INVESTIGATING EYE MOVEMENTS IN NATURAL LANGUAGE AND C++ SOURCE CODE

by

Patrick R. Peachock

Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Computing and Information Systems

YOUNGSTOWN STATE UNIVERSITY

May, 2015

INVESTIGATING EYE MOVEMENTS IN NATURAL LANGUAGE AND C++ SOURCE CODE

Patrick R. Peachock

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

Signature:

_____

Patrick R. Peachock, Student Date

Approvals:

_____

Bonita Sharif, Thesis Advisor Date

_____

Alina Lazar, Committee Member Date

_____

John Sullins, Committee Member Date

_____

Sal Sanders, Associate Dean of Graduate Studies Date

Abstract

Is there an inherent difference in the way programmers read natural language text compared to source code? Does expertise play a role in the reading behavior of programmers? In order to start answering these questions, we conduct a controlled experiment with novice and non-novice programmers while they read small short snippets of natural language text and C++ source code. The study was conducted with 33 students recruited from an Introduction to Programming class in the Computer Science and Information Systems department at Youngstown State University. The students were each given ten tasks: a set of seven programs, and three natural language texts. The order of the tasks was randomized within each type. They were asked one of three random comprehension questions after each task. We use several linearity metrics that were presented recently in a similar eye tracking study and report on the findings.

The results indicate that novices and non-novices both read source code less linearly than natural language text. We did not find any differences between novices and non-novices between natural language text and source code. We discuss the implications of this work along with directions for future work.

Acknowledgements

I'd like to acknowledge my advisor, Dr. Sharif, for putting up with my procrastination and pushing me to constantly get work done. Without her constant motivation, I would've never become interested in my study. Teresa Busjahn for her assistance in analysis and running eyeCode, and Ahraz Husain for his script writing ability that helped in analysis.

My thesis committee, Dr. Lazar and Dr. Sullins for taking time out of their busy schedules to support me, and Dr. Schueller, without him I would've never been a graduate assistant and I would've never pursued my Master's Degree.

I also want to acknowledge the staff of Uptown Pizza who put up with me hour after hour many evenings, consistently bringing me food to keep me going and keeping me motivated while I sat reading, researching, and analyzing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Programming is a complex set of activities that involves both reading and writing source code. Part of programming involves comprehending what is being read. Programming and comprehension are difficult to tease apart since they are very much intertwined. Programming languages derive many of their characteristics from natural languages. A Latin-based language is any language that uses alphabetical characters such as English or German, instead of symbols. Latin based languages are structured to be read left to right, and top to bottom in a systematic order. Programs on the other hand are inherently more structured and complex than natural language text. While most Latin-based languages are taught to be read left to right, and top to bottom, source code often does not follow the same structure since it is executed in a specific sequence (generally based on the control flow graph).

The aim of this research is to compare how reading of natural languages differs from source code. The best way to do this is to look at programmers' eye movements while reading natural language and source code. Barbara Kitchenham was the first to propose the adaptation of evidence-based paradigm in software engineering in 2004 and called it "Evidence-based Software Engineering (EBSE). The use of evidence-based medicine had been around long before the proposal of EBSE. The medical field saw a huge growth back in the 80's and 90's with this platform shift. Being able to view, analyze, and compare data helped to not only benefit the medical field but increase the amount of publications

(Kitchenham, Dyba, & Jorgensen, 2004). We believe that the eye tracker supports the concept of evidence-based software engineering.

An eye tracker is a device that uses infrared technology to record eye movements. Eye tracking is important because it gives us accurate data as to what a user is looking at when reading on a screen. It is different from filling out a questionnaire after you are done analyzing the task because sometimes a person might misreport on the answer or have the correct thought process but answer incorrectly. With eye tracking, we can see the thought processes as the person is looking at and solving the problem on the screen. Knowing how a user reads is critical in the evolution of tools and techniques to help users read and comprehend both code and natural language text. Eye tracking has been around in the field of software engineering since the early 1990's but has been gaining popularity at a rapid pace.

Eye tracking has many purposes; testing vision, reading comprehension, simulations, and even software engineering. Eye tracking has been used in the software engineering field as early as the late 1980's. One of the first and most well-known uses being Martha Crosby's dissertation (M. Crosby, 1986) in 1986 titled "Natural Versus Computer Languages: A Reading Comparison". Since the release of this dissertation, there has been a growing interest in eye tracking and software engineering. Studies have been done in many fields including program comprehension (Brooks, 1982; Busjahn & Schulte, 2013a; Sharif & Maletic, 2010), computer education (Busjahn et al., 2014), debugging (Bednarik & Tukiainen, 2008), software traceability (Ali, Sharafi, Guéhéneuc,

& Antoniol, 2014; Sharif & Kagdi, 2011; Walters, Falcone, Shibble, & Sharif, 2013; Walters, Shaffer, Sharif, & Kagdi, 2014), software visualization (Jetty, 2013) and usability.

In this thesis, we conduct a study to test how students comprehend C++ code and natural language texts. We analyzed three short English texts and seven short C++ code snippets with several students having a varied set of expertise. The goal is to determine if there are different styles of reading patterns between different categories of users. In particular we are interested in characterizing linear reading. Our participants were recruited from introductory as well as advanced classes in the Computer Science and Information Systems department at Youngstown State University.

1.1 Motivation

There has been a rapidly growing interest in eye tracking studies. During the 1990's there were less than 5 papers published. While studies have given insights as to how users read code and natural language text separately, there hasn't been much progress in the comparing of novices and non-novices reading code. There also has yet to be proof if the expertise level of a programmer determines how they read and comprehend code. The ability to read code is just as important as the ability to write code. There is a growing number of programs that are being worked on that have been built upon previous programming; knowing what a program does prior to using it as a base building block is critical to the success of the built program. With a better understanding of how a novice vs. an expert reads and comprehends code, tools can be developed to teach people to be better programmers. There is no published research that quantifies the reading approach of

natural language text and source code (besides (Busjahn et al., 2015) for Java). This study seeks to bridge that gap.

