Towards Understanding How Developers Comprehend Tests

Master's Thesis

Chak Shun Yu

Towards Understanding How Developers Comprehend Tests

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Chak Shun Yu
born in Rotterdam, the Netherlands

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

©2018 Chak Shun Yu. All rights reserved.

Towards Understanding How Developers Comprehend Tests

Author: Chak Shun Yu
Student id: 4302567
Email: c.s.yu@student.tudelft.nl

Abstract

Developers spend the majority of their time and effort on reading and comprehending source code. In order to improve this process of program comprehension for developers, a numerous amount of existing studies have looked into understanding how developers approach it and factors of influence. However, less is known in the field about how developers comprehend test code, an alternative form of source code, despite its widely acknowledged importance and benefits in both research and practise. In this paper, we perform a foundational study on understanding how developers comprehend tests by applying existing knowledge and work on program comprehension onto tests comprehension and looking at the influential factors. An online controlled experiment was conducted with 44 developers to measure three defined metrics of tests comprehensibility, namely the total time spent on reading a test suite, the ability to identify the overall purpose a test suite, and the ability to produce additional test cases to extend a test suite. The main findings of our study, with several implications for future research and real world, are that *(i)* prior knowledge of the software project decreases the total reading time, *(ii)* experience with Java affects the proportions of time spent on the Arrange and Assert sections of test cases, *(iii)* experience with Java and prior knowledge of the software project positively influences the ability to produce additional test cases of certain categories, and *(iv)* the most influential factor towards understanding and extending a test suite is experience with using tests.

Thesis Committee:

Chair: Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor: Dr. M. Aniche, Faculty EEMCS, TU Delft
Committee Member: Prof. Dr. C. Hauff, Faculty EEMCS, TU Delft

Preface

This document serves not only as my Master's thesis, but also as the end of my five year period as a student at the Delft University of Technology. Throughout these years, I have met a lot of people and made a lot of friends. Several of them have had significant impact on me as a software developer and/or helped me out during various phases of the experiment in this study. Here, I want to take the opportunity to thank all of them.

During this period, the people closest to me in my daily life have been supportive of me and helped me getting through this thesis and university in a successful manner. Without any of them, my daily life outside of the university would not have been the same. Towards all of them, I want to express my love and appreciation.

During this Master's thesis, I have been supervised by Maurício Finavaro Aniche for 9 months, which was preceded by a literature survey for roughly 4 months. My collaboration with Maurício has lasted more than a full year's length of time, in which he has supervised me on a weekly, almost daily, basis. Not only has he been an incredible supervisor during this time, but also an extremely insightful mentor in research, an enthusiastic source for the directions of this study, and a person who has helped me significantly in several aspects outside of the scope of this thesis, including mentally and as a developer. To Maurício, I want to express nothing but my sincerest respect, gratitude and, most importantly, appreciation for what he has done for me and this research during the period of our collaboration. We really did cool stuff and I enjoyed my time in the past year thoroughly.

Chak Shun Yu
Rotterdam, the Netherlands
November 9, 2018

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

2 Related Work 3
2.1 Program Comprehension 3
2.2 Research Methods in Program Comprehension 8

3 Research Design 13
3.1 Research Questions 13
3.2 Methodology 15
3.3 Independent and Dependent Variables 21
3.4 Participants Selection 26
3.5 Analysis Procedure 26

4 Results 29
4.1 Descriptive Statistics 29
4.2 Model Assumptions Statistics 31
4.3 Model Statistics 32

5 Discussion 39
5.1 Revisiting Research Questions 39
5.2 Implications 41
5.3 Threats to Validity 42

6 Conclusion 47

Bibliography 49

A Letter of Ethics Committee Approval **58**

B Screenshots of the Online Experiment **59**

B.1 Pre-Experiment Questionnaire 59
B.2 Initial Instructions to Get Familiar With the Tool 60
B.3 Tracking Tool with Reduced Viewport 60
B.4 Experiment Task Questions 61

List of Figures

3.1 Overview of the online research procedure. 16
3.2 Example of the tracking tool in which the participants are shown a fraction of the test suite. 17

4.1 Boxplots of the proportional distributions of time spent by participants on each section of the AAA test structure. 32
4.2 Pairwise scatterplots of all the numerical independent variables. 33
4.3 Residuals plotted against the fitted values of the two models, RT as is (left) and RT log transformed (right). 33
4.4 Histograms of the residuals of the two models, RT as is (left) and RT log transformed (right). 34
4.5 Q-Q Plots of the residuals of the two models, RT as is (left) and RT log transformed (right). 34

Chapter 1

Introduction

One of the most essential aspects of software development is being able to understand how a program works [57, 60]. For software engineers to be able to perform software development and maintenance related tasks on a software project, it is necessary that they understand how the underlying source code works [14]. However, this results into developers spending the most significant portion of their time on reading and understanding source code [14, 39, 60], rather than performing the actual task. As such, over the past decades the field of Computer Science has produced numerous amount of research investigating the process of program comprehension to be able to improve it. This has resulted into studies of varying angles, topics, and abstraction layers onto program comprehension.

While no single general approach exists to explain the process of program comprehension of developers [39], studies across the field have investigated the possible influences of all kinds of different specific aspects. These aspects range from analyzing how to approach and improve program comprehension from a higher level persective, focusing on the program comprehension as a whole [39, 57, 60], to properties of the software program on a source code level, like the style, quality, and length of identifier names [18, 27, 28, 40], to properties on a code construct level, like code regularity [31] and code beacons [14, 25], to higher abstractions across source code, like code smells [62], readability [54], and familiarity [36], to using code visualizations [3, 13, 33, 45], to drawing similarities with natural language comprehension [11, 52, 53].

All of these studies provide contributions towards our knowledge on program comprehension and together form our current overall understanding of program comprehension. However, in our community less is known about how developers comprehend test code and which factors are of influence on their tests comprehension, although test code is an alternative form of source code. The importance of tests has been stated numerously in the field in the form of improving the quality of software projects, ensuring the behaviour of the software project is correct, or functioning as documentation [13, 20, 21, 23, 64, 68]. Despite this, developers often disregard test code during software maintenance related tasks due to it being very complex and costly [23, 63, 64], causing the quality and usefulness of test code to decrease [6, 16, 44, 64].

To combat this, several studies have looked into ways of enhancing the tests maintenance process for developers through various methods [6, 13, 16, 23, 64]. While all of

these studies come up with ways to enhance the process of test comprehension and the theory behind their enhancements are valuable contributions towards the understanding of test comprehension, none of them have the primary focus of learning more about the underlying process of test comprehension. Similarly to improving the program comprehension process of developers, it is necessary to gain a better understanding of how developers approach the comprehension of tests to be able to improve upon it. This hole in the field is what this research attempts to fill by performing a research on understanding more of the process of test comprehension and the factors that influence this process. This will lay the foundation for future work on tests comprehension.

The main goal of this study is to look into the factors of influence on the tests comprehension process of developers. To do so, an online controlled experiment is conducted to gather data on the test comprehensibility of 44 participants in a total of 132 data points. To measure the test comprehensibility of participants, we define three metrics represented across nine dependent variables: the amount of time spent on reading a test suite, the ability to identify the overall purpose of a test suite, and the ability to extend on the test suite by producing additional test case of varying categories. The main contributions of this paper are as follows:

- A foundational study to gain an initial understanding on the tests comprehension process of developers based on existing studies towards program comprehension. The study reveals a collection of factors that are of influence on the tests comprehension process of developers, what their specific impacts are, and a discussion on their implications for research and practise.
- An overview of the current state of research towards program comprehension and discussion on the research methods involved.
- A set of quantifiable metrics to measure the tests comprehensibility of developers.
- A design methodology highlighting the design decisions regarding a study towards tests comprehension that can serve as the blueprint for future work.

The rest of this paper is structured as follows: Chapter 2 presents background information and related work on understanding program comprehension. Chapter 3 describes the design of our research by going over the research questions, the design of our online experiment, the variables of interest in this study, how participants were selected, and how the data will be analyzed. Then, Chapter 4 present statistics to describe the participants distribution and to verify the model assumptions, and the results of our study. These results are discussed in Chapter 5, together with the threats to validity of this study. Lastly, Chapter 6 concludes the paper and provides future work.

Chapter 2

Related Work

Test code comprehension is not yet a well explored domain in the field of computer science. Despite it being a subset of the domain of program comprehension, which contrary to the former is a widely explored domain. This chapter discusses related studies taken from the domain of program comprehension, alternative research methods for code comprehension, and the few existing studies regarding or touching tests comprehension.

2.1 Program Comprehension

One of the most essential aspects to software engineering is to understand how the program works. Consequently, the domain of program comprehension has attracted dozens of studies. These studies range from attempts to increase the current knowledge on the process behind program comprehension, and looking into different factors that influence the way programs are comprehended, to ways to improve the program comprehension of software engineers.

Impact of Code Lexicon on Program Comprehension In their studies, Hofmeister et al. [28], Schankin et al. [56] investigated the impact of the length of identifier names on the program comprehension of developers. In their web based experiment, 72 C# developers were provided six code snippets, and the tasks to identify and fix the present code defect. As a proxy measure of the program comprehension, besides whether they found and corrected the code defect correctly, was the time that it took for the participants to do so. The results of their study indicate that shortening the identifier names through abbreviations or only using letters negatively impacts the program comprehension. More specifically, using full words for identifiers resulted in 19% faster completions of the experiment tasks compared to using abbreviations and letters for identifiers.

Work by Fakhoury et al. [18] use an alternative form of fMRI, namely functional near infrared spectroscopy (fNIRS), in combination with eye tracking to measure the effects of poor source code lexicon on the cognitive effort required by developers in the process of program comprehension. Based on their study with 15 eligible participants, the results indicate that poor source code lexicon, in any form, has a negative impact on the comprehen-

sibility of the respective source code and the ability of developers to perform their software development related tasks.

In another study, Kosti et al. [35] worked towards a way to asses the cognitive workload of developers during software development related tasks and with it a model to assess the difficulty of these tasks. To do so, they tended towards the research method of Electroencephalography (EEG). Based on their experiment with 10 participants, comparisons between the EEG patterns of all the participants indicated clear differences in the used cognitive workload during code comprehension and finding syntax errors. Furthermore, besides the results, they also describe their method as a possible universal way of measuring, assessing, and comparing cognitive effort of developers during software development related tasks.

Code Constructs and Program Comprehension Logically, more complex source code is more difficult to comprehend than simpler code. While existing metrics exist to measure the complexity of source code like lines of code (LOC) and McCabe's cyclomatic complexity (MCC), studies by Jbara and Feitelson [31, 32] identified a mismatch between the theory behind these metrics and how the complexity is interpreted in practise in certain cases. In their work, the authors introduce the notion of code regularity, a repetitive code segment with potential small adjustments in every iteration, and measure the potential effects it has on the code complexity and thus the code comprehensibility. To do so, eye tracking technology was used to determine the differences in effort used by developers to comprehend code snippets with varying levels of regularity, which was measured by the time and number of fixations spent. Based on a data set of 105 data points across two experiments, the results indicated that a high rate of regularity in code snippets has no impact on the time spent on the tasks, but does lead to better task performance and comprehensibility. Additionally with this data, they found a diminishing amount of effort with every repetition of a code block, on which they based their conclusion that the additive nature of the syntactic complexity metrics causes an overestimation of the complexity of regular code, and should be modified with context-dependent weights.

Melo et al. [43] used eye tracking in their work in order to understand *how* developers debug programs with variability, which in short is the presence of configuration-dependencies at compile time. Melo et al. state that previous studies on debugging programs with variability only focus on quantitative questions, like the debugging time and correctness, while little attention is paid to the understanding of how developers perform the debugging process, which the vision and intend of this research is in line with. Using an experiment with 20 participants, Melo et al. display a number of findings relevant for the field, some of which are consistent with prior studies, while others are new. Specifically, they found that variability increases the debugging time for both code fragments containing variability, as well as code fragments in the proximity of variability containing code fragments, the number of saccades between definition-usages of fields and call-returns for methods, prolongs the initial scan of the program, and splits the debugging approach of developers to either consecutive or simultaneous processing of the configurations.

Work by Crosby et al. [14] investigate the role of experience on program comprehension by looking at how different groups identify beacons, important code segments, and

comparing them to one another. The results indicate that developers will identify beacons differently based on their experience. Experienced developers will focus more on identifying beacons in a software program, while novice developers are less likely to search for beacons.

Building forth on this concept of beacons, work by Hegarty-Kelly et al. [25] shows how that knowledge on differences in performance (in identifying beacons) can be used to improve the process of program comprehension for certain groups of developers. Their research consist of two stages of experiment. The first experiment contains only experts as participants, who they use to identify beacons, important code segments, in their code comprehension tasks. In their second experiment which consisted of only novice participants, Hegarty-Kelly et al. divide the experiment into one control group and two treatment groups. All of the groups are given the same code comprehension task as the experts in the first experiment, but the treatment groups are provided visual assistance. One group receives visual highlights of the beacons acquired from the experts' experiment, while the other group receives visual highlights of incorrect beacons.

Impact of Code Smells on Program Comprehension Palomba et al. [49] have conducted a large scale study on 395 releases of 30 Java open source projects, aimed at analyzing code smells and their impact on software quality. In total, the study processed 17.350 instances of 13 different code smell types. Their results indicate that the most impacting code smells are related to size and complexity, support the findings of previous literature that source code affected by code smells are more prone to change and defects, but also indicate that code smells are not necessary the direct cause of this proneness. Rather, the presence of code smells are indications of an intrinsic underlying problem with the source code causing its proneness to change and defects.

While the significant issues with code smells has been thoroughly stated by existing literature in the field, the same does not hold for the developers' perception of the extent of these issues. Based on a survey with 34 participants, of which 15 Master's students, 9 industrial developers, and 10 software project developers, Palomba et al. [48] conclude that there exists a gap between theory and practise in how significant code smells are perceived as issues. Specifically, developers are generally more aware of code smells related to complex and long source code, while other code smells are rarely identified as design problems. Additionally, Palomba et al. find that the identification of certain code smells is positively affected by the developers' experience and knowledge of the software project.

Work by Soh et al. [62] show the impact of code smells on different program comprehension activities. Soh et al. state that previous empirical studies have shown that code smells have little impact on maintenance effort at file level and credit this low effect to the fact that the effort was measured by previous studies as "sheer-effort", without any distinction between the specific activities at hand of developers. The identical dataset of a previous study was taken in order to support their statement and was annotated according to their own annotation schema to identify the different maintenance activities. In short, this annotation schema boils down to tracking the duration of all the different maintenance activities under condition of different code smells. This was regarded as the effort spent by developers in the different maintenance activities, and were then aggregated and analyzed. The results

of their analysis is in line with their statement regarding the previous studies, as different relationships were found between specific maintenance activities and code smells. Besides very low level relationships, namely specific code smells affecting specific maintenance activities more significantly, there were also more general findings made. Soh et al. show that code smells have a more significant impact on editing and navigating effort compared to increased file sizes, while the opposite holds for reading and searching effort of source code. They conclude that the field should be wary of the presence of code smells, contrary to the statement of previous studies, due to their specific influence on developers during different maintenance activities.

