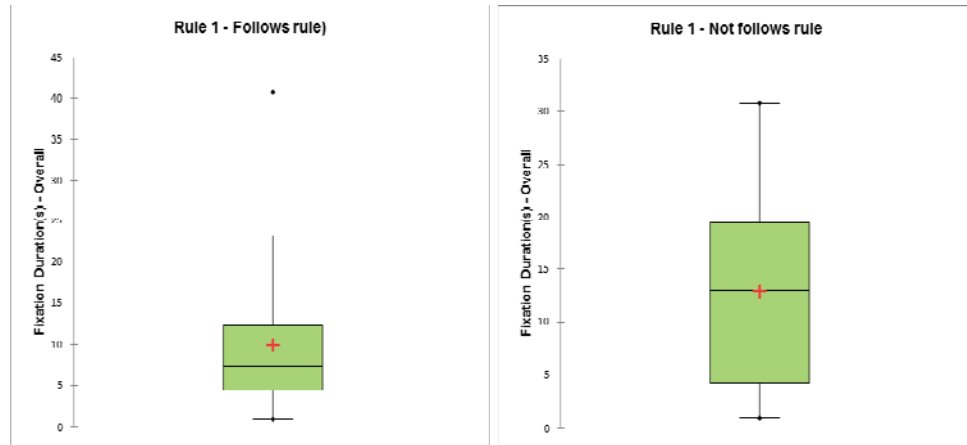
We did not find significant difference in overall fixation count ( $p=0.323$ ). Figure 29 shows the fixation count of the methods that follow minimize-nesting rule and the one that does not follow minimize-nesting rule.



**Figure 29. Overall Fixation Count for the two methods in Rule 1 (minimize-nesting) comparison task**

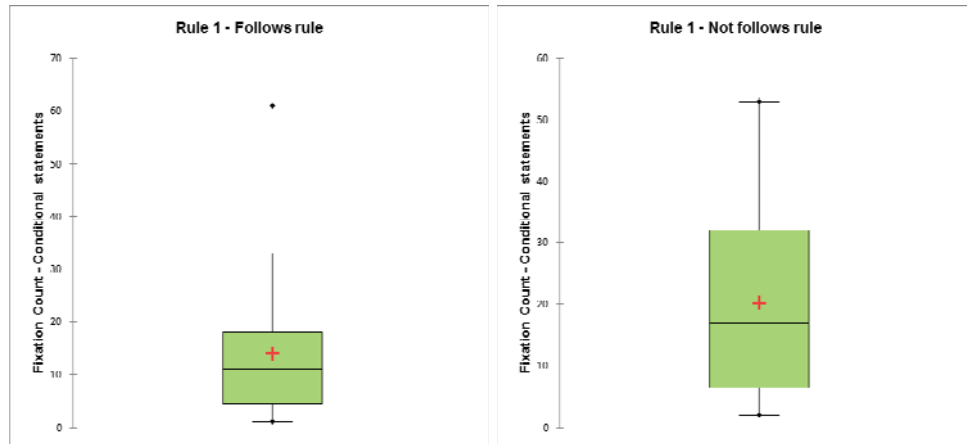
We did not find significant difference in overall fixation duration ( $p=0.291$ ).

Figure 30 shows the fixation duration of the methods that follow minimize-nesting rule and the one that does not follow minimize-nesting rule.



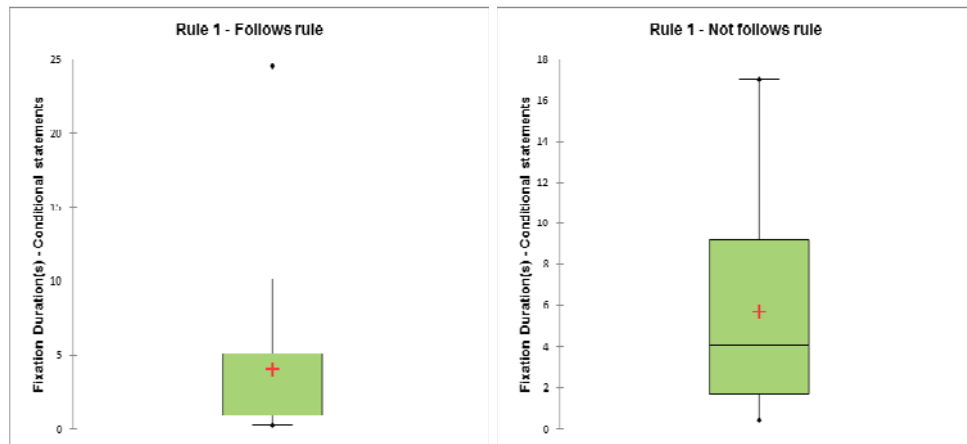
**Figure 30. Overall fixation duration for the two methods in Rule 1 (minimize-nesting) comparison task**

We did not find significant difference in fixation count of the conditional statements ( $p=0.125$ ). Figure 31 shows the conditional statements fixation count of the methods that follow minimize-nesting rule and the one that does not follow minimize-nesting rule.



**Figure 31. Conditional statements fixation count for the two methods in Rule 1 (minimize-nesting) comparison task**

We did not find significant difference in conditional statements fixation duration ( $p=0.457$ ). Figure 32 shows conditional statements fixation duration of the methods that follow minimize-nesting rule and the one that does not follow minimize-nesting rule.