1.2 Research Questions

The following are the research questions we seek to answer.

- RQ1: Is there an inherent difference in the way novice and non-novice programmers read natural language text compared to source code?
- RQ2: Does expertise play a role in the reading behavior of programmers, in particular, with respect to linear reading?

Better understanding of the different reading patterns of novice and non-novice programmers will help us to advance teaching styles, tools, and also improve coding practices. This study is only a first step towards this long-term goal.

RQ1 basically seeks to determine if the linear reading patterns found in natural language text transfers to source code. RQ2 seeks to separate the expertise and compare if there are differences in two sub groups i.e. novices vs. non-novices.

1.3 Contributions

The main contribution of this thesis is a study on the eye movements in natural language text and C++ source code within novice and non-novice programmers. The study consists of 33 test participants reading source code as well as natural text and answering a comprehension question after each. Based on the eye movements, a set of linearity measures were calculated based on recent prior work. The linearity measures are used to quantify linear reading in natural language text and source code. This study is a partial

replication of the recent study published at the International Conference of Program Comprehension (ICPC 2015) by Busjahn et al (Busjahn et al., 2015).

The differences are briefly highlighted next. Our study used a different experimental design and was part of a C++ semester long class although other students from senior CS classes were also recruited. Busjahn et al.'s study was done after students completed an online module in Java. Unlike Busjahn's study, our study did not study professional programmers, rather only novices and non-novices. We refer to any participant as a non-novice if they were recruited from a class that was at a higher standing than the Introduction to Computer Programming class (that requires no prior background in programming). Since these non-novices are not experts, we prefer not to use the term experts to refer to them.

1.4 Organization

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

CHAPTER 2

BACKGROUND AND RELATED WORK

This chapter talks about an overview of program comprehension, a history of eye tracking studies, and a general discussion on eye tracking history. It has only been within the past few years that eye tracking in software engineering has become a growing area of study. Eye tracking in software engineering has primarily been based around a few foci, model comprehension, code comprehension, and debugging. Because this study focuses on code reading and comprehension of said code, the chapter will cover a brief history of both as well as a small overview of eye trackers and eye tracking studies.

2.1 An Overview of Eye-tracking Technology

An eye tracker is a device that is able to detect where a user is looking on the screen. The screen usually is displaying a stimuli of interest (in our case source code or natural text). When using an eye tracker, there are two types of data that are typically generated: eye fixations and saccades. An eye fixation is a resting of the eye on part of the stimuli for a set amount of time. A saccade is a rapid movement from one eye fixation to the next. This amount of time can vary slightly but is often between 200 to 300 milliseconds. Fixations and saccades are connected by scan paths. Some analysis tools tell the different durations of fixations by marking a dot on the stimuli that grows the longer the eyes rest in one spot, fixation data can also be recorded as a metric of time. In eye tracking studies, the process of information happens during fixations but not during saccades (Rayner, Chace, Slattery, & Ashby, 2006).

Eye trackers come in two different forms, intrusive and non-intrusive. A non-intrusive eye tracker is not worn by the user in any way. The original non-intrusive eye trackers first functioned by reflecting light off of a user's pupils. Some non-intrusive eye trackers are video-based and use cameras to determine eye movements by other objects on the face such as the eye-brows and nose. These types of eye trackers use infrared to see a reflection off of the cornea.

The second form of eye tracker is considered an intrusive eye tracker. These types of eye trackers are worn by the user. These can consist of a headband with cameras, glasses, or goggles. The most noticeable problem with these types of eye trackers is that users are uncomfortable wearing them. The intrusive eye trackers are generally more accurate than the video-based ones. For our purposes, the video-based eye tracker works well.

2.2 Code Maintenance

A large part of handling computer programming is the maintenance and upkeep of code. Code maintenance is defined as the totality of activities required to provide cost-effective support to a software system. Activities are performed during the pre-delivery stage as well as the post-delivery stage (Lientz, Swanson, & Tompkins, 1978). It is estimated that expenditures as high as 50-80% of an Information Systems budget can be spent on software maintenance. (Banker, Davis, & Slaughter, 1998). Program maintenance can often be difficult for a programmer because the code was written by another programmer. A programmer can be more prepared for code maintenance tasks if they are already able to comprehend code on a high level. In 1987 research focused on

program comprehension had estimated more than 50% professional programming time was spent on code maintenance and upkeep of already written programs (Pennington, 1987).

2.3 Program Comprehension

Comprehension is defined by Merriam-Webster as: the act or action of grasping with the intellect. A program or computer program is defined as a sequence of instructions, written to perform a specified task with a computer. Program comprehension is the reading and understanding of the lines of code within a computer program.

Program comprehension is an important part of computer programming. Software maintenance is heavily reliant on a programmer's comprehension of the program that is being maintained. To help improve the ability to comprehend a computer program, we must first understand how a program is written and assembled. Practically, an enormous amount of time is spent developing programs, and even more time is spent debugging them, and so if we can identify factors that expedite these activities, a large amount of time and money can be saved (M. E. Hansen, Goldstone, & Lumsdaine, 2013). Finding those expediting factors can be done by studying the eye movements of programmers.

Brooks first discussed the use of beacons or orthodox segments of code (Brooks, 1983). Ruven Brooks produced a theory of comprehension on computer programs. The research discussed theories and evidence involving program comprehension and different hypotheses behind programs.

Beacons have since been studied to show that they encompass more than just code fragments (Gellenbeck & Cook, 1991). Since the founding discussions of beacons, they are still not known to be a major part in program comprehension or to be explanation of a

professional programmers' understanding and comprehension of code. Crosby et al. in 2002 provided a complete research article discussing and studying beacons and stated beacons may be in the eye of the beholder (M. E. Crosby, Scholtz, & Wiedenbeck, 2002). Fan noted similar results on a study done in 2010 affirming results that suggest beacon identification is in the eyes of the beholder but noticed that the presence of comments does have an effect on code reading (Fan, 2010).

