Test Code Comprehension: Insights from an Eye Tracker

Version of November 7, 2020

Sharanya Suresha Konandur

Test Code Comprehension: Insights from an Eye Tracker

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Sharanya Suresha Konandur
born in Shivamogga, India

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Test Code Comprehension: Insights from an Eye Tracker

Author: Sharanya Suresha Konandur
Student id: 4912020
Email: `sharanyasureshak@gmail.com`

Abstract

Software maintenance is an essential and time-consuming task during the software development cycle. Readability of test code is a crucial element for performing programming tasks, such as testing, bug fixing and maintaining code. Hence poorly written tests are difficult to maintain and lose their value to developers. In order to overcome this problem, we need to understand how programmers read test code. Therefore we conducted an empirical study to analyze the various reading patterns in novices and professionals using a sophisticated eye tracking device. Our results show that *(i)* all programmers first comprehended the production code and then switched between test and production codes, *(ii)* novices had higher fixations reading test code and assert statements, *(iii)* professionals revisited the test code more than novices, *(iv)* professionals had significantly lesser test code coverage than novices, and *(v)* there is a significant difference in reading test code between novice and professionals.

Thesis Committee:

Chair:	Dr. A.E. (Andy) Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Finavaro Aniche, Faculty EEMCS, TU Delft
Co-Supervisor:	Dr. Enrique Larios Vargas, TU Delft
Committee Member:	Dr. O.E. (Odette) Scharenborg, Faculty EEMCS, TU Delft

Preface

Originally, this foundation for this research emerged through my passion for bridging the gap between psychology and software programming. As the world continues to transition into the digital era, most industries require software developers and programming as become a new norm of the 20th century. Yet we have little knowledge about behind the psychology of program comprehension. It is my ambition not just to analyze, but to develop tools to help young programmers code better. In reality, without a strong support network I would have not accomplished my current success. In fact, without a good support network, I might not have accomplished my current level of success. I would first like to thank my research supervisors Dr. M. Finavaro Aniche and Dr. Enrique Larios Vargas. This thesis would never have been accomplished without your encouragement and commitment to every stage of the process. I want to express my most heartfelt gratitude for your support and understanding over the past 12 months. I would also like to thank the experts who were involved in the validation survey for this research project: Dr. A.E. (Andy) Zaidman and Dr. O.E. (Odette) Scharenborg. It took more than academic guidance to achieve my dissertation, and I have several people to thank for believing in me over the past two years of my masters program. I can't begin to express my sincere gratitude for their friendship. Finally, I must thank my parents for providing me with constant motivation and support through my journey in the university and writing this thesis Without them this achievement was not attainable. This thesis attests to your unconditional love and guidance.

Sharanya Suresha Konandur
Delft, the Netherlands
November 7, 2020

Contents

List of Figures

Chapter 1

Introduction

Software readability is a property which determines the ease reading and comprehending of a given piece of code. Readability of code during program comprehension is a crucial element for performing programming tasks, such as testing, bug fixing and maintaining code. These costs make up to 50% of maintenance tasks [38], which inevitably leads to the alteration of the code snippets. Hence readability must be considered a major factor in determining maintenance of code. Subsequently, better readability of code can minimize both the maintenance and the software process costs. This can result in a cheaper software development process. Reading the software code tends to take over 40% of the total time to comprehend [38]. Buse et al. [12] recognized readability as a significant factor in understanding source codes. Various studies on readability such as investigation of different code styles [41], improving software quality code readability [11] [70], readability testing [77] and effect of code features on readability [3], [74, 21] [35], provide novel approaches to measure readability.

Moreover, modern code bases are not only composed of production code; instead, developers write extensive test code bases to ensure the quality of their systems. Maintaining and writing good test code is important because the patterns of test smells are repetitive and induce high robust costs which are extensively researched by Meszaros et al. [48]. Hence understanding how programmers read test code is the first step to write better software code and reduce software maintenance costs. In the past years, there has been immense research on comprehension on source code but none of the studies uses bio-metric devices to analyze the readability of tests to measure a programmer's expertise.

Hence, our motivations for conducting this research are therefore important for two reasons. Firstly, we look into the different reading patterns during test code comprehension performed by novice and professionals programmers using an eye tracker. Analyzing the reading patterns helps us define whether a developer needs some rest to minimize potential errors or to estimate the time required to comprehend the code realistically. In our research, however, we follow an empirical method consisting of 14 professionals and 15 novices programmers, using an eye-tracker which help us understand the different reading patterns. We focus on test code complexity by generating heat maps to each participant and calculating the fixation time, average time spent and revisits in different Area Of Interest (AOI). In particular, we explore the principle of linearity. The linearity of the reading is how strongly reader adheres to natural language order(left-right,top-bottom) since software programs are different from natural languages, developers follow unique ways to understand code. This method has been proven successful by various studies, to map source code readability [73], [14], [79].

Secondly, we go beyond analyzing the basic metrics of fixation and gaze patterns. Our thesis focuses on implementing linear mixed regression model to classify the significant differences in fixation features while performing test code comprehension. Stein and Brennan [45] implemented ANalysis Of VAriance (ANOVA) to examine the effect on the outcome of the second group based on eye gazes of other respondents. Their results show that respondent's eye gaze provides essential clues to other person's performance. In our study, we use linear mixed regression models as they are more robust than ANOVA to validate the results by the participants.

Finally, we discuss the challenges faced during test code comprehension. Even though there are numerous studies in the field of code comprehension, there is not enough documentation on the difficulties faced by novices and professionals while performing test code comprehension. In this thesis, we aim to bridge this knowledge gap by providing a detailed analysis of challenges which will act as a starting point for future researchers and academics to understand test code comprehension. We can improve the readability of code by understanding the various issues encountered while performing code comprehension and aid novices with tools to write good readable tests in the future.

The research questions addressed in this thesis are:

RQ1: How do developers read test code?

RQ2: What are the differences between novices and professionals when reading test code?

Our results show that all the participants first comprehended the production code, novices had higher fixations reading test code and assert statements, professionals revisited the test code more than novices, professionals had significantly lesser test code coverage than novices, and finally, we found that there is a significant difference by applying linear mixed model in reading test code between novice and professionals. This paper contributes by providing an empirical analysis of test code readability, examines the different reading methodologies using fixation features and also provides a starting point for future research in the field on test code comprehension using bio-metric devices.

The report is structured in the following manner. Chapter 2 provides background information about the eye movements and highlights related research conducted in this pioneering field. In Chapter 3, we describe the research methodology and the experiment setup and all the required criteria for this pilot study. Further, in chapter 4, we present the results and observations gathered from our experiment. Chapter 5, we discuss the overview of the project's contribution and future work which can be performed based on the finding. Lastly, in Chapter 6, we provide our conclusions.

Chapter 2

Related Work

To address the research questions in this study, we first discuss the background of code comprehension and look into the various fields of code comprehension studies. Finally, we discuss the advancements in research methods over the last few decades.

2.1 Background on Code Comprehension

Software maintenance is an extremely demanding, vast and complex subject which requires the developers to understand the source code in detail [10]. To do the necessary maintenance work, skilled developers must know the code framework that is continuously changing. Hence code comprehension is increasingly being taken into account during software development, and it is a significant concern for software development companies. The lack of initial developers reduces code comprehension and thus affects the participants negatively [10]. A maintenance team needs to establish a fast and efficient understanding of the source code relevant to the maintenance tasks. A study by Penta et al. [23] concluded that code comprehension techniques have confirmed to obtain potential advantages in the maintenance work.

Furthermore, developers would want to re-evaluate their code and modify it to their requirements. The capacity to read and comprehend a program that others have written is also a crucial challenge. Rather of having one developer, software companies rely on team effort so that everyone writes a component of code working as a team. Code comprehension is one of the major steps to tackle many software and maintenance tasks. The accumulation of information about the program is critically important [71], [47]. This information is generally complex, which means many aspects such as maintenance [86], reporting [26], testing [95], reuse [36] and validation [18] are incorporated.

In terms of tools that support the latest software design and maintenance activities are up-to-date to meet code comprehension necessities. In recent decades, Storey's [87] methodology investigates some of the cognitive approaches of program understanding. In particular, the author explores how the theories and technologies are related, and focuses on the research methodology used to build the hypotheses and analyze the tools. The theories and devices evaluated can be further distinguished by the individual features, program, and the purpose of the various understanding tasks. Code comprehension tools help developers to take advantage of the newly implemented software code [87].

2.2 Fields of Code Comprehension

In this section, we explore the different fields in code comprehension.

2.2.1 Code Readability

We perceive readability as a subjective evaluation that software developers have difficulty in understanding code. The connection between readability and comprehension is similar to the syntactic and the semantic evaluation; the syntactic dimension is readability, whereas the semantic measurement is comprehensibility [14]. Essentially, readability is a potential obstacle to understanding which the programmer feels the need to resolve before executing with a code. Subjective interpretation measures are challenging to accomplish and require human testing. They are also fundamentally variable with the help of large-scale surveys, and rigorous statistical evaluation of survey data is required for a meaningful analysis [58].

Recent studies have developed unique code readability methodologies that have an impact on software readability. Buse and Weimer [12] suggested a readability measure and implemented a readability tool, which tests the readability measure efficiently. They choose java code snippets and used human observers for making judgments about the readability of the code. The findings from the human observers were compared to a programming tool. The total precision of the tool was 80%. However, Posnett et al. [66] asserted that code readability is a qualitative attribute and that readability grades can not be achieved using an automated readability tool. They also stated that the readability score could be estimated depending on the size and the code complexity, using the data obtained in the source code. Elements such as appropriate comments improve software readability, and poorly defined variables reduce readability [56].

Cowan [89] proposed a novel method of improving the readability of code using SGML to integrate semántic and syntactic code through content and text-database visualization. The visualization of principles attempts to link the writer's mental representation to the software reader's mental image.

Wang et al. [96] designed a new and efficient automatic readability metric which calculates faster than human judgement. To analyze code snippets and assess the complexity based on certain functions such as keywords, summary, loop, lines, etc., the investigator collected many snippets from open source databases.

Relf [69] also conducted a study of the empirical influence of naming formats on and management of code base on a team of software engineers-novice and professionals. The study compared and compiled 19 guidelines for constructive naming formats and created a java compiler that can verify the names of the identifier. The use of relevant names has been explored by Butler et al. [15], and researchers have been trying to compare their findings using the tool they developed called FinBugs. While researching further connections with software quality, the researchers extended their work with java methods.

The scenario scan technique [57], [55] is used when a person searches for and renders brief fixations and longer saccades for a goal in the code. A task inspection is a voluntary act of the person; thus, the focus sight fits the "top-down" cycle of cognitive processing and can be described by direct visual function [55]. Ambient sight is cheaper and dictates parallel scanning, while sequential searches in viewing tasks need close attention [55]. A study by Bednarik et al. [7] showed that differences in the reading styles such as gazed locations, fixation duration, and switching eye movements were recorded in novice and expert programmers. Furthermore, Busjahn et al. [14] compared the reading patterns in Java code to natural languages. It has been proven that novices follow a linear approach when they read Java code, while professionals followed a non-linear path because reading skills increase with more experience.

2.2.2 Test code

In this section, we highlight the various advancements in test code research. Unit testing is a technique of developing software where the shortest testable components of a program, termed units, are individually examined [88]. There are several experiments on readability in the past, but the challenge of defining what a subjective interpretation is a big concern with readability studies [58]. Test cases are often used as regression tests to ensure that the features of the software are always working once it is developed. Unit testing allows you to check for bugs and ensures that these bugs wouldn't induce the program to break. Errors which a unit test reveals are easy to track down and comparatively easy to fix.

In the past decade, various research has been contributed in the software engineering field to understand how programmers read test code. A study by Scanniello et al. [75] performed an empirical Test-Driven Development (TDD) research with focus groups to gain perspectives into developer viewpoints using TDD. They found that it is hard to apply TDD without learning advanced unit testing methodologies. The significance of tests during software development to enhance the efficiency of software projects ensures corrective actions are taken and serves as a document, which has been highlighted numerously in various studies [67], [81].

Li et al. [43] designed an automated stereotype-based tagging tool to aid unit test cases and testing tools. They conducted the study with 46 students and 25 developers to validate the tool and its usefulness. In a study by Kamimura and Murphy [34] proposed generating human-oriented summaries to aid the comprehension of large test cases. They implemented their methodology on JFreeChart application and successfully highlighted statements which were important to the developers.

2.2.3 Task difficulty

Task difficulty is another important area of research. Gundel et al. [28] studies the topographical distribution of EEG when performing two tasks with different difficulty levels. The test showed reduced alpha band through when performing visual scanning and increase in the theta band in the left frontal electrodes related to the overall cognitive workload. The principal benefit of EEG is to specifically test functions inside the brain rather than indirectly measuring the blood flow or metabolisms such as fMRI and fNIRS. Also, EEG-based approaches ideally suited towards cognitive processes and task difficulties due to the strongly sensitive EEG.

2.2.4 Cognitive load

A collection of bio-metric detectors such as eye trackers for evaluating the size of the pupil, EEG to assess cognitive function, skin-related electro-dermal detectors and cardiac sensors have been employed in software engineering. Crk and Kluthe [19] determined the Alpha and Theta fluctuations ERD (event-related desynchronization) seen between task test time and the relaxation time. ERD activity is measured to be intrinsic workload, and the cognitive load rises whenever tackling problems with high complexity. Siegmund et al. [84] used fMRI to analyze the brain regions stimulated during code comprehension of code snippets. Some researchers utilized eye tracker measurements to measure cognitive load at various task levels and were able to demonstrate that cognitive load decreased near the high-level task limit. Researchers have shown that increasingly demanding tasks require longer computation time, generate high psychological levels of cognitive load and induce enhanced pupil dilation reaction at significant tasks [33].

2.2.5 Attributes of the programmer

Selecting the sample group to experiment is important to get credible results. The experiments by Pennington [63] study showed that the chosen language has a significant impact on the processes of comprehending the code by programmers.

We know the importance of using electronics and software in our everyday life; hence software development is no longer a niche practice. A study by Storey [85] reviewed work from biologists, astronomers and medical researchers who use and implement specialized software without any formal training in computer science. As a consequence, methods must be used to enhance the understanding of non-experts and end-users. There is a lot of research in this field (PPIG community-www.ppig.org) to explore how code comprehension can be enhanced by supporting resources for excel and other end-user platforms. Complex software development knowledge in fields such as advanced visual interface, machine learning and hardware programming is only known to only sophisticated developers. Hence a niche group of advanced users are selected for these studies who have the required knowledge.

Fields of Code Comprehension				
Methodology	**Tools**	**Authors**	**Participants**	**Study**
Code readability	Readability tool	Buse and Weimer [12]	Experts & Novices	human observers were used to make judgments about the readability of the code. The findings from the human observers were compared to a programming tool.
	Survey	Posnett et al. [66]	Novices	asserted that code readability is a qualitative attribute and that readability grades can not be achieved using an automated readability tool.
	Code Readability Tool (CRT)	Cowan et al. [89]	Experts	proposed a novel method of improving the readability of code using SGML to integrate semántic and syntactic code through content and text-database visualization.
	Automated readability tool	Wang et al. [96]	Experts	the authors designed a new and efficient automatic readability metric which calculates faster than human judgement.
	Java compiler tool	Relf [69]	Experts & Novices	empirical influence of naming formats on and management of code base on a team of software engineers
	Scenario scan technique	Bednarik et al. [7]	Experts & Novices	showed that differences in the reading styles such as gazed locations, fixation duration, switching eye movements
	Scenario scan technique	Teresa Busjahn et al. [14]	Experts & Novices	compared the reading patterns in Java code to natural languages. It has been proven that novices follow a linear approach when they read Java code, while experts followed a non-linear path because reading skills increase with more experience.

Table 2.1 continued from previous page

Methodology	**Tools**	**Authors**	**Participants**	**Study**
Test Code	Thematic analysis template (TAT)	Scanniello et al. [75]	Novices	A empirical Test-Driven Development (TDD) research was conducted with focus groups to gain perspectives into developer viewpoints using TDD.
	Automated stereotype-based tagging tool	Li et al. [43]	Experts & Novices	designed a automated stereotype-based tagging tool to aid unit test cases and testing tools.
	Human-oriented summaries tool	Kamimura and Murphy [34]	Experts	proposed generating human-oriented summaries to aid the comprehension of large test cases.
Task difficulty	EEG	Gundel et al. [28]	Experts & Novices	studies the topographical distribution of EEG when performing two tasks with different difficulty levels.
Cognitive load	EEG	Crk and Kluthe [19]	Experts & Novices	Studied the Alpha and Theta fluctuations ERD (event-related desynchronization) seen between task test time and the relaxation time.
	fMRI	Siegmund et al. [84]	Experts & Novices	analyze the brain regions stimulated during code comprehension of code snippets
Attributes of the programmer	Target Recognition Tests	Pennington [63]	Experts	showed that the chosen language has a significant impact on the processes of comprehending the code by programmers.
Behavioral/ psychological studies	IDE	Müller and Fritz [51]	Experts & Novices	studied developers' positive and negative emotions as indicators of performance during code comprehension.
	Model-driven Testing tool	Wrobel et al. [98]	Experts & Novices	performed research to evaluate how emotions influence the efficiency of programmers at work.

Table 2.1: Summary of Fields of Code Comprehension

2.2.6 Behavioral/psychological studies

Study into emotions in software engineering has a long history. To evaluate emotions, a wide variety of psychological research has studied using biometric sensors to assess emotion-caused physiological changes. Frustration is among the most popular subjects examined through the use of sensing devices. Müller and Fritz [51] utilized developers' positive and negative emotions as indicators of performance during code comprehension.

Wrobel et al. [98] also performed research to evaluate how emotions influence the efficiency of programmers at work. They discovered that anger is the most common unpleasant feeling that also disrupts performance and that negative moods could also have a positive effect on performance for some individuals.

2.3 Advancements in Research methods

The research scope in this study consists of several important and diverse topics, resulting from changes throughout the research frameworks and technologies in the last few decades. In table 2.1 we present an investigation of the different methodologies and developments in the code comprehension.

2.3.1 Traditional methodology

The previous studies [49], [27], [76] focuses on using think-aloud protocols or questionnaire approach to analyzes reading patterns, and these do not accurately measure the visual patterns employed by experts and novices as they are subject to bias and lessen the comprehension capacity [13].

Research by Crosby and Stelovsky [20] attempted to explore similarities between reading non-procedural text and software code, as well as the effect of programming skills on comprehension. Components used in the analysis were a Pascal code with the binary search application, which includes inline comments and visualizations of the control flow of the program. They examined ten novices and nine professionals. The findings indicate a disparity between written texts: a higher amount of fixations and multiple regression induced while comprehending source code. Novices were more interested in comments and descriptions, while professionals looked at control flow statements. The findings indicate that important information about the program is found in comments and complex statements. Think aloud (CTA) is a tool often used in the code comprehension experiments. This procedure shows the dynamics of cognitive understanding but is difficult for the subjects to perform this task and affects their actions since it imposes added cognitive load. Another issue with methods such as CTA and self-reporting techniques is that subjects can have a different approach to the same task; hence it can be biased. In recent decades there have been advances in the code comprehension studies which used innovative and sophisticate tools such as bio-metrics devices and machine learning to improve their findings. We discuss them in the following sections.

2.3.2 Bio-metric devices

In this section, we look into various bio-metric devices which have been used to study the programmer's cognitive activity such as fMRI, fNIRS, EEG, eye-tracking and physiological devices. Currently, commercial based eye trackers are widely used in experimental research studies as they are cheap and readily available. In the field of software engineering, most studies are focused on cognitive load, program comprehension and emotional/psychological studies.

Studies aimed at interpreting natural language processing using eye movement offer a more in-depth understanding of the cognitive processes. Yet there are just a handful works which examined eye movements while comprehending code. The outcomes of the few conducted studies in code comprehension indicate that eye tracking is a successful method of analyzing cognitive processes, particularly those associated with think-aloud techniques.

- **Eye-tracking** Eye-tracking is a useful way to examine the mechanism of program comprehension. Assessment of the eye motions during the interpretation of the code helps scientists to determine the visual load. According to the viewing location of the visible focus item, measuring techniques are often used in the study of the area of the focused fixation, length of focus and search trend.

  A study by Rodeghero et al. [72] developed an experiment for professional java programmers to read code and summarize them in English. This study helped to build better summarizing systems in the future. [7], [80], [5], [8] also studied the effect of eye tracking on code comprehension by recording fixation duration, location and switching between eye movements. This data helped to categorize the most effective way to read source code.

  Eye-tracking is also seen as an effective tool in understanding marketing trends [97]. It was reported that heightened gaze leads to high cognitive load on the brain. Clinical studies on patients with Schizophrenia [30], [31] has provided an effective tool to understand how eye gaze relates to mental disorders. There have also been various studies combining eye tracking with fMRI, fNIRS or EEGs [62], [61], [60] to provide more novel methodology to understand a programmer's mind.

  Bednarik and Tukiainen [6] studied where less complicated tools such as the restricted focus viewer (RFV) provide equivalent data analysis. An RFV is a spatial cognition control device which only enables the visualization of a specific area in focus and blurs the remainder of the image. The minimal exposure tends to affect cognitive behaviour. Current research findings suggest correlations between the eye movements and patterns of cognitive load. But these are dynamic, and it's not easy to match. Cognitive load is generally correlated with fixation length, so complicated texts cause longer fixations, brief saccades and regular regressions [16].

- **fMRI** A variety of research has utilized brain activity control methodologies to investigate comprehension of code. Researchers use spatially sensitive devices, like fMRI scanners, to classify an area of the brain. The minimally invasive method of monitoring blood oxygen levels which increase due to concentrated brain activity is functional magnetic resonance imaging (fMRI). [94], [90], [68], [44], [91] studied the effect on the cognitive load during visualization tasks. They demonstrated experimental studies using fMRI to observe working memory load and visual load while performing visual tasks.

Siegmun et al. [82] conducted a study using a guided test in the fMRI scanner, 17 individuals reading and understanding brief source code snippets with syntax faults were recorded. Across five brain areas, a consistent and distinctive activation trend has been identified linked to memory, focus and linguistic production [83].

[60], [62], [62], [84] also studied program comprehension but they employed both fMRI and eye-tracking devices.

- **fNIRS** fNIRS is a wearable brain-imaging methodology that detects the hemodynamic response from the exterior of the skull to a specific distance. Due to its high temporal resolution and minimal user constraint, NIRS is a particularly viable tool for code comprehension experiments. Nakagawa et al. [53] observed increased blood flow when reading code and proposed the use of fNIRS to assess cognitive workload during program comprehension.

  Ikutani et al. [32] also implemented fNIRS to prove that high brain activity was in the frontal area, which related workload to short-term memory caused by variables in code. It also showed no change when arithmetic tasks were implemented. Pike et al. [64] performed four think-aloud protocols using fNIRS to study the effect of verbalization on the mental workload of the brain. Studies [93], [59], [29] proved that measuring the hemodynamic responses from fNIRS can be used to measure mental workload while doing natural tasks such as driving in a simulator and hand gestures.

- **EEG** EEG was commonly used to measure and determine cognitive loads which decreases the total capacity of the brain and makes it much harder for the activity to be done, leading to reduced flexibility of the working memory and heightened mental stress. [4], [101]. [50], [102], [17] have performed experimental tests to detect the cognitive load. Cognitive workload interventions using EEGs have not only been implemented but have also been successful in many fields, like those of adaptive learning, design performance, gameplay videos, etc.

  Lee et al. [39] used a portable EEG sensor to investigate the difference in cognitive load between novices and professionals. The primary focus in these studies was collecting metrics correlating with the efficiency of computer programmers. Zuger and Fritz [104] employed interruptibility, whereas Muller and Fritz [52] examined the software developers' positive and negative emotions as metrics for performance. The data were analyzed by various bio-metric devices and implemented supervised learning to differentiate the levels of such cognitive processes.

2.3.3 Machine learning

Machine learning is becoming popular in recent years due to its ability to learn from complex data automatically. The previous studies in test code comprehension involved automating unit test generation, instead of understanding the cognitive process. We try to bridge this knowledge gap by classifying developers expertise using linear mixed regression model, which is a powerful statistical tool used in most social science experiments. A recent study by Lee et al. [40] proved that we could use machine learning models to predict the expertise during source code comprehension accurately. They used bio-metric devices such as EEG and eye-tracking and compared the performance of the machine learning models in these bio-metric devices. Another study by Dreiseitl and Ohno-Machado [24] highlighted the popularity of logistic regression models in the field of biomedical devices, due to its advantages in interpretability of model variables and ease of use. These studies [37], [103], [42] also employ machine learning models to predict program comprehension among participants.