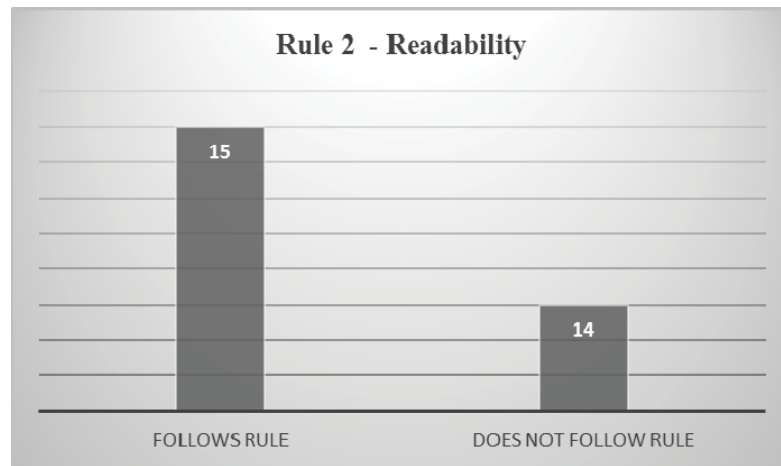


**Figure 32. Conditional statements fixation duration for the two methods in Rule 1 (minimize-nesting) comparison task**



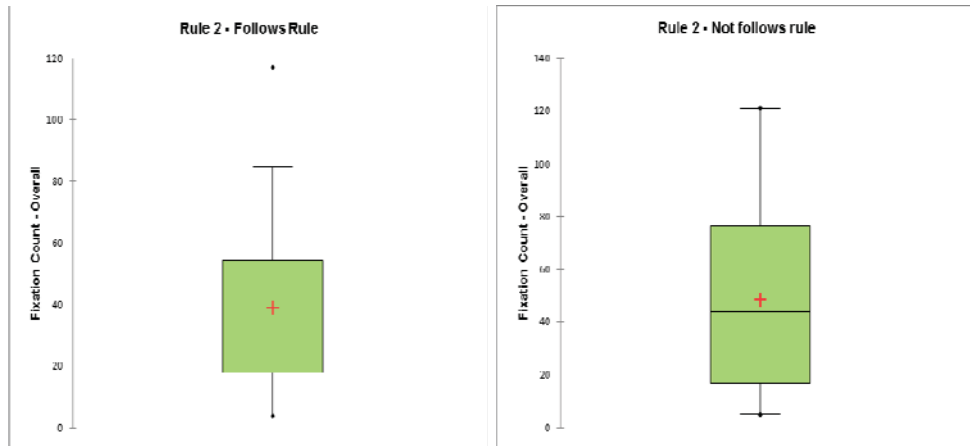
#### 4.6.2 Rule 2

The readability was slightly higher for the method that followed the avoid do/while loop rule. This shows that the participants found the method more readable when the method followed readability rules.



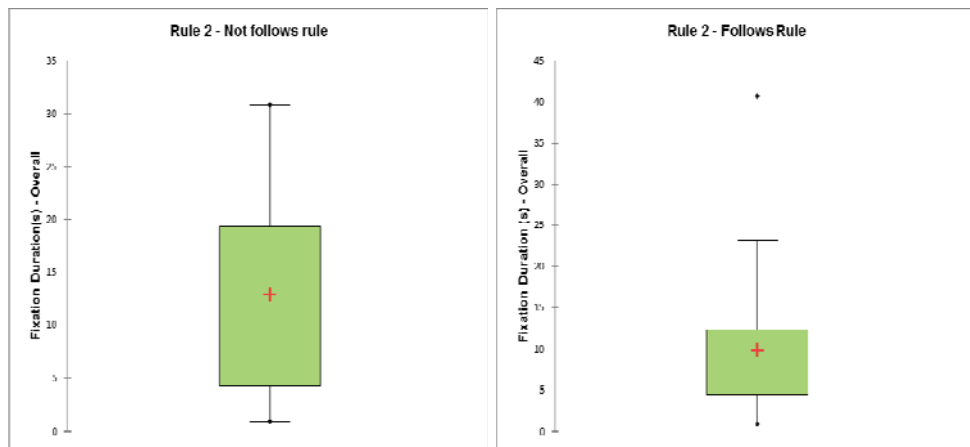
**Figure 33. Readability ranking for Rule 2 (avoid do/while)**

We did not find any significant difference in overall fixation count ( $p=0.145$ ) when the avoid do/while loop was used. Figure 34 shows the fixation count of the methods that follow avoid do/while rule and the one that does not follow avoid do/while rule.



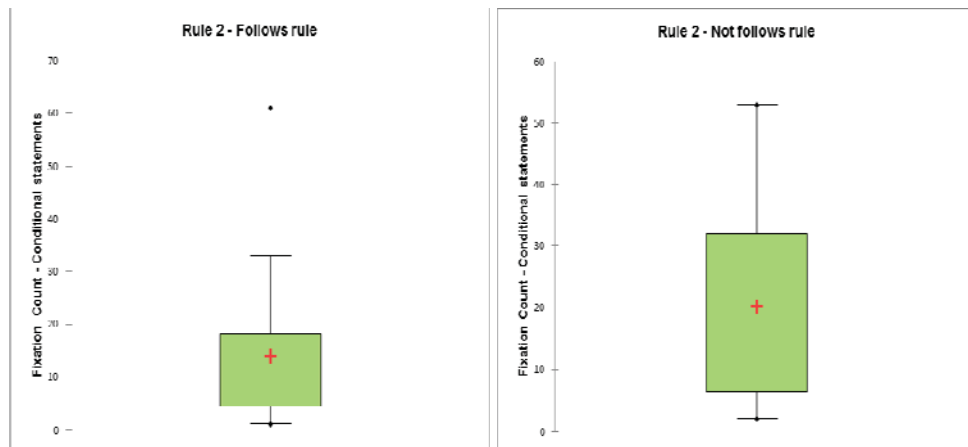
**Figure 34. Overall fixation count for the two methods in Rule 2 (avoid do/while) comparison task**

We did not find any significant difference in overall fixation duration ( $p=0.080$ ) when the avoid do/while rule was used. Figure 35 shows the fixation duration of the methods that follow avoid do/while rule and the one that does not follow avoid do/while rule.



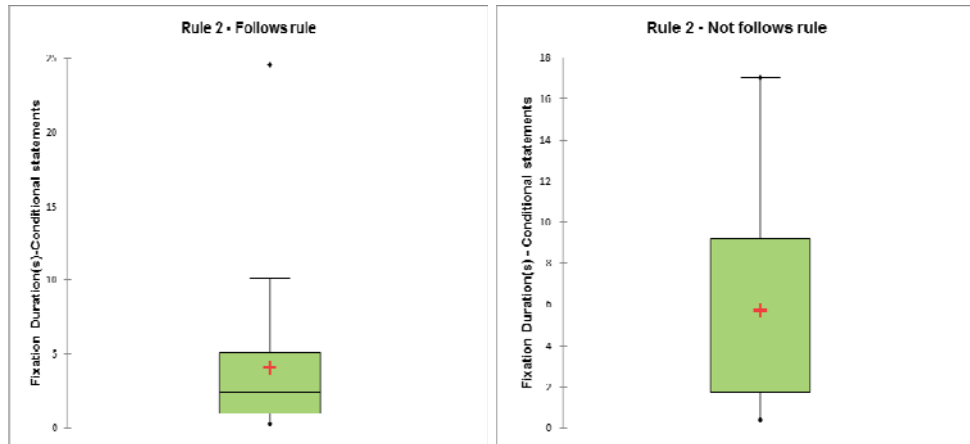
**Figure 35. Overall fixation duration for the two methods in Rule 2 (avoid do/while) comparison task**

We observed a significant difference in conditional statements fixation count ( $p=0.036$ ) when the avoid do/while loop was used. This shows that the participants looked the conditional statements less number of times when the methods followed avoid do/while rule. Figure 36 shows the fixation count of the methods that follow avoid do/while rule and the one that does not follow avoid do/while rule.