In 1988, Soloway et al. designed two studies (Soloway, Lampert, Letovsky, Littman, & Pinto, 1988) to find important conceptual relations of code. This was done by taking a look at the different focuses on software documentation. The goal was to find different representations that could benefit a programmer when making changes to that code. The first study dealt with exploring the design of software documentation for maintenance. It focused on different types of representations that could possibly help programmers when they make changes to a program. The second study dealt with exploring the types of information that are traded back and forth among the participants in formal design and code inspection.

The first study was to take a look at the different mindsets, logic, and static knowledge that professional programmers use. To do this, they gave different programmers a program that managed a small database as well as the documentation and code of the program. Given this information the programmers were given a task to restore a deleted record within the same session of its deletion. Observation was found that professional level programmers used two different types of strategies related to each the code and the documentation. After observing and creating an enhanced strategy for reading

and performing a task, it was determined that more optimization could be done to help enhance the strategy.

The second study took a look at information that was shared between programmers in a formal review process. After taking the time to monitor a formal review team, it was shown that less than half of that time was focused on the code itself while more than half was focused on the code being understandable for future programmers. The example mentioned discussed that by avoiding some optimization of functionality, time spent on code maintenance down the road could be reduced. It was determined that while delocalized plans were important in both studies, that the nature of these strategies are in the eye of the programmer doing the comprehension.

To help further understand what happens in the upkeep of a program, research was conducted on code naming. Naming in programming is the act of assigning an identifier to a construct. While simple sounding, naming doesn't have a standard and is therefore set by the programmer. A simple guideline to naming is to use a name that is meaningful or somehow related to the construct in use. Two types of naming conventions are common among programmers when it comes to naming with multiple words, underscores and camel case. Since several programming languages are picky about using a space, an underscore can be used to connect a name with multiple words. The second convention, camel case, capitalizes only the first letter of each connected word.

Common naming can also take its form from natural language. Since programming is comprised of a lot of natural text but with specific key words, this is something that occurs naturally. Looking at the Java programming language, there are different methods

within the Java class java.util.Vector that utilize common grammar such as removeElementAt, setElementAs, and setSize. Naming can also contain what is referred to as overloading. This means that one name can be common between several functions. While this can be beneficial due to decreasing the lines of code or having less names to recall, it can also be complicated. Many constructs with the same name can be difficult to read and comprehend for novice programmers, or someone unfamiliar with the programmer's coding styles.

Naming is an important part of all programs. The research explained several different ways that programmers use naming. Practical naming conventions have helped to make most programmers more efficient in naming schemes. Naming combines both natural language and a methodical thought process, and while it seems simple on the surface, there is a further cognitive layer that needs to be looked into (Liblit, Begel, & Sweetser, 2006).

The difficulty in understanding and comprehending of programming are not just isolated incidences. A multi-national study was conducted in 2001 by the "McCracken group" that proved that programming is no easy task to learn. In 2004, a group set out to find out what exactly the root of the problem is when learning how to program. They set out with multiple choice questions to determine a proper score without subjective judgement and also because this would remove any language fluency issues, leaving all of those whom were tested on an even playing field.

While there were differences between institutions such as programming language, student experience, and student motivation (impact on grades vs. volunteers) the reliability

of the questions themselves did fall below a .75 on the Cronbach's coefficient alpha scale meaning the whole set of questions was only slightly unreliable, but not worthy of dismissing, across all institutions. The study goes into detail explaining doodling, notes, and reading versus writing source code. With all of the data presented the conclusion of programming skill results in students lacking a precursor to problem-solving and ties directly to a student's ability to read code rather than write code. The study while built upon The McCraken group's research, left open room for more studies interlacing both groups research designs, reading code, and writing code (Lister et al., 2004).

Michael Hansen with Robert Goldstone and Andrew Lumsdaine published "What Makes Code Hard to Understand?". The research conducted used programmers of all ranges and was done to help determine what factors could impact the comprehension difficulty of the code. Their results led to a better understanding that experience helps when programmers believe that there may be errors but can actually hinder their ability when they haven't been trained for specific cases. Their results also showed that vertical whitespace was a factor in grouping sections of related statements together (M. E. Hansen et al., 2013).

2.4 Eye-tracking Studies in Software Engineering

Studies involving eye tracking and programming have been a slowly growing area. Studies involving eye tracking and software engineering have been few compared to other fields. Program comprehension research has been an even smaller area of eye tracking. The first paper was published in 1990 by Crosby et al. on eye tracking in source code. The next paper was not published until 2002, and then not again until 2006. The field has

quickly grown since then though to include approximately 35 papers in the software engineering field since 1990. The reason most likely being that the growing accessibility to eye trackers within the software engineering field is what is causing the increase in research.

Following Brooks (Brooks, 1983), Martha Crosby released her dissertation "Natural Versus Computer Languages: A Reading Comparison" in 1986 (M. Crosby, 1986). The study followed non-expert programmers and their ability to read and comprehend computer programs. The experiments led to a belief that computer programming language is more advanced than natural language when it comes to non-experts and their ability to comprehend programming code. Crosby did not include expert level programmers though which left the door open for research that would include expert tiered programmers.

In 2002, Martha Crosby, Jean Scholtz, and Susan Widenbeck released a study titled "The Roles Beacons Play in Comprehension for Novice and Expert Programmers" (M. E. Crosby et al., 2002). The study investigated key features of programs that assist in comprehensibility. The study was done using programmers of all ranges from non-programmers to advanced experts. The conclusions lead to a belief that those features or beacons are not necessarily the same for all programmers. What one programmer sees as a beacon may be looked over quickly by another programmer. The research once again left more room for study on eye tracking and reading code.