Using Code Visualization to Enhance Program Comprehension Another part of the field are focused on enhancing the program comprehension process of developers through code/software visualization. Through visualizations, developers are provided a structural overview of the software project, abstracting away unnecessary or less important information. The aim is to aid developers in program comprehension by providing a focused overview of the important aspects. This has been implemented by a broad range of existing tools [33] and has been subject to many studies in the field.

In modern source code editors, a mini-map visualization of the currently open source code file can often be found to assist the developer. In their work, Bacher et al. [3] make an attempt in enhancing this mini-map visualization, and thus the program comprehension process of developers, by integrating macroscopic details within the code onto the mini-map. These are conceptual structures that are not manifested directly into the source code, but do contribute to the overall program comprehension. Specifically, Bacher et al. focus on the integrating information regarding the scope chain of the currently selected source code in the mini-map. Based on the results of an experiment with 60 participants divided into two groups, they conclude that layering additional information relevant to specific (difficult) features of a programming language, like variable hoisting in the case of their scope chain information layer, on top of the existing mini-map visualization yields positive effects on the code understanding.

In their work, Mumtaz et al. [45] use two two-dimensional multivariate data visualization techniques to assist developers in identifying bad smells in source code. The two techniques complement each other in identifying data patterns, clusters, and outliers, which Mumtaz et al. connect to bad smells and software quality attributes. Based on applying and inspecting their approach onto three open source Java software projects, they conclude that these connections can be made and that their approach can help developers visually identify bad smells in source code.

Work by Cornelissen et al. [13] show a visual approach to assisting the test comprehension process of developers. Their approach creates scenario diagram models for test cases based on dynamic analysis of the test suites. These scenario diagrams focus on the interactions between objects, abstracting away unnecessary or less important information, and visualizing them in a more optimal human readable way. Varying amount and types of abstractions have to be used to achieve this, which in turn depends on the type of test cases that are subject to the visualization. Based on their case study, they conclude that test code

visualization in the form of scenario diagrams does yield useful information regarding the system's inner working.

Similarities with Natural Language Comprehension Busjahn et al. [11] conducted an eye tracking study on the linearity of developers' source code reading behaviour. In the field, the comparison between reading natural language text and source code has often been made, displaying similarities in the processes at which these tasks are performed. Busjahn et al. identify, however, that the linearity aspect, which is a significant property of natural language text, is left quite unexplored. In this study, Busjahn et al. make an attempt in exploring this aspect of linearity in source code reading behaviour. To do so, fourteen students from a Java beginner's course and six professional programmers were given comprehension tasks on short English texts, and programs written in Java and pseudocode, while being monitored with an eye tracking device. The results of their study show that novices read source code less linearly than natural language text, 70% linear eye movements compared to 80%, and, on top of that, experts read source code less linearly than novices, 60% linear eye movements.

Work by Peitek et al. [52, 53] focus on the use of functional magnetic resonance imaging (fMRI) to aid in the existing measurement methods of program comprehension [53] and determining the cognitive processes used during program comprehension and the brain regions involved [52]. Based on an experiment with 17 participants accompanied by a replication for confirmation purposes with 11 participants, their study yield a set of interesting cognitive results. No correlation was found between the programming experience of participants and the cognitive effort, while familiarity with the programming language of the code snippets decreased the cognitive efforts. Furthermore, several findings were made regarding the activation of varying brain areas, to the point that similarities were observed with natural language comprehension.

Test Code Comprehension While the aspect of program comprehension has been extensively studied in the field, the comprehension of test code has not received the same amount of attention. Despite this, several studies have looked into possible ways to enhance the program comprehension process of developers [13, 16, 23].

Greiler et al. [23] investigate a way to derive relations between levels of test cases. In particular, their approach attempts to connect higher level end-to-end tests to low level unit tests through similarity of the outputted stack traces. This will aid developers during their software engineering tasks when changes occur in requirements, by making it easier for them to trace the changes from the end-to-end tests all the way down to the involved source code, through the unit tests.

In other work, Deursen et al. [16] focus on assisting the process of test code *refactoring*. They identify the importance of test code in extreme programming and the fact that it is subject to frequent refactoring. In their work, they provide an extensive analysis of this refactor process. The result is a collection of code smells specific to test code refactoring and ways to refactor or address the specific code smell.

Bavota et al. [6], Spadini et al. [64] build upon this work by further investigating this collection of test code smells (abbreviated to ”test smells”). Through a survey with 19 par-

ticipants, Bavota et al. show that test smells generally are not recognized by developers. Based on previous studies highlighting the negative effect of test smells on code comprehension and maintenance, they further emphasize the importance of being able to (semi) automatically identify and refactor test smells. For this reason, Bavota et al. conducted a controlled empirical experiment to gain more insight into test smells. Specifically, they investigate the origin and survivability of test smells and the relationships of their presence with production code smells. Based on the analysis of the commit history of 152 open source projects, their results indicate that test smells originate during the creation of test cases, rather over time due to diminishing code quality, have a significant high survivability, and have certain relationships with the presence of code smells.

Extending upon this, Spadini et al. [64] look further into the relations between test smells and software code quality. To do so, they conducted an empirical analysis of 221 releases across ten software systems to observe trends between the presence of a subset of test smells and the measured software code quality. Based on the results, they conclude that test cases containing test smells have 81% higher risk of being defective and are 47% more likely (required) to be refactored between releases, and that production code tested by test code containing certain test smells are far more likely to contain defects.

2.2 Research Methods in Program Comprehension

Program comprehension, as a topic subject to research, is in essence the cognitive process behind understanding code. In existing studies, including but not limited to the ones considered in this chapter, various methods have been used to research program comprehension in order to gain a better understanding of the process behind it and the factors of influence on it. Table 2.1 provides an overview of all the related studies considered in this chapter. All the work entries are grouped by the used research method and extended its contribution towards the understanding of program comprehension.

Qualitative Methods Traditionally, research towards program comprehension, and in the field of Computer Science in general, has tended towards more qualitative forms of research methods, like think-aloud protocols, interviews, and comprehension summaries. The benefits of these forms of research methods are generally that they have a very low learning curve, require relatively little setup and in certain cases can yield high quantity for its effort. This means that any researcher can opt into these research methods and can start an experiment in a short amount of time. Due to this, as can be noticed in Table 2.1, studies using conventional research methods cover a wide range of varying research topics with varying success. Ranging from the impact of lower level code constructs to the general knowledge on program comprehension [39, 57]. While these conventional research methods have set the fundamental layers to a set of widely accepted program comprehension theories and models to date, there are varying limitations regarding their accuracy, preciseness, and efficiency [53]. To address these limitations, varying new research methods have emerged in the field, each with their own properties. In the following, we group the papers we intro-

duced before by their research methodologies. Additionally, we offer a reflection on the advantages and disadvantages of each of them.

Eye Tracking Technology Other studies have tended towards the use eye tracking as instrumentation. Severe improvements can be gained in the quality of the tracking of the participants' focus on particular code snippets with the use of eye tracking technology. This makes it very attractive for studies focusing on the impact of low level aspects of source code on program comprehension, as can be observed in the majority of contributions of the respective works in Table 2.1. Most of the works focus on measuring the impact of low level source code aspects or code constructs on the process of program comprehension, while the others focus on the differences between expertise level and how to use that knowledge. However, the disadvantages of eye tracking technology are the learning curve, the availability to hardware, the relatively long setup time and the low quantity of data compared to the required effort. The relatively steep learning curve is a major drawback to using eye tracking technology. Not only does it require more thorough planning of the experiment and relatively long setup times for each experiment run, it also substantially increases the considerations to take into account regarding the quality of the tracking data [29]. Additionally, obtaining data through eye tracking technology is a relatively slow process, as it is often will require an on-site presence of the participant due to the required hardware and is a sequential process.

Cognitive Methods Another part of the field has focused more towards gaining a better higher level understanding of the cognitive efforts and processes behind program comprehension. These studies have tended towards the field of cognitive science for their research methods. The advantages of cognitive research methods are that they provide better isolation of the cognitive process of interest [53], thus benefiting the reliability and preciseness of the obtained data. The studies using cognitive research methods in Table 2.1 shows a clear tendency towards measuring cognitive processes like effort and workload, and the possibilities of this information. Cognitive research methods share most of the disadvantages with eye tracking technology, but even bigger. It requires more expensive hardware and has an even steeper learning curve in the form of setup time, preparations and carrying out the actual experiment.

Instrumented Software Methods Lastly, there are studies that stay close to the field of Computer Science by creating their own research method software. In work by Hofmeister et al. [28] and Schankin et al. [56], a custom web application was built to track the program comprehension process of participants. Similar methods were used by Soh et al. [62], in which the activity of developers were automatically logged by the experiment. These custom made software research methods in terms of advantages and disadvantages lie in between the conventional methods and eye tracking technology. The main limitation that this method attempts to solve is the quantity of incoming data, as this is limited with the conventional methods, while preserving a decent amount of quality. Compared to conventional research methods, the quality is not substantially improved, as these forms of tracking

the participants' focus or activity is essentially a form of proxy. However, the scalability is greatly improved due to the possibility of parallel data data acquisition, less effort for the researcher to run the experiment and not requiring on-site presence of participants.

Method	Work	Contribution
Conventional	Bacher et al. [3]	Enhancing mini-map visualization with layer of scope chain information.
	Bavota et al. [6]	Assessing whether developers are able to identify test smells (initial survey).
	Cornelissen et al. [13]	Use scenario diagrams to assist test comprehension.
	Krüger et al. [36]	Impact of code familiarity on software development.
	Maalej et al. [39]	Study on how developers approach PC.
	Mumtaz et al. [45]	Use multivariate data visualization techniques to identify bad smells.
Eye Tracking	Busjahn et al. [11]	Linearity of developers' source code reading behaviour.
	Crosby et al. [14]	Role of experience during PC.
	Hegarty-Kelly et al. [25]	Using knowledge of experts to assist novice developers.
	Hofmeister et al. [27]	Comparing eye movements between experts and novices during PC.
	Jbara and Feitelson [31, 32]	Impact of code regularity on effort and complexity.
	Maletic and Sharif [40]	Impact of identifier naming conventions on PC.
	Melo et al. [43]	Impact of variability on debugging.
	Sharif et al. [59]	Impact of (initial) scan time on finding code defects.
Cognitive	Fakhoury et al. [18]	Impact of poor code lexicon on cognitive effort.
	Fritz et al. [19]	Detecting when tasks in software development are considered difficult by developers.
	Kosti et al. [35]	Cognitive workload during PC to asses task difficulty.
	Peitek et al. [52]	Measuring the cognitive process of PC.
	Peitek et al. [53]	Improve accuracy of measuring the cognitive process of PC.
Software	Hofmeister et al. [28]	Impact of identifier names length on PC.
	Schankin et al. [56]	Comparing the impact of compound and short identifier names on PC.
	Soh et al. [62]	Impact of code smells on PC.

Table 2.1: Overview of related studies in the field grouped by research method with details on their contribution towards the understanding of the program comprehension (PC) process.

Chapter 3

Research Design

3.1 Research Questions

The main goal of this research is to measure and determine the factors of influence on the test comprehensibility of developers. More specifically, the focus lies on identifying and establishing potential relationships between (software development related) properties of developers and the degree at which they are able to understand source code tests in a meaningful manner.

[**MRQ**: What are the factors of influence on test comprehensibility?]

To be able to determine the test comprehensibility of developers, our research focuses onto different measurements. The three factors that we associate with test comprehensibility in this research are the Reading Time (RT), the ability to Identify the Testing Purpose (ITP), and the ability to Produce Additional Cases (PAC) of the developers. In the rest of this section, we will explain the three factors in detail.

Reading Time The Reading Time (RT) is the amount of time that developers spend on reading test code before moving on the next software development related task. Developers spent a great amount of time on software maintenance related tasks [22]. During those tasks, they spend a significant portion of their time reading source code [8, 41]. While previous studies have shown potential beneficial effects of spending more time on reading source code [59], it is more often associated negatively. Namely that developers are less able to spend more time on other software development related tasks as more time is dedicated to reading code. To aid the process of potentially this matter, this research will look into the potential factors that influence the reading time either negatively or positively.

[**RQ1**: What are the factors of influence on the time that developers spend reading test code?]

Besides learning about how much time developers spend on reading tests, we are also interested in learning how this time is distributed across different parts of a test case and the factors that influence this matter. To be able to make distinguishments between parts

of the test case, tests in this research are handled using the commonly used Arrange-Act-Assert (AAA) testing pattern [2, 15, 24, 30]. This pattern enforces a certain structure and order which the source code in tests should adhere to, according to the most important components to a test. For each of the different AAA sections, we will look into how much time is spent on them, how they compare to one another, and the factors that influence how much time developers spent on them.

RQ1.2: How do the different parts of AAA compare to one another regarding the time that developers spend on them?

RQ1.3: What are the factors of influence on the time that developers spend on reading the Arrange part of test code?

RQ1.4: What are the factors of influence on the time that developers spend on reading the Act part of test code?

RQ1.5: What are the factors of influence on the time that developers spend on reading the Assert part of test code?

Identifying the Testing Purpose Another metric to measure test comprehensibility of developers is their ability to identify the overall testing purpose of a test suite (ITP), focusing on understandibility. One of the most important aspects to reading test code is to understand the higher level scenarios that are being verified inside the test cases. In general, test cases are bundled together for structure purposes in a test suite to verify similar but slightly different test scenarios. Thus, all these test cases follow a general purpose that is lead by the test suite. This is what we call the testing purpose, and the ability to identify this testing purpose and what factors influences this ability is what we are interested in.

RQ2: What are the factors of influence on the ability of developers to identify the testing purpose of a test suite?

Various previous studies have already looked into specific factors that affect the described understandability aspect of code comprehension. Work by Crosby et al. [14], Jbara and Feitelson [31, 32] are examples of studies with similar intends to this study, but in which performance measurement of a code understandibility task is solely used for the assessment of the study. In this research, however, evaluation of the relationships against the task performance (ITP) is not solely about the performance assessment itself, but rather on what can be done with the knowledge on the differences in performances and the factors of influence. Work by Hegarty-Kelly et al. [25] provide an example of such, in which they measure the impact of visually providing important code segments, based on the performance of a group of experienced participants, to novice participants.

Producing Additional Cases The last metric to measure test comprehensibility of developers is their ability to produce additional cases (PAC). In practice, plainly understanding the test code and its underlying testing purpose is not sufficient for developers. After understanding the test code, it often comes paired with the task of producing, writing, or adding

more test cases. So building forth on this, their ability to identify the testing purpose, we also look at their ability to produce additional test cases to extend the existing test suite and the factors that influence this ability.