2.3.4 Information Retrieval

Software comprehension is a required prerequisite before making any improvements to the system. The developer needs to collect information across the source code of the software components and then understandably display the obtained data. This task takes time and is susceptible to error, mainly if it is a vast and complex system. Hence much work on how to reduce the time and resources required to understand a program was conducted. Maletic and Marcus [46] used semantic and structural information to support software maintenance tasks. Another study by Denys et al. [65] combines the concept analysis (FCA) and latent semantic indexing (LSI).

Advancements in Research methods				
Methodology	**Tools**	**Authors**	**Participants**	**Study**
Traditional	Think oud	Crosby and Stelov [20]	Experts & Novices	explored similarities between reading non-procedural text and software code, as well as the effect of programming skills on comprehension
Bio-metric	Eye tracker	Bednarik and Tukiain [6]	Experts	correlations between the eye movements and patterns of cognitive load.
	EEG	Lee et al [39]	Experts & Novices	differences in cognitive load
	fMRI	Zuger and Fritz [104]	Experts	interruptibility during software programming
	Think aloud	Muller and Fritz[52]	Experts	examined the software developers'positive and negative emotions as metrics for performance
	Restricted focus viewer	Bednarik and Tukiainen [6]	Experts & Novices	correlations between the eye movements and patterns of cognitive load.
Machine learning	Eye tracking	Lee et al [40]	Experts	machine learning models to accurately predict the expertise during source code comprehension.
	EEG	Dreiseitl and Ohno-Machando [24]	Experts	highlighted the popularity of logistic regression models in the field of bio-medical devices, due to its advantages in interpretability of model variables
Information Retrieval	Integrated development environment	Maletic and Marcus[46]	Experts	semantic and structural information to support software maintenance tasks.
		Denys et al.[65]	Experts	combines concept analysis (FCA) and latent semantic indexing (LSI) to implement concept location.

Table 2.2: Summary of Advancements in Research methods

Chapter 3

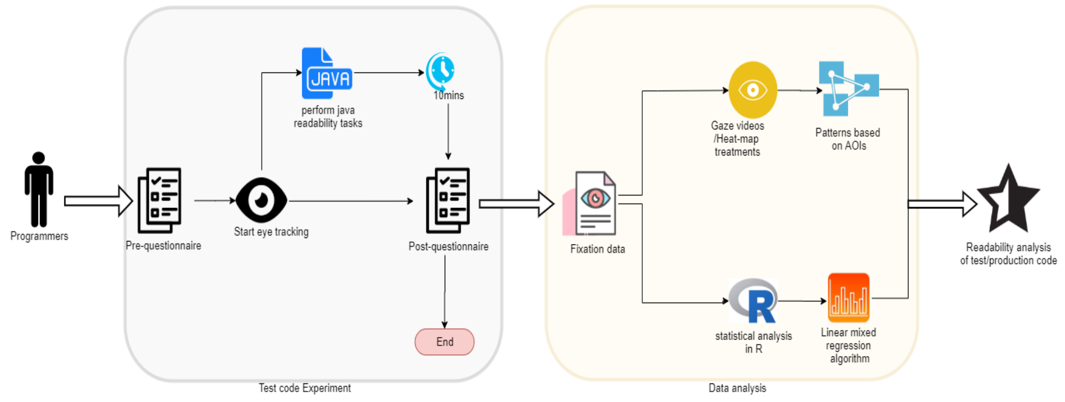
Methodology

The goal of this thesis is to understand the reading patterns of software developers when reading the test code. Thus, we formulated the following research questions.

RQ1: How do developers read test code?

RQ2: What are the differences between novices and professionals when reading test code?

In order to address these research questions, we implemented an experiment shown in figure 3.1. In a nutshell, we propose a controlled experiment where participants perform test code comprehension tasks. We monitor the participants' actions through a webcam-based eye tracking solution. Using the fixation, we first visualize the data using heat-maps and make qualitative analysis to map each participant's treatment to areas of interest. Next, we perform a statistical analysis using linear regression to measure the differences between the behaviour of novices and expert developers.



Programmers
Pre-questionnaire
perform java readability tasks
10mins
Start eye tracking
Post-questionnaire
End
Test code Experiment
Fixation data
Gaze videos /Heat-map treatments
Patterns based on AOIs
statistical analysis in R
Linear mixed regression algorithm
Data analysis
Readability analysis of test/production code

Figure 3.1: Our research methodology

3.1 Experimental Setting

In this section, we explain the various design features taken before the implementation of the experiment.

We first conducted a pre-questionnaire where we collected the demographics data from the participants to understand their level of expertise (novice/expert) in java programming. We provided two java code comprehension tasks to determine the visual load of the participants. We experimented using a webcam by Real-eye software. Real-eye is approximately 100px precise with an estimated visual angle deviation of 4.17 deg with video recording. To conduct this experiment, we use everyday web-cameras available on a computer or laptop. Web camera eye tracking is a revolutionary way to determine the gaze of an individual which is achieved in real-time in an internet browser. Hence the identity of the participant remains safe, and only the gaze points are stored in the form of the simple texts of ”Time-stamp: 08, GazePoint-X: 500, GazePoint-Y: 345” [1]. In Figure 3.1, we show the experimental process in the test code experiment grid:

1. Participants complete a pre-questionnaire survey, which is focused on demographic information such as experience in Java, Junit, gender and occupation.
2. Calibrate the eye tracker with the required guidelines.
3. Comprehend and explain the goal of two test suites, 10 minutes each.
4. Fill in the post-questionnaire after the experiment, regarding difficulty, the time required, and understandably of code.

In figure 3.1, we show the overall experiment design. First, the participants fill the post-questionnaire, which contains questions regarding the background of the participants. Once completed, the participants are redirected to the eye-tracking application to begin the tests. Each test lasts 10 minutes, and the participants can finish on click to exit answer the summary questions. Once completed, the participants fill a general overall post-questionnaire and end the experiment. We explain the questionnaires and tasks in more depth below.

In Appendix A.2, we display the interface of the experiment. The platform for the experimentation is available in shareable links. The interface first calibrates the eye movements and then starts the experiment. We showed the programming tasks in the form of images and asked the participants to do the summary task, later answer the questions related to these tasks. Participants are automatically redirected to the external survey once the task was complete.

3.2 Pre-Questionnaire

We designed the questionnaire to understand the participant’s level of expertise and background prior to the experiments. We collect the subject’s background data before beginning the experiment [100]. This questionnaire is required to evaluate the expertise and coding skills of participants with Java. We ask general questions about the participant’s history and skill level in Java, given that existing studies have proven that use features such as gender, age and expertise influence on the characteristics of individuals [25], [54]. Lime-survey was used to design the questionnaires. The participants were given a unique token to start the survey and are allowed to edit their answers anytime.

3.3 Tasks

We asked the participants to comprehend two codes snippets containing both production & test code and write the summary of the codes given. We selected four code out of which the participants selected two codes randomly. We asked the participants to explain the implementation of the code in detail which were shown are shown side-by-side to the participants. The font sizes were maintained in the same style all tasks to aid eye trackers to differentiate from each code line. Participants did not communicate using the cursor or the keyboard. Hence we obtained all the answers after the test was completed in the survey. We asked the following tasks for each test code. The tests contain two questions regarding test code and production code.

- **Summary production task:** What does the given class in the production code do? And can you briefly explain its implementation?
- **Summary test task:** What does the given class in the test code do? and can you briefly explain its implementation?

The goal of the tasks is to understand if the participants understand the basic overview of the program and its test cases. Hence the answers provided were only used to validate if the participants performed the comprehension of tasks. The participants who did not meet the standards were excluded from the experiment.

3.4 Code selection

We selected four independent code snippets with the same level of difficulty. The code snippets consisted of basic concepts of programming, such as nested loops, and basic input/output commands. All the programs are selected so that participants can finish each task within 10 minutes to prevent fatigue. We measured the programming tasks by measuring the time spent on each task received from recording the screen and by solving the tasks using Realeye software. We applied the following selection criteria to choose the code snippets for the tasks of this experiment:

1. The code should resemble ”real-world” projects used by software developers.
2. The codes must be comprehensible in a discrete and independent form
3. The code should be complicated and long enough to initiate cognitive thinking process to detect reading patterns.

Hence, four codes were selected with an equal level of difficulty, each code performing different types of array/string manipulations. The purpose of designing the tests was to ensure that everyone could comprehend them with only basic knowledge of Java and Junit(so that no real domain knowledge is required), they all have similar complexity in terms of LOC and branch statements. Hence we wrote test cases manually[1]. All the classes have well defined descriptive names to allow participants to understand the overall purpose of the code. We eliminated the risk of bias by randomly picking only 2 out of 4 codes for the participants to solve.

Code	**Test LOC**	**Production LOC**	**Test class**	**Assert**
Unique	79	25	10	10
Reverse	81	20	3	11
LastIndexOfTest	78	28	6	12
ContainsAny	61	26	4	12

Table 3.1: Summary of production and test codes

Table 3.1 summarises the lines of code for test and production codes, with number of test cases and assert statements. Below we explain the implementation of codes.

- The code Unique finds the input value of type double in the given treeset. Returns an array of unique values after sorting them in decreasing order. Empty arrays are allowed, but null arrays result in Null Pointer Exception.
- The code LastIndexOfTest finds the last index of the given value in the array starting at the given index. A startIndex which is larger than the length of the array will search from the end of the array.
- ContainsAny returns true if at least one element is in both collections of the array. In other words, this method returns true if the intersection of coll1 and coll2 is not empty.
- The code Reverse, reverses elements in a given array, with the arguments startIndexInclusive and endIndexExclusive. It returns the value after sorting in decreasing order. The starting index takes the Undervalue and is promoted to 0, and the overvalue results in no change.

3.5 Areas of interest

The area of interest, also called the AOI, is a method for choosing sections of the code shown and for collecting metrics for certain areas in particular. To better analyze the visual data, we broke the code in *"blocks"*, whose components of which fit logically into a unit

[1]The original test suite of these snippets was in a single method, which would hinder our ability to see how developers behave in test suites with multiple test methods

which may be of importance to the participants. We also sub-categorize AOI based on programming statements; they are roughly identical to sentences in natural languages. In this study, we have the following AOIs.

- *Control flow statements*: govern the sequence in which the code is executed.
- *Assert statements*: This is used for testing types, argument values and the method output.
- *Declaration statements*: are used to initialize/declare a value.

Based on these elements, we evaluate the total code coverage area differences in novices and professionals.

3.6 Variables

In this section, we discuss the dependent and independent variables that we have implemented in our research. These metrics are were derived manually using gaze videos and automatically using fixation data.

3.6.1 Independent variables

Our study includes two independent variables: if the participant is novice or expert programmer, and if they are reading a test code or production code.

1. **Expertise:** The expertise of the programmer has played a crucial role in the analysis of code comprehension in the field of software engineering in the previous years [9], [14]. The programming expertise of our participants ranged from bachelor's studies in Java to graduate school students with a limited programming background to professional programmers with a strong background in Java. The programming experience was diverse. Appendix B.2 shows the overall participants data.
2. **Total time:** In this study we consider total time as a control feature used in the linear mixed model (someone that spends more time in the code will spend more time in comprehending test/production code).

3.6.2 Dependent variables

Our dependent variables are all based on the data we collect from the eye-tracking. Table 3.2 shows the dependent variables implemented in this experiment. We explain them in more detail below.

1. **Fixation points**: We used various fixation features from the eye tracker and obtained the fixation times between saccades. During such focus gaps, the eye may search multiple lines of code, and in each fixation, the participant will move their attention from one topic to the next.

Feature	Definition
	Fixation data based on AOIs
Fixation points	The number of points of fixation on the code
Avg. Fixation time	The duration of fixation in seconds
TTFF(Time To First Fixation)	Time taken to look at the different AOIs, provides priority levels of AOIs.
Time spent	Total time spent on fixation in seconds
Total time	Total gaze time spent from the beginning of the experiment in seconds
Revisits	Number of times the participant returned their gaze to a particular AOI.
	Manual gaze metrics
Source time	The total source time taken in seconds
Test time	The total test time taken in seconds
source switch count	The number of times participants switch to source code.
test switch count	The number of times participants switch to test code.
assert time taken	The total time taken in seconds to read assert statements.
test read	The number of tests read by the participants.
Total switch	The total number of switches between test and source code.

Table 3.2: Fixation Features

2. **Avg. Fixation time**: In this metric, we calculate the average fixation time taken by participants. This data provide the results in seconds which is helpful to analyze the time taken by participants when they fixate on specific LOC.

3. **TTFF(Time To First Fixation)**: This metric gives the time at which the participants first fixated on a LOC. This data is helpful to analyze when the participants shift between test and production code.

4. **Time spent**: This metric provides the total fixation time taken by the participants to comprehend the test and production code completely.

5. **Total time**: This metric provides the total gaze time taken to comprehend test and production code in seconds. Gaze analysis provides the data of eye movements throughout the code reading and comprehension phase.

6. **Revisits**: This metric provides the number of times the participant returned their gaze to particular AOI. Using the revisit data, we can analyze which part of the code was interesting and difficult to comprehend to the participants.

The below variables were selected by manually analyzing gaze videos recorded during the experiment through the eye tracking device.

1. **Source time**: The total source time taken by participants to comprehend source code in seconds.

2. **Test time**: The total test time taken by participants to comprehend source code in seconds.

3. **Source switch count**: The number of times participants switch to source code.

4. **Test switch count**: The number of times participants switch to test code.

5. **Assert time taken**: The total time taken in seconds to read assert statements.

6. **Test read**: The number of tests read by the participants.

7. **Total switch**: The total number of switches between test and source code.

3.7 Post-Questionnaire

At the end of the experiment, we asked the open-ended questionnaire that gathers data regarding the challenges faced by participants during code comprehension. We ask questions regarding difficulty, the time required, and understandably of code. We also ask questions about how the programmers solved the comprehension tasks and their techniques used to solve them. We based the questionnaire on various similar studies related to software comprehension tasks [22], [99].

This questionnaire is an important part of the study as it helps gain insight on how programmer's read test code. We have asked specific questions such as if they first focused on test or production code, did they read line-by-line and if production code aided to comprehend test code. These questions help us see distant patterns based on the expertise of the participants.

3.8 Data Analysis

In this section, we explain the various steps involved in the analysis of different reading patterns among novices and professionals.

3.8.1 Quantitative analysis

We examined the relationship between dependent variables and independent variables using a linear mixed regression model. We choose this model because we deal with hierarchical data. We investigate the best approach to read test codes to be able to solve comprehension tasks efficiently. We first analyze the gaze videos and heat maps treatments using the eye tracker and investigate the reading strategies of novices and professionals. We select the manually dependent metrics mentioned in table 3.2. and feed the data to out linear regression model to find the significant differences. Finally, we implemented various fixations features from the eye tracker software to the linear mixed regression model to observe if there are any significant differences. We examined the developer's efficiency using visual data such as fixation points, revisits, average fixation, and total time is taken to solve the task. We evaluate whether there are any correlations on the outcomes between the skills of the participants: novice and expert using box plots to visualize our results [78].

3.8.2 Qualitative data analysis

In our study, we performed a qualitative analysis using open-ended questionnaires to analyze the readability patterns. These post-questionnaires aims to understand the participant's experience and difficulties during the experiment. We ask questions regarding the complexity of code, tasks asked, the time required, etc. We used the answers provided by the participants to correlate the quantitative data to validate our findings. We analyzed different gaze patterns for each participant and compared linear reading to that of natural language(left-to-right and top-to-bottom).

3.9 Pilot study

We performed a pilot to evaluate the study in which four participates took part voluntarily. All the participants were familiar with Java testing in Junit, and the results of the pilot study were analyzed. In a preliminary analysis, two novices and two professionals evaluated initial test code tasks. We assessed their programming skills in a questionnaire. They then proceeded to comprehend the java test code and answered post-questionnaires. We used the participant's observations and feedback in defining the final experimental code.

There were multiple updates made using feedback from the pilot study.

- Increase in the code size.
- Keep the same level of difficulty for both codes to remove bias.
- Add extra buttons and instructions to navigate the eye-tracking tool.
- Add a darker background colour to reduce the reflection on the lens and strain on eyes.
- People with glasses were instructed not to wear to reduce calibration issues during eye-tracking.

Hence the final study had significant improvements, which increased the overall participation.

3.10 Participants

In this study participants from two groups were selected, novices and professionals. We termed students who previously worked with the language as professionals and bachelor students who are still learning the language in their courses as a novice. Participation has been made known to be voluntary and does not engage in the curriculum and therefore, does not have a favourable or detrimental effect on student's grades. We recruited expert programmers by advertising on various websites to participate in the study and were 5-10 euro gift cards as compensation for their time. Our study consisted of programmer's from both software testing and software developer fields. Appendix B.2 provides an overview of all the participant demographics in this study.

Background characteristics of the participants	
Factor	**Percentage**
Gender	
Female	34.48%
Male	65.52%
Age group	
<22	13.79%
22-27	37.93%
28-35	27.59%
>35	13.79%
Experience	
Student	51.72%
Software developer	41.38%
Tester	6.90%
Years of experience	
0-2	27.59%
2-4	37.93%
4-8	17.24%
>8	3.45%

Table 3.3: Background characteristics of the participants

The experiment was conducted for two months and has 29 participants in total, of which 15 are novices, and 14 are professionals. The main distinction between novices and professionals is that professionals have a minimum industry experience of 2 years in software development and testing. In contrast, novices are bachelor/master students without proper work experience. In table 3.3, we summarise the background characteristics of participants.

Furthermore, among the professionals, 14.29% have experience in testing with two years of experience. The participants were 34.48% females, of which 60% are students, and 40% are developers, the male participants (65.52%) consisted of 42.11% students and 57.89% professionals. We show the different level of expertise for novices and professionals in Figure 3.2

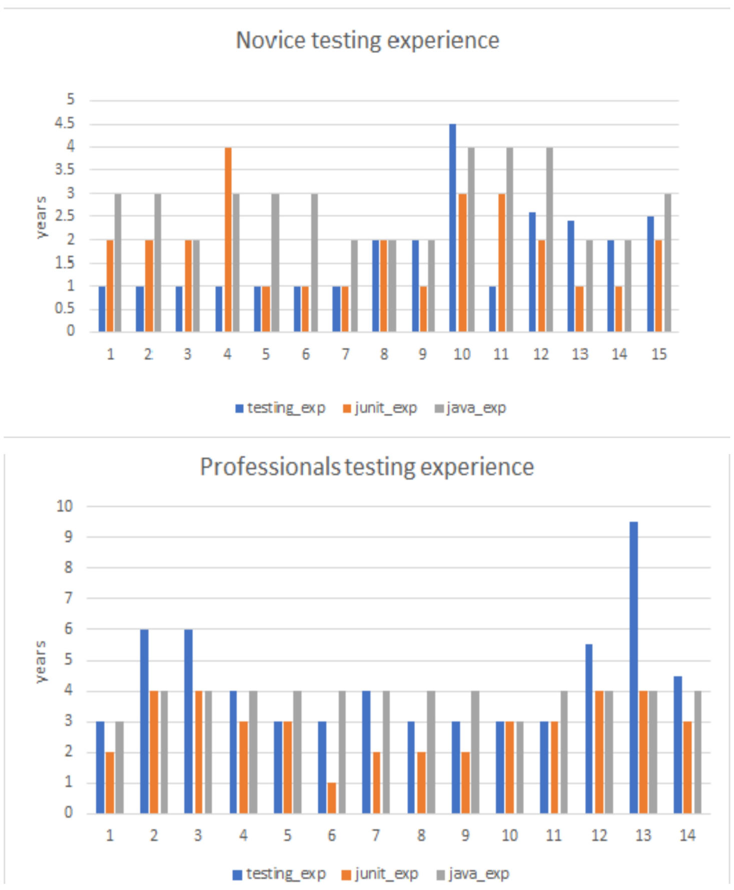


Figure 3.2: Level of expertise for novices and professionals

Chapter 4

Results and Observations

In this section we look deeper into the various distributions of dependent variables based on programming expertise. We analyze the differences in reading patterns between novices and professionals using gaze videos and heat-maps to visualize how programmer's read test code.

4.1 RQ1: How do developers read test code?

In this section we report the different reading patterns followed by novices and professionals. We use different assessment strategies such as heat-maps, open-ended questions and gaze video. Here we answer our first research question (RQ1).

We analyzed gaze videos manually which were recorded during the experiment by examining the *(i)* time taken to finish comprehension of production and test code, *(ii)* the number switches between test and source, *(iii)* time taken to read assert statements, and *(iv)* the number of tests read. To have a more accurate overview of tests read, we combined the results with heat maps.We summarized our results in Table 4.1.

- **First view:** We find that both novices and professionals on average, first fixate on the production code at 1.37 secs and 1.59 secs respectively, which is within 2 secs from the start of the experiment. According to the answers provided, all the participants first focused on the production code to understand the basic functionality/implementation of the code, next they concentrate on the test code and switch between production code and test code. Participant P10 stated *"I first read all the production code to understand what it did, and then read the test code to see how it tested the functionality."* Novices (P13, P14, P6, P28) sometimes found the production code challenging to comprehend and hence switched to test code to understand the source code. However, professional P13 choose to read the production code first because the test code in the industry is not explicit enough and has poor naming conventions and an unclear purpose. This was highlighted by P13: *"It might happen, in industry, that the test code is not explicit enough (bad naming, not clear enough what it tests), so I'm not used to rely on test code. In this exercise, the tests were well written so it could also be used as a starting point."*

- **Production time:** Novices spent on average spent 58 secs reading production code and whereas professionals spent only 47 secs of their time reading production code, which shows that novices spent 23.4% more time reading production code than professionals. This is because participants read the production code line-by-line and then switched to test code. Participant P10 stated *"I first read all the production code to understand what it did, and then read the test code to see how it tested the functionality."* Furthermore, 4 out of 15 novices also said that the production code was difficult to comprehend.

- **Test time:** We observe that the professionals spent 63.6% more time reading test code than novices, which corresponds the answers providing during the open-ended questions, where professional P2 stated: *"Taking look at the test case name first to get a rough idea what it is testing. Then looks at the implementation of the test case while switching back and forth from the production code to understand what the test is doing."*. We found that professionals spend more time reading test code because they validate the production code. Participant P1 highlighted that reading test cases also allows them to see the corner cases and boundaries of each parameter. On the other hand, novices spent more time reading production code and switched to test code after reading the production code line-by-line. In particular, a group of novices (P20, P29, P14, P26) read the test cases in a linear manner (line-by-line) and top-to-bottom, this statement was exemplified by P26 *"Just from top to bottom. Line by line"*.

- **Assertions:** Novices spent on average 70 secs of their time reading assert statements, in contrast, professionals spent only an average of 45 secs, which is 35.7% lesser than novices. We also observe that professionals had 70.29% higher assert fixation counts than novices. Using post-questionnaires we find that 9 out of 29 survey participant commented they read the method names first and used assert statements to verify the arguments. Novice P8 focused more on the inputs of the test code: *"Read test code objects to verify the functionality of source. Focus more on inputs in the test"*. Furthermore, professional P22 stated that *"I mostly focus on the overall test methods and glance through the objects in it."*. Additionally, expert P1 also provided a detailed procedure they followed while reading the test code. Participant P1 first read the inputs and outputs and then validated the methods and assertions. In the words of P1: *"For me, I follow these rules to create my tests: given (input), when (invoked method) and then (assertions)."*

- **Source switch:** We also examined the switches between test and production code during the experiment and found that professionals had 16.4% higher source switch count than novices. This is due to linear reading methodology followed by novices, whereas professionals read the code to understand the control flow. 4 out of 14 of the professionals (P16, P17, P7, P2) highlighted that they switched multiple times between production and test code. P2 stated: *"Taking look at the test case name first to get a rough idea what it is testing. Then looks at the implementation of the test case while switching back and forth from the production code to understand what the* *test is doing."* Furthermore, professional P12 said due to exact tests provided with descriptive test method names and assert statements in this experiment, it motivated P12 to switch between to test code more often to verify the production code.

- **Test revisit:** We found that professionals revisited the test code 59.14% more than novices. Looking closely, we found novices prefer reading test code when the production is complicated, in the words of P28: *"Yes, especially in the first example (replace). I trusted in the test and I ignore some parts of the implementation, that I couldn't completely understand."*. Novice P26 also highlighted that due to multiple input samples and descriptive names provided in the tests, helped P26 to switch to test code. Almost all the participants expect (P13, P1, P14) answered that test code aided them to understand the production code completely as they found reading production code rather difficult at times.

Features	**Novice(avg)**	**Professionals(avg)**
Gaze data analysis		
Test time	55 secs	90 secs
Production time	58 secs	47 secs
Assertions time	70 secs	45 secs
source switches count	6.7	7.8
All test read	82.76%	13.79%
Linear regression analysis		
source_time_fix_spen	36.3 secs	28.49 secs
source_Total_time	79.57 secs	50.44 secs
source_TTFF*	1.37 secs	1.59 secs
test_fixation	248.6 fixations	157.14 fixations
test_TTFF*	44.23 secs	27.07secs
test_Total	174.05 secs	144.05secs
test_revisits_count	21.85 counts	13.73 counts
assertions	153.86 fixations	90.35 fixations

Table 4.1: Summary of finding using fixations, gaze data, and heat maps. *TTFF=Time to first fixation.