**Figure 36. Conditional statements fixation count in Rule 2 (avoid do/while) comparison task**

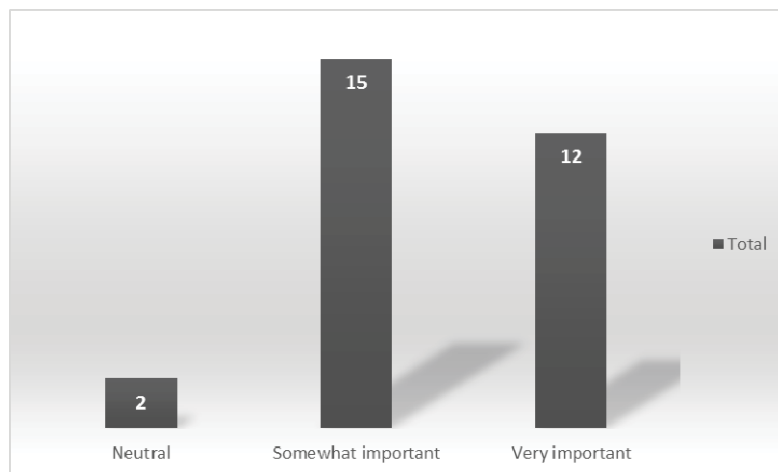
We observed a significant difference in conditional statements fixation duration ( $p=0.026$ ) when the avoid do/while loop was used. This shows that the participants took less time to analyze the methods that followed avoid do/while rule. Figure 37 shows the fixation duration of the methods that follow avoid do/while rule and the one that does not follow avoid do/while rule.



**Figure 37. Conditional statements fixation duration in Rule 2 (avoid do/while) comparison task**

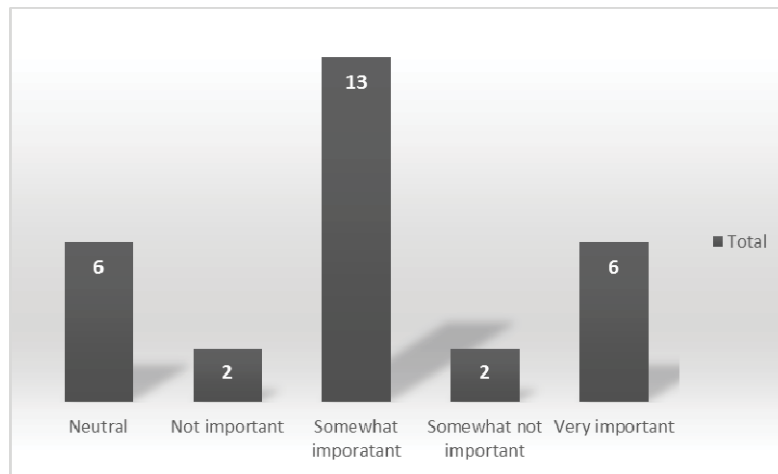
#### 4.7 Post Questionnaire Results

The participants were asked to rank the importance of minimize-nesting rule and avoid do/while loop rule used in the study in the post questionnaire form. Most of the participants thought that minimize-nesting rule is very important and somewhat important. None of the participants thought that minimize-nesting rule is not at all important.



**Figure 38. Importance of minimize-nesting rule**

Most of the participants thought that avoiding do/while loop rule is somewhat important and very important. Some of the participants also thought that avoiding do/while loop rule is not important and somewhat not important. Figure 39 shows the graph that the participants ranked in post questionnaire.



**Figure 39. Importance of avoid do/while loop rule**

#### **4.8 Observations and Discussion**

Based on the analysis presented, we find that developers tend to read the source code accurately when the two coding practices were used and the methods are correct. We did not observe much progress in accuracy for the incorrect methods that followed the rules. Since the participants did not know that the method may contain bugs, it is possible that there is no significant difference in accuracy for buggy solutions. However, there was a tendency to understand the source code better when the methods are correct and follow the rule. There is no significant difference in visual effort for the buggy

solution that does not follow the rule. Once the participants click the left mouse button, they cannot see the method while answering the questions. We think that the results may differ for the buggy solutions if the participants were informed about the errors in the method before they analyzed and allowed them to look back at the method whenever required.

Most of the participants analyzed the methods from minimize-nesting rule very quickly when compared to avoid do/while loop rule. While analyzing the last methods in the study, some of the participants showed lack of interest. When the participants were asked to answer if the method is logically correct or not, few participants gave the reason about why the method is logically incorrect based on their analysis. Participants took more time to answer about the correctness of the method based on certain input, as one cannot look at the code while answering the questions. Two of the participants took more than an hour to complete the study. Table 6 shows whether the hypotheses are accepted or rejected based on the results. As shown below, we do find a significant difference in the level of confidence and ease of readability rankings from participants. We also found significant differences on conditional statements when the minimize-nesting rule was followed. However, we did not get any significant difference in any of the other hypotheses. One reason for this could be due to the low sample size in our studies. We plan to perform this study with more participants in order to increase our confidence in the results and improve our external validity threat.

**Table 6. Hypotheses results**

<b>Hypothesis</b>	<b>R1 Minimize-Nesting Rule</b>	<b>R2 Avoid do/while Rule</b>
H1 <sub>0</sub> : R1-Time	Rejected	-
H2 <sub>0</sub> : R1-Accuracy	Rejected	-
H3 <sub>0</sub> : R1-Level of confidence	Accepted	-
H4 <sub>0</sub> : R1-Ease of readability	Accepted	-
H5 <sub>0</sub> : R1-Visual Effort	Accepted for overall method and conditional statements	-
H6 <sub>0</sub> : R2-Time	-	Rejected
H7 <sub>0</sub> : R2-Accuracy	-	Rejected
H8 <sub>0</sub> : R2-Level of confidence	-	Rejected
H9 <sub>0</sub> : R2-Ease of readability	-	Rejected
H10 <sub>0</sub> : R2-Visual Effort	-	Rejected

#### **4.9 Threats to Validity**

To overcome the influence of human factors we collected demographic data to make sure that the participants had a proficient java programming level, acceptable English reading skills, and no reading disorder. The participants were also asked (i) not to assume the survey as an assessment of their programming skills, (ii) complete the survey in a single session without any pause or interruption, (iii) not to use other tools to answer the respective questions, (iv) not to worry about the time to analyze the method, and (v) to be focused on understanding each method, since the participant cannot go back to the method once the set of questions appear on the screen.