RQ3: What are the factors of influence on the ability of developers to produce additional test cases to extend the test suite?

This metric does not focus on the implementation details of the additional test cases, but rather on the higher level test scenarios that the developers can come up with. In similar occasions in practise that developers are provided the task of extending a test suite, implementing additional test cases is often a matter of following and replicating the existing test classes to a certain degree. The more difficult part is to actually come up with meaningful additional test cases to implement. Therefore, this will be the main focus of this metric and will be accounted for in the design of this experiment.

3.2 Methodology

The main objective of this research is to observe and investigate the tests reading behaviour of developers when performing software development related tasks and the factors of influence on their test comprehensibility during this process. Specifically, it is necessary to be able to describe in a structured form how the participating developers read a test class. Based on similar experiments in the field, the most logical data representation would be to keep track of the specific lines of code in a test class that developers are reading or looking at during the execution of a software development task, as commonly done in software engineering research [11, 27, 31, 32, 34, 42, 47, 51, 65].

Besides the difficulty in properly tracking and representing how developers read a specific test in an accurate manner, the other difficulty is to design the experiment in an as realistic and representative manner as possible. This requirement holds for the design of the software maintenance related task given to the participants, as well as for the used test classes as this research involves tests.

In the rest of this section, there will be extensive elaboration and discussions on what this meant for their respective aspects, the final design of all these different aspects, the decisions and trade-offs that went into the designs, and the overall procedure that the participants go through in this experiment. Additionally, we will cover the online web application that was created through which our experiment was conducted and information was gathered to address and elaborate on the different research questions (Section 3.1).

3.2.1 Overall Procedure

Our online experiments consists of two main parts, as illustrated in Figure 3.1. First, the participants are provided a pre-experiment questionnaire. This questionnaire focuses on demographical information which will serve as independent variables for our models.

After the participants finish the pre-experiment questionnaire, they arrive at the practical experiment part. The steps of this experimental procedure are illustrated in figure 3.1. First,

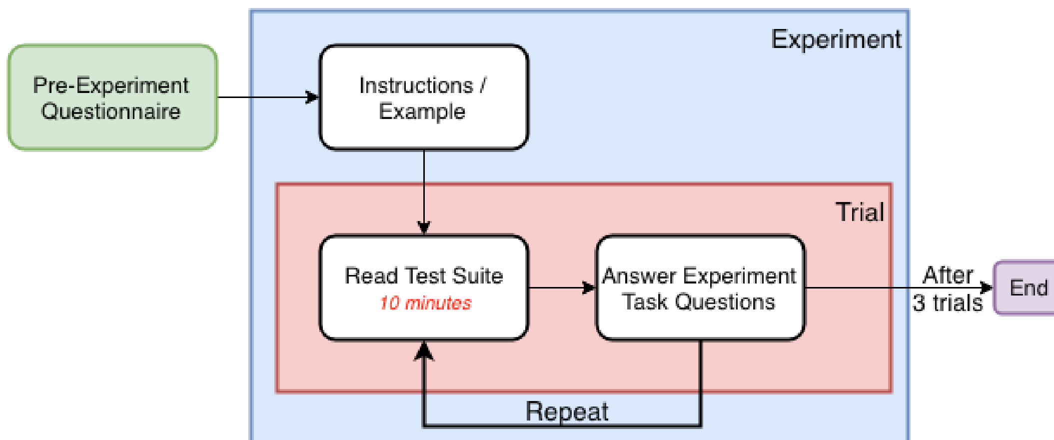


Figure 3.1: Overview of the online research procedure.

they are made familiar with our experiment tool through an example. Then, the participants enter the actual experiment in which they have to read through test suites and extract the necessary knowledge to complete the experiment task, after which they perform the actual task. This process of reading a test suite, performing the experiment task, and answering the questions together is referred to as a *trial*. In total, every participant will complete three trials and thus read through three different test suites, one low, one medium, and one high in difficulty.

3.2.2 Pre-Experiment Questionnaire

To start off the experiment for this research, a participant of the research has to fill in a pre-experiment questionnaire. Specifically, the information that was requested from the participants were their gender, age, software development role, amount of experience with software development in years, amount of experience with Java in years, current programming language of choice, and amount of experience with proper usage of tests. Their experience with Java and current programming language of choice together form a proxy for their familiarity and comfortability with Java, which is important as the main programming language used in this research is Java. Their experience with software development and proper usage of tests, similarly, are important factors to this research and will serve significant roles.

3.2.3 Trials

An essential component to this experiment is to be able to track what line of code the developer is reading at a specific moment in time during the experiment tasks, the focus, and in the end structure this information in a way that represents the amount of time that the developer has spend on every line of code.

During the experiment, the participants are shown test classes and the main purpose of this research is to look into how they read those tests. As this experiment is conducted in

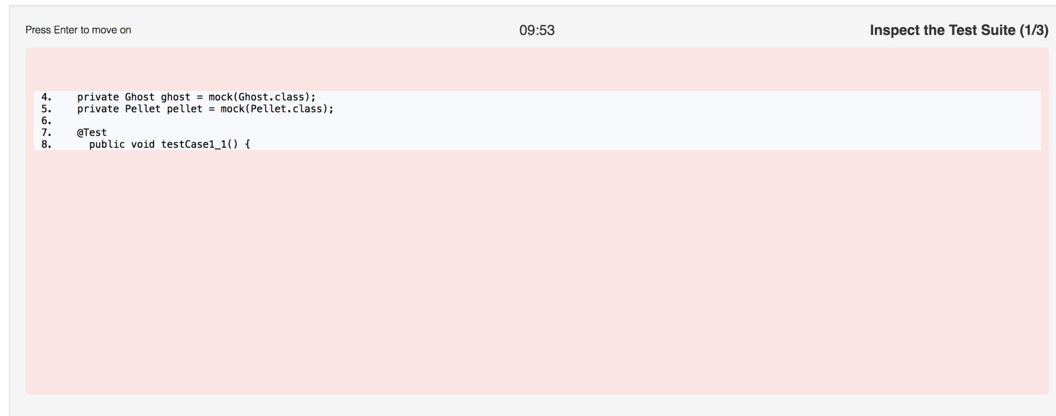


Figure 3.2: Example of the tracking tool in which the participants are shown a fraction of the test suite.

an online manner, it is impossible to obtain that kind of data by showing the entire test. A conscious description of their reading patterns from the participants is also not desirable, due to the bias of the participants and the fact that we are also looking into their unconscious behaviour.

To address this, Hofmeister et al. [28] created an online tool that made the participants only able to see a limited amount of lines of code at a certain time. We called this feature the *viewport*, as it is the only visible area of the test suite to the participant. The participants could then shift the viewport over the test to view different parts one line at a time using arrow keys of the keyboard. On the background, the application would then keep track of these key presses and the location of the viewport. This way, the result would be a series of event data containing which lines of code were visible to the participant, and thus most likely reading, in a chronological order.

Functionality wise, the tracking tool created in this research is similar to the design as described in the work by Hofmeister et al. [28]. Configuration wise, there are a few differences with their specific implementation.

While Hofmeister et al. configured the size of the viewport as approximately one third of the code snippet, 7 lines of code, we decided to set this number to 5. This specific number is based on the statistics of the different test classes as described in Tables 3.1, 3.2, and 3.3. As we are looking into the reading patterns across the different AAA parts, the size of the viewport should not be too large that it covers too much of multiple AAA parts, as that will aggregate the proxy focus of the participants too much and thus reduce the accuracy of our data. However, the size should also not be too small, as that will potentially hinder the participants in the execution of the experiment task. Also, to avoid potential confusion or external influences that are not accounted for, we decided to make this number constant across all the three trials (see Figure 3.1).

Based on the statistics of the test classes across the different complexity levels, setting the size of the viewport to 5 lines of code seemed to be the most fitting. In the case of the easy test classes, this number means that the viewport shows slightly less than the whole of

the original test class. For the medium and hard test classes, it is on average slightly larger the size of a single AAA part. Together, these indicate that it is not too large according to the described requirements. In the case of the largest test classes, it is approximately one third of the original test class. This is in line with the configuration of Hofmeister et al., meaning that it should also not be too small.

One of the possible biases that can occur in this experiment is that of a learning curve. This means that the participants will perform better as they progress in the trials. To address this, the order at which the three test suites are provided to each participant is randomized. This minimizes the possible bias introduced by a learning curve due to either the order of the test suites, the order of complexity, or the objectives becoming clearer as the participants progress through the experiment.

Another possible bias that can occur in the reading behaviour of the participants is that of linearity. As stated in work by Busjahn et al. [11], although less when compared to reading natural language text, developers still show a certain degree of linear reading patterns when reading source code. This linearity would be strengthened if every participant's viewport starts on the first line of the test suite. To address this potential bias, the starting viewport of every trial is randomized across the test suite.

Before the participants start with the actual trials of experiment, they will be given instructions as an example to get familiar with our created tracking tool. The instructions contain information regarding how to navigate through the source code with our tracking tool, regarding the context of the test classes, and the participant's objectives are during the experiments. During each official trial, the participants can keep reading the test code until they actively indicate that they are done or until the 10 minute time limits expires. Afterwards, the test suite is removed and the experiment task questions are provided to the user to fill in.

3.2.4 Task Design

During the experiment, the participants are provided test classes with an accompanying software maintenance task. The task consists of two questions related to the purposes of tests in realistic development environments.

1. What is the purpose of the test suite?
2. Describe the additional tests cases that you would write to extend the current test suite. Use 1 line per case.

The objective of the first question is to measure the degree at which the participant has grasped the overall testing purpose of the original test suite. However, in general it is not safe to assume that every existing test suite has a single testing purpose and that every test class evaluates a single test case derived from that testing purpose.

For that reason, in this research the choice was made to extract the test classes from an academic software project (See Section 3.2.5). One of the benefits of this academic software project is that it provides a certain guarantee to the described matter. Due to the structure of the related course and the software project, most of the tests are structured in a way that

every test suite has a single testing purpose with each test class describing another test case. With this, the question is plausible in the sense that it yields an assessable answer which can be compared to one another.

While grasping the overall testing purpose of a test suite is one of the aspects to using tests, another aspect is using that obtained knowledge to be able to work with and extend upon the test suite at hand. This, specifically the latter part, is exactly what the second question of the experiment task assesses. The participants are expected to use the knowledge obtained for the first question and create additional test cases that would extend the provided test suite in a meaningful way. Because the objective is to observe how well the participants are able to express their understanding of the test suite in an extending way, they are requested to form concise, natural language answers that are at most one line in length.

3.2.5 Test Code Selection

As the main objective of this experiment is to measure the tests reading behaviours of developers, it is necessary that the software project of which the test classes originate from and the test classes themselves fulfill a few criteria accordingly:

1. The test classes should be understandable in an isolated and standalone manner. [18]

2. The test classes should adhere to the AAA testing structure in an unambiguous manner so that the all obtained time distributions can be categorized and processed properly.

3. The amount of cognitive load from external factors required to understand the test classes should be as minimal as possible so that the developer can focus on what the test code does rather than what it is.

As external software projects failed to satisfy these criteria properly, the decision was made to use JPacman, a Pacman game implemented in Java and being used for teaching software testing at the Delft University of Technology. Due to the educational purposes of JPacman, the extracted test classes naturally fulfill criteria 1 and 2. The decision also helps significantly with keeping criteria 3 in check. As it is part of an academic course, the project and the accompanying tests were designed with the intention in mind that everyone should be able to understand them with just basic knowledge. Another benefit regarding criteria 3 is the domain of the software project. It is safe to assume that everyone knows the concept of pacman, which reduces the cognitive effort required from participants to understand the underlying purpose.

To avoid any bias through manually picking out test classes for this experiment, test classes were selected randomly according to a systematic process. First, all the test classes in the project are scraped and gathered together. In the case that a test suite also has setup or teardown code specified, the gathered individual test classes are reconstructed with these segments of code.

Test Name	LOC	Arrange	Act	Assert
testSetup	6	1	1	4
deadStart	6	3	1	2
testPlayerPellet	6	2	1	3
nonMovingCollider	6	2	1	3

Table 3.1: Statistics of the test classes of complexity level easy (6 LOC).

Test Name	LOC	Arrange	Act	Assert
s2_1_MoveOverPelletAndConsume	10	4	2	3
s4_Suspend	10	4	3	3
freshStart	9	5	1	3
parsePlayer	9	3	2	4

Table 3.2: Statistics of the test classes of complexity level medium (9.5 LOC).

Test Name	LOC	Arrange	Act	Assert
startAndMove	14	9	3	2
win	14	9	4	1
readOneCharFromResource	14	11	1	2

Table 3.3: Statistics of the test classes of complexity level hard (14 LOC).

Then, this collection is rearranged into ascending order based on their complexity. As a proxy metric for the complexity of a test class, we used the length of the reconstructed test class in terms of lines of code. Blank lines were not counted towards this metric as they add no meaningful information code-wise. However, as previous studies have shown that blank lines may affect the readability and comprehensibility of source code [10, 17, 61], they were included in the test code when used for the experiment tasks.

To reduce the influence of the complexity of the test class on our results, in trend with criteria 3, three test classes from different test suites are extracted from the collection. From the ordered collection, three sets of tests are extracted based on their complexity. Respectively the first quartile, the second quartile (median), and the third quartile represent the complexity of the tests as easy, medium, and hard.

Lastly, for every test class in each of the sets, we then look at the ratio in proportions between the three parts of the AAA pattern. The test class with the most evenly distributed proportions between the three parts is then chosen to represent its complexity level. The reason that this is important is to make sure that every part has an as equal representation and chance of being read as possible.

The results of this process for JPacman are stated in Tables 3.1, 3.2, and 3.3. For the test classes of complexity level easy, as the first test has the worst proportions, the second test describes a confusing interaction in the game, and the remaining two are basically the same test case, *testPlayerPellet* was chosen as the easy test for this experiment. As the medium

complexity level test class, *parsePlayer* was chosen due to the fact that it has the most even proportions, and for the hard complexity level *win* was chosen over *startAndMove* because it statistically is not worse proportioned, but does describe an easier understandable game interaction.

While these three single test classes should be sufficient enough to answer the first question of our experiment ask, regarding the overall testing purpose of the encapsulating test suite, it is most likely not sufficient information to be able to come up with additional test cases based on one single test case. To address this issue, one additional *neighbour* test class from the same test suite was extracted as well. This decision was based on its similarity with the *main*, originally extracted, test class. In all three cases, a *neighbour* test class was found and included in the test suite that either was a slightly modified test case or had slightly modified input to the method under test.