- **Fixation counts:** We observe that in novices, there is a higher fixation count than professionals for test code. This is because professionals switched between test and production code but did not fixate (comprehend) the test code line-by-line. Hence novices spent more time fixating on production code than test code. Novice P10 commented: *"I read line by line and then check the source code with input and outputs ."* This confirm the previous studies [9] [13] conducted in this field, where novices show higher fixation count than professionals.

- **Tests read** we observed that novices had higher test code coverage compared to professionals in both production and test code. Professionals read (comprehended) only

13.79% of the test cases, whereas novices read (line-by-line) 82.76% of the test cases. We analyzed the number of test read using heat maps, and the results indicate that 13 out of 15 novices fixate on the whole test code; however, only 2 out of 14 professionals fixated on the entire test code. The most preferred reading patterns (P1, P2, P5-P7, P13, P9-P11) to comprehend test is by first reading the method names and read the verifying the assert statements using input and output arguments which was reported by 9 out of 29 survey participants. Expert P10 exemplified *"The title of the test method already gives me an indication of what the test is for. Then I gloss over the input given and the expected outcome."*

Result RQ1: Our results show that *(i)* all programmers first comprehended production code and then switched between tests and production, *(ii)* novices had higher fixations reading test code and assert statements and also took longer time to comprehend them, *(iii)* Professionals revisited the test code more than novices in-order to verify the assert statements wheras novices read the test code line-by-line and switched to test code when they found production code difficult to understand, and *(iv)* professionals had significantly lesser test code coverage than novices.

4.2 RQ2: What are the differences between novices and professionals when reading test code?

In this section, we report on the various dependent variables using the linear mixed model. We have also verified our model assumptions for linearity using Q-Q plot and residuals vs fitted plot in Appendix E. We provide the formula used in this model below:

$$lmer(test\_Total \approx total\_time + professional + (1|task\_id) + (1|participant\_id) + (1+task\_id|order))$$

Here we choose the total time (time taken to complete production code and test code) and professional (novice/professionals) as the independent variable in order to predict the dependent variables in column one(features). We have reported the results of LMM in table 4.2 for general fixation features in the form of the p-values of all fixed effect per dependent variable. When the result of the p-value is lower than ($\alpha \leq 0.05$) then it is significant with symbol marked by "*". Statistical significant value shows that changes in independent variables lead to shifts in the dependent variable, where the independent variables are the total time of test code/production and expertise of the participants[2].

4.2.1 Model variables analysis

We visualize the distribution of various dependent variables using box plots. We use box plots in Figure 4.1 we to understand the relationship between continuous data and categorical data between professionals for significant dependent variables.

- **source_time_fix_spent:** The linear mixed model also indicates that participants had a significant difference (p-value = 0.0175*) where novices had higher fixation time on production code than professionals. The total_time variable is also increasingly affected by the total time taken to read production code by 0.039±0.016. This finding suggests that professionals can stay focused on the important elements of the production code than novices, which is consistent with previous expertise research.

- **source_Total_time:** The regression model confirms significance for the total time taken on production code (p-value = 0.000238*) and shows that the total time taken to read production code is higher for novices. The source_Total_time variable is increasingly affected by the total time taken by 0.136±0.0347 to complete the task between novice and professionals.

- **test_fixation:** During test code comprehension, we see that professionals have 36.7% lesser fixation counts than novices. We tested these differences using LMM and found that it is statistically significant (p-value = 4.51E-05***) for expertise where novices had higher test fixation counts than professionals. The test_fixation variable is decreasingly affected by the amount of experience in Java in years by -86.319± 19.453, indicating that professionals fixated only on methods names and necessary tests, as discussed in section 3.1.2.

- **test_TTFF(time to first fixation):** In this feature, we observe the time taken to switch to test code. After LMM analysis, we found that test_TTFF is highly significant (p-value = 3.01E-07***) which shows that novices took longer to switch to test code than professionals. The test_TTFF variable is decreasingly affected by the amount of experience in Java by -18.307±3.133 Hence professionals took on average 38.78 lesser times to switch to test code than novices. This shows that professionals chose non-linear reading methodology and focused on the control flow of the production code rather than reading line-by-line.

- **test_Total:** The total time taken to comprehend test code is highly significant (p-value = <2E-16***) and where novices had higher time comprehending test code. The test_TTFF variable is decreasingly affected by the total time taken to comprehend the code by 0.863±0.034. This is because they read all tests instead of focusing on reading methods/class names, which was highlighted by participants in section 3.1.2.

- **test_revisits_count:** Upon running the LMM we discovered that the result is indeed highly significant the expertise of the programmers (p-value = 2.38E-05***) where professionals had higher test revisit count than novices. With increase in years of expertise, the odds of being able to identify the test revisits count increased by 8.389 ± 1.654. This implies that professionals revisited to test code multiple times to corelate the production code with the test code in order to validate the samples, as answered by participants in open-ended questions. This shows that novices prefer to read all the test cases line by line instead on reading the most significant test cases.

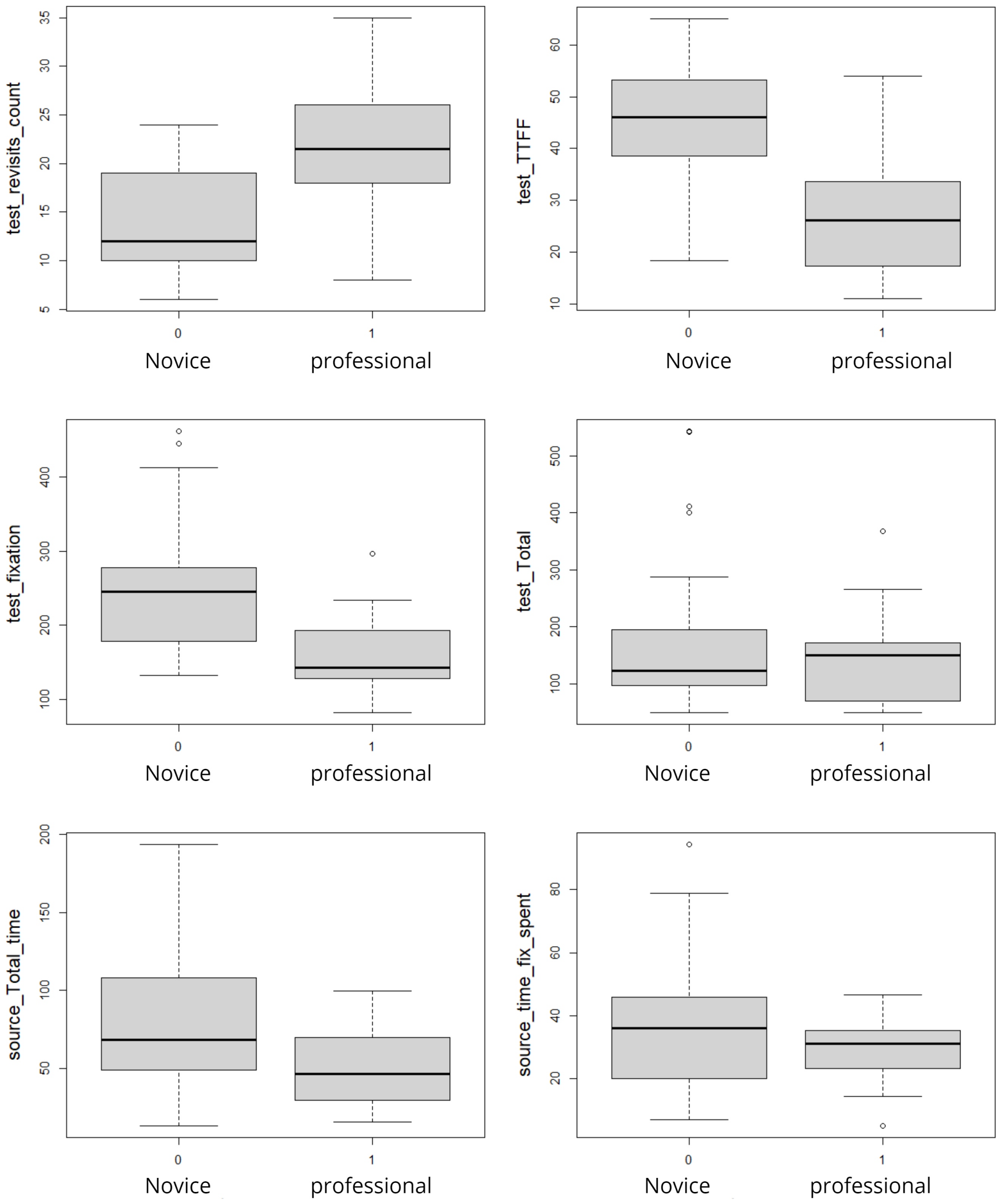


Figure 4.1: Box plots showing relationship between professionals with significant dependent variables

Linear mixed model fit by REML. t-tests use Satterthwaite's method ['lmerModLmerTest']						
Fixed effects	**Features**	**Estimate**	**Std. Error**	**df**	**t value**	$Pr(>\|t\|)$
source_fixation_count	total_time	0.10364	0.06146	52.93109	1.686	0.0976
	professional	-16.66941	15.35758	27.80421	-1.085	0.2871
source_Avg	total_time	9.63E-06	2.74E-05	4.88E+01	0.351	0.727
	professional	5.81E-03	6.84E-03	2.54E+01	0.849	0.404
source_TTFF	total_time	-0.0004481	0.0020815	53.5967187	-0.215	0.8304
	professional	0.1541709	0.5157746	53.9962501	0.299	0.7662
source_time_fix_spent	total_time	0.03936	0.01605	54.12681	2.452	**0.0175***
	professional	-5.36674	3.96982	54.42151	-1.352	0.182
source_Total_time	total_time	0.13688	0.03472	53.0202	3.943	**0.000238*****
	professional	-20.32861	9.92868	27.99235	-2.047	0.050101
source_revisits	total_time	0.003402	0.00407	53.98899	0.836	0.40691
	professional	0.983937	1.006678	54.138919	0.977	0.33271
test_fixation	total_time	0.05326	0.07858	53.94695	0.678	0.501
	professional	-86.31939	19.4534	54.20273	-4.437	**4.51E-05*****
test_Avg	total_time	2.46E-06	2.64E-05	5.50E+01	0.093	0.926
	professional	-5.71E-03	6.53E-03	5.50E+01	-0.875	0.385
test_TTFF	total_time	-0.01333	0.01249	52.44036	-1.067	0.291
	professional	-18.30731	3.1339	54.29997	-5.842	**3.01E-07*****
test_tim_fix_spent	total_time	-0.006363	0.023504	55	-0.271	0.788
	professional	-5.399673	5.809452	55	-0.929	0.357
test_Total	total_time	0.86312	0.03472	53.02017	24.862	**<2E-16*****
	professional	20.3286	9.92867	27.99239	2.047	0.050101
test_revisits_count	total_time	0.004493	0.006513	54.367332	0.69	0.493
	professional	8.389503	1.654151	27.542058	5.072	**2.38E-05*****

Table 4.2: Linear mixed model for fixation features.* Statistically significant effect ($\alpha \leq 0.05$).

In Figure 4.2, we visualize the total time taken by both novices and professionals to comprehend source code and test code. We see that the participates less test total time in comprehending Contains_Any (code 1) program and the maximum time comprehending Reverse (code 3) program. This shows that participants perceived Contains_Any to be easier to comprehend because it has only 4 test cases with few samples of 61 lines of code (LOC). On the contrary Reverse program had 3 test cases with multiples samples of 81 (LOC). Similarly, participants took minimum time to comprehend source code for Contains_Any (code 1) program and maximum time to comprehend Unique (code2) program.

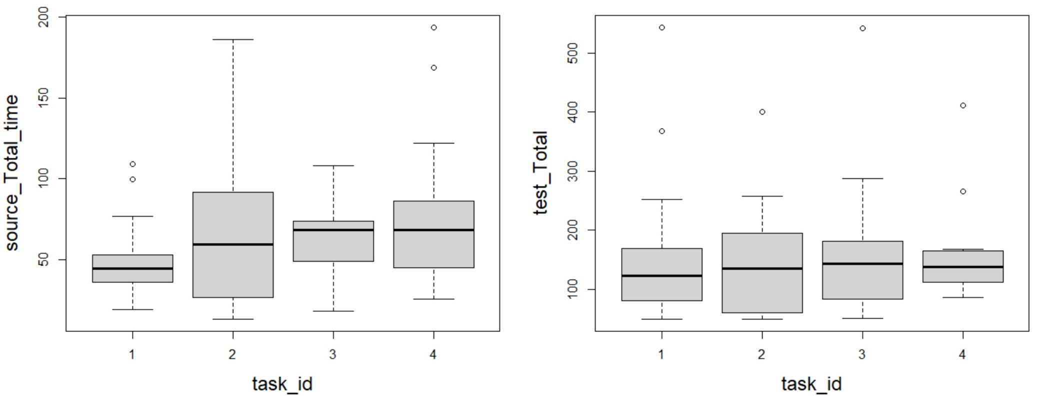


Figure 4.2: Total time taken to comprehend source code and test code

AOI based Linear mixed model						
Fixed effects	**Features**	**Estimate**	**Std. Error**	**df**	**t value**	$Pr(>\|t\|)$
test_name	total_time	0.04249	0.02604	54.65182	1.632	0.1085
	professional	-12.93515	6.68982	28.10284	-1.934	0.0633
assertions	total_time	0.004879	0.109218	52.431708	0.045	0.9645
	professional	65.698936	27.234565	54.812723	2.412	**0.0192***
method_first	total_time	0.009305	0.009486	52.224479	0.981	0.3312
	professional	-0.958852	2.388759	53.617516	-0.401	0.6897
method_last	total_time	0.00501	0.004353	54.982771	1.151	0.254775
	professional	-4.187072	1.171034	28.11434	-3.576	**0.001289****
control flow	total_time	-0.07581	0.04503	54.28334	-1.683	0.098
	professional	-18.49755	11.12392	54.24394	-1.663	0.102

Table 4.3: Linear mixed model based on AOI features.* Statistically significant effect ($\alpha \leq 0.05$)

4.2.2 Linear mixed model (LMM) based on AOI features

In this section, we report our results on AOI based features. We found that assertions and method_last feature has statistically significant results. Table 4.3 shows the results from LMM.

- **assertions**: The total fixation time is significant (p-value = 0.0192*) where professionals have lower fixations on assert statements. The assert_fix variable is increasingly affected by the amount of the expertise by 65.698±27.234, where novices spend 41.2% less time than professionals. Participants (P5-P7, P1, P2, P13, P9-P11) said they read the assert statements using input and output arguments to verify the production code.

- **method_last**: We see significant difference for the last method of the test code (p-value = 0.001289**), where novices read the test code line-by-line and had higher test code coverage than professionals. The method_last variable is decreasingly affected for expertise by -4.187±1.171 to fit the linear mixed model. This implies that novices spent 67.1% less time reading the last test method than professionals.

Other features such as Test_name, method_first and control flow did not yield any significant results.

Result RQ2: There is a statistical significance between novices and professionals for source_time_fix_spent and source_Total_time while comprehending the production code. Most importantly, we observe a significant difference in test code for test_fixation count, test_TTFF, test_total, test_revisits, assertions and method_last.

Chapter 5

Discussions

In this section, we first revisit the research questions in this study. Next, we reflect on the challenges of testing observed during our experiments. Finally, we address risks that could undermine our study's validity.

5.1 Research question review

In this section, we revisit the main research question using the results observed, and we discuss various reading patterns followed by programmers during test code comprehension. Based on the results, we have identified differences in reading patterns. The main research questions explored in this study are:

RQ1: How do developers read test code?

- All the participants started with production code and then switched between tests and production.
- Novices took more time to comprehended test code and spent more time fixating on test code.
- Professionals spend 63.6% more time reading test code and than novices.
- Professionals spent 35.7% lesser time comprehending assert statements and also had 41.2% lesser assert fixation counts than novices.
- Professionals revisited the test code 59.14% more than novices.
- Professionals read 13.79% of the test cases, whereas novices read 82.76% of the test cases.

In our study, 28 out of 29 participants confirmed that source code aided them in the comprehension of source code. This is because the production provides the overall functionality and test code helped them test the corner cases and boundary parameters which they missed to observe in the production code. We believe the professionals use their software experience in the industry to enhance their performance during comprehension tasks. We also verified our results using post-questionnaires provided by participants after the experiment.

RQ2: What are the differences between novices and professionals when reading test code?

We implemented a linear mixed regression model to verify our results using the eye tracking device. We see a significant difference between novices and professionals for test code comprehension, and we observed significant difference for test_fixation_count, test_TTFF, test_total and test_revisit. We also see similar trends for source_time_fix_spent and source_Total_time while comprehending the production code. These results confirms the previous studies in the field of software engineering [14], [92], [79].

5.2 Challenges

Our study not only provides a readability analysis of test code comprehension, but we also provide analysis of various challenges faced during testing. Table 5.1 shows the overall analysis of the challenges.

1. **Method/class names:** Method names play a crucial role in understanding the overall purpose of the tests. From our survey, 7 out of 29 participants believe that descriptive and clear method names help reduce the time taken and indicates the critical test cases. Participant P4 also mentioned that the clear and descriptive method names provided in this experiment helped them comprehend the code faster. P4 stated *"generally understanding test codes and their purpose when it is not clearly written like in these tests."* Hence to understand the purpose of the test code quickly when the production code is complicated, it is highly essential to write descriptive method/class names.

2. **Complexity of unit tests:** 4 out of 29 participants found it challenging to correlate test cases to source when very long and multiple assertions are present. P28 also highlighted that the tests codes need to be well organized and well commented on reducing the time taken to read tests cases. Furthermore, P15 exemplified *:"Test code can be written in many ways and sometimes it can be hard to see exactly what the methods do. Moreover, if the tester thinks of other cases than you do, it may be challenging to correlate them."* P6 added that having only a single test covering all the functionalities of the code becomes complex. P6 says *"I think the big problem is when the code test has no too many samples about the behaviour of the production class or when It has only one method that tests everything with many confusing variables."*

3. **Setup of mock tests:** P1 answered that due to various design methodology by followed by each developer, the tests becomes complex due to inconsistent naming, multiple assertions and complex variable. In our survey, 13.79% of the participants

Challenges	Participant	Percentage
Method/class names	P2, P3, P4, P26, P29, P21, P23	20.69%
Complexity of unit tests	P28, P15, P6, P26	13.79%
Setup of mock tests	P21, P19, P1, P17	13.79%
Corner case tests	P8, P9	6.9%

Table 5.1: Summary of challenges

(P21, P19, P1, P17) agree that mock test is complicated to write, specifically when it involves backend databases. P21 states *"I haven't faced such a challenge yet. Most of the time I find it difficult to mock a backend service that is not a database neither an HTTP server, for instance: SMTP server or SMPP server."*

4. **Corner case tests** Another challenge participants (P8, P9) faced during test code comprehending is finding all the corner cases for the production code. Participant P29 emphasizes that *"Writing tests for corner cases is difficult for me and when the tests are not written properly with case classes and methods names, it difficult to understand what it does."* Difficulties finding the corner cases is indicated by 2 out of 29 survey participants.

5.3 Threats to validity

Eye-tracking provides useful insights into code comprehension experiments through the collection of visual stimulation data from participants. They provide researchers with observations that can not be gained through interviews or questionnaires. That being said, these studies are not defects free and pose methodological and ethical issues.

5.3.1 Internal Validity

We focused our study mainly on code readability for test code. Since LOC for production code was lesser than that of test code, we could not receive sufficient analysis for production code comprehension using fixation data. However, the test code analysis had an adequate amount of manually written test cases which provided the required amount of fixation data needed for this study, as we are mainly interested in test code readability metrics. In our research, we implemented seven fixation features using eye tracker, which validated using gaze videos manually by seeing videos of all participants. The summary tasks check the cognitive process of the participant. We implemented the following criteria to evaluate if the participates performed the experiment completely, and participants who did not meet the standards were excluded from the experiment.

3Descriptive explanations of the implementation of code The eye-tracking data is captured according to the explanations Minimum of 50 fixations per task should be captured.

In our experiment, we found that all the participant satisfied the criteria except for one participant since the explanations were too short, and no accurate corresponding fixation data was captured. Hence our experiment had a total of 29 participants.

5.3.2 External Validity

We observed that, while each participant's eye tracker was calibrated before each mission, the gaze orientation of the captured sometimes looked too tiny or too large while mapping gaze on code. It was sometimes shifted to the left or the right in even more extreme situations. Such a mistake will occur if the subject moves too far from the camera. In such cases, the gaze data was reviewed and was readjusted to fit the correct offset. As the study was conducted in an online platform using realeye.io, we could not run a controlled experiment as performed in previous research using an eye-tracking device [92], [79], [14].

Chapter 6

Conclusion

In our study, we conducted empirical research aiming at investigating the readability of test cases. In particular, we focused on the difference in code readability between *(i)* production code and corresponding test code *(ii)* reading patterns in novices and professionals *(iii)* fixation metrics from eye tracker and manual analysis of gaze data. Our preliminary results are open to new interesting research directions.

This research contributes to the current field by identifying various reading patterns between novices and professionals such as *(i)* detect how often participants transition between regions test code and the production code, *(ii)* discovered various trends in comprehending test code *(iii)* the most influential factors affecting the comprehension of test code. We have selected several manual and automated dependent features to study the patterns using linear mixed regression model and also verified our results according to the answers provided by the participants during the post-questionnaire.

Our results show that *(i)* all programmers first comprehended the production code and then switched between test and production codes, *(ii)* novices had higher fixations reading test code and assert statements and also took longer time to comprehend them, *(iii)* professionals revisited the test code more than novices in-order to verify the assert statements whereas novices read the test code line-by-line and switched to test code when they found production code difficult to understand, *(iv)* professionals had significantly lesser test code coverage than novices, and *(v)* there is a significant difference in reading test code between novice and professionals.

Our methodology has some limitations which need to be addressed. Firstly, an online experiment can affect the ability to accurately detect the differences in reading patterns among participants. Since it is not possible to monitor the experiment, it is difficult to validate the authenticity of the results. Secondly, given the small sample size, there is minimal generalization. Hence further studies are required to achieve generalizability.

Our long-term goal is to build technologies that can support young programmers to comprehend test code in real-time using their eye-tracking data. This study is the first step in this direction. We call other researchers to validate and further contribute by improving experimental techniques using other bio-metric devices such as EEG and fMRI to better understand how programmers read test code.

Bibliography

[1] • The online eye-tracking platform for distributed research using a webcam gaze-tracking. `www.realeye.io/`. Accessed: 2020-04-24.

[2] How to interpret regression models that have significant variables but a low r-squared. `https://statisticsbyjim.com/regression/low-r-squared-regression/#:~:text=The\%20statistical\%20significance\%20indicates\%20that,variability\%20in\%20the\%20dependent\%20variable`. Accessed: 2020-10-12.

[3] Mohammed Akour and Bouchaib Falah. Application domain and programming language readability yardsticks. In *2016 7th International Conference on Computer Science and Information Technology (CSIT)*, pages 1–6. IEEE, 2016.

[4] Erik W Anderson, Kristin C Potter, Laura E Matzen, Jason F Shepherd, Gilbert A Preston, and Cláudio T Silva. A user study of visualization effectiveness using eeg and cognitive load. In *Computer graphics forum*, volume 30, pages 791–800. Wiley Online Library, 2011.

[5] Christoph Aschwanden and Martha Crosby. Code scanning patterns in program comprehension. In *Proceedings of the 39th hawaii international conference on system sciences*, 2006.

[6] Roman Bednarik and Markku Tukiainen. Effects of display blurring on the behavior of novices and experts during program debugging. In *CHI'05 Extended abstracts on human factors in computing systems*, pages 1204–1207, 2005.

[7] Roman Bednarik and Markku Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, pages 125–132, 2006.

[8] Roman Bednarik, Niko Myller, Erkki Sutinen, and Markku Tukiainen. Program visualization: Comparing eye-tracking patterns with comprehension summaries and performance. In *Proceedings of the 18th Annual Psychology of Programming Workshop*, pages 66–82, 2006.

[9] Roman Bednarik, Shahram Eivazi, and Hana Vrzakova. A computational approach for prediction of problem-solving behavior using support vector machines and eye-tracking data. In *Eye Gaze in Intelligent User Interfaces*, pages 111–134. Springer, 2013.

[10] David Binkley and Dawn Lawrie. Information retrieval applications in software maintenance and evolution. *Encyclopedia of software engineering*, pages 454–463, 2010.