To avoid any poorly designed artifacts, we conducted a pilot study with two graduate students before we started the actual experiment. This helped us to detect unclear problem descriptions, deficiencies in the questionnaires used in the study, and too complex code snippets or questions. We also monitored the overall feasibility and the time required to complete each task of the study.

To avoid any carryover effects, we organized the trials in such a way that each participant analyzed four different problems for a readability rule and each of the four solutions proposed for these problems is a different treatment. These solutions were again presented to each participant in a random order. External validity threats reduce the degree of generalization of our results. Firstly, these results could be valid for languages with similar loop and conditional constructs since we restricted the scope of the experiment to test the impact of rules related to simplifying loops and logic. However, if the code snippets used are more complex and have dependencies on other methods, the results may be somewhat different. Secondly, the subjects were mostly graduate and undergraduate students, so that it is plausible that our conclusions cannot be fully applied to the population of professional developers. Therefore, replications involving industrial subjects or other types of source code samples are highly desirable.



## CHAPTER 5

### Conclusions and Future Work

Program reading is an important skill and should be explicitly taught when learning how to program. Reading code is important because it is the first thing developers do to maintain a software system. Developers spend lot of time maintaining code, even more than writing the code from scratch. They need to read and understand the code before fixing bugs and implementing new features to existing code. We conducted an empirical study to see how two coding practices impact the degree of understandability of source code on effort.

For both the readability rules, we hypothesize that developers will be more accurate to analyze the correct methods following the rule. We find this to be true, however, our developers were not accurate analyzing the incorrect methods that followed the rule. The level of confidence and ease of readability of the programmers to analyze the methods increased using minimize-nesting rule. The visual effort of developers (determined through eye fixation counts and durations) was less to analyze the overall method and the conditional statements when the method followed the minimize-nesting rule. From the method comparison tasks, we observed that developers felt the methods are more readable when the rules are applied. In the method comparison task, when the avoid do/while rule is used, there was a significant difference in the visual effort to analyze conditional statements when the method followed the rule.

Future work includes collecting more data to minimize our external validity threat even further. We plan on running this study with industry professionals to determine if

we find any differences in the results we got with students used in this study. In addition, the visual effort of students/practitioners based on keywords used in the methods that can be used to provide valuable feedback to developers on task difficulty or determine when they are fatigued and need to take a break.

Researchers and practitioners can use this information to create coding style guides based on these rules. We can provide evidence to teach programming to students using specific rules that have been shown to be more effective.

## **APPENDIX Study Material**

You will find all the tasks here including the pre-questionnaire, post-questionnaire, tasks used in the study and the questions related to the tasks.

### **A.1. Study Instructions**

This study is concerned with software readability. Software readability is a property that influences how easily a given piece of code can be read and understood.

- Your task is to carefully read some code snippets and then you will be asked to answer questions about them.
- We will record the time you need to read the code snippets and answer the questions.
- Your answers and timing will only be used to study the readability of code and not to assess your performance.
- Please note that you cannot go back to a question that you have already finished answering.
- In total, you will see eight java methods in two parts. The whole exercise will take about 20 minutes. In the first part, you will be asked to read eight Java methods after which you will be asked very specific questions about it. You will not be able to see the source code while you are answering the question. So please take your time to understand the methods before you move on to answer the questions. When you are done with the reading the code on the screen, click the LEFT mouse button only once to advance to the next screen.

- In the second part, you will be shown two different ways in which a method is written to accomplish the same task. You will need to choose which one is more readable in your opinion and state why.
- Please do not guess the answers.
- You will fill in your answers in a web form that will pop up automatically.
- For each question, you will be asked to rate the difficulty level you faced and your confidence in the answer you provided. Use the mouse to select the options. 7
- Please try to maintain your position in the chair while you do the study so that we do not lose the tracking of your eyes. Moving the chair back or moving yourself back in the chair will cause the eye tracker to stop tracking. Small head movements such as looking at the keyboard to type should be fine.
- Find a comfortable position so we can begin. We will first begin with calibrating your eyes. Look at the black dot in the centre of the red circle and follow it around on the screen.

## **A.2. Pre-Questionnaire**

1. ID: \* \_\_\_\_\_
2. What is your mother tongue (native language)? \*
3. In what country do you live? \*
4. Do you have some form of reading disorder, such as dyslexia? \*
  - a. Yes
  - b. No
5. Are you currently working as a developer in a software company? \*
  - a. Yes
  - b. No

6. Are you currently an undergraduate student? \*
  - a. Yes
  - b. No
  
7. Are you currently a graduate student? \*
  - a. Yes
  - b. No
  
8. Please rate your English proficiency (in reading) \*
  - a. Very Poor
  - b. Poor
  - c. Satisfactory
  - d. Good
  - e. Very Good
  
9. Please rate your knowledge of the Java programming language \*
  - a. Very Poor
  - b. Poor
  - c. Satisfactory
  - d. Good
  - e. Very Good
  
10. Please indicate the number of years you have been actively programming (in any language) \*
  - a. Less than 1 year
  - b. Between 1 and 3 years
  - c. Between 3 and 5 years
  - d. More than 5 years
  
11. Please indicate the number of years you have been actively programming in Java \*
  - a. Less than 1 year
  - b. Between 1 and 3 years
  - c. Between 3 and 5 years
  - d. More than 5 years
  
12. How many years have you been working as a professional programmer? \*
  - a. Less than 1 year
  - b. Between 1 and 3 years
  - c. Between 3 and 5 years
  - d. More than 5 years
  - e. I have not worked as a professional programmer

### **A.3. Tasks and Comprehension**

The tasks used in the study and the questions given for each of the tasks are presented in this section. Note that the version numbers indicate the type of solution presented in the study. For instance, Version one indicates that the method follows the readability rule and is the correct solution for the problem specification, whereas version two indicates that the method does not follow the readability rule and is the correct solution for the problem specification.