Before these tests can be used for this experiment, they need to post-processed. The objective of this post-processing is to reduce cognitive effort from external factors, criteria 3, even further if possible, and make sure that no external, relative to the test code, factors influence how the participants meet the experiment task objectives.

Regarding the former objective, some tests cases had to be partly rewritten or complemented with source code defined elsewhere but not recognized and picked up by the automatic scraping process. Various test classes contained syntax or code constructs related to mocking. In general, these code construct were left untouched as knowledge on mocks is to a certain point expected for this experiment and the syntax is descriptive enough of their functionalities. An exception were the test classes in the medium complexity test suite, which contained mocking syntax that was not assumed to be common knowledge, specifically *ArgumentCaptors*. The way at which this was rewritten was by replacing the *ArgumentCaptors* with an actual instance of the same object type.

Another construct that was treated for the same reason were fields. Although the usage of field variables was often clear from a domain perspective due to the naming of the variable, the incompleteness of the test code or the lack of object information could still lead to confusion by the participants. Therefore, field declarations and field initializations were manually extracted from the original test suites and added to all of the extracted test suites.

Regarding the latter objective, the test names were removed from the test classes. Test classes are often named towards its testing purpose or the exact test case that it implements. Due to the way the software maintenance related task of this experiment is designed, the name of the test class will influence how people perform on the task, as it will give away critical knowledge. For this reason, all the test names were replaced by a placeholder which has no such influence.

The test source code after post-processing used in this research is available in the appendix.

3.3 Independent and Dependent Variables

In this section, the set of all possible influential factors used in this study will be discussed. This set consists of two types of variables, independent and dependent variables. Indepen-

dent variables are variables that are considered to not be influenced by any other variable in any of our models. These variables are exclusively used as input to the models to answer our research questions and, thus, will not be the variables predicted by any of our models. Dependent variables are variables that are expected to be dependent on other variables. All of the dependent variables will be an output variable to one of our models. However, a subset of dependent variables will also be used as input to the model of other dependent variables.

3.3.1 Independent Variables

This section will focus on the independent variables in our research, which are mainly static properties, and the expectations towards their potential influence on the different metrics of code comprehensibility.

Age The age of the participants. Its main purpose is to provide us insight on the demographical distribution of the participants. While the variable of age has been common subject to research in the field of linguistics regarding its effect on language comprehension [12, 38], it has rarely been so in the field of Computer Science. Although, it is the main variable of interest in this study, it will be included in all the models to discover potential influences. Contrary to language, however, knowledge about software development is severely less correlated to age, as software development is not educated to everyone at the same stage in their lives. Thus, expectations for this study are that the age of the participants will not have any significant effects on their test comprehensibility.

Gender The gender of the participants. Similarly to age, this variable mainly exists to provide insight on the demographical distribution of participants. Although being a controversial topic, gender has already been subject to research in the field of Computer Science in a few previous studies [7, 9, 37, 58]. The focus of these studies is on the potential differences between how genders approach different software development related tasks, with varying results. The general observation is that the main difference lies in the way at which software development related tasks are approached, while the results and performance show no correlation with gender. While gender is not the main variable of interest in this study, expectations are that, similarly to previous studies, that gender will have no influence on the performance related independent variables, namely ITP and PAC. Possibilities are that differences might be observed in the amount of time that participants spend on reading the tests or how they distribute their time across the different AAA sections, as the study by Sharafi et al. [58] showed that female subjects take more time to carefully elaborate on their decisions.

Experience. One of the most common relationships subject to evaluation in the field is between the process of program comprehension and the expertise level of the developer [11, 14, 25, 27, 34, 40, 47, 51, 65]. Building upon these examples of previous work, the research performed in this work will also be targeting and evaluating the relationship between the process of program comprehension and the expertise level of the developer. More

specifically, the aim is to evaluate whether any of our test comprehensibility metrics is correlated to or affected by the experience of participants.

In this research, we will take three different types of relevant experience into considerations. For all these three types the expectation is that, similar to the results of previous studies, more experience in relevant matters will positively affect the task performance and the degree of code comprehensibility [14, 25, 31, 32]. The most general type is the experience of participants as a **developer**. It captures the general software development knowledge of the participant.

Additionally, another type of experience that is interesting for this research is the experience of participants with the **Java** programming language. As is custom in this research field, the source code used in this research will be written in Java. Similarly to other studies, it is thus necessary and interesting to capture the influence of the experience with the programming language on test comprehensibility, if it is present. A previous study by Peitek et al. [53] has already highlighted the positive influence of familiarity with the programming language on program comprehension.

The last type of experience that is accounted for is the experience of participants in using automated source code **tests**. This is also the most specific type of experience relevant to this research. Writing and reading tests is another programming skill on its own, compared to writing production source code. Using this experience, the goal is to capture the influence of experience in this skill on test comprehensibility, if present.

Prior Knowledge of Software Project A binary variable indicating whether the participants have any (prior) knowledge on the domain of the software project. The process at which this variable is derived is described in Section 3.4. Another aspect to understanding and writing tests for a software project is having knowledge on the internals of the software project and what the software does, the domain. Expectations are that domain knowledge will affect all the different dependent variables positively.

Trial The index of the trial (see Section 3.2.1) that the participant is currently working on. Together with *UUID*, this variable is necessary in the data analysis stage to prevent potential patterns and biases occurring due to either of these variables (see Section 3.5).

UUID The unique identifier for the respective participant. Together with *Index*, this variable is necessary in the data analysis stage to prevent potential patterns and biases occurring due to either of these variables (see Section 3.5).

3.3.2 Dependent Variables

This section will focus on the dependent variables in our research to answer our research questions regarding test comprehensibility. We will cover the details of these variables and the way at which these variables are derived or converted in an assessable variable.

TotalTimeInSecs To be able to address the research question regarding the RT of participants (RQ1.1), the experimental tracking data has to be processed into a singular value. This

was calculated by taking the difference in the timestamps of the moment the participant was provided the test suite, and the moment they indicated to have sufficiently understood the test suite and moved on to the experiment task. For our conveniences, the resulting values were also converted from milliseconds to second and named as *totalTimeInSecs*.

%Arrange, %Act, %Assert To be able to address the research questions regarding the time spent on the different sections of the AAA structure (RQ1.2, RQ1.3, RQ1.4), we need numerical variables to capture these properties. For this, a manual analysis was performed over the three extracted test suites to annotate the test code statements belonging to each of the AAA sections. For the majority, this was only done for the semantically contributing test code statements. Code statements with solely syntactic purposes, like curly braces, were not annotated and accounted for, unless they contributed to understanding test code statements that were already categorized in any of the AAA sections. Then, using these annotations, the time that was spent on each of the annotated lines was extracted from the experimental tracking data and summed together per AAA section and per test suite. These absolute numbers per AAA section are highly dependent on the reading time of the participant for that specific trial. To account for this, the numbers were taken proportional to the summed reading time spent on each of the AAA sections of the developer per trial to normalize between all the participants and trials. The resulting variables were named *%AR(range)*, *%AC(t)*, and *%AS(sert)*.

Identifying Testing Purpose To be able to address the research question regarding the ability to ITP of the participants (RQ2), the qualitative data obtained from the open question on general testing purpose (Section 3.2.4) has to be processed into quantitative data. The new independent variable will capture whether the participant has correctly understood and determined the testing purpose of the provided test suite. This will be a binary variable, named *purposeScore*, with a value of 0 indicating that the participant did not identify the testing purpose of the respective test suite and a value of 1 that they did. The quantification criteria of each respective test suite are as following:

Test Suite 1 Quantification Criteria

- 0 if either or both of the individual purposes of the test classes are summed up, player collides onto pellet and ghost respectively, rather than an overall test suite purpose.
- 0 if the answer mentions the testing of collisions, limited to the provided two test cases or without any additional information.
- 1 if the answer mentions additional units besides pacman, player, ghost, and pellet, either implicitly or explicitly.

Test Suite 2 Quantification Criteria

- 0 if either or both of the individual provided test classes are summed up, the mapParser parsing a player and a ghost.

- 0 if the answer is focused on the verification of starting positions or creation of the player and non-player characters (NPCs).
- 1 if the answer mentions the focus being on the mapParser function, having a string as input and a board as output.

Test Suite 3 Quantification Criteria

- 0 if either or both of the individual test classes are summed up, winning and losing the game.
- 1 if the answers mentions related to end game conditions, completion states, or anything else that is a valid abstraction of the two provided test cases.

Producing Additional Cases To be able to address the research question regarding the ability to PAC of the participants (RQ3), similar to ITP, the qualitative data obtained from the open question on producing additional test cases (Section 3.2.4) has to be processed into quantitative data. For this, all the sensible produced additional test cases from the answers to the respective open question were pooled together and manually analyzed. This analysis consisted of grouping additional test cases together in categories that contributed meaningful to the PAC. Furthermore, the amount of categories should not be too large that categories would be too specific and thus cause a skewed distribution towards people not hitting the category, but also not too small that categories would be too general and thus cause a skewed distribution towards everyone hitting the category. In the end, the pool of categories categories of additional test cases that were used in this research are the following four:

- **Basic**. Extension of the provided cases in a limited manner based on the information in the provided cases.
- **Domain**. Extending cases to test valid scenarios and input, but not limiting it to the examples given in the provided cases, from the perspective of the game of the software project. Like additional units.
- **Software**. Extending cases to test valid scenarios and input, but not limiting it to the examples given in the provided cases. Focusing on software development related aspects, rather than the domain. For example, certain conditions like empty arrays or testing classes interactions.
- **Error**. Additional cases with invalid input or dedicated to fail.

Each of these four categories was represented in an additional dependent variable. Similar to the ITP these variables were binary, with a value of 1 indicating that the answer of the respective participant includes at least one test case in that specific category, while a value of 0 means they did not. Thus, producing multiple cases for the same category is not rewarded for in this processing schema, although it is definitely beneficial in practise. However, the focus of this research question is the ability of participants to produce additional

test cases and the variety of types of test cases that they come up with, prioritizing diversity over quantity.

3.4 Participants Selection

For this research, we focused on getting two groups of participants, namely a group that has prior domain knowledge of the software project and another group that does not have this prior domain knowledge.

For the former group, all the students of the 2018 Software Quality and Testing course on the Delft University of Technology, in which the software project is used, were invited to participate in our experiment. Students were invited at the end of the course, after having two months of experience with the JPacman code base. Participation was made clear to be on a voluntarily basis, to not be part of the course, and to not affect the grade of the students in any positive or negative manner. Additionally, fellow students who had completed the course in a previous academic year were also invited to participate in our experiment. For the latter group, the online experiment was shared on several social media platforms to invite developers to participate.

Besides this separation in participants based on whether they have prior domain knowledge of the software projects, there were no additional separations or requirements for participating in our experiment. Significant experience with Java or programming in general, for example, was not required. Rather, diversity in these factors was highly desired and appreciated for the generality of this research.

3.5 Analysis Procedure

In this section, we will cover the procedure at which the models will be analyzed according to the results. This will also cover the criteria of the used research models and the decisions taken regarding satisfying those criteria. Lastly, the statistics that will be used to assess the accuracy of the models are stated.

Reading Time To be able to test the hypotheses regarding RT and determine relevant factors of influence on it, a Linear Mixed Model (LMM) analysis method was used. For each of the dependent variables of interest, *totalTimeInSecs, %AR, %AC, and %AS*, a LMM was constructed with all the independent variables described before as fixed effects. As each participant completes three trials in our experiment and these three trials are the same for every participant, similarities and patterns caused by this overlap in trials and participants can affect the results of our models. To account for these, both of the variables (*index* and *uuid*) were represented as random effects in the models [66].

The resulting model for a LMM analysis method holds several assumptions [21, 67]: *(i)* Linearity in the data of the model, *(ii)* there should not be collinearity between fixed effects, *(iii)* absence of heteroskedasticity, which means that the residuals in the model need to have a similar amount of variation for all the predicted values, *(iv)* the residuals of the LMM

model need to be normally distributed, *(v)* there should be no influential data points, and *(vi)* independence should hold across the data of the model:

- Assumptions *(i), (iii),* and *(iv)* are verified by means of inspecting visual plots of the residuals [67]. Manual visual inspection of the histogram and Q-Q plot of the residuals provides us evidences whether the residuals of the model are normally distributed. A visual plot of the residuals against the fitted values of the models provides us information to manually verify the linearity of the model and the property of heteroskedasticity. In the scenario that these visualizations display significant indications that violate any or multiple of these properties, several options to adjust the model are available to solve these violations [67]. The most common one is to apply a non-linear transformation on the data of the dependent variable (e.g. log transformation) [21, 67].
- Assumption *(ii)* is verified by means of a visual plot of the linearity between every pair of independent variables.
- Assumption *(v)* is verified by means of manual inspection and comparing the full model against reduced models. These reduced models that were tested have a certain part of the head, tail, and both of the data removed based on the dependent variable. None of the reduced models show a significant difference with the full model.
- Assumption *(vi)* was adhered to by conforming to a mixed effect model, rather than just a linear model [66, 67].

The models of these independent variables will be created with R [55] and the *lme4* package [5]. The fitness of the models will be verified by means of marginal and conditional R^2 values [46], using implementations of the *MuMIn* package [4]. For every independent variable, a likelihood ratio test will be conducted of the full model against a reduced model without the effect in question. As is common with testing statistical significance, independent variables were deemed influential over the dependent variable, RT, when there was a probability of at most 5% of committing Type-I error ($\alpha = 0.05$).

Identifying Testing Purpose and Producing Additional Cases Contrary to the RT, the variables capturing the participants' abilities to ITP, *purposeScore*, and PAC, *BasicCase, SoftwareCase, DomainCase, and ErroCase*, are nominal categorical (binary) variables. It takes either the value of 0 or 1, depending on the described quantification criteria and whether they produced an additional case from the respective categories. Due to this, applying a LMM analysis method will result into specific undesired behaviour in the residuals of resulting models [67], meaning that the dependent variables are unsuitable for a linear model.

To still be able to test the hypotheses regarding ITP and PAC, and determine the relevant factors of influence on it, a logistic regression analysis method was performed. Specifically, we chose Binomial Logistic Regression (BLR) which predicts the probability the variable takes either value and the influences of the fixed effects on this probability. The latter will be

the method at which the interactions between the independent variables and the dependent variable is measured.

All the five dependent variables, *purposeScore, BasicCase, SoftwareCase, DomainCase, and ErrorCase*, will be modelled with the same collection of fixed effects, namely the previously described independent variables. On top of those, the variables *RT, %AR, %AC,* and *%AS* are also included in the models as fixed effects. To verify and assess the resulting models, we will use the McFadden's pseudo-R^2 [26] and analyze the deviance tables [1]. Similarly to RT, independent variables are deemed statistical significant and thus influential on the dependent variables when $\alpha \leq 0.05$.