Following in the line of beacons, in 2010 Quyin Fan released the study "The Effects of Beacons, Comments, and Tasks on Program Comprehension Process in Software

Maintenance" (Fan, 2010). This study was built upon Crosby et al's previous research in beacons. The observations of the study showed that comments within programming code helped to increase comprehension levels if the code is familiar to the programmer. When it came to beacons, Fan noticed the same patterns as Crosby, beacons types are identified by the individual programmer.

Taking a different direction from the reading and maintenance area of code comprehension, a new study was done to compare bug tracking in two different programming languages. Sharif et al. took a look into something that hadn't been done yet, a comparison of C++ versus Python source code in the context of debugging (Turner, Falcone, Sharif, & Lazar, 2014). The choice of the two coding languages were simple, C++ is a very common and continually used programming language, and research led to showing that Python was similar to C++ but easier and more graceful. An eye tracker was used to check if there was a difference in eye gaze behavior between novices and non-novices across both languages, if there was a difference in visual effort, and also if the programming language had an impact on efficiency of solving different tasks, in this case, bug finding and overview tasks.

The study was setup using students studying programming, they were broken up into groups dependent on their answers of their knowledge of each given programming language. The total ended up having more participants in the C++ programming group, this is correlated to the amount of programming courses that the students were able to take in each programming language. The study itself consisted of three different find-bug tasks and two overview tasks.

The results that were analyzed were accuracy, time, fixation counts, and fixation duration for each group. Accuracy was higher for all parties in the C++ group, also in the C++ group, novices took longer overall. For the C++ group, novices had a higher fixation count when compared to Python. The Python group took longer to complete all of their tasks. With all of the data analyzed, there was no statistical difference when it came to comparing C++ and python with accuracy and time, but there was a noticeable difference when it came to fixation rate on bugs. Being that this is the first study of its kind, this left open a follow up study that will be focusing on constructs within the two languages where the users get to see both the C++ and Python codes.

Without having concrete evidence on expedited code reading factors, we rely more strictly on the data that is retrieved from eye-tracking for program comprehension. To be able to understand how comprehension is understood through eye tracking data, we must look at how natural language text is read and understood. Reading can be analytically divided into perception and comprehension (Busjahn, Schulte, & Busjahn, 2011).

To further look into identifier styles Sharif et al. took an in-depth look at eye movements on camel case versus underscores (Binkley et al., 2012; Sharif & Maletic, 2010). The study was a replication of Binkley et al.'s research in 2009 that used response time to get data. Since eye trackers are able to gather more data, the hope was for a better overall look into identifier styles.

The study consisted of eight phrases followed by an identifier that matched what they saw exactly. The other three identifiers were distractors that change the identifier. The goal of doing eye tracking in the study was to focus and see if there was a trigger for

comprehension within a given task. Using AOIs, the study and focus was easily able to be narrowed down and analyzed.

With the use of an eye tracker it was able to tell that identifier style significantly affects time as well as the visual effort that it takes to correctly detect identifiers within a phrase. While Binkley's study did not show a relation between the length of a phrase and its style, eye tracking data reported that camel-cased tasks took longer than underscored tasks. A higher accuracy for camel-case was found in Binkley's study. Camel-casing did noticeably affect the speed of novices over experts though in this study while in Binkley's there was a direct correlation to the programmers' familiarity with their identifier style. The final conclusions being that there may not have been a difference in accuracy, but there is a large improvement when it comes to time and visual effort with underscoring (Sharif & Maletic, 2010).

Since code maintenance is becoming a larger part of programming every day, it's important to not just focus on a programmer's point of view, but also a code reviewer's point of view. One group of programmers wanted to take a look directly at the reading of code. Uwano et al. took a look in 2002 on the eye movements of code readers by doing a code review study (Uwano, Nakamura, Monden, & Matsumoto, 2006). Source code review, not just by the writer of the code, but by peers, has been a proven and longtime used technique to improve code. There are several different types of code review including Ad-Hoc Review, Perspective-Based Reading, Check-list Based Reading, Defect-Based Reading, and Usage-Based Reading. While some of the different review types have been shown to perform better over others, it was never proven via actual eye movements.

The researchers set up a well-organized study that looked at line reading, gazes and glances, and line identification and duration. The researchers developed their own software that worked with their hardware that did a lot of work for them. It was able to get the data from the eye tracker, and fit it into their requirements. Five graduates participated in the study. Each participant had six small programs that they were to review. To ensure the validity of the data, a manual check was done on data obtained and if more than 30 percent of data was bad, the whole trial was discarded.

The study showed that a common practice in code reading was to review the whole block of code in one continuous reading and then look at individual parts of code. The study also showed a correlation between code reading time, code reading patterns and how well a code reviewer was able to find an error within the code. The results also proved that code reviewers have different pattern types when reading, such as recursively going back and looking at related variables. While the study didn't show a relationship between these reading patterns and performance, it is still important to note that code reviewers do read in a pattern which leaves room for a larger and more in depth study of code reviewers eye movements.

Following on the same path of Uwano, Falcone and Sharif (Sharif, Falcone, & Maletic, 2012) conducted a study mimicking that of Uwano et al. but using more participants with varied programming experience. The goal of the research was to once again look at how a programmer reads code when given a specific task, with the end goal being to recognize logical defects.

The study looked at two different questions. First, did the time taken to scan affect accuracy, time, and visual effort of defect detection? Second, does the experience of a programmer come in to play when doing a code review? The experiment was setup using a non-intrusive eye tracker. All 15 volunteers were tasked with finding errors in four different pieces of source code, these codes were used in the Uwano study. A speak aloud method was used and the programmers once given the stimuli were asked to think out loud where they thought the bugs were in the code. Once an answer was given, the programmer moved onto the next stimulus and the test continued.

The study itself varied only slightly from the Uwano study. There were 15 programmers with varied experience instead of only five with the same field of experience. The Uwano study used six programs total, while this study used four. The Uwano study also waited until a programmer found the bug or for five minutes maximum, this study did not. Also, a different eye tracker was used.