#### **Rule 1 - Problem 1**

##### **Version One**

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = 0;
    if (numTwo < numThree) {
        theLargest = numThree;
    } else {
        if (numTwo > numOne) {
            theLargest = numTwo;
        }
    }
    if ((numOne > numTwo) && (numOne > numThree)) {
        theLargest = numOne;
    }
    return theLargest;
}
```

## Version Two

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest = 0;
    if ((numOne > numTwo) && (numOne > numThree)) {
        theLargest = numOne;
    }
    if ((numTwo > numOne) && (numTwo > numThree)) {
        theLargest = numTwo;
    }
    if ((numThree > numTwo) && (numThree > numOne)) {
        theLargest = numThree;
    }
    return theLargest;
}
```

## Version Three

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {
    int theLargest;
    if (numOne < numTwo) {
        if (numTwo < numThree) {
            theLargest = numThree;
        } else {
            theLargest = numTwo;
        }
    } else {
        if (numOne < numThree) {
            theLargest = numThree;
        } else {
            theLargest = numOne;
        }
    }
    return theLargest;
}
```

#### Version 4

```
public int findTheBiggest(int numOne, int numTwo, int numThree) {  
    int theLargest = numOne;  
    if (numTwo > theLargest) {  
        theLargest = numTwo;  
    }  
    if (numThree > theLargest) {  
        theLargest = numThree;  
    }  
    return theLargest;  
}
```

#### Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? \***

1. The method does not work properly when the input numbers are 1, 1, and 2
2. The method does not work properly when the input numbers are 2, 2, and 2
3. The method does not work properly when the input numbers are 1, 2, and 3
4. The method works properly when the input numbers are all zero.

**How confident are you in your own level of comprehension of the previous method? \***

\*

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No



**If you selected No above, please explain why the method is not logically correct**

**Rule 1 - Problem 2**

**Version One**

```
public static char findGrade(int marks) {  
    if (marks >= 90) {  
        return 'A';  
    }  
    if (marks < 90 && marks > 80) {  
        return 'B';  
    }  
    if (marks < 80 && marks > 70) {  
        return 'C';  
    }  
    if (marks < 70 && marks >= 60) {  
        return 'D';  
    }  
    return 'E';  
}
```

### Version Two

```
public static char findGrade(int marks) {
    char grade;
    if (marks >= 90) {
        grade = 'A';
    } else {
        if (marks >= 80) {
            grade = 'B';
        } else {
            if (marks >= 70) {
                grade = 'C';
            } else {
                if (marks >= 60) {
                    grade = 'D';
                } else {
                    grade = 'F';
                }
            }
        }
    }
    return grade;
}
```

### Version Three

```
public static char findGrade(int marks) {
    if (marks >= 90) {
        return 'A';
    }
    if (marks >= 80) {
        return 'B';
    }
    if (marks >= 70) {
        return 'C';
    }
    if (marks >= 60) {
        return 'D';
    }
    return 'F';
}
```

### Version Four

```
public static char findGrade(int marks) {
    char grade;
    if (marks >= 90) {
        grade = 'A';
    } else if (marks <= 89 && marks > 80) {
        grade = 'B';
    } else if (marks <= 79 && marks > 70) {
        grade = 'C';
    } else if (marks <= 69 && marks > 60) {
        grade = 'D';
    } else {
        grade = 'F';
    }
    return grade;
}
```

## Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? \***

1. The method does not work properly when the input mark is 80
2. The method works properly when the input mark is 70
3. The method does not work properly when the input mark is 50
4. The method does not work properly when the input mark is 0

**How confident are you in your own level of comprehension of the previous method? \***

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No

**If you selected No above, please explain why the method is not logically correct**

## Rule 1 - Problem 3

### Version One

```
public static String bodyMassIndexCategory(double bmi) {  
  
    if (bmi < 15.00) {  
        return "Very severely underweight";  
    }  
    if (bmi >= 15.00 && bmi < 16.00) {  
        return "Severely underweight";  
    }  
    if (bmi >= 16.00 && bmi < 18.5) {  
        return "Underweight";  
    }  
    if (bmi > 18.50 && bmi < 25.00) {  
        return "Normal (healthy weight)";  
    }  
    return "Overweight";  
}
```

## Version Two

```
public static String bodyMassIndexCategory(double bmi) {
    String result;
    if (bmi < 15.00) {
        result = "Very severely underweight";
    } else {
        if (bmi < 16.00) {
            result = "Severely underweight";
        } else {
            if (bmi < 18.5) {
                result = "Underweight";
            } else {
                if (bmi < 25.00) {
                    result = "Normal (healthy weight)";
                } else {
                    result = "Overweight";
                }
            }
        }
    }
    return result;
}
```

### Version Three

```
public static String bodyMassIndexCategory(double bmi) {

    if (bmi < 15.00) {
        return "Very severely underweight";
    }
    if (bmi < 16.00) {
        return "Severely underweight";
    }
    if (bmi < 18.5) {
        return "Underweight";
    }
    if (bmi < 25.00) {
        return "Normal (healthy weight)";
    }
    return "Overweight";
}
```

## Version Four

```
public static String bodyMassIndexCategory(double bmi) {
    String result;
    if (bmi < 15.00) {
        result = "Very severely underweight";
    } else {
        if (bmi >= 15.00 && bmi < 16.00) {
            result = "Severely underweight";
        } else {
            if (bmi >= 16.00 && bmi < 18.5) {
                result = "Underweight";
            } else {
                if (bmi > 18.50 && bmi < 25.00) {
                    result = "Normal (healthy weight)";
                } else {
                    result = "Overweight";
                }
            }
        }
    }
    return result;
}
```

## Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? \***

1. The method does not work properly when the bmi is 18.5
2. The method does not work properly when the bmi is 10



3. The method does not work properly when the bmi is 40
4. None of the previous statements is true

**How confident are you in your own level of comprehension of the previous method?**

\*

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No

**If you selected No above, please explain why the method is not logically correct**

## Rule 1 - Problem 4

### Version One

```
public int countPosNumbers(int numOne, int numTwo, int numThree) {
    if (numOne > 0) {
        if (numTwo > 0) {
            if (numThree > 0) {
                return 3;
            } else {
                return 2;
            }
        } else if (numThree > 0) {
            return 2;
        } else {
            return 1;
        }
    } else if (numTwo > 0) {
        if (numThree > 0) {
            return 1;
        } else {
            return 2;
        }
    } else if (numThree > 0) {
        return 1;
    } else {
        return 0;
    }
}
```

## Version Two

```
public int countPosNumbers(int numOne, int numTwo, int numThree) {
    int count = 0;
    if (numOne >= 0) {
        count++;
    }
    if (numTwo >= 0) {
        count++;
    }
    if (numThree >= 0) {
        count++;
    }
    return count;
}
```

### Version Three

```
public int countPosNumber(int numOne, int numTwo, int numThree) {
    if (numOne > 0) {
        if (numTwo > 0) {
            if (numThree > 0) {
                return 3;
            } else {
                return 2;
            }
        } else if (numThree > 0) {
            return 2;
        } else {
            return 1;
        }
    } else if (numTwo > 0) {
        if (numThree > 0) {
            return 2;
        } else {
            return 1;
        }
    } else if (numThree > 0) {
        return 1;
    } else {
        return 0;
    }
}
```

## Version Four