Chapter 4

Results

In the previous chapter, the approach for participant selection, quantifying our dependent variables and analysis of our data, the necessary models per dependent variable and the procedure at which the models need to be verified were described, specifically in Section 3.5. In this chapter, we will thoroughly report the statistical results of these processes and the resulting models. Outlining the structure of this chapter, we first will go over descriptive statistics related to the results of the participants selection and quantification of the dependent variables. Then, we need to address potential correlations between independent variables and verify all the potential assumptions of the LMM and BLR models. Lastly, we will go over the results of all the models and discuss them individually. This will focus on the potential statistically significant independent variables, the statistics of the influence of these variables on the dependent variable, and the resulting fitness value for each model.

4.1 Descriptive Statistics

In this section, we will report descriptive statistics of the participants distribution and the variables in this research. First, the relevant numerical demographical variables of the participants are covered. This includes a separation of the data based on prior domain knowledge, as described in the participants selection (Section 3.4). Then, the resulting distributions of the quantifying process (Section 3.3.2) are reported per dependent binary variable. Lastly, the statistics of the remaining dependent (numerical) variables are reported and elaborated on.

4.1.1 Participants Statistics

After three months of hosting the online experiment a total of 44 developers participated in our research, 22.7% of which have (had) prior domain knowledge of the software project. Furthermore, 20.5% of the participants were female and roughly 86% of the participants were either a developer (39%) or a student (48%). By far the most preferred programming language by the participants is Java (43%), with Python (11%) and C# (9%) trailing behind it.

In Table 4.1, descriptive statistics of the numerical factors are described based on whether the participant had prior domain knowledge of the software project. This includes the set of independent variables, namely age, years of with software development, Java and tests, and the set of dependent variables, namely the percentage of time spent on the arrange, act and assert part, and the total reading time. The biggest notable differences between the two domain groups are in the different types of experience. Participants with prior domain knowledge of the software project generally have less experience across the board, which can be expected based on the criteria of the target group.

	Prior Knowledge of Software Project					**No Prior Knowledge of Software Project**				
Factor	**Min**	**Median**	**Mean**	**SD**	**Max**	**Min**	**Median**	**Mean**	**SD**	**Max**
Age	18	19	19	1.44	23	19	24	26.7	5.94	43
Developer	1.0	2.5	3.2	2.12	7.0	0.0	5.5	7.37	5.40	20
Java	1.0	2.5	3.0	2.18	7.0	0.0	4.0	4.59	3.90	15
Tests	0.0	1.0	1.1	0.84	3.0	0.0	4.0	4.24	3.83	18

Table 4.1: Statistics of the numerical factors from the participants separated based on their prior domain knowledge of the software project.

4.1.2 ITP and PAC

In Section 3.3, the quantification process was described for the resulting dependent binary variables related to the ability of participants to identify the testing purpose (ITP) of and to produce additional cases (PAC) for an existing test suite. In the rest of this section, we will cover the statistics of the resulting distributions based on this quantification process. For both ITP and PAC, it is important make sure that the distributions of all the representing variables are collectively not too skewed towards either of the binary values.

The resulting distribution of the binary variables after the quantification process are described in Table 4.2. In the case of the ITP variables, the results show that every trial has a similar succession rate. For every trial, roughly 18% to 32% of the participants correctly identify the general testing purpose of respective test suite.

Regarding the PAC variables, participants are evenly capable of producing Basic (55%) and Domain (47%) categorized test cases, while participants are less likely to produce Software (19%) and Error (9%) categorized test cases. While the latter two, particular the Error categorized test cases, are skewed towards the failure of the participants, the average succession rate across the test case categories is 32.5%.

4.1.3 Dependent Numerical Variables

In Table 4.3, the distribution statistics of all the numerical dependent variables are listed. These are the proportions of time that participants spent on each respective AAA section during our experiment and the total amount of reading time spent on the test suite. The margins are significant between spending almost the minimal amount of time possible (25.74 seconds) and the maximum amount of time available of 10 minutes (590 seconds).

	ITP			PAC			
	Trial 1	**Trial 2**	**Trial 3**	**Basic**	**Software**	**Domain**	**Error**
FAIL	36	30	36	59	107	70	120
SUCCESS	8	14	8	73	25	62	12

Table 4.2: Distribution statistics of the resulting binary quantified variables for the ability of participants to identify testing purpose (ITP) and produce additional cases (PAC).

Factor	Min	Median	Mean	SD	Max
Arrange (%)	15.27	48.17	49.20	16.94	82.70
Act (%)	7.38	20.11	20.45	5.96	42.10
Assert (%)	4.26	34.12	30.35	17.31	61.70
Reading Time (sec)	25.74	93.04	129.59	107.28	590.35

Table 4.3: Distribution statistics of the numerical dependent variables.

In Figure 4.1, the distributions of the proportions of time spent on each respective AAA section are visualized in the form of boxplots. Based on it, we can observe that in general participants spend the least amount of time on the Act section. The Arrange section of tests are generally where participants spend most of their time, compared to the other AAA sections, but also has the largest absolute differences between participants. While the proportion of time that participants spend on the Assert section is generally in between the other two sections, there are possibilities that it is higher than the proportion of time spent on the Arrange section and/or lower than the proportion of time spent on the Act section.

4.2 Model Assumptions Statistics

Before being able to model our dependent variables and verify them, it is necessary to look into the independent variables and deal with any pair with signs of collinearity as early as possible. In Figure 4.2, scatterplots of all possible pairs of numerical independent variables are displayed. From those scatterplots, we can observe that there is a clear sign of negative correlation between the proportion of their time that participants spend on the *Arrange* section and the *Assert* section of a test. The easiest solution to address this linear relationship between the two variables is to remove one of the variables entirely from all the models, as it will be represented by the other variable through their linear relationship. As between these two variables neither show any significant difference in the scatterplots with the other independent variables, there are no clear benefits or indications to choice for either of them. So, without any particular deterministic reasoning, the proportion of time that participants spend on the *Assert* section was chosen over the *Arrange* section of the test.

After dealing with collinearity between independent variables, the resulting variables are used to model the dependent variables. For these models, several assumptions and/or requirement have to be verified. In particular, for the LMM of the Reading Time (RT) we have to verify the linearity of the model *(i)* and the distribution of the residuals *(iii) and*

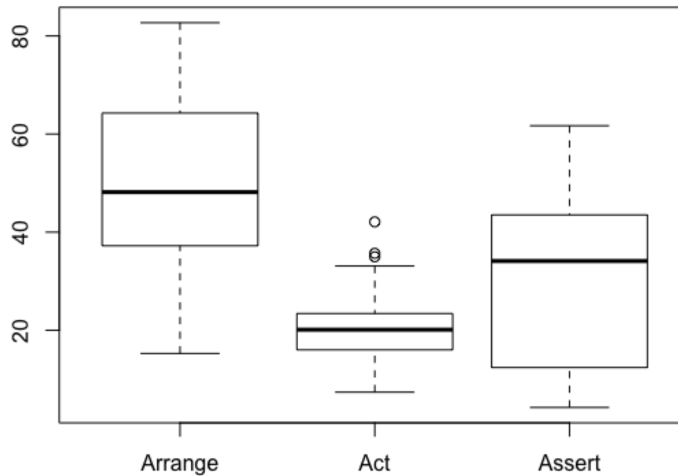


Figure 4.1: Boxplots of the proportional distributions of time spent by participants on each section of the AAA test structure.

(iv). Manual inspection of the histogram and Q-Q plot (left plots in Figures 4.4 and 4.5) show decent but not marginal indications of violations to the linearity of the model and the required normal distribution of residuals. When inspecting the residuals against the fitted values of the model however, displayed in Figure 4.3 (left), there is a noticeable pattern in the graph. Higher fitted values have larger residuals, indicating that the variance is larger in the higher range and smaller in the lower range. This is against the assumed absence of heteroskedasticity *(iii)* and thus renders the model inaccurate.

To address this violation, we applied one of the most common solutions by taking the log transformation of the RT variable. Statistical and graphical analysis showed us that the RT data is log normally distributed, making this a justified transformation to apply. This is further supported by re-inspecting the histogram and Q-Q plot of the newly created model, displayed in Figure 4.4 and 4.5 (right plots), which are improved upon the original model and display better indications of linearity of the model and normality of the residuals. Furthermore, the residuals plot against the fitted values of the new model, displayed in Figure 4.3 (right), shows no marginal pattern in the variations of the residuals. Thus, the dependent variable RT is log transformed.

4.3 Model Statistics

The results of all the LMM and BLR models are reported in Table 4.4 in the form of the p-values of all fixed effect per dependent variable. All the statistically significant results,

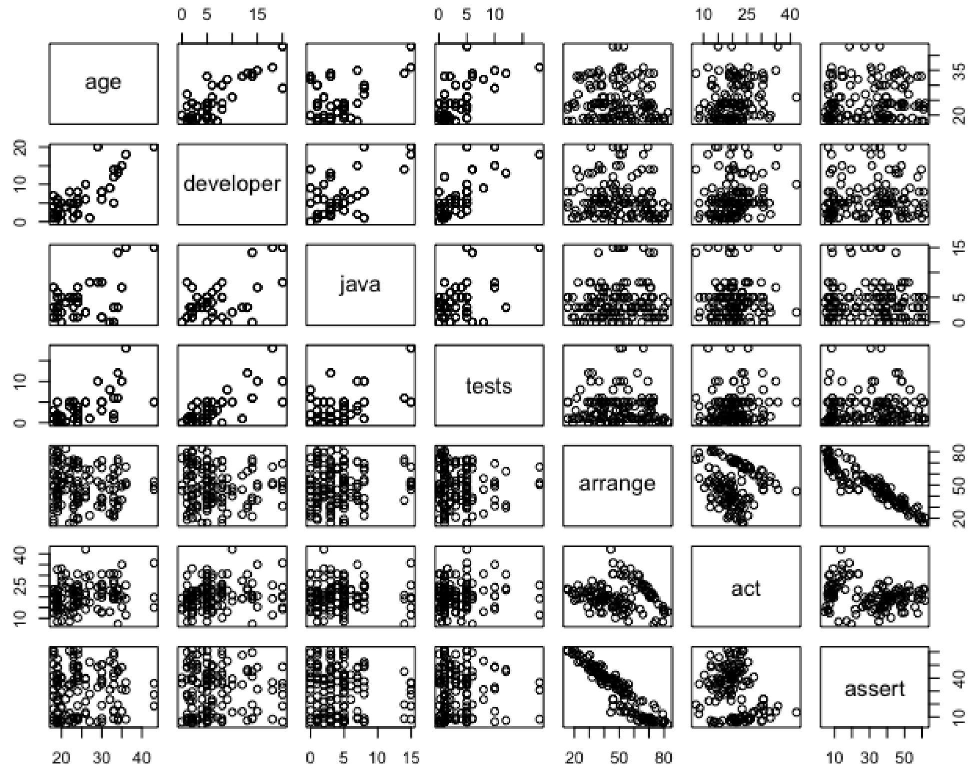


Figure 4.2: Pairwise scatterplots of all the numerical independent variables.

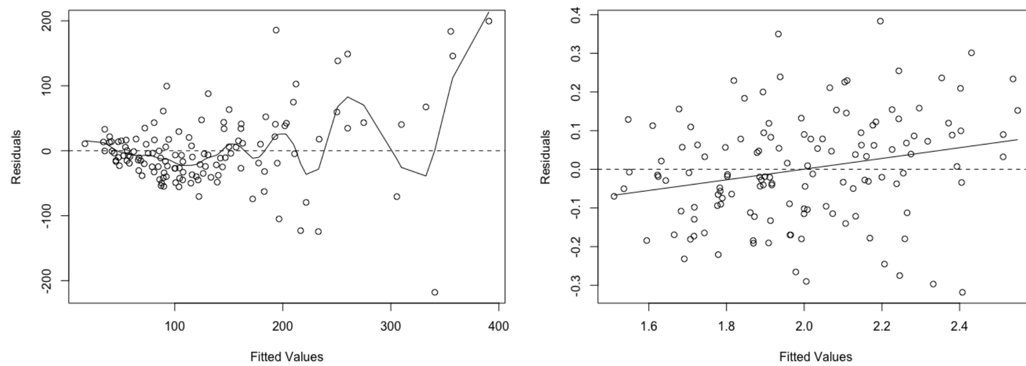


Figure 4.3: Residuals plotted against the fitted values of the two models, RT as is (left) and RT log transformed (right).

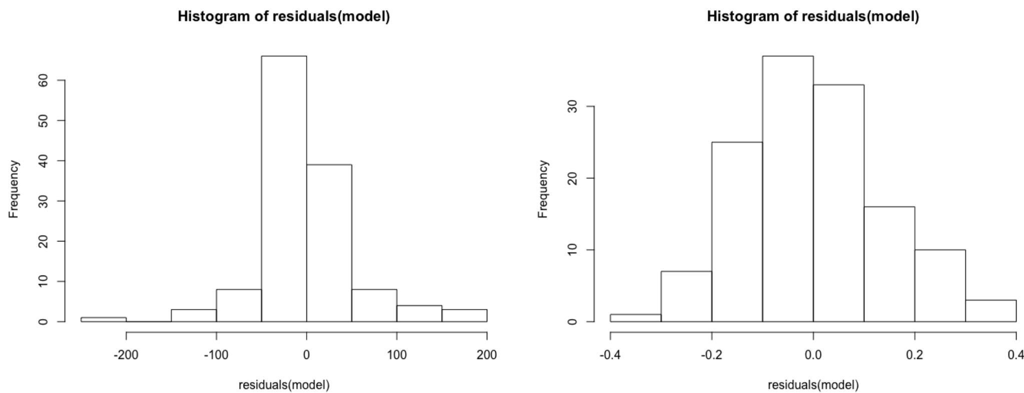


Figure 4.4: Histograms of the residuals of the two models, RT as is (left) and RT log transformed (right).

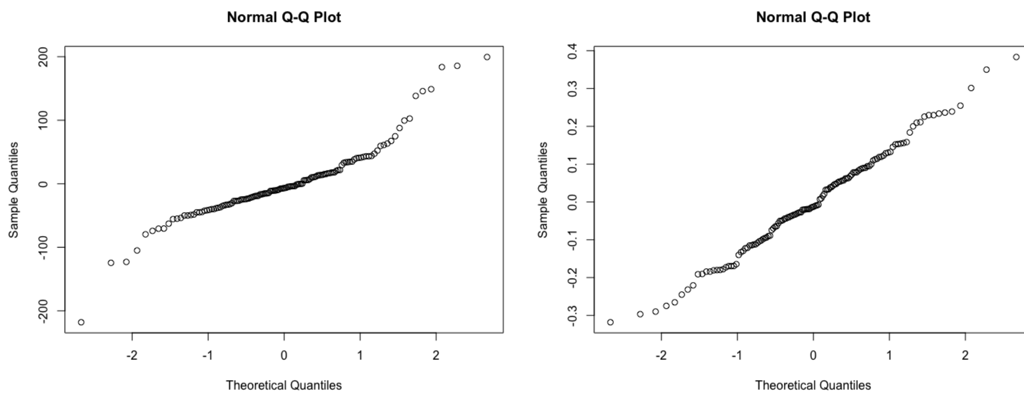


Figure 4.5: Q-Q Plots of the residuals of the two models, RT as is (left) and RT log transformed (right).

when the p-value is lower than or equal to $\alpha = 0.05$, are marked with a * symbol.