[11] Raymond P.L. Buse and Westley R. Weimer. A metric for software readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, page 121–130, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605580500. doi: 10.1145/1390630.1390647. URL `https://doi.org/10.1145/1390630.1390647`.

[12] Raymond PL Buse and Westley R Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130, 2008.

[13] Teresa Busjahn, Carsten Schulte, and Andreas Busjahn. Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, pages 1–9, 2011.

[14] Teresa Busjahn, Roman Bednarik, Andrew Begel, Martha Crosby, James H Paterson, Carsten Schulte, Bonita Sharif, and Sascha Tamm. Eye movements in code reading: Relaxing the linear order. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 255–265. IEEE, 2015.

[15] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. Relating identifier naming flaws and code quality: An empirical study. In *2009 16th Working Conference on Reverse Engineering*, pages 31–35. IEEE, 2009.

[16] Manuel Carreiras and Charles Clifton Jr. *The on-line study of sentence comprehension: Eyetracking, ERPs and beyond.* Psychology Press, 2004.

[17] Debatri Chatterjee, Arijit Sinharay, and Amit Konar. Eeg-based fuzzy cognitive load classification during logical analysis of program segments. In *2013 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, pages 1–6. IEEE, 2013.

[18] Song C. Choi and Walt Scacchi. Extracting and restructuring the design of large systems. *Ieee Software*, 7(1):66–71, 1990.

[19] Igor Crk and Timothy Kluthe. Toward using alpha and theta brain waves to quantify programmer expertise. In *2014 36th Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, pages 5373–5376. IEEE, 2014.

[20] Martha E Crosby and Jan Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):25–35, 1990.

[21] Amit Rajendra Desai. Eeg-based evaluation of cognitive and emotional arousal when coding in different programming languages, 2017.

[22] Amit Rajendra Desai. Eeg-based evaluation of cognitive and emotional arousal when coding in different programming languages, 2017.

[23] Massimiliano Di Penta, RE Kurt Stirewalt, and Eileen Kraemer. Designing your next empirical study on program comprehension. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 281–285. IEEE, 2007.

[24] Stephan Dreiseitl and Lucila Ohno-Machado. Logistic regression and artificial neural network classification models: a methodology review. *Journal of biomedical informatics*, 35(5-6):352–359, 2002.

[25] Sukru Eraslan, Yeliz Yesilada, and Simon Harper. Eye tracking scanpath analysis techniques on web pages: A survey, evaluation and comparison. 2016.

[26] Letha H Etzkorn, Lisa L Bowen, and Carl G Davis. An approach to program understanding by natural language understanding. *Natural Language Engineering*, 5(3): 219–236, 1999.

[27] Yongqi Gu. To code or not to code: Dilemmas in analysing think-aloud protocols in learning strategies research. *System*, 43:74–81, 2014.

[28] Alexander Gundel and Glenn F Wilson. Topographical changes in the ongoing eeg related to the difficulty of mental tasks. *Brain topography*, 5(1):17–25, 1992.

[29] Samuel W Hincks, Daniel Afergan, and Robert JK Jacob. Using fnirs for real-time cognitive workload assessment. In *International Conference on Augmented Cognition*, pages 198–208. Springer, 2016.

[30] Philip S Holzman, Leonard R Proctor, Deborah L Levy, Nicholas J Yasillo, Herbert Y Meltzer, and Stephen W Hurt. Eye-tracking dysfunctions in schizophrenic patients and their relatives. *Archives of general psychiatry*, 31(2):143–151, 1974.

[31] William G Iacono, Vicente B Tuason, and Roger A Johnson. Dissociation of smooth-pursuit and saccadic eye tracking in remitted schizophrenics: An ocular reaction time task that schizophrenics perform well. *Archives of General Psychiatry*, 38(9):991–996, 1981.

[32] Yoshiharu Ikutani and Hidetake Uwano. Brain activity measurement during program comprehension with nirs. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 1–6. IEEE, 2014.

[33] Shamsi T Iqbal, Xianjun Sam Zheng, and Brian P Bailey. Task-evoked pupillary response to mental workload in human-computer interaction. In *CHI'04 extended abstracts on Human factors in computing systems*, pages 1477–1480, 2004.

[34] Manabu Kamimura and Gail C Murphy. Towards generating human-oriented summaries of unit test cases. In *2013 21st International Conference on Program Comprehension (ICPC)*, pages 215–218. IEEE, 2013.

[35] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *11th International Software Product Line Conference (SPLC 2007)*, pages 223–232. IEEE, 2007.

[36] Yongbeom Kim and Edward A Stohr. Software reuse: survey and research directions. *Journal of Management Information Systems*, 14(4):113–147, 1998.

[37] Imran Kurt, Mevlut Ture, and A Turhan Kurum. Comparing performances of logistic regression, classification and regression tree, and neural networks for predicting coronary artery disease. *Expert systems with applications*, 34(1):366–374, 2008.

[38] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006.

[39] SeolHwa Lee, Andrew Matteson, Danial Hooshyar, SongHyun Kim, JaeBum Jung, GiChun Nam, and HeuiSeok Lim. Comparing programming language comprehension between novice and expert programmers using eeg analysis. In *2016 IEEE 16th International Conference on Bioinformatics and Bioengineering (BIBE)*, pages 350–355. IEEE, 2016.

[40] Seolhwa Lee, Danial Hooshyar, Hyesung Ji, Kichun Nam, and Heuiseok Lim. Mining biometric data to predict programmer expertise and task difficulty. *Cluster Computing*, 21(1):1097–1107, 2018.

[41] Taek Lee, Jung Been Lee, and Hoh Peter In. A study of different coding styles affecting code readability. *International Journal of Software Engineering and Its Applications*, 7(5):413–422, 2013.

[42] Stephenie C Lemon, Jason Roy, Melissa A Clark, Peter D Friedmann, and William Rakowski. Classification and regression tree analysis in public health: methodological review and comparison with logistic regression. *Annals of behavioral medicine*, 26(3):172–181, 2003.

[43] Boyang Li, Christopher Vendome, Mario Linares-Vásquez, and Denys Poshyvanyk. Aiding comprehension of unit test cases and test suites with stereotype-based tagging. In *Proceedings of the 26th Conference on Program Comprehension*, pages 52–63, 2018.

[44] David EJ Linden, Robert A Bittner, Lars Muckli, James A Waltz, Nikolaus Kriegeskorte, Rainer Goebel, Wolf Singer, and Matthias HJ Munk. Cortical capacity constraints for visual working memory: dissociation of fmri load effects in a fronto-parietal network. *Neuroimage*, 20(3):1518–1530, 2003.

[45] Damien Litchfield, Linden J Ball, Tim Donovan, David J Manning, and Trevor Crawford. Viewing another person's eye movements improves identification of pulmonary nodules in chest x-ray inspection. *Journal of Experimental Psychology: Applied*, 16 (3):251, 2010.

[46] Jonathan I Maletic and Andrian Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*, pages 103–112. IEEE, 2001.

[47] Shneiderman Mayer Mayer. R., shneiderman, b.: Syntactic/semantic interactions in programmer behavior: a model and experimental results. *International Journal of Computer and Information Sciences*, 8(3):219–238, 1979.

[48] Gerard Meszaros and Jim Doble. Metapatterns: A pattern language for pattern writing. In *The 3rd Pattern Languages of Programming conference, Monticello, Illinois*. Citeseer, 1996.

[49] Joel Meyers, Susan Lytle, Donna Palladino, Gillian Devenpeck, and Michael Green. Think-aloud protocol analysis: An investigation of reading comprehension strategies in fourth-and fifth-grade students. *Journal of Psychoeducational Assessment*, 8(2): 112–127, 1990.

[50] Caitlin Mills, Igor Fridman, Walid Soussou, Disha Waghray, Andrew M Olney, and Sidney K D'Mello. Put your thinking cap on: detecting cognitive load using eeg during learning. In *Proceedings of the seventh international learning analytics & knowledge conference*, pages 80–89, 2017.

[51] Matthias M. Mueller and Oliver Hagner. Experiment about test-first programming. *IEE Proceedings-Software*, 149(5):131–136, 2002.

[52] Sebastian C Müller and Thomas Fritz. Stuck and frustrated or in flow and happy: sensing developers' emotions and progress. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 688–699. IEEE, 2015.

[53] Takao Nakagawa, Yasutaka Kamei, Hidetake Uwano, Akito Monden, Kenichi Matsumoto, and Daniel M German. Quantifying programmers' mental workload during program comprehension based on cerebral blood flow measurement: a controlled experiment. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 448–451, 2014.

[54] Unaizah Obaidellah, Mohammed Al Haek, and Peter C-H Cheng. A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys (CSUR)*, 51(1):1–58, 2018.

[55] Pavel A Orlov. Ambient and focal attention during source-code comprehension. *FACHBEREICH MATHEMATIK UND INFORMATIK SERIE B INFORMATIK*, page 12, 2017.

[56] Ankit Pahal and RS Chillar. Code readability: a review of metrics for software quality. *Int J Comput Trends Technol*, 46(1):1–58, 2017.

[57] Sebastian Pannasch, Jens R Helmert, Katharina Roth, Ann-Katrin Herbold, and Henrik Walter. Visual fixation durations and saccade amplitudes: Shifting relationship in a variety of conditions. 2008.

[58] James H Paterson and Katrin Hartmann. Readability metrics for program code: How is reading ease reflected in gaze? *Eye Movements in Programming: Models to Data*, page 19, 2016.

[59] Evan M Peck, Daniel Afergan, Beste F Yuksel, Francine Lalooses, and Robert JK Jacob. Using fnirs to measure mental workload in the real world. In *Advances in physiological computing*, pages 117–139. Springer, 2014.

[60] Norman Peitek, Janet Siegmund, and André Brechmann. Enhancing fmri studies of program comprehension with eye-tracking. In *Proc. Int'l Workshop on Eye Movements in Programming. Freie Universität Berlin*, pages 22–23, 2017.

[61] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, and André Brechmann. Toward conjoint analysis of simultaneous eye-tracking and fmri data for program-comprehension studies. In *Proceedings of the Workshop on Eye Movements in Programming*, pages 1–5, 2018.

[62] Norman Peitek, Janet Siegmund, Chris Parnin, Sven Apel, Johannes C Hofmeister, and André Brechmann. Simultaneous measurement of program comprehension with fmri and eye tracking: A case study. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018.

[63] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[64] Matthew F Pike, Horia A Maior, Martin Porcheron, Sarah C Sharples, and Max L Wilson. Measuring the effect of think aloud protocols on workload using fnirs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3807–3816, 2014.

[65] Denys Poshyvanyk and Andrian Marcus. Combining formal concept analysis with information retrieval for concept location in source code. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 37–48. IEEE, 2007.

[66] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th working conference on mining software repositories*, pages 73–82, 2011.

[67] Yahya Rafique and Vojislav B Mišić. The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856, 2012.

[68] Christina Regenbogen, Maarten De Vos, Stefan Debener, Bruce I Turetsky, Carolin Mößnang, Andreas Finkelmeyer, Ute Habel, Irene Neuner, and Thilo Kellermann. Auditory processing under cross-modal visual load investigated with simultaneous eeg-fmri. *PloS one*, 7(12), 2012.

[69] Phillip A Relf. Tool assisted identifier naming for improved software readability: an empirical study. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10–pp. IEEE, 2005.

[70] Phillip Anthony Relf. Achieving software quality through source code readability. *Quality Contract Manufacturing LLC*, 2004.

[71] Robert S Rist et al. Plans in programming: definition, demonstration, and development. In *Empirical studies of programmers*, pages 28–47, 1986.

[72] Paige Rodeghero, Collin McMillan, Paul W McBurney, Nigel Bosch, and Sidney D'Mello. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering*, pages 390–401, 2014.

[73] Jonathan Saddler. Understanding eye gaze patterns in code comprehension. 2020.

[74] Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. Improving code readability models with textual features. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016.

[75] Giuseppe Scanniello, Simone Romano, Davide Fucci, Burak Turhan, and Natalia Juristo. Students' and professionals' perceptions of test-driven development: a focus group study. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1422–1427, 2016.

[76] Gonny Schellings, Cor Aarnoutse, and Jan Van Leeuwe. Third-grader's think-aloud protocols: Types of reading activities in reading an expository text. *Learning and Instruction*, 16(6):549–568, 2006.

[77] Todd Sedano. Code readability testing, an empirical study. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 111–117. IEEE, 2016.

[78] Zohreh Sharafi, Zéphyrin Soh, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Women and men—different but equal: On the impact of identifier style on source code reading. In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 27–36. IEEE, 2012.

[79] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 381–384, 2012.

[80] Kshitij Sharma, Patrick Jermann, Marc-Antoine Nüssli, and Pierre Dillenbourg. Understanding collaborative program comprehension: Interlacing gaze and dialogues. 2013.

[81] Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, and Hakan Erdogmus. What do we know about test-driven development? *IEEE software*, 27(6):16–19, 2010.

[82] Janet Siegmund, André Brechmann, Sven Apel, Christian Kästner, Jörg Liebig, Thomas Leich, and Gunter Saake. Toward measuring program comprehension with functional magnetic resonance imaging. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–4, 2012.

[83] Janet Siegmund, Christian Kästner, Sven Apel, Chris Parnin, Anja Bethmann, Thomas Leich, Gunter Saake, and André Brechmann. Understanding understanding source code with functional magnetic resonance imaging. In *Proceedings of the 36th International Conference on Software Engineering*, pages 378–389, 2014.

[84] Janet Siegmund, Norman Peitek, Chris Parnin, Sven Apel, Johannes Hofmeister, Christian Kästner, Andrew Begel, Anja Bethmann, and André Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017.

[85] Susan Elliott Sim, Margaret-Anne Storey, and Andreas Winter. A structured demonstration of five program comprehension tools: lessons learnt. In *Proceedings Seventh Working Conference on Reverse Engineering*, pages 210–212. IEEE, 2000.

[86] Elliot Soloway and Sitharama Iyengar. Empirical studies of programmers: Papers presented at the first workshop on empirical studies of programmers, 1986.

[87] M-A Storey. Theories, methods and tools in program comprehension: Past, present and future. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 181–191. IEEE, 2005.

[88] Yiyi Sun. Unit testing. In *Practical Application Development with AppRun*, pages 247–264. Springer, 2019.

[89] Yahya Tashtoush, Zeinab Odat, Izzat M Alsmadi, and Maryan Yatim. Impact of programming features on code readability. 2013.

[90] D Tomasi, L Chang, EC Caparelli, and T Ernst. Different activation patterns for working memory load and visual attention load. *Brain research*, 1132:158–165, 2007.

[91] Dardo Tomasi, Thomas Ernst, Elisabeth C Caparelli, and Linda Chang. Common deactivation patterns during working memory and visual attention tasks: An intra-subject fmri study at 4 tesla. *Human brain mapping*, 27(8):694–705, 2006.

[92] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. An eye-tracking study assessing the comprehension of c++ and python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 231–234, 2014.

[93] Anirudh Unni, Klas Ihme, Henrik Surm, Lars Weber, Andreas Lüdtke, Daniela Nicklas, Meike Jipp, and Jochem W Rieger. Brain activity measured with fnirs for the prediction of cognitive workload. In *2015 6th IEEE International Conference on Cognitive Infocommunications (CogInfoCom)*, pages 349–354. IEEE, 2015.

[94] Lotte F Van Dillen, Dirk J Heslenfeld, and Sander L Koole. Tuning down the emotional brain: an fmri study of the effects of cognitive load on the processing of affective images. *Neuroimage*, 45(4):1212–1219, 2009.

[95] Anneliese von Mayrhauser and A Marie Vans. *Program Understanding: A Survey*. Colorado State Univ., 1994.

[96] Xiaoran Wang, Lori Pollock, and K Vijay-Shanker. Automatic segmentation of method code into meaningful blocks to improve readability. In *2011 18th Working Conference on Reverse Engineering*, pages 35–44. IEEE, 2011.

[97] Michel Wedel and Rik Pieters. A review of eye-tracking research in marketing. In *Review of marketing research*, pages 123–147. Routledge, 2017.

[98] Michal R Wrobel. Emotions in the software development process. In *2013 6th International Conference on Human System Interactions (HSI)*, pages 518–523. IEEE, 2013.

[99] Chak Shun Yu. Towards understanding how developers comprehend tests. 2018.

[100] Chak Shun Yu, Christoph Treude, and Maurício Aniche. Comprehending test code: An empirical study. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 501–512. IEEE, 2019.

[101] Pega Zarjam, Julien Epps, and Fang Chen. Characterizing working memory load using eeg delta activity. In *2011 19th European Signal Processing Conference*, pages 1554–1558. IEEE, 2011.

[102] Pega Zarjam, Julien Epps, and Nigel H Lovell. Beyond subjective self-rating: Eeg signal classification of cognitive workload. *IEEE Transactions on Autonomous Mental Development*, 7(4):301–310, 2015.

[103] Ji Zhu and Trevor Hastie. Classification of gene microarrays by penalized logistic regression. *Biostatistics*, 5(3):427–443, 2004.

[104] Manuela Züger and Thomas Fritz. Interruptibility of software developers and its prediction using psycho-physiological sensors. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*, pages 2981–2990, 2015.

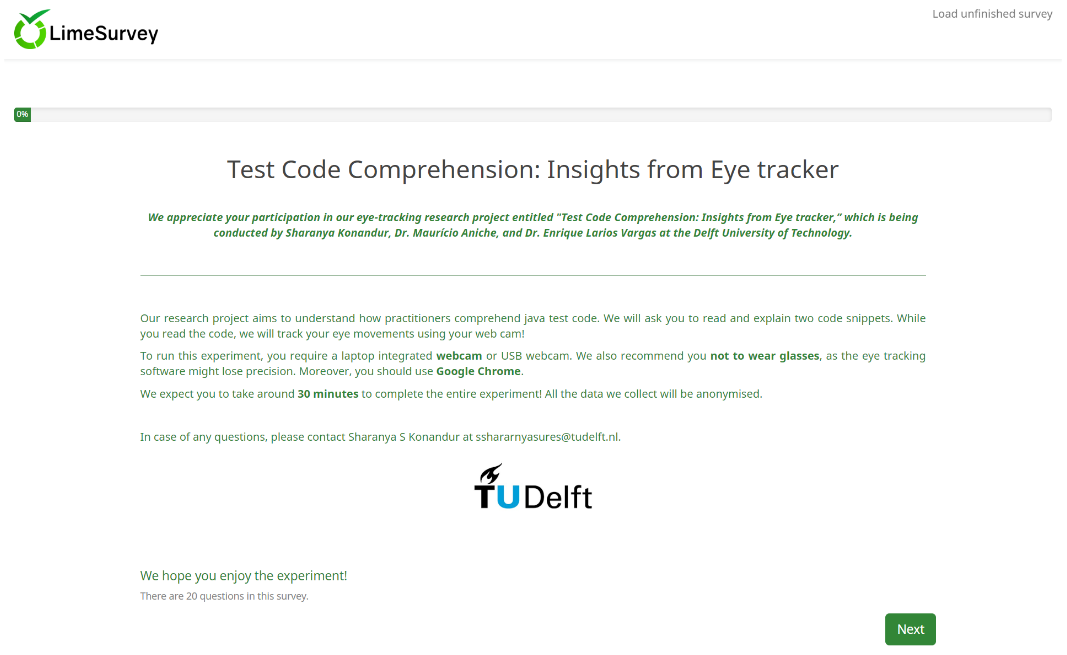
Appendix A

Experimental Design

In this appendix provide various experimental data collected and the survey overview.

A.1 Survey format

Below we provide the complete survey implemented in this experiment.



Test Code Comprehension: Insights from Eye tracker

We appreciate your participation in our eye-tracking research project entitled "Test Code Comprehension: Insights from Eye tracker," which is being conducted by Sharanya Konandur, Dr. Mauricio Aniche, and Dr. Enrique Larios Vargas at the Delft University of Technology.

Our research project aims to understand how practitioners comprehend java test code. We will ask you to read and explain two code snippets. While you read the code, we will track your eye movements using your web cam!

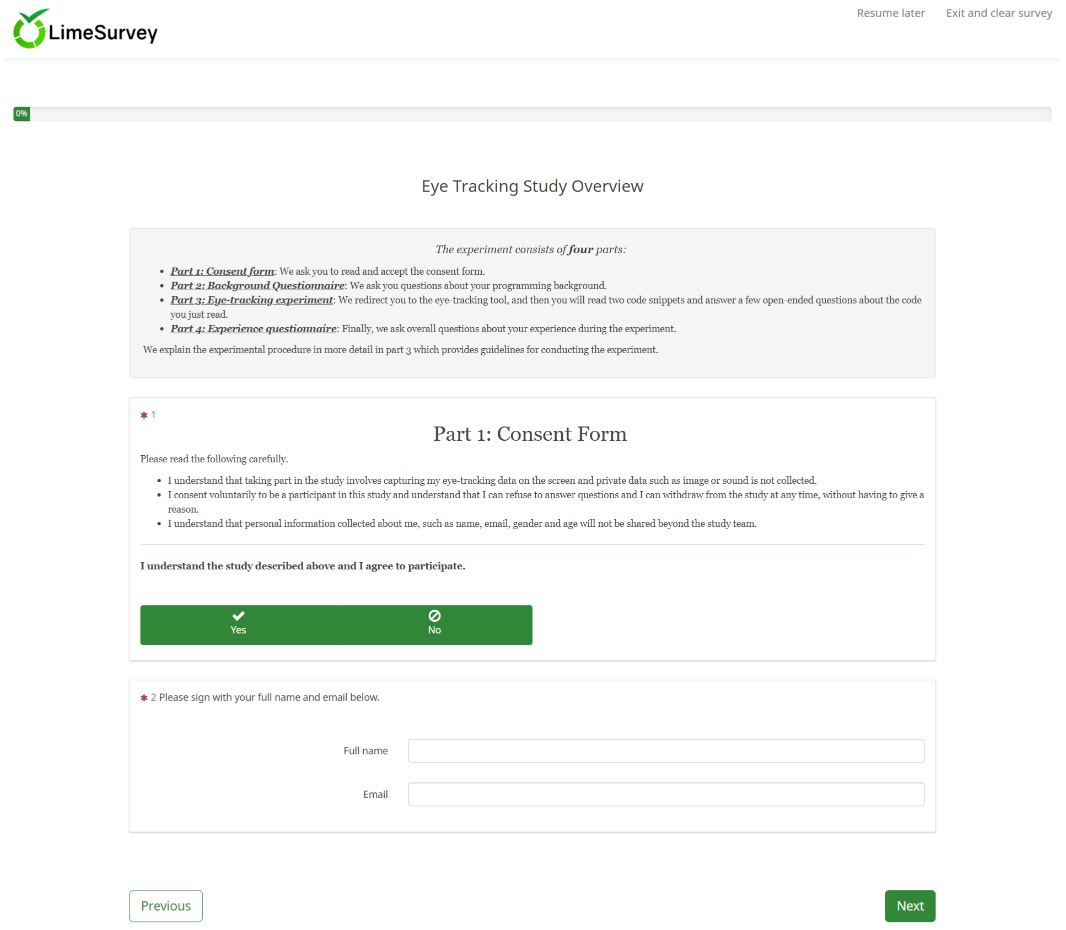
To run this experiment, you require a laptop integrated **webcam** or USB webcam. We also recommend you **not to wear glasses**, as the eye tracking software might lose precision. Moreover, you should use **Google Chrome**.

We expect you to take around **30 minutes** to complete the entire experiment! All the data we collect will be anonymised.

In case of any questions, please contact Sharanya S Konandur at sshararnyasures@tudelft.nl.

We hope you enjoy the experiment!

There are 20 questions in this survey.



Eye Tracking Study Overview

*The experiment consists of **four** parts:*

- ***Part 1: Consent form***: We ask you to read and accept the consent form.
- ***Part 2: Background Questionnaire***: We ask you questions about your programming background.
- ***Part 3: Eye-tracking experiment***: We redirect you to the eye-tracking tool, and then you will read two code snippets and answer a few open-ended questions about the code you just read.
- ***Part 4: Experience questionnaire***: Finally, we ask overall questions about your experience during the experiment.

We explain the experimental procedure in more detail in part 3 which provides guidelines for conducting the experiment.