Although there were modifications from the Uwano study, the study gives good results. The study validates that scan time does play an important role in the amount of time that it takes to detect errors in code. Also, the varied range of programmers gave an insight to how expert programmers and novice readers take a look at code. Experts tend to have longer fixations while novices tend to take broader sweeps of the code when looking for errors. Noticing that there are different eye movement patterns leaves room to look deeper into scan patterns on code reading and reviewing.

While code reading is an important part of code comprehension, code summarization is important as well. Having a good understanding of what a program does

helps with program maintenance. Some tools can do this for you but they may not always be best. The backbone of any automatic code summarization is in the keywords and identifiers programmed into it. In 2014, a study was completed that looked to use eye movements to better code summarization software (Rodeghero, McMillan, McBurney, Bosch, & D'Mello, 2014).

The study was conducted by having 10 programmers read Java methods and then write their own summaries for the code. The research was looking to see how close programmer's choice of keywords are to a vector space model, term frequency-inverse document frequency model (VSM, tf/idf). Also, do programmers focus on signature, control flow, or invocations of a method any more than other parts of the code?

Using a high frequency eye tracker, the 10 programmers were asked to read Java methods and write summaries for each method. The choices that the programmers made were compared to a VSM tf/idf approach. The eye tracking results showed that half of the top 10 keywords from VSM were the same ones that were most commonly read by the programmers. The eye-tracking results also showed that programmers seemed to pick signatures over invocation keywords in Java methods. While the paper goes into further detail about code summarization, and the tools created, it was from eye-tracking data that they were able to come to their conclusions.

Code reading and reviewing has been proven to be a multi-leveled comprehensive task. Code comprehension is becoming a key part of understanding how a program works. A programmer should be able to understand what problem a program is trying to solve, be

able to read the code and analyze how it's solving the problem, as well as know how to adapt code to the problem it is solving.

Different methods for analysis of eye tracking data are being used, and while some are becoming nearly universal, there is always a possibility for improvement in analysis methods. The results of one such method, ScanMatch, were reported by Cristino et al. in 2010 (Cristino, Mathôt, Theeuwes, & Gilchrist, 2010). Their experiment was designed to quantitatively explain how similar two eye movement sequences are by using algorithms within bioinformatics.

Their experiments consisted of data that shows their approach in relation to the Levenshtein measure which measures the distance between two sequences. Results showed that ScanMatch outperformed Levenshtein when it came to noisy data. The second experiment was to check the performance of ScanMatch. Given stimuli and instructions to follow a number pattern, ScanMatch proved to cluster all of the trials successfully after comparing every other sequence with ScanMatch. In the third experiment, human eye movements were tested on a search task to test if ScanMatch would be able to tell the difference in conditions. One condition had 19 distractors in which different colors were used. After completing trials, each participant was analyzed on an individual basis, ScanMatch proved to have a higher normalized score across tasks.

They concluded that ScanMatch was able to outperform previous string edit methods. ScanMatch is able to tell the difference in conditions and able to withstand and work with noisy data. With the use of ScanMatch, the data received from eye trackers are able to be better analyzed with more precision.

One major enhancement to eye tracking analysis was the introduction of Areas of Interest or AOIs. An AOI is commonly a rectangle that covers a line of code. Each individual line gets an AOI and each AOI has its own identifier. By using an AOI, it is easy to monitor the eye movements within a specific line or segment of code, allowing a narrowing down to a specific keyword or section of code. If there is at least one fixation within the AOI, the AOI is considered hit, otherwise the AOI is missed. The use of an AOI not only helps to better organize data but drastically causes a reduction in scan-path data. A scan-path is the chronological order of AOIs or fixations.

2.5 Eye-tracking Studies in Computer Education

Teresa Busjahn and Carsten Schulte describe the style of reading code as a teaching tool instead of limiting to only writing as a way of learning to program (Busjahn & Schulte, 2013b). They studied the potential to facilitate code reading, the aspects of code reading that are worthy of being read, inquiries as to what should be known about code reading, and what comes to mind when thinking about code reading. After tasking 6 participants, several different aspects of code reading were brought to light. It was learned that code reading has a direct link to comprehension of all aspects of programming including algorithms and semantics of constructs. This leaves room open to further study code reading and techniques.

To confirm that eye tracking was a plausible resource to use in computing education, a case study and workshop were conducted that would help determine the usage of eye trackers and the quality of data that could be obtained. It was determined that eye tracking did allow for a deeper look into how a novice reads and understand code. It was

also decided that the detail of data that is found form eye tracking research can help to advance different learning tools for new programmers. The information gathered isn't strictly related to novices though. From the workshop examples that were gathered, there was a notice in difference in the way that expert programmers read code as well. Knowing this information could give further direction to one's own approach when it comes to reading and writing programming. The paper concludes that eye tracking is a great field to research for teaching computing. Eye trackers are able to give detailed data into how individual programmers work as well as help to study the different methods of computing education (Busjahn et al., 2014). More information about this study is given below.

Over the time of one programming course, it is possible for a novice's eye movements to change. There can be several different patterns of code reading namely, JumpControl, Linear, LineScan, Scan, Flicking, Thrashing, and Signatures. These cover all categories of gaze patterns from natural text reading (left to right, top to bottom) to gazes focusing on methods, or gazes jumping all over the code with no recognizable pattern. They also describe different strategies of reading, DesignAtOnce, Debugging, ProgramFlow, TestHypothesis, Trail&Error, and FlowCycle. These strategies help an analyst determine what's happening in the programmer's mind, such as ProgramFlow which describes the following of the execution order of the code, or Debugging which notices eye gazes look at all elements evenly, suggesting that the reader is trying to find errors. The research takes a look at one programmer over the length of a programming course and notices that eye movements change quite a bit.