From the results, we can first observe that all the demographical variables, both the age and gender of the participants, have no statistical significant influences on any of the dependent variables. On top of that, the experience of the participant as a developer and the proportions of the time that they spend on each individual AAA section (Act, Assert, and Arrange through Assert) also has no statistically significant effect on any of the dependent variables. In the following, we will discuss each model in detail.

Reading Time (RT) In the case of the reading time of the participants, the results indicate that none of the measured experience of the participants, as a developer, with Java, and with using tests, has statistically significant effect on the dependent variable. The only statistically significant independent variable is whether the participant has any prior domain

Table 4.4: All the models represented per row with the relevant independent variables in the columns. The values are the coefficients of the fixed effects on the dependent variable.

Predicted Variables	**Age**	**Gender**	**Develop**	**Java**	**Tests**	**Domain**	**%Ac**	**%As**	**RT**
RT (log)	0.005	-0.090	-0.002	-0.010	-0.013	**-0.259***	-0.002	0.002	-
%Ar	-0.442	-4.300	-0.092	**0.867***	0.051	-0.480	-	-	0.008
%Ac	0.056	1.93	0.169	-0.223	-0.121	-2.285	-	-	-0.001
%As	0.383	2.408	-0.073	**-0.636***	0.071	2.895	-	-	-0.005
purposeScore	0.063	0.018	0.007	-0.135	**0.272***	0.773	0.046	-0.038	-0.000
BasicCase	-0.016	0.042	-0.063	0.106	0.096	0.194	-0.067	0.021	0.000
DomainCase	0.021	0.077	-0.090	-0.051	**0.339***	**1.975***	-0.001	-0.037	0.000
SoftwareCase	-0.129	1.224	-0.030	0.049	**0.266***	0.478	0.021	0.008	-0.000
ErrorCase	-0.168	0.285	-0.061	**0.345***	0.236	1.982	0.159	-0.019	**0.000**†*

* Statistically significant effect ($\alpha \leq 0.05$).
† = *1.028e-5*

knowledge of the software project that the test suites are targeted at. The log transformed RT is decreasingly affected by the prior domain knowledge ($\chi^2(1) = 5.364, p = 0.022$), lowering it by about 0.259 ± 0.109 log transformed seconds, which converted is 1.815 ± 1.284 seconds. The marginal and conditional R^2 of the model are respectively 0.165 and 0.726.

Arrange (%Ar) In the case of the proportion of their time that participants spend on the Arrange section of the test classes, the results indicate that their experience with Java is statistically significant effect on it, while experience in using tests and domain knowledge are not. The *%Ar* variable is increasingly affected by the amount of experience in Java in years ($\chi^2(1) = 6.730, p = 0.009$), increasing it by $0.867\% \pm 0.328$. The marginal and conditional R^2 of the model are respectively 0.049 and 0.586.

Act (%Ac) In the case of the proportion of their time that participants spend on the Act section of the test classes, the results indicate that none of the independent variables used in this research have statistically significant effects on the dependent variable. The marginal and conditional R^2 of the model are respectively 0.100 and 0.298.

Assert (%As) In the case of the proportion of their time that participants spend on the Arrange section of the test classes, the results indicate that their experience with Java is the only statistically significant effect on it. The *%As* variable is decreasingly affected by the amount of experience in Java in years ($\chi^2(1) = 4.920, p = 0.02$), lowering it by $0.636\% \pm 0.276$. The marginal and conditional R^2 of the model are respectively 0.021 and 0.776.

Purpose Score (ITP) In the case of the ability of participants to identify the testing purpose of a provided test suite, the results indicate that the only independent variable that has

a statistically significant effect on it is their experience with using tests. On the contrary, their experience as a developer and Java, whether they have any prior domain knowledge, the proportion of their time that they spend on any of the AAA sections and the total reading time of the participants have no statistically significant effects with the ITP. The odds to ITP is increasingly affected by experience with using tests (*p-value* = 0.008), increasing the log-odd by 0.272 ± 0.103. For every unit increase in years of experience with using tests, the odds of being able to identify the testing purpose of a test suite increased by $e^{0.272} = 1.313$ times. The McFadden pseudo-R^2 value is 0.216.

Basic Cases (PAC) In the case of the ability of participants to produce at least one additional test case that satisfies the criteria of being a basic case, the results indicate that none of the independent variables used in this research have statistically significant effect on the dependent variable. The McFadden pseudo-R^2 is 0.052.

Domain Cases (PAC) In the case of the ability of participants to produce at least one additional test case that satisfies the criteria of being a domain related extending case, the results indicate that the independent variables that are of statistically significant influence are their experience with using tests and whether they have prior domain knowledge. Experience as a developer and with Java, and the time related independent variables, on the contrary, show no statistically significant relation. The McFadden pseudo-R^2 value is 0.157.

The odds to produce additional test cases in the *Domain* category is increasingly affected by experience with using tests (*p-value* = 0.002), increasing the log-odd by 0.339 ± 0.108 times. For every unit increase in years of experience with using tests, the normal odds of being able to produce at least one domain extending case is increased by $e^{0.339} = 1.404$ times.

These odds are also increasingly affected by having prior domain knowledge (*p-value* = 0.001), increasing the log-odd by 1.975 ± 0.619. Thus, having prior domain knowledge of the software project increases the odds of producing at least one domain extending test case by $e^{1.975} = 7.207$ times compared to not having any prior knowledge.

Software Cases (PAC) In the case of the ability of participants to produce at least one additional test case that satisfies the criteria of being a software related extending case, the results indicate that the only factor with statistically significant effect is their experience with using tests. The odds to produce additional test cases in the *Software* category is increasingly affected by experience with using tests (*p-value* = 0.024), increasing the log-odd by 0.266 ± 0.118 times. This means that for every unit of years of experience with using tests, the normal odds of producing at least one software extending test case is increased by $e^{0.266} = 1.304$ times. The McFadden pseudo-R^2 value is 0.153.

Error Cases (PAC) In the case of the ability of participants to produce at least one additional test case that satisfies the criteria of testing an error, the results indicate that the factors of statistically significant influence are their experience in Java and the total RT on that particular test suite. On the other hand, whether they have prior domain knowledge of

the software project or their experience as a developer or with using tests is not statistically relevant regarding this dependent variable. The McFadden pseudo-R^2 value is 0.279.

The odds to produce additional test cases in the *Error* category is increasingly affected by experience with Java (*p-value* = 0.023), increasing the log-odd by 0.345 ± 0.152 times. This means that every unit increase in years of experience with Java, the normal odds of producing at least one error handling case is increased by $e^{0.345} = 1.411$ times.

Similarly, the RT of the participant for a specific test suite also holds a statistically significant increasing effect on these odds. Specifically, the log-odd is increased by $1.028e-05 \pm 3.769e-06$ for every unit increase in seconds that the participant spends reading the specific test suite, which converts to an increase in normal-odds of 1.00001 times per additional second spent.

Chapter 5

Discussion

In this chapter, we revisit the research question, formulate answers based on the obtained results of our experiment, and elaborate how our findings compare against existing relevant literature. Then, we present the practical implications of our results and findings for both scientific and real world environments. Lastly, we elaborate on threats that can affect the validity of our study.

5.1 Revisiting Research Questions

In this section, we will revisit the stated research questions of this research. Based on the results of our experiment, answers will be formulated for each respective research question. An overview of out findings are reported in Table 5.1. Besides formulating answers onto the research questions, the findings of this research will be reflected upon using related work in the field where possible.

> **RQ**: What are the factors of influence on the time that developers spend reading test code?

Based on the results of our experiment, the sole influential factor in the set of variables used in this research on their reading time is whether participants had any prior knowledge of the software project. Having prior knowledge of the software project has a decreasing effect on the time that developers spend on reading test code. Contrary to similar studies in the field focusing on aspects relevant to reading source code [11, 14, 25, 27, 28, 40, 51, 56, 59], our results indicate no differences in the reading time between experts and novices. While observing significant differences based on expertise level is common in studies in this field, none of the three types of considered experience in this experiment show significant influence on the reading time of participants.

> **RQ**: How do the different parts of AAA compare to one another regarding the time that developers spend on them and the influential factors?

Based on the distributions of proportional time spent by participants on each section of the AAA test structure, several observations can be made. In general, developers spend the

least amount of their time on the Act section of test classes, while most of their time is spent on the Arrange section. The time spent by developers on the Assert section is on average between the other two sections, but in certain scenarios can be less than the Act section and/or more than the Arrange section. In terms of variation, the Act section has the lowest variation of all the sections, while the Arrange section has the largest variation.

Our results indicate that the only factor of influence on the time that developers spend on reading the Arrange part of test code is their experience with the Java programming language. More specifically, the time spent on reading the Arrange part of developers is increasingly affected by their amount of experience with Java. For the Act part of test cases, our results indicate that none of the considered factors in this research has a significant impact on the time spend by developers on reading it. Similar to the Arrange part, the time spent on reading the Assert part of test code is only affected by the amount of Java experience of participants, but in a decreasing manner.

> **RQ**: What are the factors of influence on the ability of developers to identify the testing purpose of a test suite?

As a metric of understandibility, the ability of participants to identify the testing purpose (ITP) of a test suite was used in this research. Based on the results of our experiment, we can conclude that the sole influential factor on ITP is the experience of the participants with using tests. Every additional year of experience with using tests has an increasing effect on the odds of correctly identifying the testing purpose of a test suite.

> **RQ**: What are the factors of influence on the ability of developers to produce additional test cases to extend the test suite?

As a metric of extensibility, the ability of participants to produce additional test cases (PAC) to extend a test suite was used in this research. To answer the respective research question, a collection of test case categories was constructed based on the answers of the participants. In this research, this collection consists of four different categories. For each respective category, our results have indicated several findings. First, there are no influential factors on the ability of developers to produce basic test cases (i.e., test cases only based on information in the provided test cases). Prior knowledge of the software project and the experience of participants with using tests have increasing effects on the ability of participants to produce domain test cases (i.e., test cases extending the provided test suite with scenarios of the game of the software project). The experience of participants is the sole influential factors on the ability of participants to produce software test cases (i.e., test cases extending the provided test suite with software development related aspects). Lastly, the influential factors for being able to produce error test cases (i.e., test cases validating invalid or erroneous scenarios) are the experience with the Java programming language and the time spent on reading the test code of participants, both having an increasing effect.

Contrary to the reading time, the rest of the findings in our study are in line with existing literature stating the differences in program comprehension between developers with different levels of expertise [11, 14, 25, 27, 28, 40, 51, 56, 59]. However, our findings also differ from existing literature by going beyond solely finding these differences based

Dependent Variable	**Significant Independent Variables**
Reading Time (RT)	Domain (↓)
Arrange (%)	Java (↑)
Act (%)	*None*
Assert (%)	Java (↓)
Purpose	Tests (↑)
Basic Case	*None*
Domain Case	Tests (↑), Domain (↑)
Software Case	Tests (↑)
Error Case	Java (↑), RT (↑)

Table 5.1: Overview of the dependent variables measured in this research with their respective independent variables that have a statistically significant influence, if any. Increasing influences are depicted with the up facing arrow and decreasing influences with the down facing arrow.

on expertise level. Particularly, compared to existing studies this study also looks into the impact of different types of experience (i.e., as a developer, with the Java programming language, and with using tests) and state the specific impact that independent variables have on the different metrics of tests comprehension. All the dependent variables regarding the understandability and extensibility metrics of tests comprehension, for which at least one significant relationship was found, are positively impacted by some type of experience. With this, the differences in impact of different types of experience can be observed. Experience as a developer has no impact on tests comprehensibility, while experience with Java positively affects the likelihood of producing an erroneous scenario testing test case. The most influencing factor is the developers' experience with using tests, positively influencing the likelihood of almost all of the remaining dependent variables.

5.2 Implications

Based on the results and answers described in the previous section, several implications can be observed relevant to both scientific and real world environments.

Interestingly, the only independent variable to significantly impact how long developers spend on reading provided source code is whether they have prior knowledge of the software project. Researchers in the field should be wary of the source code that they use and whether their participants have any previous familiarity with it if reading time is of their interest. They should either avoid using more popular software projects or code snippets or control for whether the participants had any prior knowledge of the software project to dismiss this influence. This influence is, in the context of our research, only applicable to the amount of time that developers spend on reading the test code and whether they are able to produce additional test case regarding the domain of the software project.

For real world environments, this influence means that developers spend less time on reading test code if they are more familiar with the software project. Companies and teams

should focus on getting newcomers and developers familiar with their software project to reduce the amount of time that they spend on reading the test code and thus increase the time available for other software development or maintenance related tasks.

For the more practical variables of this research, whether the participants are able to identify the testing purpose of a test suite (ITP) and produce additional cases to extend the test suite (PAC), the most important factor of influence their expertise level. Across all the dependent variables regarding these factors, experience with Java and particularly experience with using tests are of positively influence on them. Developers with more experience in using tests are more likely to correctly understand a test suite and extend upon it using test cases related to both the domain and programming aspects of the software project. The same holds for the participants' experience with Java and their likelihood of producing additional error testing test cases. From a scientific point of view, it is another emphasis that the expertise level of developers is an important factor to a form of program comprehension, namely tests comprehension. Distinctions should be made between different types of experience, however, as their specific impact is varying based on our results.

Based on this information, in real world environments, the focus should be to educate developers on the principles, benefits and nuances of testing and increase their experience with using tests in order to improve their test comprehensibility capabilities. From the results we can also observe that prior knowledge of the software project also increases the likelihood of producing additional domain related test cases and that experience with Java and the amount of time spent on reading the test suite increases the likelihood of producing an error testing case. Depending on what factor of tests comprehensibility is valued more, certain specific variables and aspects are more important towards improving that factor. In general however, experience with using tests is the most important and widely covering factor of influence on the practical aspects of tests comprehensibility and thus should be focused on.

5.3 Threats to Validity

In this section, the threats that could affect the validity of the results and conclusions of this research are discussed. Similar to other work in the field [21, 28], the threats and the remaining of this section are structured according to the different types, namely internal validity, construct validity, and external validity.