```
public int countPosNumbers(int numOne, int numTwo, int numThree) {
    int count = 0;
    if (numOne > 0) {
        count++;
    }
    if (numTwo > 0) {
        count++;
    }
    if (numThree > 0) {
        count++;
    }
    return count;
}
```

## Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? \***

1. The method does not work properly when the input numbers are 1, 1, and 1
2. The method does not work properly when the input numbers are 1, 0, and 2
3. The method does not work properly when the input numbers are 0, 2, and 3
4. None of the previous statements is true

**How confident are you in your own level of comprehension of the previous method? \***

\*

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No

**If you selected No above, please explain why the method is not logically correct**

## Rule 2 - Problem 1

### Version One

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain = "y";
    while (tryAgain.compareTo("y") == 0) {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulation!");
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }
        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    sc.close();
}
```

## Version Two

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain;
    do {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulations!");
            break;
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }
        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    while (tryAgain.compareTo("y") == 0);
    sc.close();
}
```



## Version Three

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain = "y";
    while (tryAgain.compareTo("y") == 0) {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulations!");
            break;
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }

        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    sc.close();
}
```

## Version Four

```
public static void whoIsTheAuthor() {
    Scanner sc = new Scanner(System.in);
    System.out.println("Who is the author of the book 'David Copperfield'?");
    System.out.println("a. Lewis Carroll");
    System.out.println("b. Mark Twain");
    System.out.println("c. Charles Dickens");
    System.out.println("d. Oscar Wilde");
    String choice;
    String tryAgain;
    do {
        System.out.print("Enter your choice:");
        choice = sc.nextLine();
        if (choice.compareTo("c") == 0) {
            System.out.println("Congratulation!");
        } else if (choice.compareTo("q") == 0 || choice.compareTo("e") == 0) {
            System.out.println("Exiting...!");
            break;
        } else {
            System.out.println("Incorrect!");
        }
        System.out.print("Again? press 'y' to continue:");
        tryAgain = sc.nextLine();
    }
    while (tryAgain.compareTo("y") == 0);
    sc.close();
}
```

## Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? \***

1. The method does not end when the user types “q” or “e”
2. The method does not end when the user chooses the correct answer to the question

3. The method ends when the user chooses an incorrect answer to the question
4. The method ends when the user chooses the option c

**How confident are you in your own level of comprehension of the previous method?**

\*

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No

**If you selected No above, please explain why the method is not logically correct**

## Rule 2 - Problem 2

### Version One

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass = oldPass;
    boolean valid = false;
    while (newPass.equals(oldPass) || !valid) {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() != 4) {
            valid = false;
        } else {
            for (int i = 0; i < newPass.length(); i++) {
                for (int j = i + 1; j < newPass.length(); j++) {
                    if (newPass.charAt(j) == newPass.charAt(i)) {
                        valid = false;
                    }
                }
            }
        }
    }
    sc.close();
    return newPass;
}
```

## Version Two

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass;
    boolean valid;
    do {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() != 4) {
            valid = false;
        } else {
            for (int i = 0; i < newPass.length(); i++) {
                for (int j = i + 1; j < newPass.length(); j++) {
                    if (newPass.charAt(j) == newPass.charAt(i)) {
                        valid = false;
                    }
                }
            }
        }
    }
    while (newPass.equals(oldPass) || !valid);
    sc.close();
    return newPass;
}
```

### Version Three

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass = oldPass;
    boolean valid = false;
    while (newPass.equals(oldPass) || !valid) {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() < 4) {
            valid = false;
        }
        for (int i = 0; i < newPass.length(); i++) {
            for (int j = i + 1; j < newPass.length(); j++) {
                if (newPass.charAt(j) == newPass.charAt(i)) {
                    valid = false;
                }
            }
        }
    }
    sc.close();
    return newPass;
}
```

## Version Four

```
public String changePassword(String oldPass) {
    Scanner sc = new Scanner(System.in);
    String newPass;
    boolean valid;
    do {
        System.out.println("Please, write a new password");
        newPass = sc.nextLine();
        valid = true;
        if (newPass.length() < 4) {
            valid = false;
        }
        for (int i = 0; i < newPass.length(); i++) {
            for (int j = i + 1; j < newPass.length(); j++) {
                if (newPass.charAt(j) == newPass.charAt(i)) {
                    valid = false;
                }
            }
        }
    }
    while (newPass.equals(oldPass) || !valid);
    sc.close();
    return newPass;
}
```

## Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? (Assuming that the user gives new passwords that are different from the old one) \***

1. The method accepts new passwords with exactly 4 characters and no duplicate characters
2. The method accepts new passwords with exactly 4 characters with or without duplicate characters
3. The method accepts new passwords with at least 4 characters and no duplicate characters
4. The method accepts new passwords with at least 4 characters with or without duplicate characters

**How confident are you in your own level of comprehension of the previous method?**

\*

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No

**If you selected No above, please explain why the method is not logically correct**



## Rule 2 - Problem 3

### Version One

```
public int sumPositiveNums(int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    while (counter < limit) {
        if (nums[counter] > 0) {
            sum += nums[counter];
        }
        counter++;
    }
    return sum;
}
```

### Version Two

```
public int sumPositiveNums(int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    if (limit == 0) {
        return 0;
    }
    do {
        if (nums[counter] > 0) {
            sum += nums[counter];
        }
        counter++;
    }
    while (counter < limit);
    return sum;
}
```

### Version Three

```
public int sumPositiveNums(int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    while (counter < limit) {
        if (nums[counter] > 0) {
            sum += nums[counter];
            counter++;
        }
    }
    return sum;
}
```

### Version Four

```
public int sumPositiveNums(int[] nums) {
    int limit = nums.length;
    int sum = 0;
    int counter = 0;
    do {
        if (nums[counter] > 0) {
            sum += nums[counter];
            counter++;
        }
    }
    while (counter < limit);
    return sum;
}
```

## Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? \***

1. The method throws an exception when the array is empty
2. The method never ends when the array has one or more positive numbers
3. The method never ends when the array has one or more zeros
4. None of the previous statements is true

**How confident are you in your own level of comprehension of the previous method? \***

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No

**If you selected No above, please explain why the method is not logically correct**

## Rule 2 - Problem 4

### Version One

```
public int countLetter(String phrase, char letter) {
    int max = phrase.length();
    int counter = 0;
    int index = 0;
    while (index < max) {
        if (phrase.charAt(index) != letter) {
            index++;
            continue;
        }
        index++;
        counter++;
    }
    return counter;
}
```

### Version Two

```
public int countLetter(String phrase, char letter) {
    int max = phrase.length();
    if (max == 0) {
        return 0;
    }
    int counter = 0;
    int index = 0;
    do {
        if (phrase.charAt(index) != letter) {
            index++;
            continue;
        }
        index++;
        counter++;
    }
    while (index < max);
    return counter;
}
```

### Version Three