The first part of the research was at the beginning of the course. The student read through their code in a linear manner, left to right, top to bottom. The second recording was during the middle of the course and already changes in code reading patterns were visible. While the participant started in a linear fashion, there were more regressions, and these regressions seemed to be focused on variables. This suggests that the reader was checking to follow the history of the variable. At the end of the course, a third recording was completed. Again, the participant started off in a linear fashion but soon jumped straight to the method of the program, after gathering enough information to understand the code, the reader's eye movements then start to move in a random pattern.

Although the test participant was only one individual, the data on the gaze and eye movements were enough to prove that within a short period of time, a reader's eye movements can change quite a bit. Regressions that were seen were part of a confirmation strategy, the test participant was making sure that their program was doing that they thought. This could be an issue due to the fact that programs do not always work in a natural text fashion, so confirming in a linear order isn't always going to work. The data does come to light though and bring into consideration that any novice programmer that changes from story reading to code reading could be a possible step on the way to becoming a better code reader.

With an ever growing interest in teaching computer science and finding ways to understand computer science, program comprehension is becoming more and more a main player in the field of education. In 2010, Schulte et al. reviewed program comprehension for CS educators (Schulte, Clear, Taherkhani, Busjahn, & Paterson, 2010). The review is

explained from a CS educator's perspective. Program comprehension is broken down into sections, external representation, cognitive structure, and the assimilation process. Any part of a program that is not already known by the programmer is considered an external representation while cognitive structure pertains to the already known information that each programmer possesses. The final step, assimilation is the process of going about trying to strategize and solve the programming process at hand.

The research goes into different models of program comprehension that represent different focuses of thought within program comprehension. The different models include the Block Model, Soloway and Ehrlich, Pennington, Letovsky, and more. The research goes into empirical work, characterization, external representations, assimilation, cognitive structure, and mapping to Block Model. The research shows common part for each different model. For example, representation of comprehension and assimilation were similar for several different models.

Schulte concludes with the primary foci of the different models. The first being domain knowledge for comprehension being an underestimated way of teaching. The second being that there are several different possible learning tasks for reading and comprehending programs. Plus the possible use of instruments could be helpful for learning. Finally, experts have a flexible and navigational mental representation of what a program does by reading code and knowledge learned by novice programmers can be likened to a patches being sewn together.

2.6 Other Biometric-based Studies on Program Comprehension

Since the comprehension of natural text has been understood for decades. Getting into the mind of a programmer is the best way to understand how programs are comprehended. Siegmund et al. took program comprehension to a new level when they ran experiments using functional magnetic resonance imaging (fMRI) on programmers while reading different code segments. The goal of the study was to find the different parts of the brain that are activated when a programmer is reading and processing code.

Preparation for the study was important when it came to doing the fMRI study. Avoiding biases was key and given little wiggle room, the source code snippets had to remain small, which was a difficult task because the programs had to have enough of a challenge to allow for the fMRI to read brain activity. This limited the process to only comprehending one aspect of complex comprehension, but still left an opening for studies to follow.

The study showed that different areas of the brain that processed attention, language, memory, categorization, problem solving, verbal and numerical memory, and reading all came in to play when a programmer was reading and solving code. Reading source-code statements showed close to reading sentences in that the part of the brain was active that is used for silent word reading. Problem solving was also triggered due to it being similar to different psychological tasks involving abstracting patterns and constructing how they work. Verbal and numeric memory were of course important, keeping manipulated variables in memory would activate this part of the brain.

The conclusion of the fMRI study brought to light the neuroscience aspect of program comprehension. While having been used for decades, it is now clear that different parts of the brain are activated when reading and comprehending code. These five parts of the brain all fit the bottom-up program comprehension method. The study also confirmed Dijkstra's notice of language processing being an integral part of program comprehension. Without an understanding of language processing, there can be all sorts of problems when it comes to comprehending code. The study, while small, proved accurate with clear results which left open the door for more neuropsychological studies on code comprehension (Siegmund et al., 2014).

Being able to look into how a programmer thinks can assist in the prevention of bugs before a program is ever even compiled. Begel et al. did research into using psycho-physiological sensors to determine a programmer's mental state when programming (Fritz, Begel, Müller, Yigit-Elliott, & Züger, 2014). They look a look at using eye tracking, EDA, and EEG sensors to see if they could predict if a task had different difficulties, which sensors worked best, and can they use psycho- physiological measure to predict task difficulty while the programmer is working.

Their experiment consisted of 15 professional programmers, each programming eight different comprehension tasks. Data was gathered from the participants using different sensors, eye-tracking, EDA, and EEG. Video of both the participant as well as a screen capture, and audio recordings were performed as well. After each task the participants were given a survey to fill out several different questions; mental, physical, and temporal demands, performance, effort, and frustration.

The results came to conclude that regular stress can cause a programmer to create bugs in their software. These stress levels can be predicted using psycho-physiological sensors with precision of 70% over a 62% recall. The test also was able to show that a multitude of sensors doesn't have to be used to show these results. The data found has given an extra insight into helping develop tools that could possibly prevent a programmer from introducing more bugs into their code.

2.7 Discussion

With the continual cost of maintaining code there are further studies that need to be done to help alleviate the weight of code maintenance. With studies showing that maintenance is at least 50% of the time spent programming is on maintenance and reading code, it is important that studies focus on key elements that are linked to reducing the amount of time spent reading code. Since reading code is a comprehensive task and eye tracking is the primary link to reading, we need to take a look at how natural comprehension of text is done. To do this we need to compare how users read and comprehend natural text versus how they read programming code.

With a new insight on the process of research from Kitchenham (evidence-based software engineering) (Kitchenham et al., 2004) and easier access to eye-trackers, we have seen a continual growth in eye-tracking research, whether the research is on code reading, code comprehension, psycho-physiological evaluation, or eye movements in general. As long as there is a possibility to find the connection between expert code readers and novice code readers, we will continue to use eye tracking to help us find the primary connections

between the two as well as study naming conventions, programming comprehension, and bug-finding processes.