5.3.1 Threats to Internal Validity

Contrary to the study by Schankin et al. [56], in our experiment we made the decision to have a static viewport size of 5 lines based on the proportions of the test cases rather than it being dynamically computed based on the displayed source code (one fourth of the total lines). As our experiment consists of three trials using test suites with a varying amount of lines of code, the viewport shows a different proportion of the code during each trial. While we derived the number 5 based on the varying lengths of the trial test suites and of their AAA sections, this proportion could affect the measured times spent on the different AAA sections and the total reading time as the tool is unable to accurately track significant

eye moments like saccades. To account for this, future studies could either make sure that the provided code for every trial has an equal amount of lines, similarly to Schankin et al. [56], or perform a dynamic computation of the viewport for each respective trial. The former poses an additional considerable restriction on the source code selection process, while the latter imposes an additional threat, namely the impact of a varying viewport on the adjustability of participants.

All the participants were invited on an invitational basis, which possibly has an influence on the results due to volunteers generally being more motivated. To account for, we did not set any requirements for developers to participate in our experiment and performed no post-processing of the data based on independent variables like filtering out outliers.

In our experiment, the participants were given a maximum amount of time to read the provided test code, contrary to scenarios in realistic environments. While this time limit exists to prevent participants from spending a significant amount of time on the test code until they have over-analyzed it or registering enormous numbers due to web browser related issues, there is the possibility that participants feel rushed by the timer and thus cause the answers to the open questions to be of lower quality. Inspection of the respective distribution in Table 4.3 shows that the distribution is not skewed towards the upper time limit (600 seconds). Rather, none of the participants has spent the full amount of time in any of the trials and during 97.0% of the recorded trials less than 400 seconds was spent on reading the provided test code. Based on this we conclude that the time limit of ten minutes had no influence on the quality of our results.

5.3.2 Threats to Construct Validity

In this research, we have associated and measured test comprehensibility with three different factors, namely the total time that participants spend on reading the test suite (RT), their ability to identify the testing purpose of a test suite (ITP) and their ability to produce additional cases to extend the test suite (PAC). While the factor of time is a representative partial metric of program comprehensibility, as used by previous studies [27, 28, 31, 40, 43, 56, 59], no other studies have looked into factors similar to ITP and PAC or argued about metrics of tests comprehensibility in general. To assess the participants' program comprehension ability, previous studies require them to perform software maintenance or development related tasks like finding a defect in a program [28, 43, 50, 56, 59], as this is a reasonable representation of the program comprehension workflow of developers in reality. This, however, does not apply similarly to the workflow of tests comprehension. For this reason, we came up with our own metrics based on what we identified to be important factors in a realistic workflow of tests comprehension, namely the ability to understand a test suite and then to extend it with additional cases. Future work can focus on investigating the representability and accuracy of these factors.

Contrary to many similar studies in this field [11, 25, 32, 40, 51, 59, 65], in this experiment it was decided to not use eye tracking as a tool to track the data about where developers are looking at in a test class, but rather create our own tracking software for this purpose similar to studies by Hofmeister et al. [27, 28], Schankin et al. [56]. The main trade-off that contributed to this decision was between quality and quantity of the data to be gathered.

Eye tracking technology provides better fine-grained tracking data, laying out the specific location of the focus of the participating developer on a chronological timeline. In terms of quality of the data, our approach of forcing participants to look at the test code through a reduced window viewport provides less fine-grained information. It is only possible to measure the focus of participants on a line of code basis, rather than the specific location on the screen, and spreads the tracked focus over all the lines in the viewport, possibly causing an overestimation of time spent on certain lines of code if they are shown in the viewport but not actually focused on by the participants.

While performing an eye tracking experiment does yield major benefits in terms of the quality, it does come with major drawbacks on how and how much data can be acquired. Conducting an eye tracking experiment requires a simultaneous physical presence of both the participant and the researcher. Although an eye tracking device was available to this research, the experiment would have had to be conducted at a specific location at the university of the researcher. Additionally, processing participants in an eye tracking experiment would only be possible in a sequential manner. Taking into account the time to set up such an eye tracking experiment and run all parts of it, conducting such an experiment would require a significant amount of time for a limited amount of data.

Looking at the nature of this research, it falls under the category of descriptive research. The main objective is to learn more in details about the tests reading behaviours of developers when performing general software maintenance tasks. The results of this research are aimed to be fundamental statements regarding this matter, which can serve as a foundation for future research and work to come. To be able to make these statements as general and representative as possible, having sufficient data to base these statements on is crucial. For this reason, the decision was made to opt out of conducting an eye tracking experiment and opt into conducting this research in an online matter by building our own tracking tool. While the hit in fine-grained data by opting out of eye tracking is a significant one, the proxy that was created in our own tool through the reduced window viewport is sufficient enough for what is needed in this research. Weighting this loss against the gain in quantity of data by opting into creating our own tracking tool and conducting the experiment in an online matter was deemed worth it, especially when keeping the context and objective of this research in mind.

5.3.3 Threats to External Validity

For our study, a portion of the invited participants were students (22.7%), thus diminishing the generalizability of our findings across the population of professional developers. Besides a set of general advantages of using students as participants [21], the reason specific to our research is their prior knowledge of the software project. Whether the participants had any prior knowledge of the software project is a factor that has rarely been looked into in research towards program comprehension, while it can have considerable influence (as also shown by our results). While it is possible to create an experimental group by making them familiar with the used software project as part of the (online) experiment, this would require significant time and effort to control this group and extend the experiment. Instead, we had the availability of students already familiar with the software project and opted to

invite them on an optional basis.

All of the established models in this study have a R^2 value lower than 0.28, which means that the models only explain a small proportion of the variability of the data, have a low fitness rate, and thus are not quite accurate in predicting the values for the predicted variables. In this study, however, the importance of the models are not the prediction values, but rather the coëfficients and the correlations between the fixed effects and the predicted variable. The presence and significance of these correlations are independent of the R^2 value. Thus, these low values are not considered as critical harm for the results and findings of this study.

Furthermore, the participants with no prior knowledge of the software project were invited through the social media platforms Twitter and Facebook, and a mailing list. Developers in the social media circle and/or the mailing list are likely to share similar software development interests and thus specializations. This could potentially affect the generalizability of our findings due to differences in specializations and how these groups of developers approach our experiment compared to our expectations. Particular for this research, the descriptive statistics in Section 4.1 provide us with the insight that the majority of the participants has a tendency towards Java. While this meets our expectations, this means that our findings are skewed towards the Object Oriented Programming paradigm or the Java programming language. Future studies should keep their distribution developers in mind based on their research focus.

Chapter 6

Conclusion

In this paper, we present an online empirical study to investigate influential factors on the tests comprehension process of developers. To measure the degree of tests comprehension, we decompose it into three different metrics: *(i)* how much time developers spend on reading a test suite, measured by their total reading time and the proportions of it that they spend on each respective section of the Arrange-Act-Assert test structure, *(ii)* whether developers are able to correctly identify the testing purpose of a test suite and *(iii)* their ability to extend the provided test suite with additional test cases of four different categories, Basic, Domain, Software, and Error.

The main findings based on our results are that *(i)* having prior knowledge of the software project decreases the amount of time developers spend on reading the provided test suite, *(ii)* experience with the Java programming language affects the proportions of time spent on the Arrange and Assert section of tests, *(iii)* experience with the Java programming language and having prior knowledge of the software projects increases the likelihood of producing certain categories of additional test cases and *(iv)* the most influential factor towards the understanding and extending of a test suite is experience with using tests, having an increasing effect.

Based on these findings, we speculate that companies should focus on getting developers familiar with the software project and educating developers on the principles, benefits and nuances of testing to decrease the amount of time spent on reading tests, increase their experience with using tests, and increase their tests comprehension capabilities. In scientific context, the results of this study *(i)* indicate that future studies in the field should be wary of the influence of developers having prior knowledge of the software project on their reading time and *(ii)* are in line with existing literature in the field stating the differences in program comprehension between developers with different levels of expertise. Our study differs from existing literature by going beyond solely stating these differences and provide the specific impact that these factors have on the different metrics of tests comprehension.

To our knowledge, this is the first study to use existing work on understanding program comprehension to extensively look into the process of tests comprehension. This paper serves as fundamental basis for future work towards the understanding of tests comprehension. Besides our findings, this study also contributes an overview of the current state of research towards understanding the process of program comprehension, focusing on the

different research topics and research methods. As such, we call on researchers to expand and improve upon work conducted in this study, either in directions discussed in this work or others. In order to improve the realistic relevancy of research in the field and the quality of produced tools in the community, it is necessary to gain a better understanding of how developers perform software development and maintenance related tasks, to which tests comprehension is no exclusion.

Bibliography

[1] Michy Alice. How to perform a logistic regression in r, Sep 2015. URL https://datascienceplus.com/perform-logistic-regression-in-r/.

[2] Mauricio Finavaro Aniche, Gustavo Ansaldi Oliva, and Marco Aurelio Gerosa. What do the asserts in a unit test tell us about code quality? a study on open source and industrial projects. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 111–120. IEEE, 2013.

[3] Ivan Bacher, Brian Mac Namee, and John D Kelleher. The code mini-map visualisation: Encoding conceptual structures within source code. *2018 IEEE Working Conference on Software Visualization*, 2018.

[4] Kamil Barto'n. *MuMIn: Multi-Model Inference*, 2018. URL https://CRAN.R-project.org/package=MuMIn. R package version 1.42.1.

[5] Douglas Bates, Martin Mächler, Ben Bolker, and Steve Walker. Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48, 2015. doi: 10.18637/jss.v067.i01.

[6] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Softw. Engg.*, 20(4): 1052–1094, August 2015. ISSN 1382-3256. doi: 10.1007/s10664-014-9313-0. URL http://dx.doi.org/10.1007/s10664-014-9313-0.

[7] Laura Beckwith, Margaret Burnett, Susan Wiedenbeck, Curtis Cook, Shraddha Sorte, and Michelle Hastings. Effectiveness of end-user debugging software features: Are there gender issues? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '05, pages 869–878, New York, NY, USA, 2005. ACM. ISBN 1-58113-998-5. doi: 10.1145/1054972.1055094. URL http://doi.acm.org/10.1145/1054972.1055094.

[8] Frederick P. Brooks, Jr. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-83595-9.

[9] Margaret Burnett, Scott D. Fleming, Shamsi Iqbal, Gina Venolia, Vidya Rajaram, Umer Farooq, Valentina Grigoreanu, and Mary Czerwinski. Gender differences and programming environments: Across programming populations. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '10, pages 28:1–28:10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0039-1. doi: 10.1145/1852786.1852824. URL http://doi.acm.org/10.1145/1852786.1852824.

[10] Raymond P. L. Buse and Westley R. Weimer. Learning a metric for code readability. *IEEE Trans. Softw. Eng.*, 36(4):546–558, July 2010. ISSN 0098-5589. doi: 10.1109/TSE.2009.70. URL http://dx.doi.org/10.1109/TSE.2009.70.

[11] 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 *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 255–265, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2820282.2820320.

[12] Jasone Cenoz. The influence of age on the acquisition of english: General proficiency, attitudes and code mixing. *Age and the acquisition of English as a foreign language*, pages 77–93, 2003.

[13] Bas Cornelissen, Arie van Deursen, Leon Moonen, and Andy Zaidman. Visualizing testsuites to aid in software understanding. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering*, CSMR '07, pages 213–222, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2802-3. doi: 10.1109/CSMR.2007.54. URL https://doi.org/10.1109/CSMR.2007.54.

[14] Martha E Crosby, Jean Scholtz, and Susan Wiedenbeck. The roles beacons play in comprehension for novice and expert programmers. In *Proceedings of the 14th Annual Psychology of Programming Interest Group Conference*, PPIG 14, 2002.

[15] Peli de Halleux and Nikolai Tillmann. Parameterized test patterns for effective testing with pex. *Research in Software Engineering, Microsoft research*, 21:16, 2008.

[16] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In M. Marchesi, editor, *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[17] Rodrigo Magalhães dos Santos and Marco Aurélio Gerosa. Impacts of coding practices on readability. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 277–285, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5714-2. doi: 10.1145/3196321.3196342. URL http://doi.acm.org/10.1145/3196321.3196342.

[18] Sarah Fakhoury, Yuzhan Ma, Venera Arnaoudova, and Olusola Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 286–296, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5714-2. doi: 10.1145/3196321.3196347. URL http://doi.acm.org/10.1145/3196321.3196347.

[19] Thomas Fritz, Andrew Begel, Sebastian C. Müller, Serap Yigit-Elliott, and Manuela Züger. Using psycho-physiological measures to assess task difficulty in software development. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 402–413, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568266. URL http://doi.acm.org/10.1145/2568225.2568266.

[20] D. Fucci, H. Erdogmus, B. Turhan, M. Oivo, and N. Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE Transactions on Software Engineering*, 43(7):597–614, July 2017. ISSN 0098-5589. doi: 10.1109/TSE.2016.2616877.

[21] Davide Fucci, Simone Romano, Maria Teresa Baldassarre, Danilo Caivano, Giuseppe Scanniello, Burak Thuran, and Natalia Juristo. A longitudinal cohort study on the retainment of test-driven development. *arXiv preprint arXiv:1807.02971*, page 10, 07 2018.

[22] Robert L. Glass. Facts and fallacies of software engineering, 2002.

[23] Michaela Greiler, Arie van Deursen, and Andy Zaidman. Measuring test case similarity to support test suite understanding. In *Proceedings of the 50th International Conference on Objects, Models, Components, Patterns*, TOOLS'12, pages 91–107, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0_8. URL http://dx.doi.org/10.1007/978-3-642-30561-0_8.

[24] J. Grigg. *Arrange Act Assert*, 2012. URL http://c2.com/cgi/wiki?ArrangeActAssert. [Online].

[25] Emlyn Hegarty-Kelly, Susan Bergin, and Aidan Mooney. Using focused attention to improve programming comprehension for novice programmers. In *Proceedings of the Third International Workshop on Eye Movement in Programming: Model to Data*, EMIP '15, pages 8–9. The University of Eastern Finland, 2016.

[26] Giselmar A. J. Hemmert, Laura M. Schons, Jan Wieseke, and Heiko Schimmelpfennig. Log-likelihood-based pseudo-r2 in logistic regression: Deriving sample-sensitive benchmarks. *Sociological Methods & Research*, 47(3):507–531, 2018. doi: 10.1177/0049124116638107. URL https://doi.org/10.1177/0049124116638107.

[27] Johannes Hofmeister, Jennifer Bauer, Janet Siegmund, Sven Apel, and Norman Peitek. Comparing novice and expert eye movements during program comprehension. In

Proceedings of the Fourth International Workshop on Eye Movement in Programming, EMIP '17, pages 17–18, 2017.

[28] Johannes C. Hofmeister, Janet Siegmund, and Daniel V. Holt. Shorter identifier names take longer to comprehend. *Empirical Software Engineering*, Apr 2018. ISSN 1573-7616. doi: 10.1007/s10664-018-9621-x. URL https://doi.org/10.1007/s10664-018-9621-x.