```
public int countLetter(String phrase, char letter) {
    int max = phrase.length();
    int counter = 0;
    int index = 0;
    while (index < max) {
        if (phrase.charAt(index) != letter) {
            continue;
        }
        index++;
        counter++;
    }
    return counter;
}
```

## Version Four

```
public int countLetter(String phrase, char letter) {
    int max = phrase.length();
    if (max == 0) {
        return 0;
    }
    int counter = 0;
    int index = 0;
    do {
        if (phrase.charAt(index) != letter) {
            continue;
        }
        index++;
        counter++;
    }
    while (index < max);
    return counter;
}
```

## Questions

**Based on your programming experience, how would you rate the readability of the previous piece of code? \***

1. Very difficult to read
2. Difficult to read
3. Neutral
4. Easy to read
5. Very easy to read

**Which of the following statements is true about the code you just read? \***

1. The method never reviews the last character of the phrase
2. The method does not work properly when the phrase is empty
3. The method works properly when the letter is "x" and the phrase is "awerx"
4. The method works properly only when all the characters of the phrase are equals to the letter

**How confident are you in your own level of comprehension of the previous method?**

\*

1. Very confident
2. Somewhat confident
3. Neutral
4. Not very confident
5. Not at all confident

**Is the method you just saw logically correct? \***

1. Yes
2. No

**If you selected No above, please explain why the method is not logically correct**

#### **A.4. Post Questionnaire**

1. ID: \* \_\_\_\_\_

2. Rank the importance of minimizing nesting in source code. \*

- a. Very important
- b. Somewhat important
- c. Neutral
- d. Somewhat not important
- e. Not important

3. Rank the importance of avoiding do-while statements in source code. \*

- a. Very important
- b. Somewhat important
- c. Neutral
- d. Somewhat not important
- e. Not important

4. Please explain any difficulties you encountered during the study.

5. What do you consider to be the most important aspects for the readability and comprehensibility of code?

6. What is your personal favourite technique, tool, or approach to help you read and comprehend code?

7. Please feel free to leave comments about the code and questions included in this study



## References

- Abbas, Nadeem. 2009. "Properties of 'Good' Java Examples." Sweden: Umeå University.
- Barik, Titus, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. "Do Developers Read Compiler Error Messages?" Accessed April 14. <https://people.engr.ncsu.edu/ermurph3/papers/icse17.pdf>.
- Bednarik, Roman. 2012. "Expertise-Dependent Visual Attention Strategies Develop over Time During Debugging with Multiple Code Representations." *International Journal of Human-Computer Studies* 70 (2): 143–155. doi:10.1016/j.ijhcs.2011.09.003.
- Bednarik, Roman, and Markku Tukiainen. 2006. "An Eye-Tracking Methodology for Characterizing Program Comprehension Processes." In *Proc. of the Symposium on Eye Tracking Research & Applications*, 125–132. ACM.
- Beelders, Tanya, and Jean-Pierre du Plessis. 2016. "The Influence of Syntax Highlighting on Scanning and Reading Behaviour for Source Code." In *Proceedings of the Annual Conference of the South African Institute of Computer Scientists and Information Technologists*, 5:1–5:10. SAICSIT '16. New York, NY, USA: ACM. doi:10.1145/2987491.2987536.

- Borstler, Jergen, Caspersen Michael, and Nordstrom Marie. 2016. “Beauty and the Beast” 24 (2): 231–46. doi:10.1007/s11219-015-9267-5.
- Börstler, Jürgen, Marie Nordström, and James H. Paterson. 2011. “On the Quality of Examples in Introductory Java Textbooks.” *Trans. Comput. Educ.* 11 (1): 3:1–3:21. doi:10.1145/1921607.1921610.
- Boswell, Dustin, and Trever Foucher. 2011. *The Art of Readable Code*. First. O’Reilly Media.
- Buse, Raymond P. L., and Westley R. Weimer. 2010. “Learning a Metric for Code Readability.” *IEEE Trans. Softw. Eng.* 36 (4): 546–558. doi:10.1109/TSE.2009.70.
- Busjahn, Teresa, Roman Bednarik, Andrew Begel, Martha Crosby, James Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. 2015. “Eye Movements in Code Reading: Relaxing the Linear Order.” In *International Conference on Program Comprehension*, 255–65. IEEE. <http://dl.acm.org/citation.cfm?id=2820320>.
- Busjahn, Teresa, Carsten Schulte, Bonita Sharif, Simon, Andrew Begel, Michael Hansen, Roman Bednarik, et al. 2014. “Eye Tracking in Computing Education.” In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, 3–10. New York, NY, USA: ACM. doi:10.1145/2632320.2632344.
- Cowen, Laura, Linden J.s Ball, and Judy Delin. 2002. “An Eye Movement Analysis of Web Page Usability.” In *People and Computers XVI - Memorable Yet Invisible:*

- Proceedings of HCI 2002*, edited by Kristine Faulkner, Janet Finlay, and Françoise Détienne, 317–335. London: Springer London.  
[http://dx.doi.org/10.1007/978-1-4471-0105-5\\_19](http://dx.doi.org/10.1007/978-1-4471-0105-5_19).
- Cristino, Filipe, Sebastiaan Mathôt, Jan Theeuwes, and Iain D. Gilchrist. 2010. “ScanMatch: A Novel Method for Comparing Fixation Sequences.” *Behavior Research Methods* 42 (3): 692–700. doi:10.3758/BRM.42.3.692.
- Crosby, Martha E., and Jan Stelovsky. 1990. “How Do We Read Algorithms? A Case Study.” *Computer* 23 (1): 24–35.
- D’Angelo, Sarah, and Andrew Begel. 2017. “Improving Communication Between Pair Programmers Using Shared Gaze Awareness.” In *Proceedings of CHI*. ACM.  
<https://www.microsoft.com/en-us/research/publication/improving-communication-pair-programmers-using-shared-gaze-awareness/>.
- Deimel, Lionel E., Jr. 1985. “The Uses of Program Reading.” *SIGCSE Bull.* 17 (2): 5–14.  
doi:10.1145/382204.382524.
- Dibble II, Christopher, and Paul Gestwicki. 2014. “Refactoring Code to Increase Readability and Maintainability: A Case Study.” *Journal of Computing Sciences in Colleges* 30 (1): 41–51.