The previous studies did not consider to quantify linearity between source code and natural language texts besides our very recent work related to Java code (Busjahn et al., 2015). We attempted to extend what was done in (Busjahn et al., 2015) with C++.

CHAPTER 3

THE STUDY

This chapter presents the details of the study. We describe our experimental design, tasks and stimuli, participant recruiting, measures we used as well as a detailed description of the study instrumentation. The appendix contains a complete listing of all study materials.

3.1 Study Overview

This study seeks to analyze and compare the reading and comprehension of natural language text and C++ source code. The main purposes of this study are to determine if natural text is read differently from source code as well as if novice programmers read differently than non-novices. We consider a non-novice to be a student who was exposed to programming in a prior course. They are typically students enrolled in a higher level course. Novices on the other hand had little or no programming background.

3.2 Independent and Dependent Variables

The independent variable in this study is the type of stimulus being presented: source code (SC) or natural language text (NT). We had seven source code programs and three English language texts to be tested. The dependent variables we collect are related to our research questions. Table 1 describes the independent and dependent variables for the study.

Table 1. Overview of the Independent and Dependent Variables

Independent Variables	• Type of stimulus (SC or NT) • Expertise (Novice or Non-novice)

Dependent Variables	• Linearity Metrics
	  ○ Vertical Next Text
	  ○ Vertical Later Text
	  ○ Horizontal Later Text
	  ○ Regression Rate
	  ○ Line Regression Rate
	• Saccade Length
	• Element Coverage

The dependent variables are explained below. These measures are all based on where the person fixated at and the time they fixated on that location (in other words the fixation counts and the fixation durations). We use eyeCode (M. Hansen, 2015) to map the gaze coordinates to line and word positions in the stimuli for both NT and SC. These have been adapted from (Busjahn et al., 2015).

- Vertical Next Text (%): The percentage of forward saccades that either stay on the same line or move one line down.

- Vertical Later Text (%): The percentage of forward saccades that either stay on the same line or move down any number of lines.

- Horizontal Later Text (%): The percentage of forward saccades within a line.

- Regression Rate (%): The percentage of backward saccades of any length.
- Line Regression Rate (%): The percentage of backward saccades within a line.
- Saccade Length: Average Euclidean distance between every successive pair of fixations.
- Element Coverage (%): Fraction of words the participant looked at.

The first five measures indicate reading behavior and will be mainly used to compare the SC to the NT's which are related to RQ1. They are based on fixations and take into account both horizontal and vertical reading.

The last two measures indicate differences between non-novices and novices. Non-novices for example tend to have longer saccades than novices and also tend to focus on relevant parts of the code i.e. element coverage is smaller.

3.3 Participants

For this study we recruited a total of 33 participants all of whom were students at Youngstown State University. Participants were recruited of their own free will with the option to opt out of the study at any time. The participants ranged from 0 years of programming experience to 5 years or greater programming experience. We wanted to capture a broad range of programmers so as to help with our second research question RQ2. We broke down participants into two separate groups determined by their experience in programming, Students who had taken higher level classes and/or have programmed professionally were classified as non-novices. While those in the Introduction to

Programming class were classified as novices. Note that most of the novices did not have any kind of programming background prior to taking the Introduction to Programming class. In total there were 11 non-novices programmers and 22 novice programmers.

The students participating from the Introduction to Programming class were given this study as an extra credit assignment. They were tested during week 6 and 7 after the semester started. This means that they had six weeks to learn C++ notation as well as the concepts such as data types, arithmetic operators, variables, declarations, assignment and initialization, selection structures, and looping structures.

The participants filled out a pre-questionnaire before they did the study. There were 15 female and 18 male students. The age range was between 18 to 27 years of age. All students were well versed in the English language. A majority of them were native English speakers. Figure 1 and Figure 2 present some demographics on our participants.

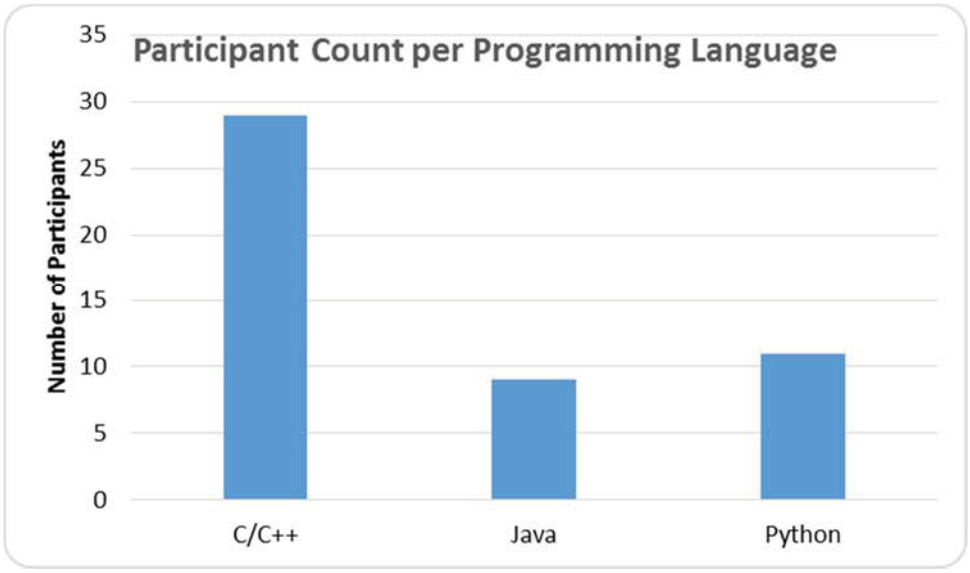

Figure 1. The number of participants self reporting the programming languages they knew or are currently learning.