[29] Kenneth Holmqvist, Marcus Nyström, and Fiona Mulvey. Eye tracker data quality: What it is and how to measure it. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '12, pages 45–52, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1221-9. doi: 10.1145/2168556.2168563. URL http://doi.acm.org/10.1145/2168556.2168563.

[30] Marcin Jamro and Bartosz Trybus. Testing procedure for iec 61131-3 control software. *IFAC Proceedings Volumes*, 46(28):192 – 197, 2013. ISSN 1474-6670. doi: https://doi.org/10.3182/20130925-3-CZ-3023.00018. URL http://www.sciencedirect.com/science/article/pii/S1474667015373237. 12th IFAC Conference on Programmable Devices and Embedded Systems.

[31] Ahmad Jbara and Dror G. Feitelson. On the effect of code regularity on comprehension. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 189–200, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2879-1. doi: 10.1145/2597008.2597140. URL http://doi.acm.org/10.1145/2597008.2597140.

[32] Ahmad Jbara and Dror G. Feitelson. How programmers read regular code: A controlled experiment using eye tracking. In *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension*, ICPC '15, pages 244–254, Piscataway, NJ, USA, 2015. IEEE Press. URL http://dl.acm.org/citation.cfm?id=2820282.2820319.

[33] H. M. Kienle, A. Kuhn, K. Mens, M. van den Brand, and R. Wuyts. Tool building on the shoulders of others. *IEEE Software*, 26(1):22–23, Jan 2009. ISSN 0740-7459. doi: 10.1109/MS.2009.25.

[34] Dimosthenis Kontogiorgos and Konstantinos Manikas. Towards identifying programming expertise with the use of physiological measures. In *Proceedings of the Third International Workshop on Eye Movement in Programming: Model to Data*, EMIP '15, pages 10–11. The University of Eastern Finland, 2016.

[35] Makrina Viola Kosti, Kostas Georgiadis, Dimitrios A. Adamos, Nikos Laskaris, Diomidis Spinellis, and Lefteris Angelis. Towards an affordable brain computer interface for the assessment of programmers mental workload. *International Journal of Human-Computer Studies*, 115:52 – 66, 2018. ISSN 1071-5819. doi: https://doi.org/10.1016/j.ijhcs.2018.03.002. URL http://www.sciencedirect.com/science/article/pii/S1071581918300934.

[36] Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Do you remember this source code? In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 764–775, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180215. URL `http://doi.acm.org/10.1145/3180155.3180215`.

[37] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[38] Karen Lidzba, Eleonore Schwilling, Wolfgang Grodd, Inge Krgeloh-Mann, and Marko Wilke. Language comprehension vs. language production: Age effects on fmri activation. *Brain and Language*, 119(1):6 – 15, 2011. ISSN 0093-934X. doi: https://doi.org/10.1016/j.bandl.2011.02.003. URL `http://www.sciencedirect.com/science/article/pii/S0093934X11000332`.

[39] Walid Maalej, Rebecca Tiarks, Tobias Roehm, and Rainer Koschke. On the comprehension of program comprehension. *ACM Trans. Softw. Eng. Methodol.*, 23(4): 31:1–31:37, September 2014. ISSN 1049-331X. doi: 10.1145/2622669. URL `http://doi.acm.org/10.1145/2622669`.

[40] J. I. Maletic and B. Sharif. An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension (ICPC 2010)(ICPC)*, volume 00, pages 196–205, 06 2010. doi: 10.1109/ICPC.2010.41. URL `doi.ieeecomputersociety.org/10.1109/ICPC.2010.41`.

[41] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

[42] Ian McChesney and Raymond Bond. Do computer programmers with dyslexia see things differently? a computational eye tracking study. In *Proceedings of the Fourth International Workshop on Eye Movement in Programming*, EMIP '17, pages 19–21, 2017.

[43] Jean Melo, Fabricio Batista Narcizo, Dan Witzner Hansen, Claus Brabrand, and Andrzej Wasowski. Variability through the eyes of the programmer. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 34–44, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-0535-6. doi: 10.1109/ICPC.2017.34. URL `https://doi.org/10.1109/ICPC.2017.34`.

[44] Leon Moonen, Arie van Deursen, Andy Zaidman, and Magiel Bruntink. *On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension*, pages 173–202. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-76440-3. doi: 10.1007/978-3-540-76440-3_8. URL `https://doi.org/10.1007/978-3-540-76440-3_8`.

[45] Haris Mumtaz, Fabian Beck, and Daniel Weiskopf. Detecting bad smells in software systems with linked multivariate visualizations. *2018 IEEE Working Conference on Software Visualization*, 2018.

[46] Shinichi Nakagawa and Holger Schielzeth. A general and simple method for obtaining r2 from generalized linear mixed-effects models. *Methods in Ecology and Evolution*, 4:133–142, 02 2013.

[47] Pavel A Orlov. Experts vs novices in programming: "who knows where to look?". In *Proceedings of the Third International Workshop on Eye Movement in Programming: Model to Data*, EMIP '15, pages 16–18. The University of Eastern Finland, 2016.

[48] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrea De Lucia. Do they really smell bad? a study on developers' perception of bad code smells. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 101–110, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6146-7. doi: 10.1109/ICSME.2014.32. URL http://dx.doi.org/10.1109/ICSME.2014.32.

[49] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: A large scale empirical investigation. *Empirical Softw. Engg.*, 23 (3):1188–1221, June 2018. ISSN 1382-3256. doi: 10.1007/s10664-017-9535-z. URL https://doi.org/10.1007/s10664-017-9535-z.

[50] Sebastiano Panichella, Annibale Panichella, Moritz Beller, Andy Zaidman, and Harald C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 547–558, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884847. URL http://doi.acm.org/10.1145/2884781.2884847.

[51] Patrick Peachock and Bonita Sharif. Investigating eye movements in natural language and c++ source code - a replication experiment. In *Proceedings of the Fourth International Workshop on Eye Movement in Programming*, EMIP '17, pages 4–5, 2017.

[52] N. Peitek, J. Siegmund, S. Apel, C. Kstner, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. A look into programmers' heads. *IEEE Transactions on Software Engineering*, pages 1–1, 2018. ISSN 0098-5589. doi: 10.1109/TSE.2018.2863303.

[53] 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*, ESEM

'18, pages 24:1–24:10, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5823-1. doi: 10.1145/3239235.3240495. URL `http://doi.acm.org/10.1145/3239235.3240495`.

[54] Daryl Posnett, Abram Hindle, and Premkumar Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 73–82, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0574-7. doi: 10.1145/1985441.1985454. URL `http://doi.acm.org/10.1145/1985441.1985454`.

[55] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL `https://www.R-project.org/`.

[56] Andrea Schankin, Annika Berger, Daniel V. Holt, Johannes C. Hofmeister, Till Riedel, and Michael Beigl. Descriptive compound identifier names improve source code comprehension. In *Proceedings of the 26th Conference on Program Comprehension*, ICPC '18, pages 31–40, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5714-2. doi: 10.1145/3196321.3196332. URL `http://doi.acm.org/10.1145/3196321.3196332`.

[57] Ivonne Schröter, Jacob Krüger, Janet Siegmund, and Thomas Leich. Comprehending studies on program comprehension. In *Proceedings of the 25th International Conference on Program Comprehension*, ICPC '17, pages 308–311, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-0535-6. doi: 10.1109/ICPC.2017.9. URL `https://doi.org/10.1109/ICPC.2017.9`.

[58] Z. Sharafi, Z. Soh, Y. Guhneuc, and G. 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, June 2012. doi: 10.1109/ICPC.2012.6240505.

[59] 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*, ETRA '12, pages 381–384, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1221-9. doi: 10.1145/2168556.2168642. URL `http://doi.acm.org/10.1145/2168556.2168642`.

[60] Janet Siegmund. Program comprehension: Past, present, and future. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 5, pages 13–20. IEEE, 2016.

[61] P. Sivaprakasam and V. Sangeetha. An accurate model of software code readability. *International Journal of engineering*, vol. 1(6), 2012.

[62] Z. Soh, A. Yamashita, F. Khomh, and Y. G. Guhneuc. Do code smells impact the effort of different maintenance programming activities? In *2016 IEEE 23rd International*

Conference on Software Analysis, Evolution, and Reengineering (SANER), volume 1, pages 393–402, March 2016. doi: 10.1109/SANER.2016.103.

[63] Davide Spadini, Maurcio Aniche, Margaret-Anne Storey, Magiel Bruntink, and Alberto Bacchelli. *When Testing Meets Code Review: Why and How Developers Review Tests*, pages 677–687. Association for Computing Machinery (ACM), United States, 2018. doi: 10.1145/3180155.3180192. Accepted Author Manuscript.

[64] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME). IEEE.*, 2018.

[65] Jozef Tvarozek, Martin Konopka, Pavol Navrat, and Maria Bielikova. Studying various source code comprehension strategies in programming education. In *Proceedings of the Third International Workshop on Eye Movement in Programming: Model to Data*, EMIP '15, pages 25–26. The University of Eastern Finland, 2016.

[66] Bodo Winter. *A very basic tutorial for performing linear mixed effects analyses (Tutorial 2)*, 2013. URL `http://www.bodowinter.com/tutorial/bw_LME_tutorial2.pdf`. University of California, Merced, Cognitive and Information Sciences.

[67] Bodo Winter. *Linear models and linear mixed effects models in R: Tutorial 1*, 2016. URL `http://www.bodowinter.com/tutorial/bw_LME_tutorial1.pdf`. University of California, Merced, Cognitive and Information Sciences.

[68] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, December 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL `http://doi.acm.org/10.1145/267580.267590`.

Appendix A

Letter of Ethics Committee Approval

Date 10-05-2018
Contact person Ir. J.B.J. Groot Kormelink, secretary HREC
Telephone +31 152783260
E-mail j.b.j.grootkormelink@tudelft.nl

Human Research Ethics Committee
TU Delft
(http://hrec.tudelft.nl/)
Visiting address
Jaffalaan 5 (building 31)
2628 BX Delft
Postal address
P.O. Box 5015 2600 GA Delft
The Netherlands

Ethics Approval Application: Test Reading Behaviour
Applicant: Yu, Chak Shun

Dear Chak Shun Yu,

It is a pleasure to inform you that your application mentioned above has been approved.

Good luck with your research!

Sincerely,

Dr. Ir. W.P. Brinkman
Vice chair HREC
Faculty EEMCS

Appendix B

Screenshots of the Online Experiment

B.1 Pre-Experiment Questionnaire

Research on Tests Understanding

Please fill in the following information

Gender *

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

What description fits your current role the most? *

- Software Developer
- Software Tester
- Manager
- Student
- Researcher
- Other

What is your age? *

For how many years have you been a developer? *

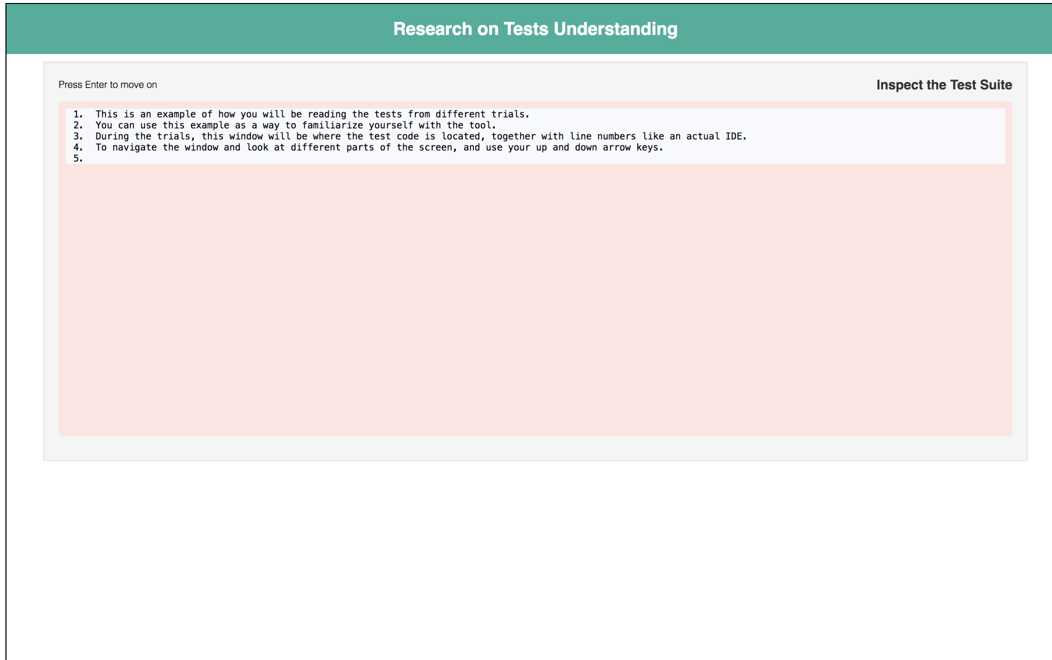
What is your current programming language of choice? *

How meany years of experience do you have with Java? *

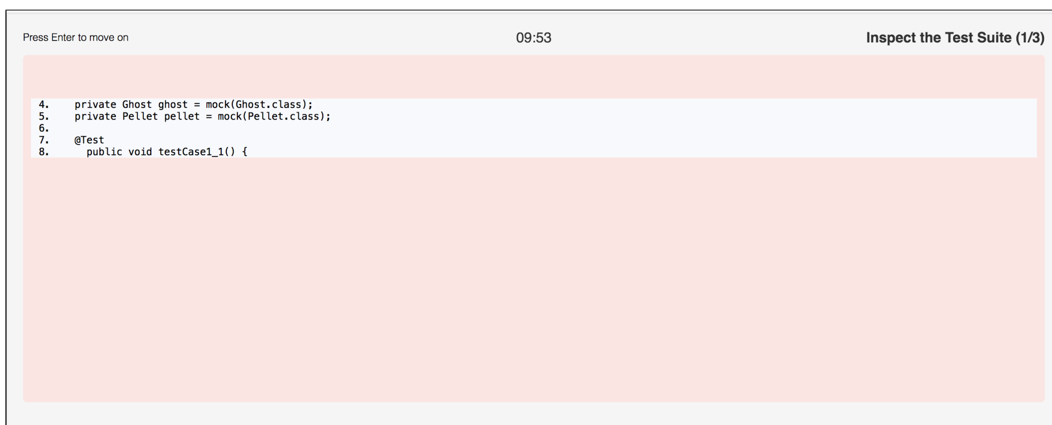
How many years of proper experience do you have with using tests? *

SUBMIT

B.2 Initial Instructions to Get Familiar With the Tool



B.3 Tracking Tool with Reduced Viewport



B.4 Experiment Task Questions

Research on Tests Understanding

Please fill in the following information

What is the purpose of the test suite? *

Describe the additional tests cases that you would write to extend the current test suite. Use 1 line per case. *

SUBMIT