* 1

Part 1: Consent Form

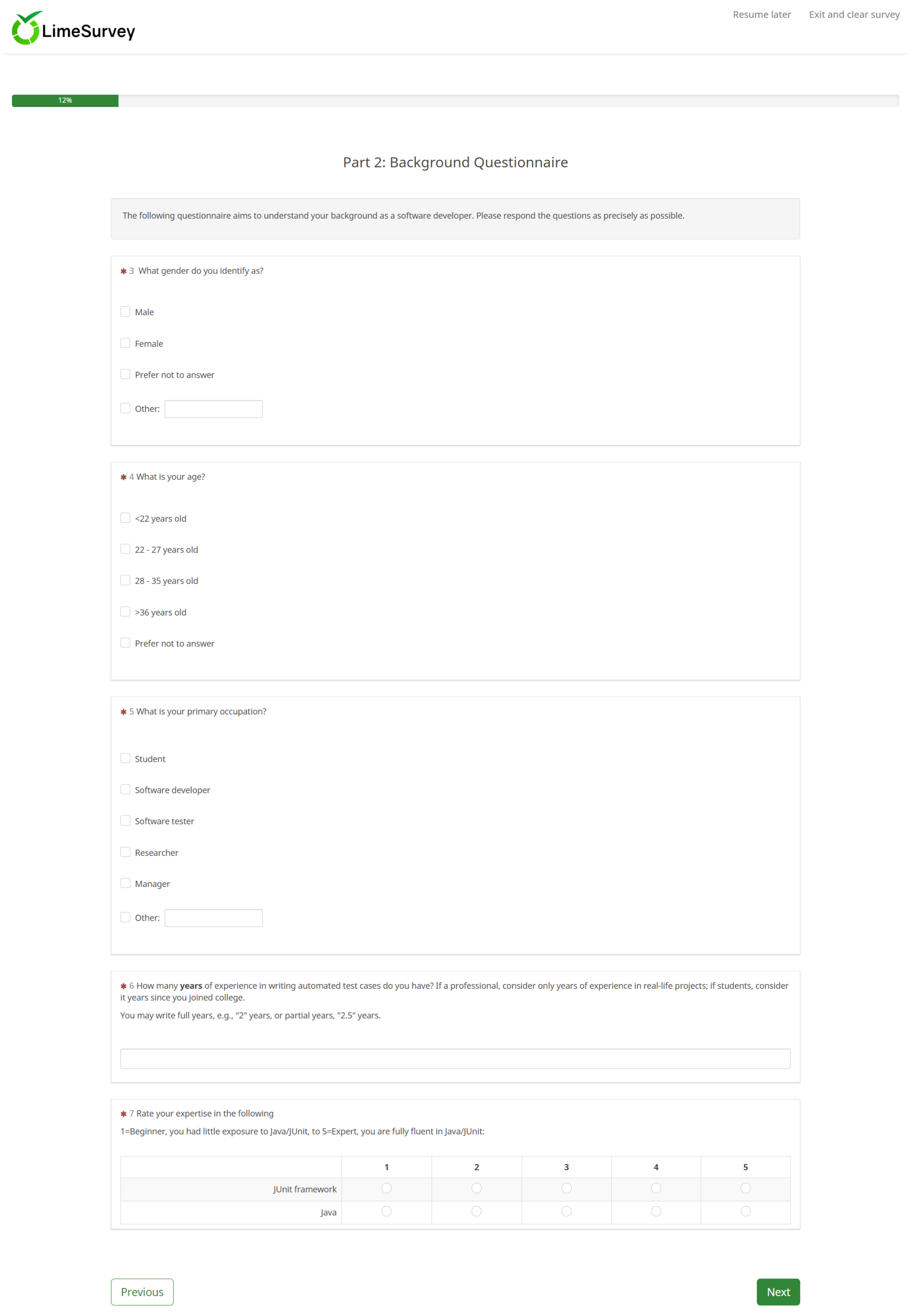
Please read the following carefully.

- I understand that taking part in the study involves capturing my eye-tracking data on the screen and private data such as image or sound is not collected.
- I consent voluntarily to be a participant in this study and understand that I can refuse to answer questions and I can withdraw from the study at any time, without having to give a reason.
- I understand that personal information collected about me, such as name, email, gender and age will not be shared beyond the study team.

I understand the study described above and I agree to participate.

* 2 Please sign with your full name and email below.

Full name	
Email	



Part 2: Background Questionnaire

The following questionnaire aims to understand your background as a software developer. Please respond the questions as precisely as possible.

* 3 What gender do you identify as?

- Male
- Female
- Prefer not to answer
- Other:

* 4 What is your age?

- <22 years old
- 22 - 27 years old
- 28 - 35 years old
- >36 years old
- Prefer not to answer

* 5 What is your primary occupation?

- Student
- Software developer
- Software tester
- Researcher
- Manager
- Other:

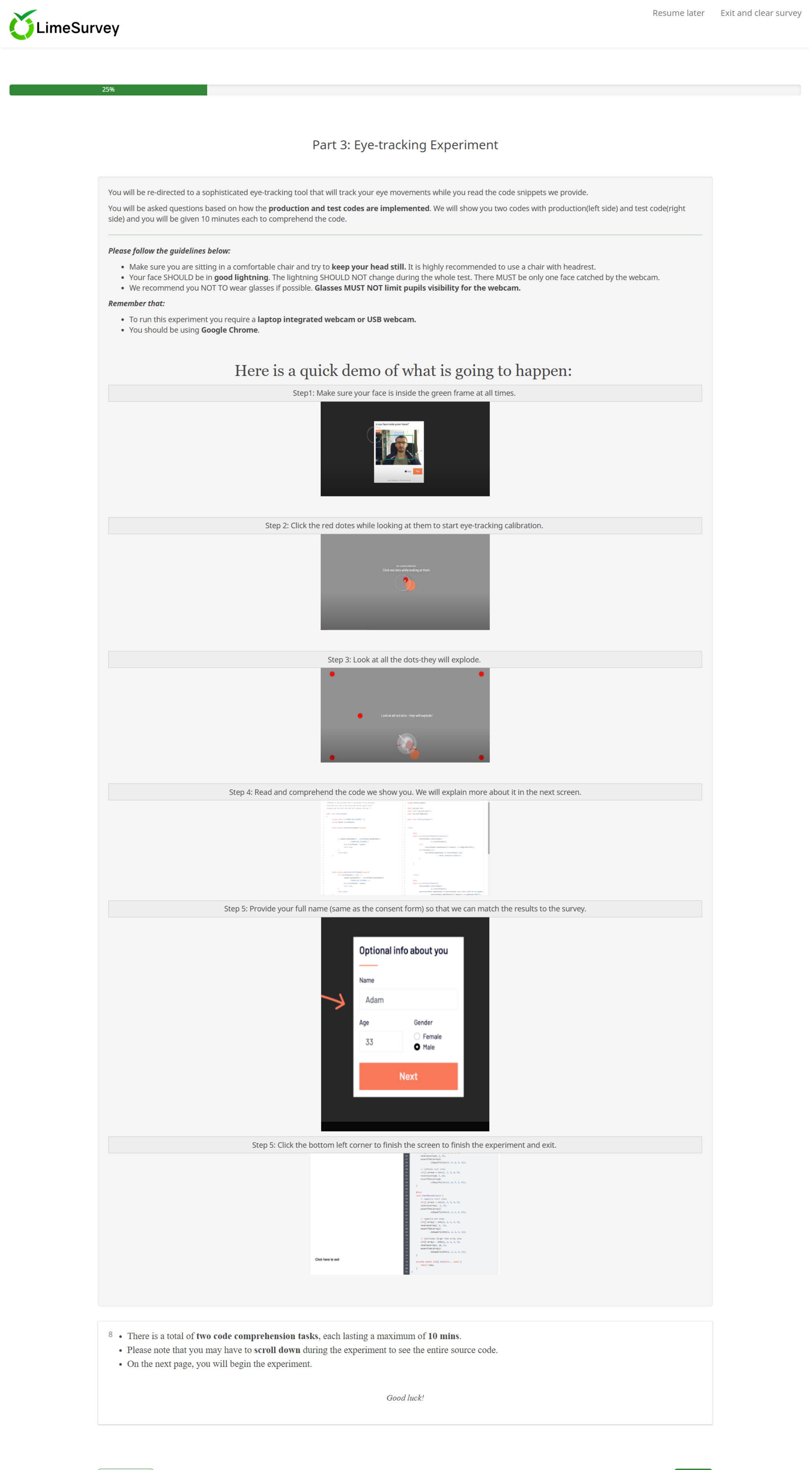
* 6 How many **years** of experience in writing automated test cases do you have? If a professional, consider only years of experience in real-life projects; if students, consider it years since you joined college.

You may write full years, e.g., "2" years, or partial years, "2.5" years.

* 7 Rate your expertise in the following

1=Beginner, you had little exposure to Java/JUnit, to 5=Expert, you are fully fluent in Java/JUnit:

	1	2	3	4	5
JUnit framework	○	○	○	○	○
Java	○	○	○	○	○



LimeSurvey

Resume later   Exit and clear survey

25%

Part 3: Eye-tracking Experiment

You will be re-directed to a sophisticated eye-tracking tool that will track your eye movements while you read the code snippets we provide.

You will be asked questions based on how the **production and test codes are implemented**. We will show you two codes with production(left side) and test code(right side) and you will be given 10 minutes each to comprehend the code.

Please follow the guidelines below:

- Make sure you are sitting in a comfortable chair and try to **keep your head still.** It is highly recommended to use a chair with headrest.
- Your face SHOULD be in **good lightning**. The lightning SHOULD NOT change during the whole test. There MUST be only one face catched by the webcam.
- We recommend you NOT TO wear glasses if possible. **Glasses MUST NOT limit pupils visibility for the webcam.**

Remember that:

- To run this experiment you require a **laptop integrated webcam or USB webcam.**
- You should be using **Google Chrome**.

Here is a quick demo of what is going to happen:

Step1: Make sure your face is inside the green frame at all times.

Step 2: Click the red dotes while looking at them to start eye-tracking calibration.

Step 3: Look at all the dots-they will explode.

Step 4: Read and comprehend the code we show you. We will explain more about it in the next screen.

Step 5: Provide your full name (same as the consent form) so that we can match the results to the survey.

Optional info about you

Name: Adam

Age: 33

Gender: Female [✓ Male]

Next

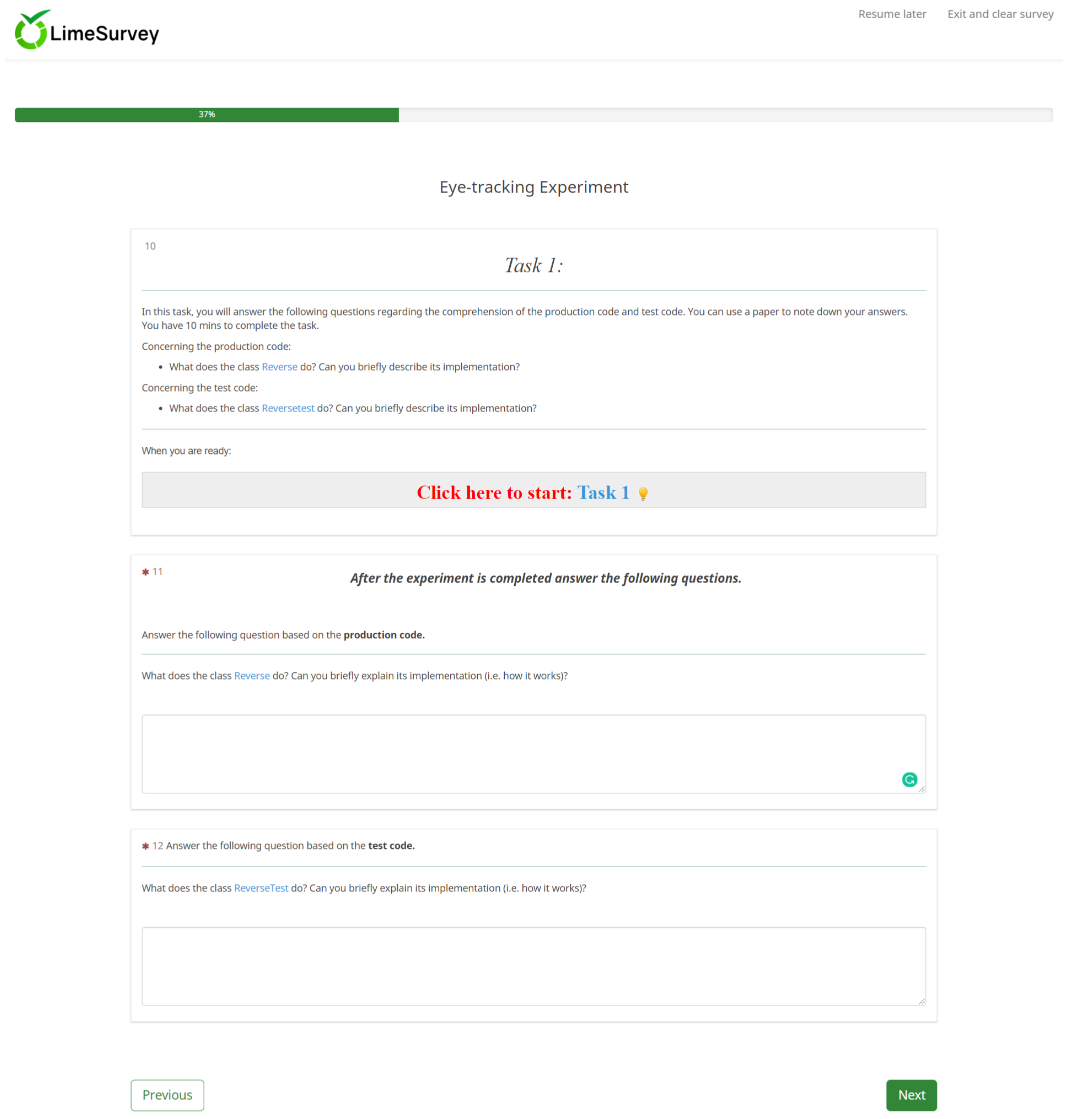
Step 5: Click the bottom left corner to finish the screen to finish the experiment and exit.

Click here to exit

8

- There is a total of **two code comprehension tasks**, each lasting a maximum of **10 mins**.
- Please note that you may have to **scroll down** during the experiment to see the entire source code.
- On the next page, you will begin the experiment.

Good luck!



LimeSurvey

Resume later   Exit and clear survey

37%

Eye-tracking Experiment

10

Task 1:

In this task, you will answer the following questions regarding the comprehension of the production code and test code. You can use a paper to note down your answers. You have 10 mins to complete the task.

Concerning the production code:

- What does the class Reverse do? Can you briefly describe its implementation?

Concerning the test code:

- What does the class Reversetest do? Can you briefly describe its implementation?

When you are ready:

Click here to start: Task 1

* 11 ***After the experiment is completed answer the following questions.***

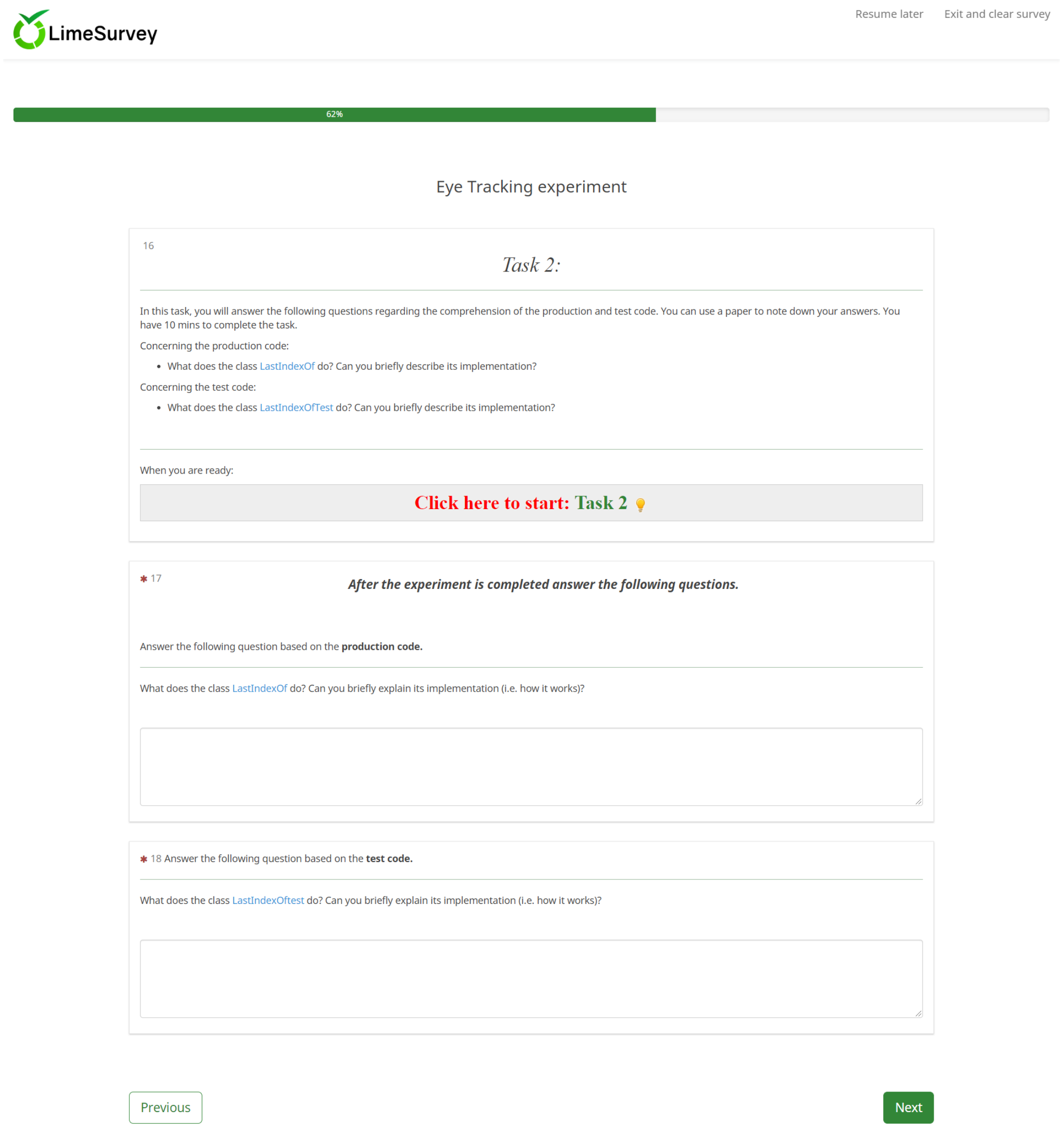
Answer the following question based on the **production code.**

What does the class Reverse do? Can you briefly explain its implementation (i.e. how it works)?

* 12 Answer the following question based on the **test code.**

What does the class ReverseTest do? Can you briefly explain its implementation (i.e. how it works)?

Previous   Next


LimeSurvey

Resume later Exit and clear survey

62%

Eye Tracking experiment

16

Task 2:

In this task, you will answer the following questions regarding the comprehension of the production and test code. You can use a paper to note down your answers. You have 10 mins to complete the task.

Concerning the production code:

- What does the class LastIndexOf do? Can you briefly describe its implementation?

Concerning the test code:

- What does the class LastIndexOfTest do? Can you briefly describe its implementation?

When you are ready:

Click here to start: Task 2

* 17 ***After the experiment is completed answer the following questions.***

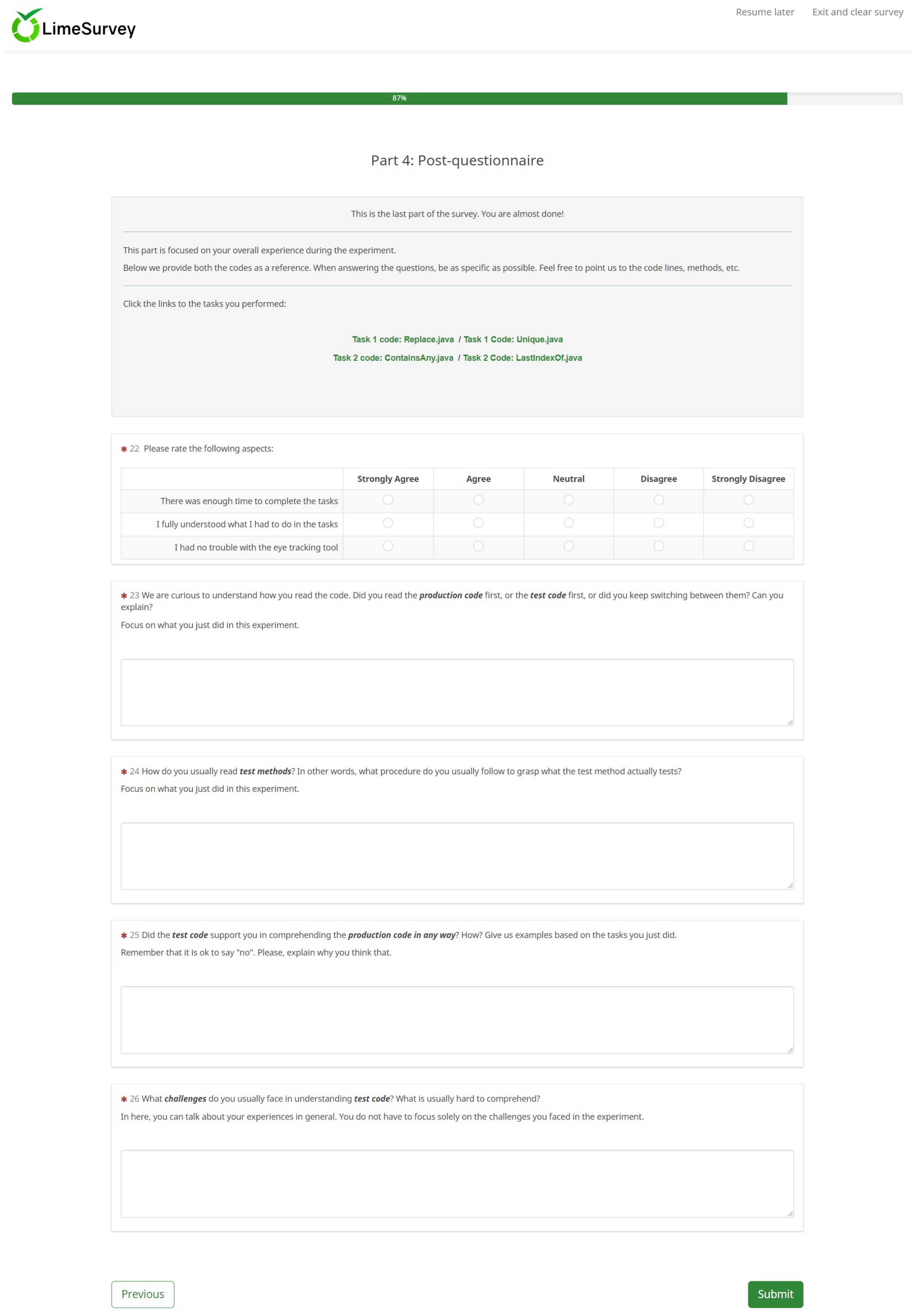
Answer the following question based on the **production code.**

What does the class LastIndexOf do? Can you briefly explain its implementation (i.e. how it works)?

* 18 Answer the following question based on the **test code.**

What does the class LastIndexOftest do? Can you briefly explain its implementation (i.e. how it works)?

Previous Next



Resume later Exit and clear survey

87%

Part 4: Post-questionnaire

This is the last part of the survey. You are almost done!

This part is focused on your overall experience during the experiment.
Below we provide both the codes as a reference. When answering the questions, be as specific as possible. Feel free to point us to the code lines, methods, etc.

Click the links to the tasks you performed:

Task 1 code: Replace.java / **Task 1 Code: Unique.java**
Task 2 code: ContainsAny.java / **Task 2 Code: LastIndexOf.java**

* 22 Please rate the following aspects:

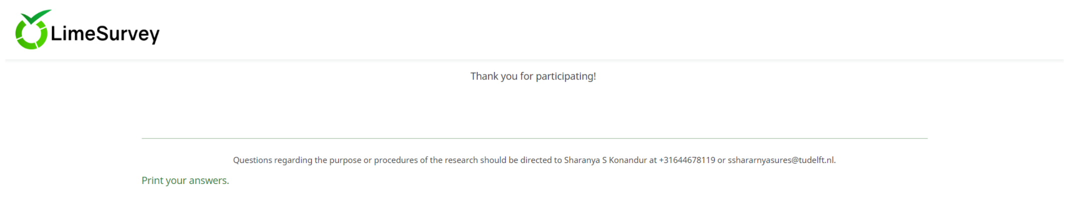
	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
There was enough time to complete the tasks	○	○	○	○	○
I fully understood what I had to do in the tasks	○	○	○	○	○
I had no trouble with the eye tracking tool	○	○	○	○	○

* 23 We are curious to understand how you read the code. Did you read the ***production code*** first, or the ***test code*** first, or did you keep switching between them? Can you explain?
Focus on what you just did in this experiment.

* 24 How do you usually read ***test methods***? In other words, what procedure do you usually follow to grasp what the test method actually tests?
Focus on what you just did in this experiment.

* 25 Did the ***test code*** support you in comprehending the ***production code in any way***? How? Give us examples based on the tasks you just did.
Remember that it is ok to say "no". Please, explain why you think that.