- Dubay, William H. 2004. “The Principles of Readability.” *Costa Mesa, CA: Impact Information*.
- Duchowski, Andrew T., Eric Medlin, Anand Gramopadhye, Brian Melloy, and Santosh Nair. 2001. “Binocular Eye Tracking in VR for Visual Inspection Training.” In , 1–8. VRST ’01. New York, NY, USA: ACM. doi:10.1145/505008.505010.

- Ehmke, Claudia, and Stephanie Wilson. 2007. "Identifying Web Usability Problems from Eye-Tracking Data." In *Proceedings of the 21st British HCI Group Annual Conference on People and Computers: HCI...But Not As We Know It - Volume 1*, 119–128. BCS-HCI '07. Swinton, UK, UK: British Computer Society. <http://dl.acm.org/citation.cfm?id=1531294.1531311>.
- Faro, A., D. Giordano, C. Spampinato, D. De Tommaso, and S. Ullo. 2010. "An Interactive Interface for Remote Administration of Clinical Tests Based on Eye Tracking." In , 69–72. ETRA '10. New York, NY, USA: ACM. doi:10.1145/1743666.1743683.
- Halverson, Tim, and Anthony J. Hornof. 2007. "A Minimal Model for Predicting Visual Search in Human-Computer Interaction." In , 431–434. CHI '07. New York, NY, USA: ACM. doi:10.1145/1240624.1240693.
- Høst, Einar W., and Bjarte M. Østvold. 2009. "Debugging Method Names." In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, 294–317. Genoa. Berlin, Heidelberg: Springer-Verlag. doi:10.1007/978-3-642-03013-0\_14.
- Imants, Puck, and Tjerk de Greef. 2014. "Eye Metrics for Task-Dependent Automation." In , 23:1–23:4. ECCE '14. New York, NY, USA: ACM. doi:10.1145/2637248.2637274.
- Isokoski, Poika, Markus Joos, Oleg Spakov, and Benoît Martin. 2009. "Gaze Controlled Games." *Univers. Access Inf. Soc.* 8 (4): 323–337. doi:10.1007/s10209-009-0146-3.

- Jacob, Robert J. K., and Keith S. Karn. 2003. "Eye Tracking in Human-Computer Interaction and Usability Research: Ready to Deliver the Promises." *Mind* 2 (3): 4.
- Kasarskis, Peter, Jennifer Stehwien, Joey Hickox, Anthony Aretz, United States, Air Force Academy, and Chris Wickens. 2001. "Comparison of Expert and Novice Scan Behaviors during Vfr Flight." In *In Proceedings of the 11th International Symposium on Aviation Psychology*.
- Liblit, Ben, Andrew Begel, and Eve Sweetser. 2006. "Cognitive Perspectives on the Role of Naming in Computer Programs." In *Proceedings of the 18th Annual Psychology of Programming Workshop*. Brighton, England, United Kingdom.
- Lionel, E.Deimel, and Naveda J.Fernando. 1990. "Reading Computer Programs: Instructor's Guide and Exercises." CMU/SEI-90-EM-3. <http://www.literateprogramming.com/em3.pdf>.
- Madsen, Adrian, Adam Larson, Lester Loschky, and N. Sanjay Rebello. 2012. "Using ScanMatch Scores to Understand Differences in Eye Movements Between Correct and Incorrect Solvers on Physics Problems." In , 193–196. ETRA '12. New York, NY, USA: ACM. doi:10.1145/2168556.2168591.
- Mason, Andrew, and Chandralekha Singh. 2011. "Assessing Expertise in Introductory Physics Using Categorization Task." *Physical Review Special Topics - Physics Education Research* 7 (2). doi:10.1103/PhysRevSTPER.7.020110.
- Rosengrant, David. 2010. "Gaze Scribing in Physics Problem Solving." In , 45–48. ETRA '10. New York, NY, USA: ACM. doi:10.1145/1743666.1743676.

- Rudolph, Flesch. 1948. "A New Readability Yardstick." *Journal of Applied Psychology* 32 (3): 221–33. doi:<http://dx.doi.org/10.1037/h0057532>.
- Sedano, T. 2016. "Code Readability Testing, an Empirical Study." In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, 111–17. doi:[10.1109/CSEET.2016.36](https://doi.org/10.1109/CSEET.2016.36).
- Sharafi, Zohreh, Z  phyrin Soh, and Yann-Ga  l Gu  h  neuc. 2015. "A Systematic Literature Review on the Usage of Eye-Tracking in Software Engineering." *Inf. Softw. Technol.* 67 (C): 79–107. doi:[10.1016/j.infsof.2015.06.008](https://doi.org/10.1016/j.infsof.2015.06.008).
- Sharafi, Zohreh, Z  phyrin Soh, Yann-Gael Gueheneuc, and Giuliano Antoniol. 2012. "Women and Men - Different but Equal: On the Impact of Identifier Style on Source Code Reading." *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, 27–36. doi:[10.1109/ICPC.2012.6240505](https://doi.org/10.1109/ICPC.2012.6240505).
- Sharif, Bonita, Katja Kevic, Braden M. Walters, Timothy R. Shaffer, David C. Shepherd, and Thomas Fritz. 2015. "Tracing Software Developers' Eyes and Interactions for Change Tasks." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 202–213. ESEC/FSE 2015. New York, NY, USA: ACM. doi:[10.1145/2786805.2786864](https://doi.org/10.1145/2786805.2786864).
- Sharif, Bonita, and Jonathan I. Maletic. 2010a. "An Eye Tracking Study on camelCase and Under\_score Identifier Styles." In *18th IEEE International Conference on Program Comprehension (ICPC'10)*, 196–205. IEEE Computer Society. doi:[dx.doi.org/10.1109/ICPC.2010.41](https://doi.org/10.1109/ICPC.2010.41).

- Sharif, Bonita, and Jonathan I Maletic. 2010b. "An Eye Tracking Study on the Effects of Layout in Understanding the Role of Design Patterns." *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 1–10. doi:10.1109/ICSM.2010.5609582.
- Sharif, Bonita, Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, and Michael Falcone. 2015. "iTrace: Enabling Eye Tracking on Software Artifacts Within the IDE to Support Software Engineering Tasks." In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 954–957. ESEC/FSE 2015. New York, NY, USA: ACM. doi:10.1145/2786805.2803188.
- Tashtoush, Yahya, Odat Zeinab, and Maryan Yatim. 2013. "Impact of Programming Features on Code Readability" 7 (6): 441–58. doi:http://dx.doi.org/10.14257/ijseia.2013.7.6.38.
- Turner, Rachel, Michael Falcone, Bonita Sharif, and Alina Lazar. 2014. "An Eye-Tracking Study Assessing the Comprehension of C++ and Python Source Code." In *Proc. of the Symposium on Eye Tracking Research & Applications*, 231–234. Safety Harbor, Florida: ACM.