* 26 What ***challenges*** do you usually face in understanding ***test code***? What is usually hard to comprehend?
In here, you can talk about your experiences in general. You do not have to focus solely on the challenges you faced in the experiment.


LimeSurvey

Thank you for participating!

Questions regarding the purpose or procedures of the research should be directed to Sharanya S Konandur at +31644678119 or ssharamyasures@tudelft.nl.

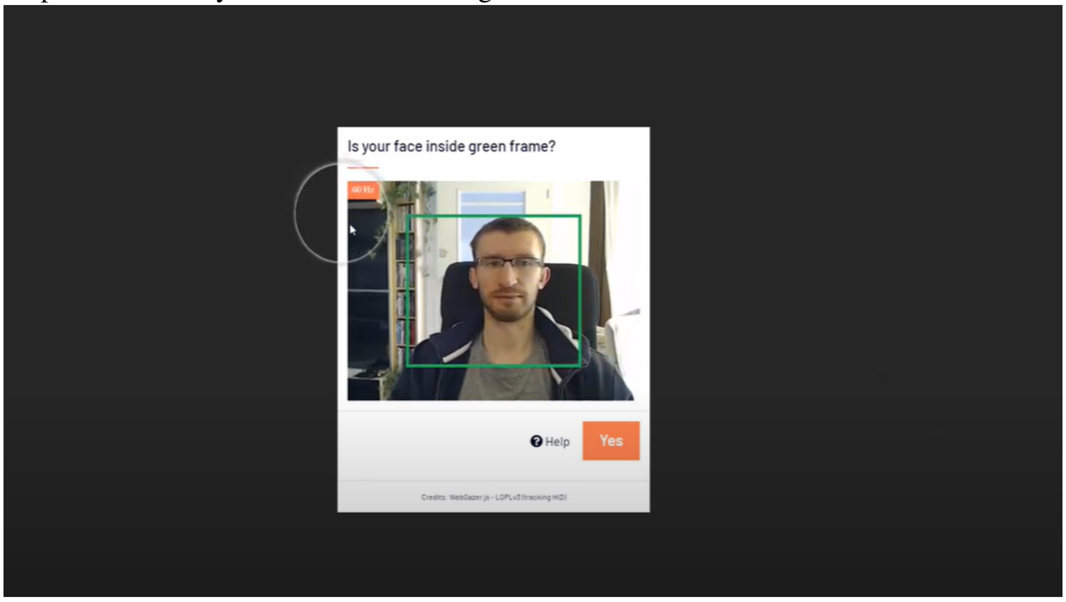
Print your answers.

A.2 Eye tracking experiment

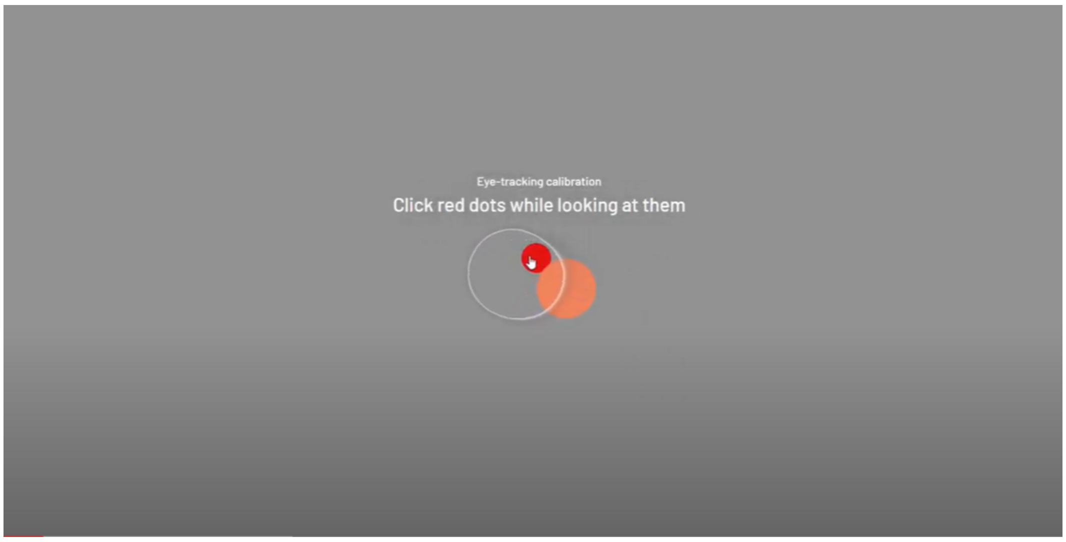
Below we provide the steps involved in eye tracking experiment.

A.3 Procedure

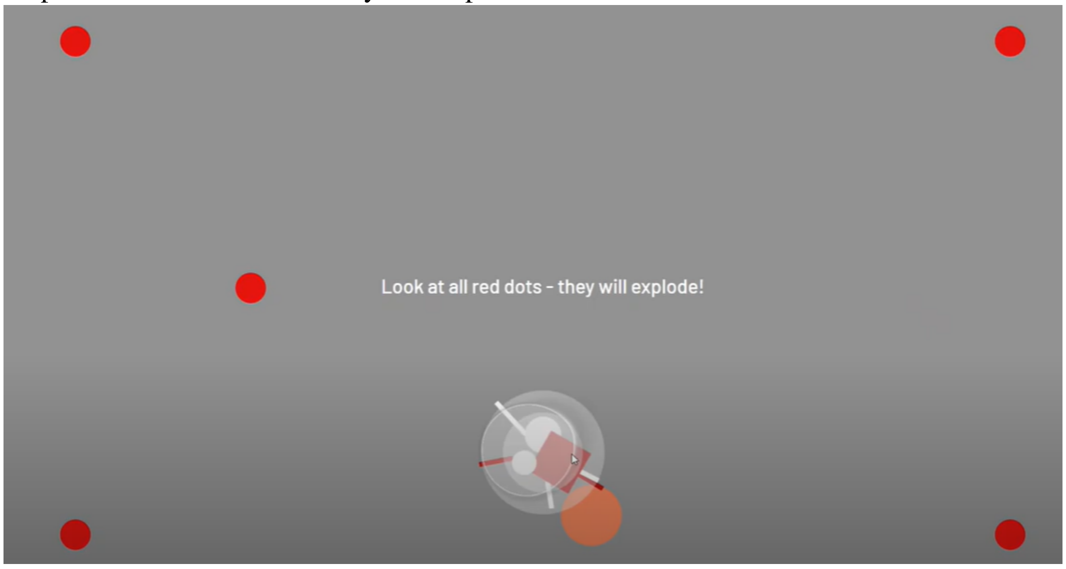
- Step1: Make sure your face is inside the green frame at all times.


Is your face inside green frame?
60 Hz
Help
Yes
Credits: WebGazer.js - LGPLv3 (tracking HID)

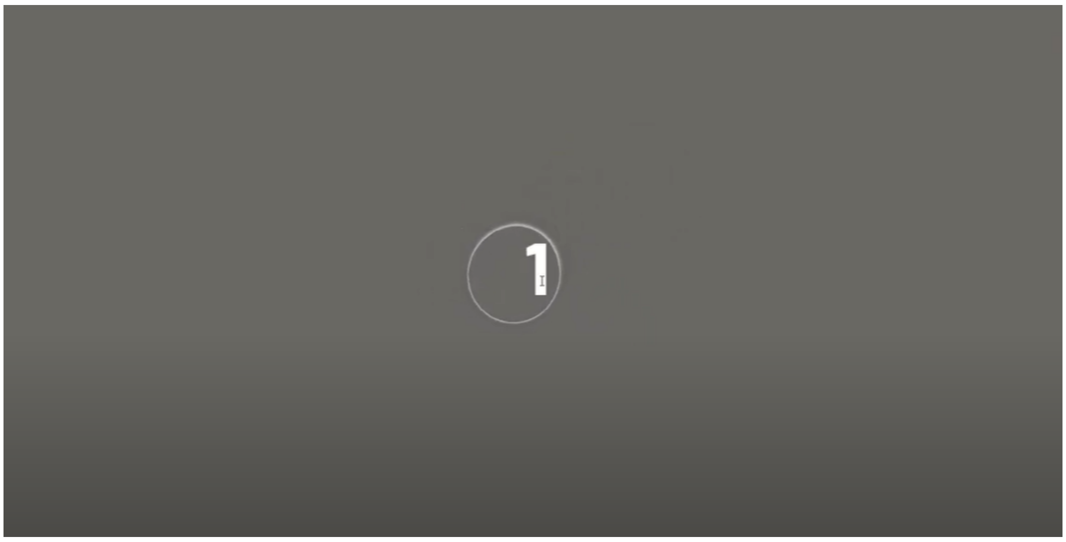
- Step 2: Click the red dotes while looking at them to start eye-tracking calibration.



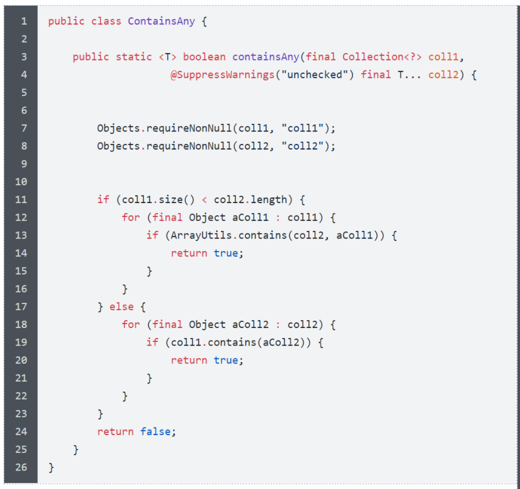
- Step 3: Look at all the dots-they will explode.



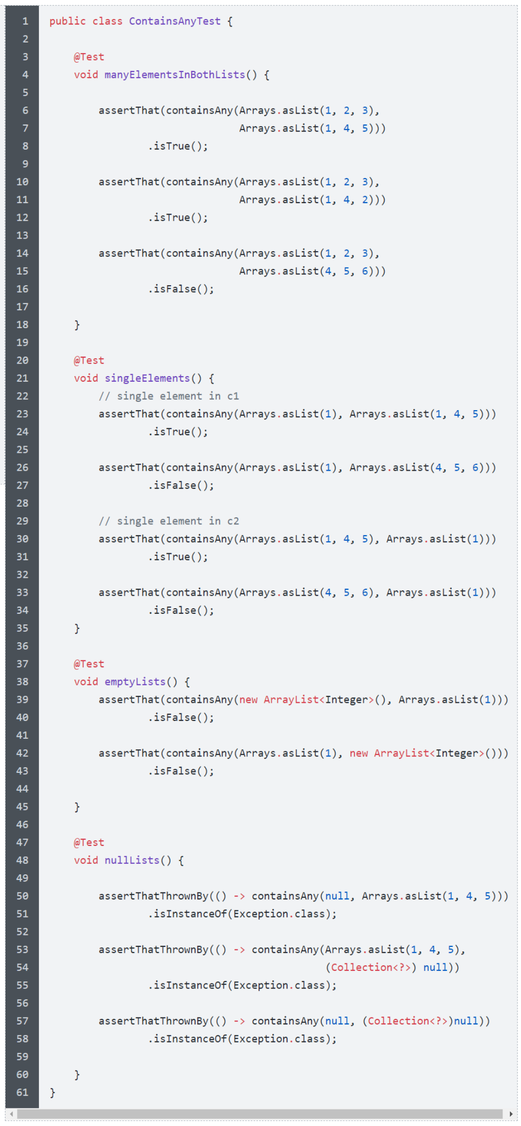
- Step 4: Start the experiment in 3..2..1..



- Step 5: Read and comprehend the code.

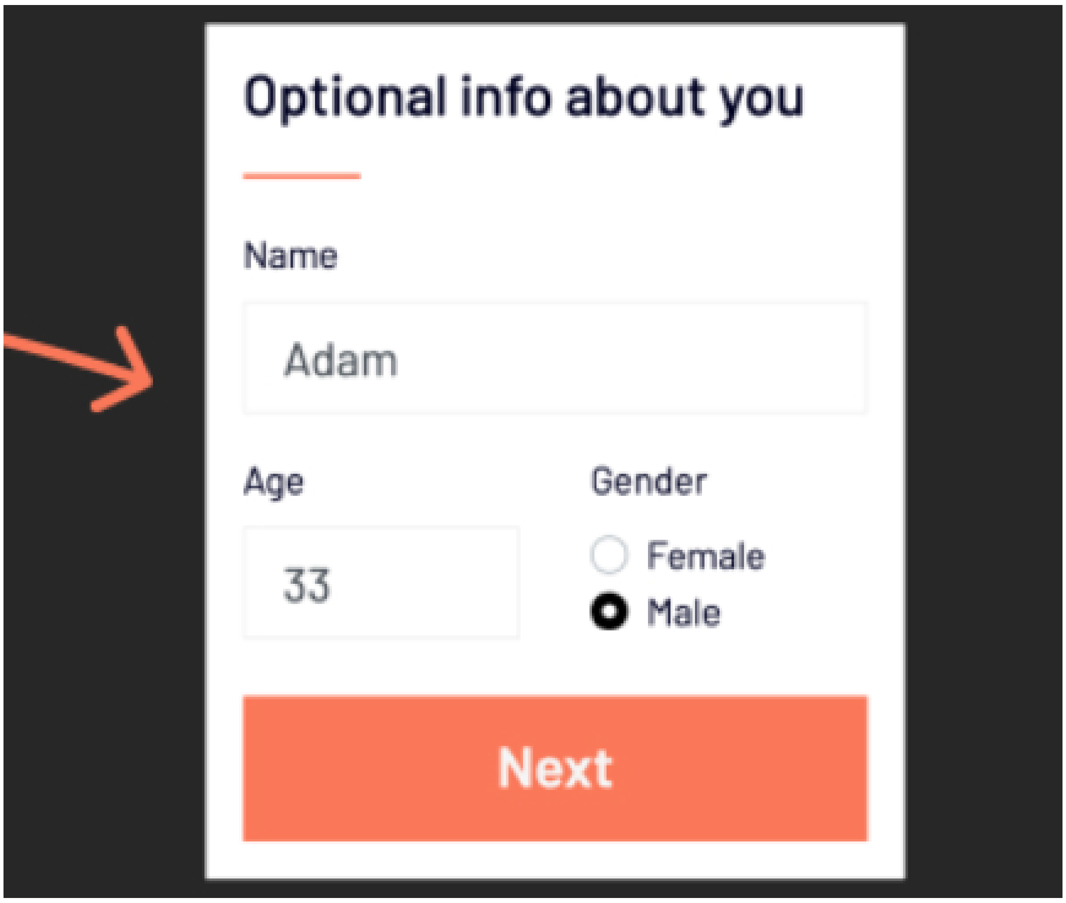


```java
public class ContainsAny {

    public static <T> boolean containsAny(final Collection<?> coll1,
                    @SuppressWarnings("unchecked") final T... coll2) {


        Objects.requireNonNull(coll1, "coll1");
        Objects.requireNonNull(coll2, "coll2");


        if (coll1.size() < coll2.length) {
            for (final Object aColl1 : coll1) {
                if (ArrayUtils.contains(coll2, aColl1)) {
                    return true;
                }
            }
        } else {
            for (final Object aColl2 : coll2) {
                if (coll1.contains(aColl2)) {
                    return true;
                }
            }
        }
        return false;
    }
}
```



```java
public class ContainsAnyTest {

    @Test
    void manyElementsInBothLists() {

        assertThat(containsAny(Arrays.asList(1, 2, 3),
                               Arrays.asList(1, 4, 5)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(1, 2, 3),
                               Arrays.asList(1, 4, 2)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(1, 2, 3),
                               Arrays.asList(4, 5, 6)))
                .isFalse();

    }

    @Test
    void singleElements() {
        // single element in c1
        assertThat(containsAny(Arrays.asList(1), Arrays.asList(1, 4, 5)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(1), Arrays.asList(4, 5, 6)))
                .isFalse();

        // single element in c2
        assertThat(containsAny(Arrays.asList(1, 4, 5), Arrays.asList(1)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(4, 5, 6), Arrays.asList(1)))
                .isFalse();
    }

    @Test
    void emptyLists() {
        assertThat(containsAny(new ArrayList<Integer>(), Arrays.asList(1)))
                .isFalse();

        assertThat(containsAny(Arrays.asList(1), new ArrayList<Integer>()))
                .isFalse();

    }

    @Test
    void nullLists() {

        assertThatThrownBy(() -> containsAny(null, Arrays.asList(1, 4, 5)))
                .isInstanceOf(Exception.class);

        assertThatThrownBy(() -> containsAny(Arrays.asList(1, 4, 5),
                                             (Collection<?>) null))
                .isInstanceOf(Exception.class);

        assertThatThrownBy(() -> containsAny(null, (Collection<?>)null))
                .isInstanceOf(Exception.class);

    }
}
```

- Step 6: Provide your full name (same as the consent form) so that we can match the results to the survey.



Optional info about you

Name: Adam

Age: 33

Gender: Female [✓ Male]

Appendix B

Data Analysis

In this chapter we provide all the data collected from the tools.

B.1 Gaze data

Below we provide the gaze data collected from all the participants which has been updated with YouTube links to get easy access. The data is anatomized to preserve the identities of the participants.

participant_id	**task_id**	**gaze video**
	Noivce	
1392341774	1	https://youtu.be/czughxmi1G0
1814760247	1	https://youtu.be/gUw9zbh-nm4
867061267	1	https://youtu.be/4An1HKhC9_I
799810770	1	https://youtu.be/ZV9ExePK−−k
1787537632	1	https://youtu.be/X3OuIG6BfSk
1444712459	1	https://youtu.be/OPLU-2LPunk
1916717354	2	https://youtu.be/N4oA0HBbtM8
1213879844	2	https://youtu.be/mzMbNdGvF-I
2097377084	2	https://youtu.be/0E-_8U34a_Q
1590805764	2	https://youtu.be/MCWh6750iBo
859391687	2	https://youtu.be/2eTMVD5itn0
1470195088	2	https://youtu.be/wlp1Jo_G6kQ
1860228703	2	https://youtu.be/A2G2KCMWegI
615697347	2	https://youtu.be/dGjjX5F_RwY
1373918450	2	https://youtu.be/8_0teSiOm_Q
1392341774	3	https://youtu.be/IRG1_8P9JZ4
1814760247	3	https://youtu.be/TWoCsEtigmk
867061267	3	https://youtu.be/ZwjcUndcm28
799810770	3	https://youtu.be/koHlKj4JmlE
1787537632	3	https://youtu.be/sTRIEhoE_3Y

Table B.1 continued from previous page

participant_id	task_id	gaze video
1444712459	3	https://youtu.be/dyFiIDoBTgw
1916717354	4	https://youtu.be/HlHY8VAV6CI
1213879844	4	https://youtu.be/lREuK4Ltb_k
2097377084	4	https://youtu.be/R2hCpnERMQA
1590805764	4	https://youtu.be/Ybfb5LFyrHQ
859391687	4	https://youtu.be/FjZM_vgunsw
1470195088	4	https://youtu.be/kIn6GOcICbI
1860228703	4	https://youtu.be/iCljsr3W6q8
615697347	4	https://youtu.be/gJwb9oCuCTE
1373918450	4	https://youtu.be/XkiuI9jSJfE
Expert		
1874230670	1	https://youtu.be/QQRwq3ID_nU
1761982197	1	https://youtu.be/8TGFoWVwApk
1644468044	1	https://youtu.be/3rgf2H3HpAQ
1110950035	1	https://youtu.be/VLVm9fY5FUA
1792232827	1	https://youtu.be/JmrjpEWJH9E
719395131	1	https://youtu.be/O9NCMq8sHeI
1527991542	1	https://youtu.be/9UEHx7OCCls
777734110	1	https://youtu.be/eLqzxgWcjjY
1565132263	1	https://youtu.be/OWio2TqMkVM
225699910	2	https://youtu.be/jdDkYRyY3VQ
784768767	2	https://youtu.be/AzaM2wUgjwY
603351179	2	https://youtu.be/IE3vKwJRu3c
175830142	2	https://youtu.be/MCWh6750iBo
1527991542	2	https://youtu.be/q9FpakziMKM
1874230670	3	https://youtu.be/xWw01PukEbk
1761982197	3	https://youtu.be/orhrDHNQysQ
1644468044	3	https://youtu.be/O-So5c-pFD0
1110950035	3	https://youtu.be/buBfDrZDOkU
1792232827	3	https://youtu.be/9b2Cqh_4YkY
719395131	3	https://youtu.be/YLI4Wv51zHE
1527991542	3	https://youtu.be/uvPbA7_Y5rI
777734110	3	https://youtu.be/HI_zB6jaLBM
1565132263	3	https://youtu.be/zJs2aD-ZjEA
1110950035	4	https://youtu.be/cJOVivcpqrM
603351179	4	https://youtu.be/oz59-fBIpl8
1792232827	4	https://youtu.be/SDqSP5uJ6fo
719395131	4	https://youtu.be/hVg2S4445rI
1527991542	4	https://youtu.be/vfRV9-kV3UA

Table B.1: Gaze videos of all participants

B.2 Participant's Data

In this section we provide the participant's background data.

Participant No.	Male	Female	Student	Developer	Tester	Years_exp	Lvl_exp_junit	Lvl_exp_java
P1	Yes	No	No	Yes	No	5.5	4	4
P2	Yes	No	No	Yes	No	6	4	4
P3	Yes	No	No	Yes	No	3	2	4
P4	Yes	No	Yes	No	No	2.6	2	4
P5	No	Yes	No	No	Yes	2	2	3
P6	Yes	No	No	Yes	No	3	3	4
P7	Yes	No	No	Yes	No	3	3	4
P8	No	Yes	Yes	No	No	2	2	2
P9	No	Yes	Yes	No	No	2	1	2
P10	No	Yes	Yes	No	No	1	4	3
P11	No	Yes	No	Yes	No	4	2	4
P12	Yes	No	No	Yes	No	4.5	3	4
P13	Yes	No	Yes	No	No	1	1	3
P14	No	Yes	Yes	No	No	2.4	1	2
P15	No	Yes	Yes	No	No	1	2	3
P16	Yes	No	Yes	No	No	2.5	2	3
P17	Yes	No	Yes	No	No	4.5	3	4
P18	Yes	No	No	Yes	No	3	3	3
P19	Yes	No	No	Yes	No	9.5	4	4
P20	No	Yes	Yes	No	No	1	1	2
P21	Yes	No	No	Yes	No	3	1	4
P22	No	Yes	No	Yes	No	4	3	4
P23	Yes	No	Yes	No	No	2	1	2
P24	No	Yes	No	Yes	No	3	2	4
P25	No	No	Yes	No	No	1	2	2
P26	Yes	No	Yes	No	No	1	3	4
P27	Yes	No	No	No	Yes	3	2	3
P28	Yes	No	Yes	No	No	1	2	3
P29	Yes	No	Yes	No	No	1	1	3

Table B.2: Participant's data

B.3 Analysis of Gaze videos

participant_id	task_id	source switch count	test switch count	assert time taken	source time	test time	test read	total switch
				Novice				
1392341774	1	9	10	1.45	1.33	2.01	100	1.33
1814760247	1	9	8	1.39	2.07	1.21	100	2.07
867061267	1	5	4	1.02	0.54	1.4	100	0.54
799810770	1	7	9	1.25	1.14	1.36	100	1.14
1787537632	1	8	7	1.13	1.03	1.35	80	1.03
1444712459	1	6	5	1.54	1.16	2.06	100	1.16
1916717354	2	7	8	0.5	0.5	1.33	40	0.5
1213879844	2	8	8	0.34	0.42	0.53	100	0.42
2097377084	2	10	11	1.28	1.03	1.43	50	1.03
1590805764	2	7	6	0.3	0.54	0.42	100	0.54
859391687	2	8	7	0.55	1.13	1.14	100	1.13
1470195088	2	5	6	1.44	1	2.03	100	1
1860228703	2	9	8	1.12	1.05	1.35	100	1.05
615697347	2	4	4	1	1.43	1.13	100	1.43
1373918450	2	7	7	1.33	0.53	1.56	100	0.53
1392341774	3	8	23	1.21	1.19	2.11	100	1.19
1814760247	3	4	7	0.48	1.43	1.03	100	1.43
867061267	3	3	5	0.35	0.56	2.04	100	0.56
799810770	3	6	6	1.09	1.32	1.45	80	1.32
1787537632	3	11	10	0.44	1.35	1.04	100	1.35
1444712459	3	7	8	1.03	1.5	1.24	100	1.5
1916717354	4	8	10	1.15	1.29	1.44	100	1.29
1213879844	4	7	8	0.46	1.23	1.04	100	1.23
2097377084	4	8	6	1.21	0.57	1.32	100	0.57
1590805764	4	6	7	0.21	0.56	0.41	100	0.56
859391687	4	3	2	1.07	0.32	1.34	100	0.32
1470195088	4	6	8	1	0.46	1.18	0	0.46
1860228703	4	8	10	1.17	1.02	1.42	100	1.02
615697347	4	3	5	1.47	1.21	1.52	100	1.21
1373918450	4	4	4	1.06	1.03	1.27	100	1.03
				Expert				
1874230670	1	5	6	0.34	1.29	0.48	50	1.29
1761982197	1	6	6	0.37	0.32	0.55	75	0.32
1644468044	1	10	11	0.17	0.36	0.32	75	0.36
1110950035	1	6	7	0.23	0.45	0.45	100	0.45
1792232827	1	7	8	0.23	0.51	0.31	100	0.51
719395131	1	8	8	0.38	1.03	0.54	50	1.03
1527991542	1	7	7	0.33	0.54	0.45	100	0.54
777734110	1	4	6	0.17	0.51	0.3	75	0.51

Table B.3 continued from previous page

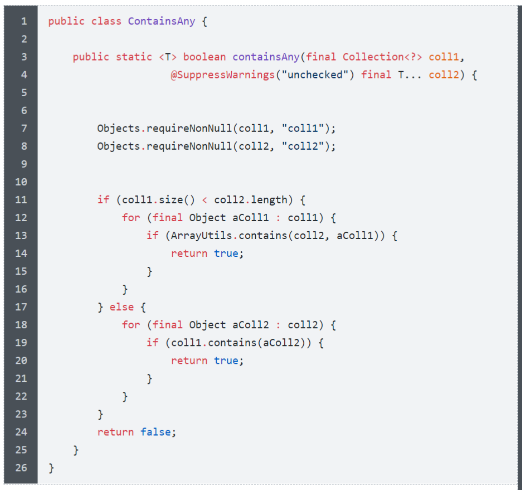
participant_id	**task_id**	**source switch count**	**test switch count**	**assert time taken**	**source time**	**test time**	**test read**	**total switch**
1565132263	1	8	9	0.35	1.04	0.45	75	1.04
225699910	2	10	11	0.49	1.02	1	66	1.02
784768767	2	4	5	0.28	1.03	0.48	83	1.03
603351179	2	8	7	1.21	0.59	1.48	50	0.59
175830142	2	9	11	1.52	1.02	2.25	66	1.02
1527991542	2	11	9	0.25	1.19	0.4	100	1.19
1874230670	3	8	9	0.44	0.21	1.04	60	0.21
1761982197	3	6	8	1.49	0.36	2.01	70	0.36
1644468044	3	12	11	0.31	0.5	0.41	45	0.5
1110950035	3	7	9	0.46	0.42	0.57	60	0.42
1792232827	3	8	7	0.14	0.34	0.39	60	0.34
719395131	3	6	7	0.14	1.03	0.29	70	1.03
1527991542	3	8	10	0.25	0.54	0.33	80	0.54
777734110	3	8	7	0.43	0.35	0.51	40	0.35
1565132263	3	6	7	0.37	1.16	0.44	80	1.16
1110950035	4	7	6	0.31	0.19	0.43	63	0.19
603351179	4	11	9	0.46	0.31	1.02	54	0.31
1792232827	4	8	7	0.43	0.21	0.57	60	0.21
719395131	4	6	7	1.02	0.36	1.23	81	0.36
1527991542	4	12	8	0.18	0.41	0.35	81	0.41

Table B.3: Gaze video analysis

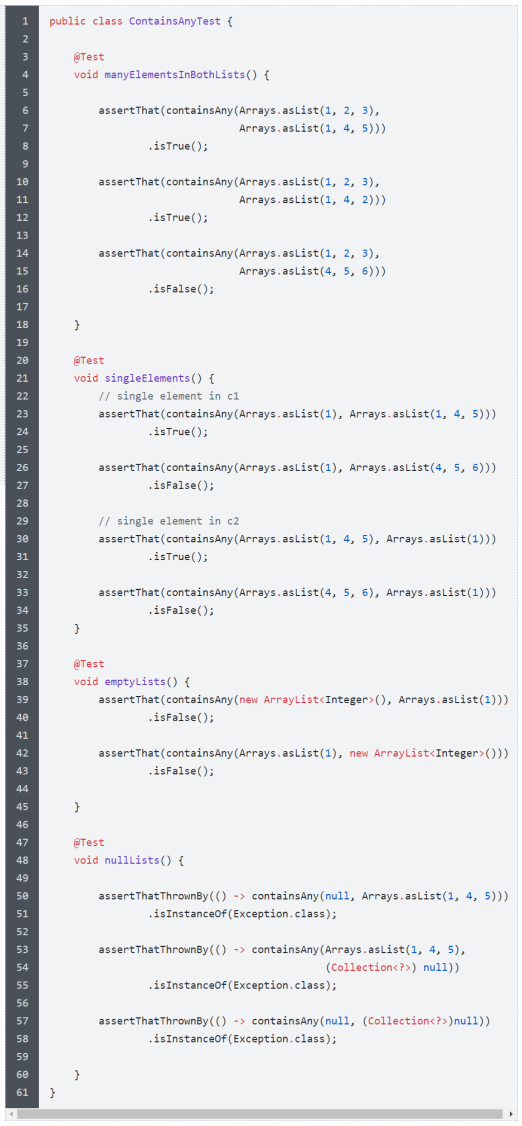
Appendix C

Experimental Codes

C.1 ContainsAny

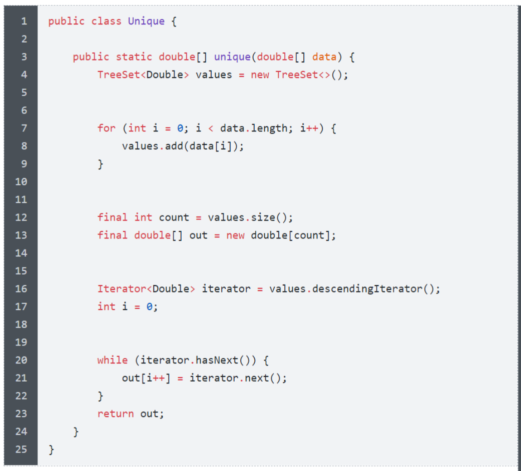


```java
public class ContainsAny {

    public static <T> boolean containsAny(final Collection<?> coll1,
                    @SuppressWarnings("unchecked") final T... coll2) {


        Objects.requireNonNull(coll1, "coll1");
        Objects.requireNonNull(coll2, "coll2");


        if (coll1.size() < coll2.length) {
            for (final Object aColl1 : coll1) {
                if (ArrayUtils.contains(coll2, aColl1)) {
                    return true;
                }
            }
        } else {
            for (final Object aColl2 : coll2) {
                if (coll1.contains(aColl2)) {
                    return true;
                }
            }
        }
        return false;
    }
}
```

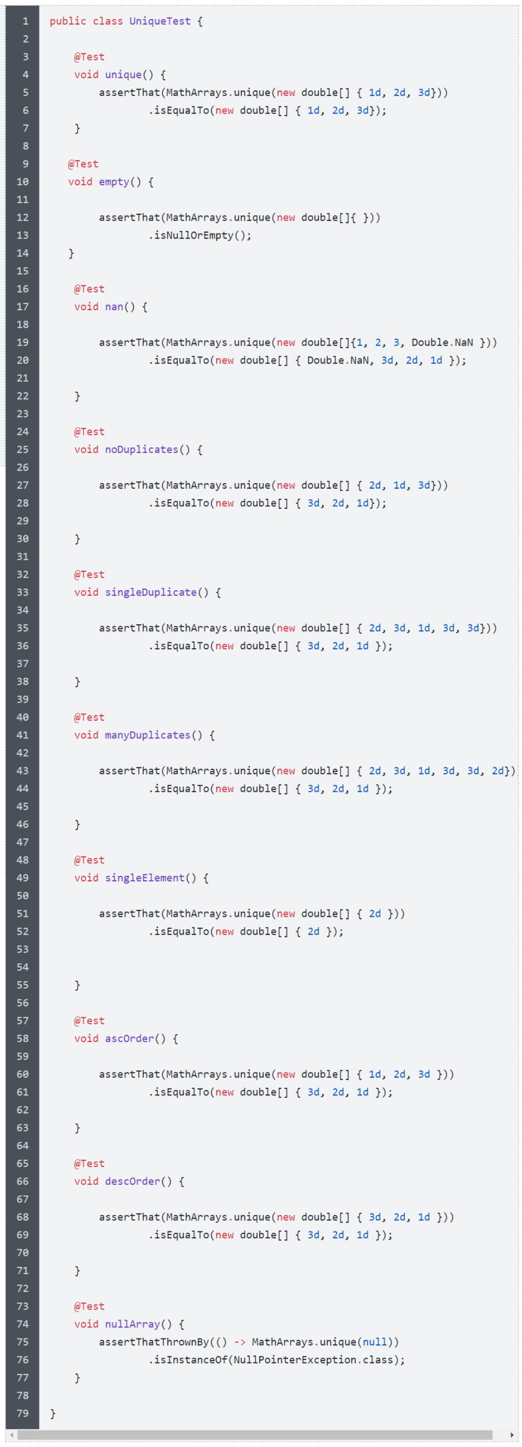


```java
public class ContainsAnyTest {

    @Test
    void manyElementsInBothLists() {

        assertThat(containsAny(Arrays.asList(1, 2, 3),
                               Arrays.asList(1, 4, 5)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(1, 2, 3),
                               Arrays.asList(1, 4, 2)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(1, 2, 3),
                               Arrays.asList(4, 5, 6)))
                .isFalse();

    }

    @Test
    void singleElements() {
        // single element in c1
        assertThat(containsAny(Arrays.asList(1), Arrays.asList(1, 4, 5)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(1), Arrays.asList(4, 5, 6)))
                .isFalse();

        // single element in c2
        assertThat(containsAny(Arrays.asList(1, 4, 5), Arrays.asList(1)))
                .isTrue();

        assertThat(containsAny(Arrays.asList(4, 5, 6), Arrays.asList(1)))
                .isFalse();
    }

    @Test
    void emptyLists() {
        assertThat(containsAny(new ArrayList<Integer>(), Arrays.asList(1)))
                .isFalse();

        assertThat(containsAny(Arrays.asList(1), new ArrayList<Integer>()))
                .isFalse();

    }

    @Test
    void nullLists() {

        assertThatThrownBy(() -> containsAny(null, Arrays.asList(1, 4, 5)))
                .isInstanceOf(Exception.class);

        assertThatThrownBy(() -> containsAny(Arrays.asList(1, 4, 5),
                                             (Collection<?>) null))
                .isInstanceOf(Exception.class);

        assertThatThrownBy(() -> containsAny(null, (Collection<?>)null))
                .isInstanceOf(Exception.class);

    }
}
```

C.2 Unique



```java
public class Unique {

    public static double[] unique(double[] data) {
        TreeSet<Double> values = new TreeSet<>();


        for (int i = 0; i < data.length; i++) {
            values.add(data[i]);
        }


        final int count = values.size();
        final double[] out = new double[count];


        Iterator<Double> iterator = values.descendingIterator();
        int i = 0;


        while (iterator.hasNext()) {
            out[i++] = iterator.next();
        }
        return out;
    }
}
```



```java
public class UniqueTest {

    @Test
    void unique() {
        assertThat(MathArrays.unique(new double[] { 1d, 2d, 3d}))
                .isEqualTo(new double[] { 1d, 2d, 3d});
    }

   @Test
   void empty() {

        assertThat(MathArrays.unique(new double[]{ }))
                .isNullOrEmpty();
   }

    @Test
    void nan() {

        assertThat(MathArrays.unique(new double[]{1, 2, 3, Double.NaN }))
                .isEqualTo(new double[] { Double.NaN, 3d, 2d, 1d });

    }

    @Test
    void noDuplicates() {

        assertThat(MathArrays.unique(new double[] { 2d, 1d, 3d}))
                .isEqualTo(new double[] { 3d, 2d, 1d});

    }

    @Test
    void singleDuplicate() {

        assertThat(MathArrays.unique(new double[] { 2d, 3d, 1d, 3d, 3d}))
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void manyDuplicates() {

        assertThat(MathArrays.unique(new double[] { 2d, 3d, 1d, 3d, 3d, 2d})
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void singleElement() {

        assertThat(MathArrays.unique(new double[] { 2d }))
                .isEqualTo(new double[] { 2d });


    }

    @Test
    void ascOrder() {

        assertThat(MathArrays.unique(new double[] { 1d, 2d, 3d }))
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void descOrder() {

        assertThat(MathArrays.unique(new double[] { 3d, 2d, 1d }))
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void nullArray() {
        assertThatThrownBy(() -> MathArrays.unique(null))
                .isInstanceOf(NullPointerException.class);
    }

}
```

Click here to exit

C.3 Reverse

```java
public class Reverse {

    public static void reverse(final boolean[] array,
                               final int startIndexInclusive,
                               final int endIndexExclusive) {
        if (array == null) {
            return;
        }
        int i = Math.max(startIndexInclusive, 0);
        int j = Math.min(array.length, endIndexExclusive) - 1;
        boolean tmp;
        while (j > i) {
            tmp = array[j];
            array[j] = array[i];
            array[i] = tmp;
            j--;
            i++;
        }
    }
}
```

```java
public class ReverseTest {

    @Test
    void startIndex() {
        // startIndex < endIndex
        int[] array1 = ints(1, 2, 3, 4, 5);
        reverse(array1, 1, 4);
        assertThat(array1)
                .isEqualTo(ints(1, 4, 3, 2, 5));

        // startIndex > endIndex
        int[] array2 = ints(1, 2, 3, 4, 5);
        reverse(array2, 4, 1);
        assertThat(array2)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // startIndex == endIndex
        int[] array3 = ints(1, 2, 3, 4, 5);
        reverse(array3, 1, 1);
        assertThat(array3)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // startIndex zero
        int[] array4 = ints(1, 2, 3, 4, 5);
        reverse(array4, 0, 4);
        assertThat(array4)
                .isEqualTo(ints(4, 3, 2, 1, 5));
    }

    @Test
    void endIndex() {
        // endIndex zero
        int[] array1 = ints(1, 2, 3, 4, 5);
        reverse(array1, 0, 0);
        assertThat(array1)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // endIndex larger than array size
        int[] array2 = ints(1, 2, 3, 4, 5);
        reverse(array2, 1, 10);
        assertThat(array2)
                .isEqualTo(ints(1, 5, 4, 3, 2));

        // endIndex precisely array size
        int[] array3 = ints(1, 2, 3, 4, 5);
        reverse(array3, 1, 5);
        assertThat(array3)
                .isEqualTo(ints(1, 5, 4, 3, 2));

        // endIndex last index
        int[] array4 = ints(1, 2, 3, 4, 5);
        reverse(array4, 1, 4);
        assertThat(array4)
                .isEqualTo(ints(1, 4, 3, 2, 5));
    }

    @Test
    void indexBoundaries() {
        // negative start index
        int[] array1 = ints(1, 2, 3, 4, 5);
        reverse(array1, -1, 3);
        assertThat(array1)
                .isEqualTo(ints(3, 2, 1, 4, 5));

        // negative end index
        int[] array2 = ints(1, 2, 3, 4, 5);
        reverse(array2, 1, -1);
        assertThat(array2)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // startIndex larger than array size
        int[] array3 = ints(1, 2, 3, 4, 5);
        reverse(array3, 10, 3);
        assertThat(array3)
                .isEqualTo(ints(1, 2, 3, 4, 5));
    }

    private static int[] ints(int... nums) {
        return nums;
    }
}
```

Click here to exit

C.4 LastIndexOf

```java
public class LastIndexOf {

    public static final int INDEX_NOT_FOUND = -1;

    public static int lastIndexOf
        (final byte[] array, final byte valueToFind, int startIndex) {


        if (array == null) {
            return INDEX_NOT_FOUND;
        }


        if (startIndex < 0) {
            return INDEX_NOT_FOUND;
        } else if (startIndex >= array.length) {
            startIndex = array.length - 1;
        }


        for (int i = startIndex; i >= 0; i--) {
            if (valueToFind == array[i]) {
                return i;
            }
        }
        return INDEX_NOT_FOUND;
    }
}
```

```java
public class LastIndexOfTest {

    @Test
    void nullArray() {

        assertThat(lastIndexOf((int[])null, 1, 2))
                .isEqualTo(INDEX_NOT_FOUND);

    }

    @Test
    void negativeIndex() {

        assertThat(lastIndexOf(values(0, 1, 2), 2, -1))
                .isEqualTo(INDEX_NOT_FOUND);

    }

    @Test
    void emptyArray() {

        assertThat(lastIndexOf(values(), 0, 1))
                .isEqualTo(INDEX_NOT_FOUND);

    }


    @Test
    void singleElementArray() {

        assertThat(lastIndexOf(values(1), 1, 0))
                .isEqualTo(0);

        assertThat(lastIndexOf(values(1), 2, 0))
                .isEqualTo(INDEX_NOT_FOUND);

    }

    @Test
    void multipleElementsArray() {

        assertThat(lastIndexOf(values(0, 1, 2), 1, 2))
                .isEqualTo(1);

        assertThat(lastIndexOf(values(0, 1, 1, 2), 1, 2))
                .isEqualTo(2);

        assertThat(lastIndexOf(values(0, 1, 2), 3, 0))
                .isEqualTo(INDEX_NOT_FOUND);

    }

    @Test
    void index() {

        // index 0
        assertThat(lastIndexOf(values(0, 1, 2), 0, 0))
                .isEqualTo(0);

        assertThat(lastIndexOf(values(0, 1, 2), 3, 0))
                .isEqualTo(INDEX_NOT_FOUND);

        // index bigger than array
        assertThat(lastIndexOf(values(0, 1, 2), 1, 5))
                .isEqualTo(1);

        // element at index
        assertThat(lastIndexOf(values(0, 1, 2), 1, 1))
                .isEqualTo(1);

    }


    private int[] values(int... array) {
        return array;
    }

}
```

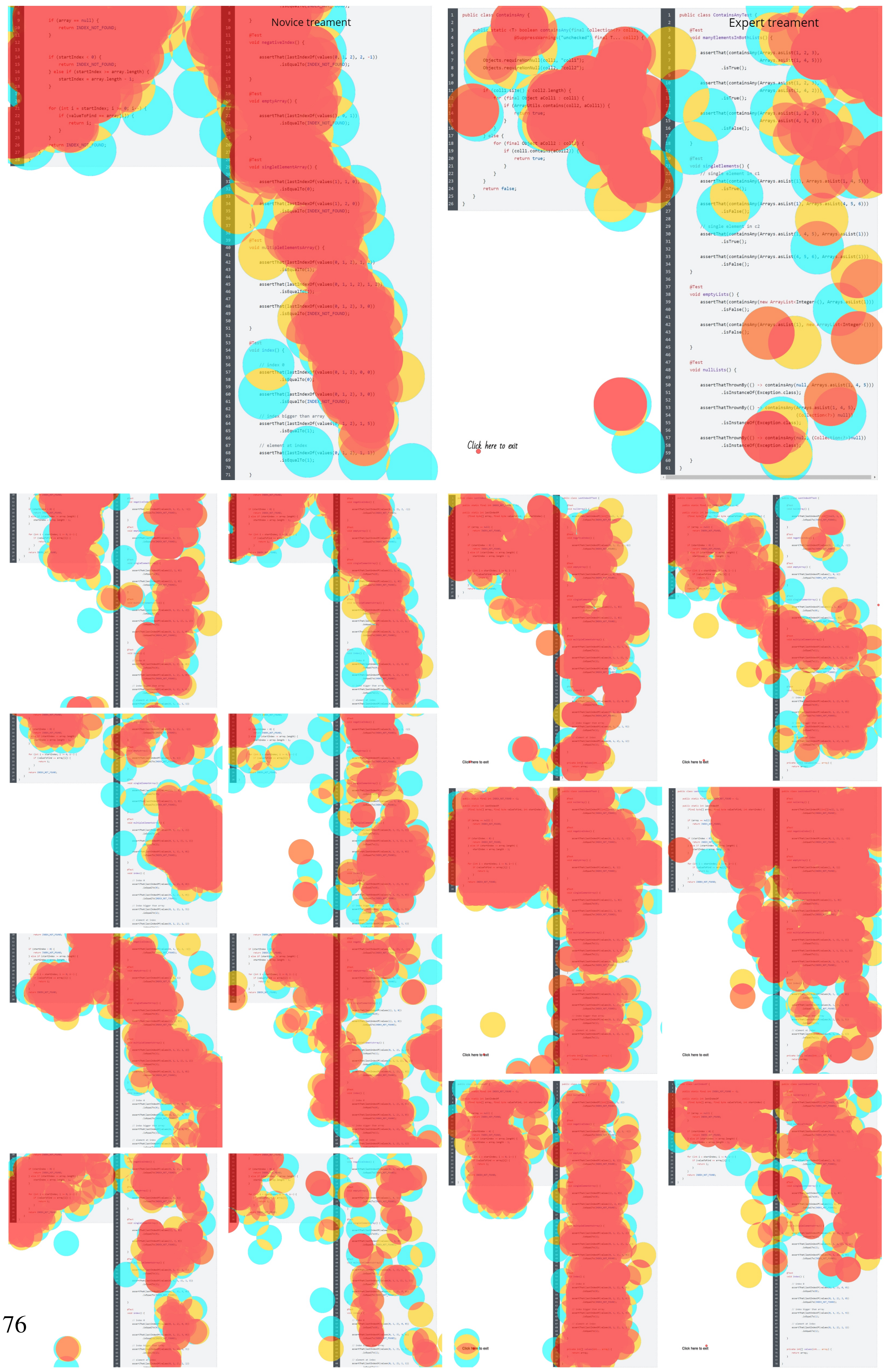
Click here to exit

Appendix D

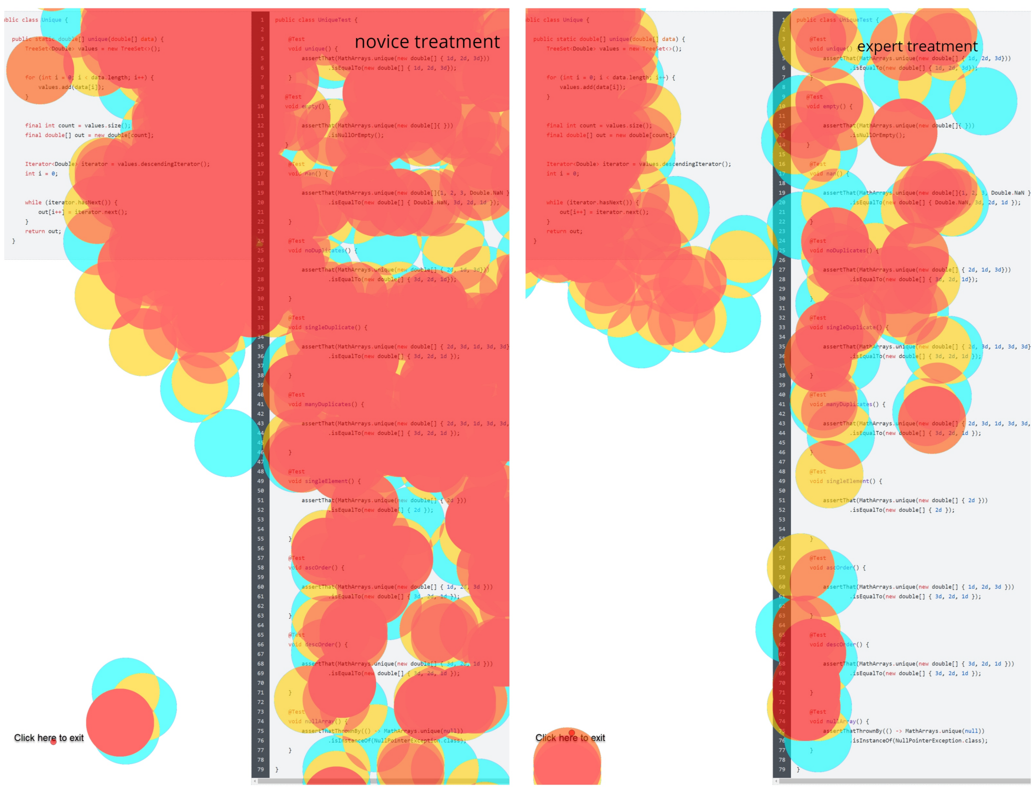
Heat Maps

In this section we show differences in heat maps for each code between novice and experts.

D.1 ContainsAny

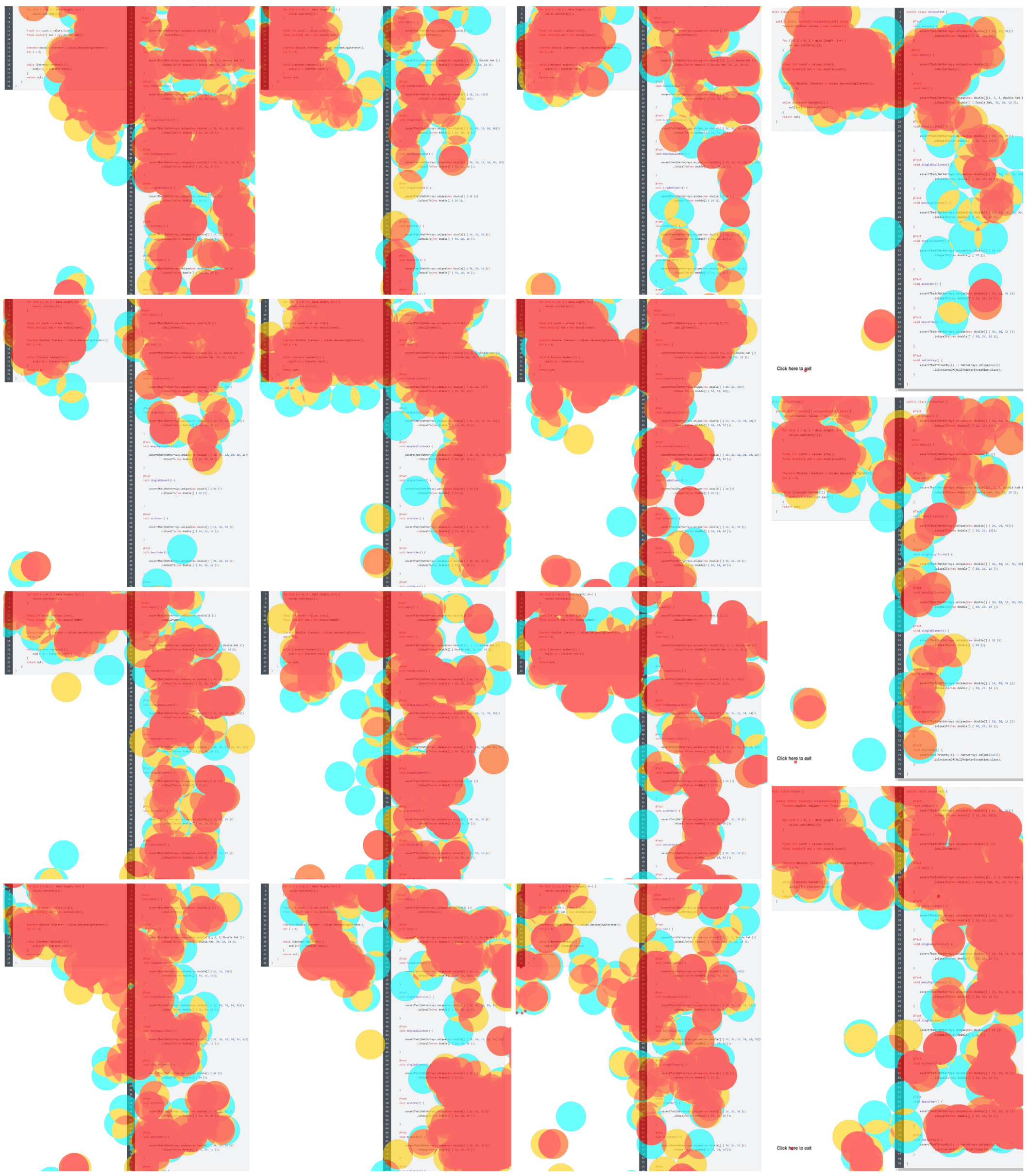


D.2 Unique

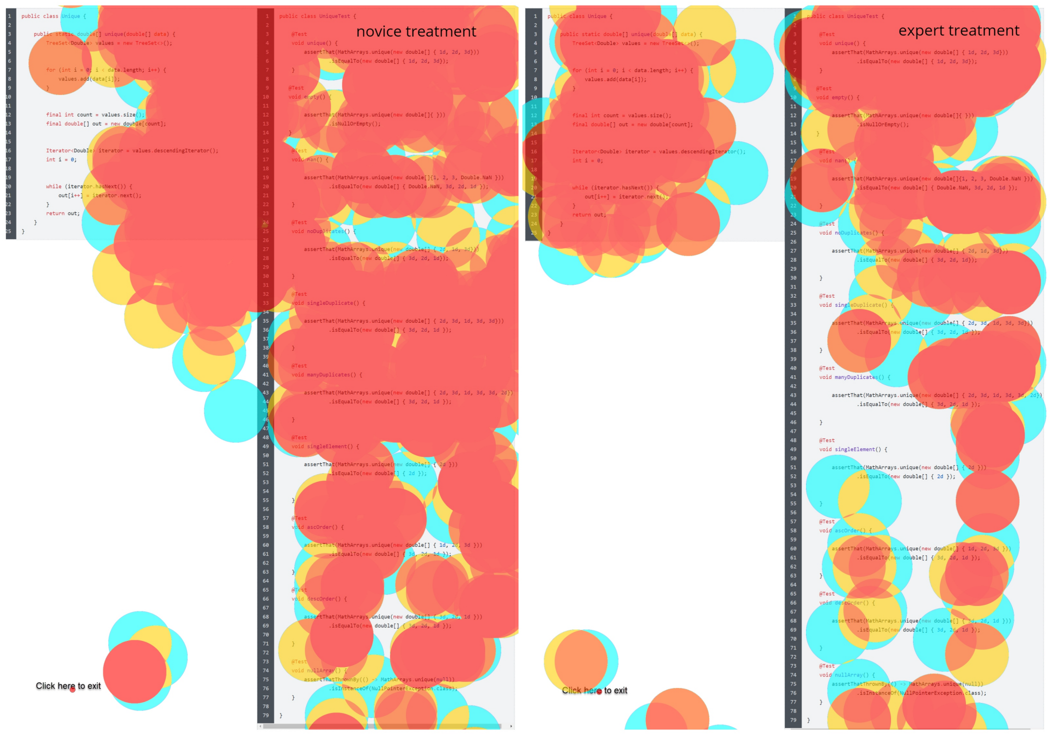


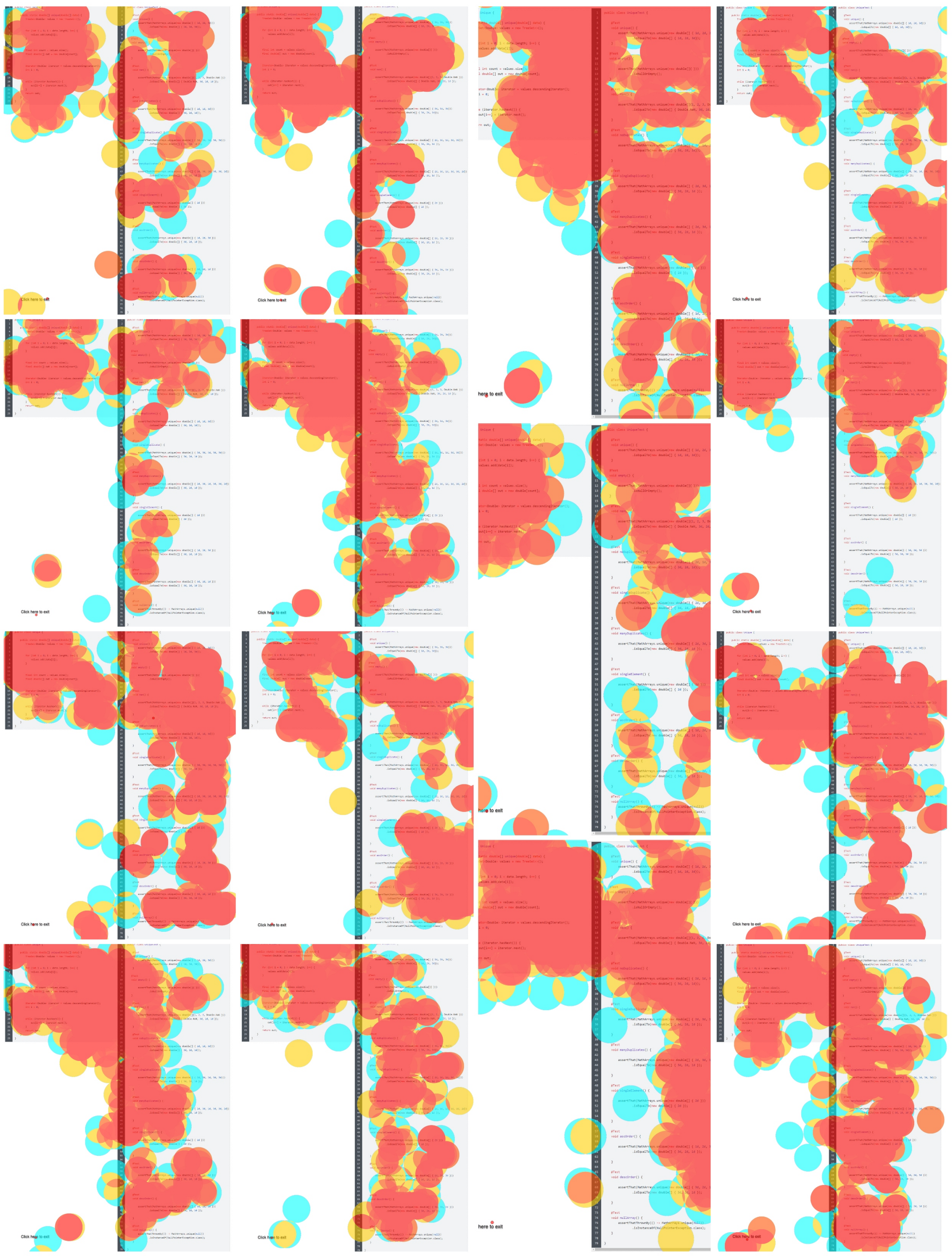
```java
public class Unique {

    public static double[] unique(double[] data) {
        TreeSet<Double> values = new TreeSet<>();

        for (int i = 0; i < data.length; i++) {
            values.add(data[i]);
        }

        final int count = values.size();
        final double[] out = new double[count];

        Iterator<Double> iterator = values.descendingIterator();
        int i = 0;

        while (iterator.hasNext()) {
            out[i++] = iterator.next();
        }
        return out;
    }
```

```java
public class UniqueTest {

    @Test
    void unique() {
        assertThat(MathArrays.unique(new double[] { 1d, 2d, 3d}))
                .isEqualTo(new double[] { 1d, 2d, 3d});
    }

    @Test
    void empty() {

        assertThat(MathArrays.unique(new double[]{ }))
                .isNullOrEmpty();
    }

    @Test
    void nan() {

        assertThat(MathArrays.unique(new double[]{1, 2, 3, Double.NaN }
                .isEqualTo(new double[] { Double.NaN, 3d, 2d, 1d });

    }

    @Test
    void noDuplicates() {

        assertThat(MathArrays.unique(new double[] { 2d, 1d, 3d}))
                .isEqualTo(new double[] { 3d, 2d, 1d});

    }

    @Test
    void singleDuplicate() {

        assertThat(MathArrays.unique(new double[] { 2d, 3d, 1d, 3d, 3d}
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void manyDuplicates() {

        assertThat(MathArrays.unique(new double[] { 2d, 3d, 1d, 3d, 3d,
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void singleElement() {

        assertThat(MathArrays.unique(new double[] { 2d }))
                .isEqualTo(new double[] { 2d });


    }

    @Test
    void ascOrder() {

        assertThat(MathArrays.unique(new double[] { 1d, 2d, 3d }))
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void descOrder() {

        assertThat(MathArrays.unique(new double[] { 3d, 2d, 1d }))
                .isEqualTo(new double[] { 3d, 2d, 1d });

    }

    @Test
    void nullArray() {
        assertThatThrownBy(() -> MathArrays.unique(null))
                .isInstanceOf(NullPointerException.class);
    }

}
```

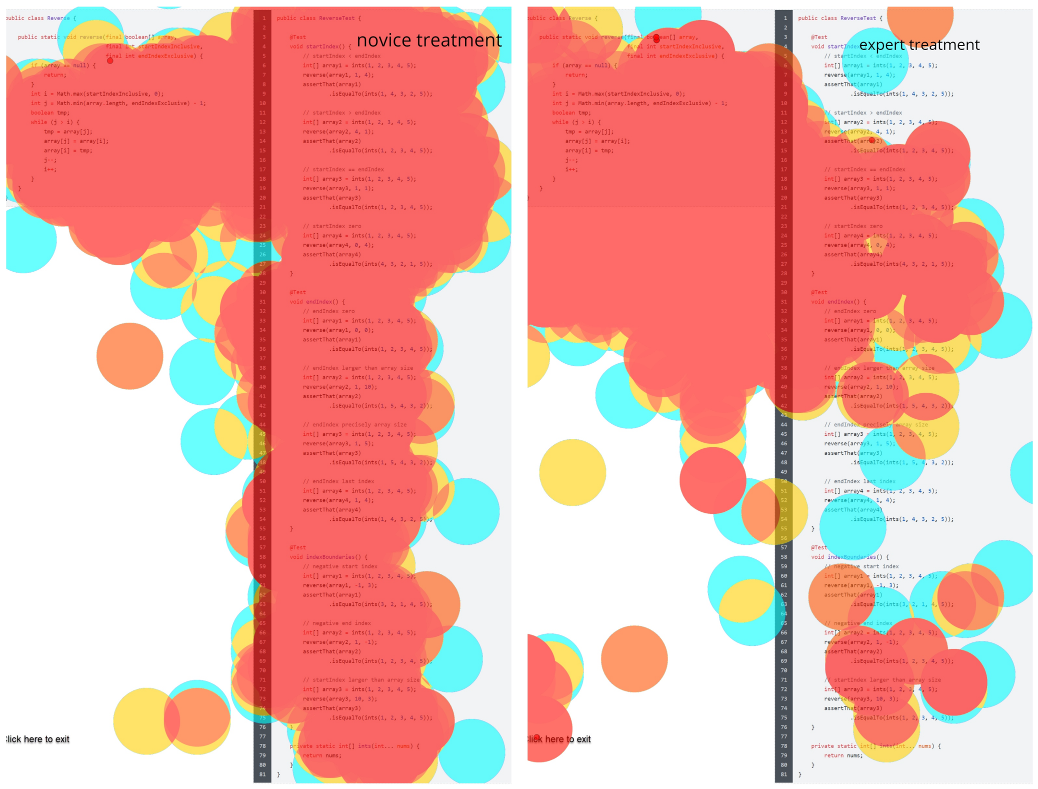

Click here to exit
Click here to exit
Click here to exit

D.3 Reverse


novice treatment
expert treatment
Click here to exit
Click here to exit

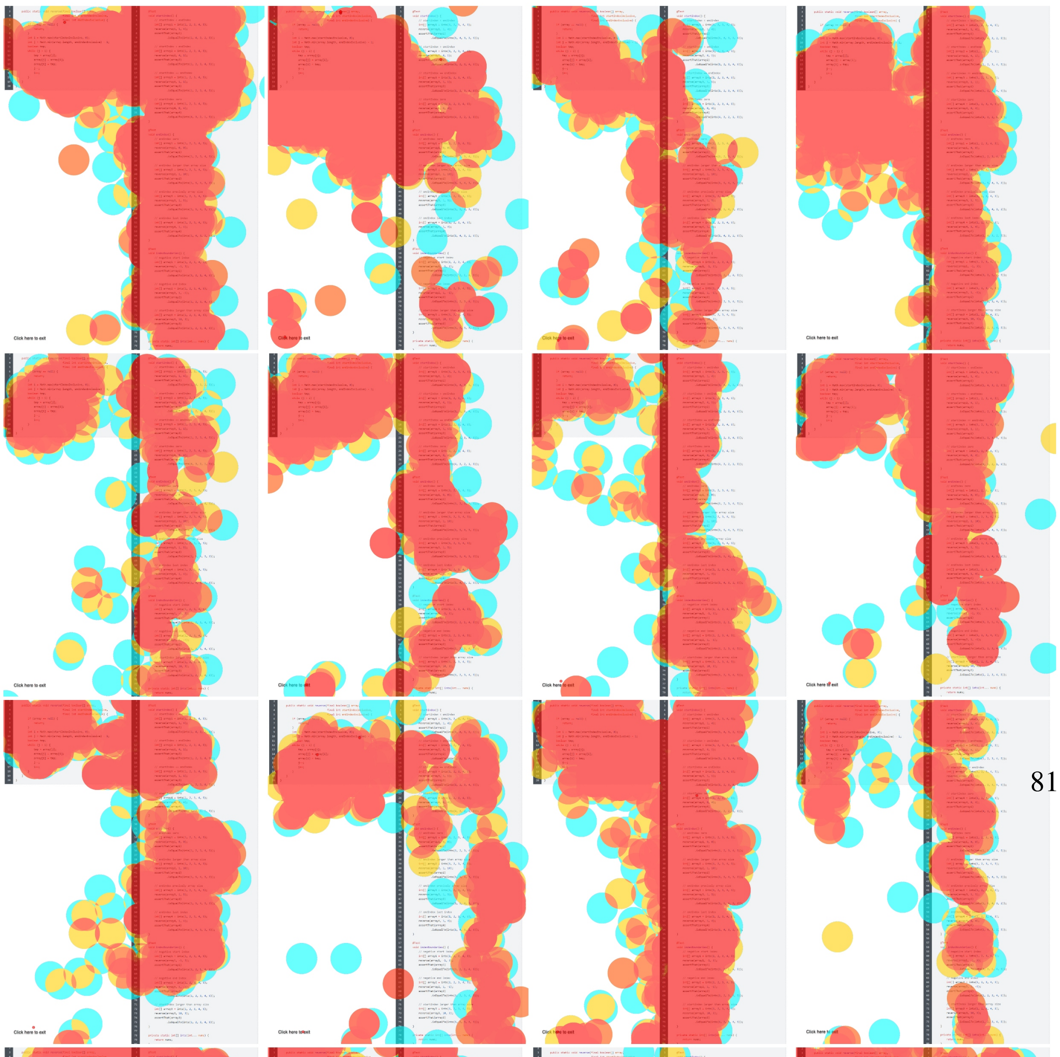

Click here to exit

D.4 LastIndexOf



```
public class Reverse {

    public static void reverse(final boolean[] array,
                               final int startIndexInclusive,
                               final int endIndexExclusive) {
        if (array == null) {
            return;
        }
        int i = Math.max(startIndexInclusive, 0);
        int j = Math.min(array.length, endIndexExclusive) - 1;
        boolean tmp;
        while (j > i) {
            tmp = array[j];
            array[j] = array[i];
            array[i] = tmp;
            j--;
            i++;
        }
    }
}
```

```
public class ReverseTest {

    @Test
    void startIndex() {
        // startIndex < endIndex
        int[] array1 = ints(1, 2, 3, 4, 5);
        reverse(array1, 1, 4);
        assertThat(array1)
                .isEqualTo(ints(1, 4, 3, 2, 5));

        // startIndex > endIndex
        int[] array2 = ints(1, 2, 3, 4, 5);
        reverse(array2, 4, 1);
        assertThat(array2)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // startIndex == endIndex
        int[] array3 = ints(1, 2, 3, 4, 5);
        reverse(array3, 1, 1);
        assertThat(array3)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // startIndex zero
        int[] array4 = ints(1, 2, 3, 4, 5);
        reverse(array4, 0, 4);
        assertThat(array4)
                .isEqualTo(ints(4, 3, 2, 1, 5));
    }

    @Test
    void endIndex() {
        // endIndex zero
        int[] array1 = ints(1, 2, 3, 4, 5);
        reverse(array1, 0, 0);
        assertThat(array1)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // endIndex larger than array size
        int[] array2 = ints(1, 2, 3, 4, 5);
        reverse(array2, 1, 10);
        assertThat(array2)
                .isEqualTo(ints(1, 5, 4, 3, 2));

        // endIndex precisely array size
        int[] array3 = ints(1, 2, 3, 4, 5);
        reverse(array3, 1, 5);
        assertThat(array3)
                .isEqualTo(ints(1, 5, 4, 3, 2));

        // endIndex last index
        int[] array4 = ints(1, 2, 3, 4, 5);
        reverse(array4, 1, 4);
        assertThat(array4)
                .isEqualTo(ints(1, 4, 3, 2, 5));
    }

    @Test
    void indexBoundaries() {
        // negative start index
        int[] array1 = ints(1, 2, 3, 4, 5);
        reverse(array1, -1, 3);
        assertThat(array1)
                .isEqualTo(ints(3, 2, 1, 4, 5));

        // negative end index
        int[] array2 = ints(1, 2, 3, 4, 5);
        reverse(array2, 1, -1);
        assertThat(array2)
                .isEqualTo(ints(1, 2, 3, 4, 5));

        // startIndex larger than array size
        int[] array3 = ints(1, 2, 3, 4, 5);
        reverse(array3, 10, 3);
        assertThat(array3)
                .isEqualTo(ints(1, 2, 3, 4, 5));
    }

    private static int[] ints(int... nums) {
        return nums;
    }
}
```



Appendix E

Model Assumptions

In this section we verify the model assumptions for linear mixed for model.

E.1 Homoskedasticity

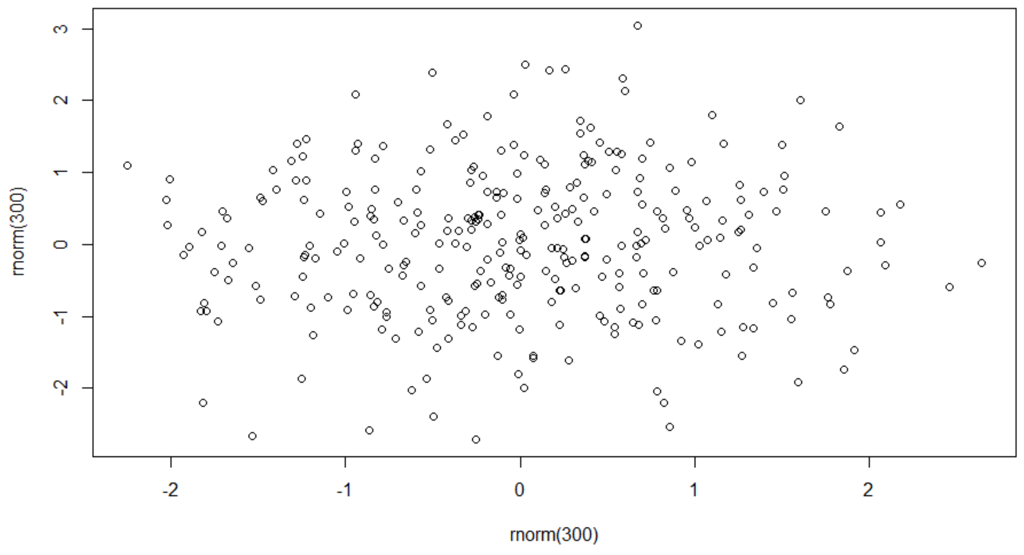


Figure E.1: Homoskedasticity

E.2 Normality of residuals

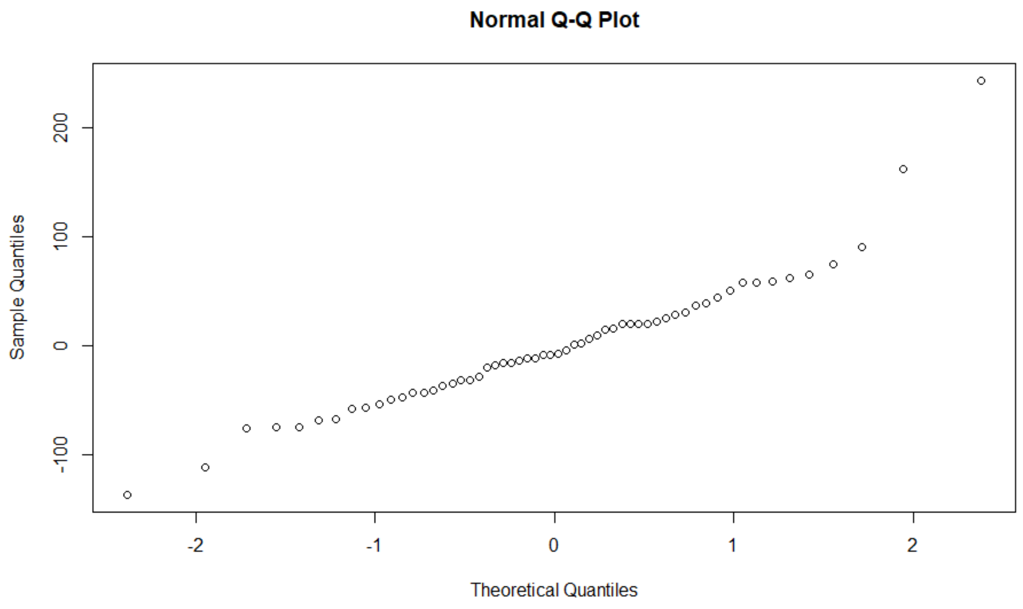


Figure E.2: Normality of residuals