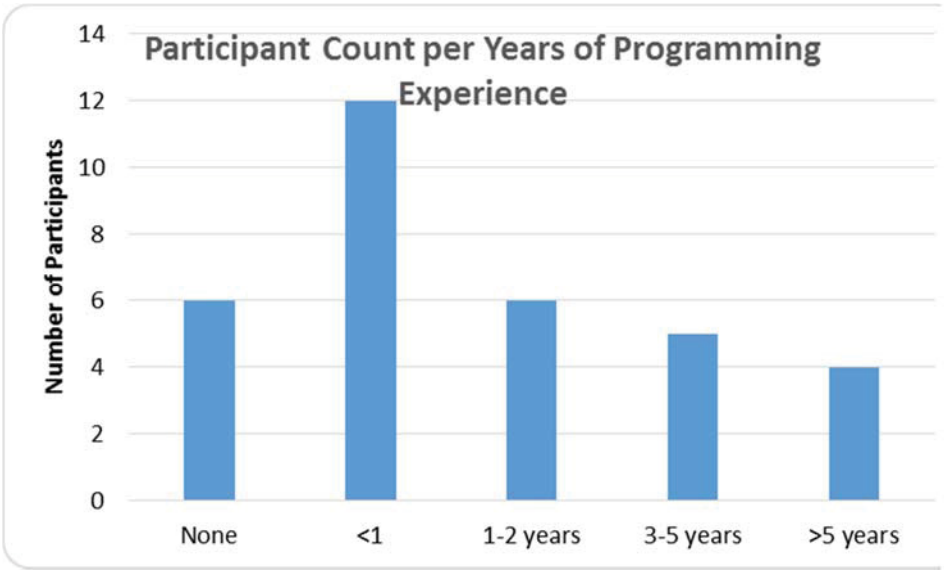

Figure 2. Years of programming experience within participants.

3.4 Tasks

A total of 10 tasks (11 including the demo task) were given to the test participants. Three of these tasks were a natural text paragraph (NT). Each NT discussed a different topic, one involving the history of black powder, one involving the adaptation of dung beetles into a new environment, and one discussing government and economy. The other 7(8) tasks were all short C++ source code programs. All of the programs were of an easy enough difficulty that the Introduction to Computer Programming class would have been able to figure out the output of the programs. The programs ranged from simple output and input statements to nested loops, so the range of difficulty did vary, though not to impossible standards.

The test participants were given as much time as they needed to read the NT and SC tasks. A comprehension question was given after each test. The comprehension question was randomly chosen between the following: A summary of the NT / SC, a multiple choice of the NT / SC, a fact about the NT, or the output of the SC. There was also an option for participants to put that they didn't know and were encouraged not to guess. The order in which the tasks were displayed was determined by a random number generator. Each comprehension question per task was also determined by running a random number generator over the NTs first and the SCs second. This not only gave us an overall understanding how the participants answered as a whole, but helped to prevent the participants from discussing the answers of the questions with each other. Table 2 and Table 3 present the tasks.

Table 2 Overview of source code used in the study. The programs were marked as SC1 through SC7 for C++. The difficulty was set based on the types of constructs used.

SC ID	Program	LOC	Constructs	Difficulty
SC1	CakePrice.txt	7	pseudocode, if/else	Easy
SC2	CalculateAverage.cpp	17	Average, division	Medium
SC3	CountDown.cpp	13	while loop	Easy
SC4	SimpleExpression.cpp	8	Output statement	Easy
SC5	SignChecker.cpp	21	If/else statement	Difficult
SC6	PrintPattern.cpp	14	Nested for loop	Difficult
SC7	DollarsAndCents.cpp	12	division and mod	Medium

Table 3 Overview of the natural language texts used in the study. The paragraphs were marked as NT1 through NT3

NT ID	Description	Number of Words
NT1	Rockets	78
NT2	Beetles	74
NT3	Government	80

Please refer to Appendix A for details on each of these stimuli.

3.5 Data Collection and Apparatus

The eye tracker used in the study was the Tobii X60 (www.tobii.com) eye tracker. The eye tracker is a 60 Hz video-based binocular remote eye tracker that is not intrusive. The eye tracker does not require a head piece, it sits on the desk directly in front of the monitor. At 60 Hz, the eye tracker is able to generate 60 samples of eye data per second.

The Tobii X60 has an accuracy of roughly 0.5 degrees or approximately 15 pixels. The eye tracker is able to compensate for head movement during the study. The study itself was conducted on a 24 inch monitor with a 1920x1080 resolution. While there were two monitors in extended desktop setting, the test participant only used one monitor while the experimenter used the other to setup, initiate, and monitor the study. Besides the eye tracker, audio and video of the participants were also recorded via a Logitech webcam. The eye gaze data recorded included timestamps, gaze positions, fixations, durations, pupil size, validity codes, areas of interest, start time, and end time for each trial. In this study we use fixations, durations, validity, and area of interest. The areas of interest were generated using eyeCode (M. Hansen, 2015). The entire study was conducted in Tobii Studio.

3.6 Study Procedure and Instrumentation

The study was conducted in accordance with IRB policies (IRB Protocol Number: 081-2015) and procedures at the SERESL (Software Engineering Research and Empirical Studies Lab) on the Youngstown State University Campus. On the day of the study, test subjects were informed that the purpose of the study was to understand how programmers read both natural language text and C++ programming source code. Subjects were then asked to answer a pre-questionnaire online. Each participant was issued an ID. The subjects were seated approximately 60-65 cm away from the screen. Before the natural text tasks, the user was presented with a screen giving instructions on the upcoming tasks. Before the C++ source code tasks, the users were presented with a set of instructions as well as a demo source code along with its questions and the form in which they should be answered. The natural text tasks were presented first (in a random order) followed by the

seven source code tasks (in a random order). Subjects announced out loud when they were ready to begin the test and used the mouse to continue on to the next slide. Subjects selected one of three numbers (in a random order) after each task and answered the question that followed using the mouse or keyboard. After each task, subjects were presented with a short questionnaire about the difficulty of the task and their confidence in their answer. After all tasks were answered, test subjects were presented with a post questionnaire that asked about difficulty, time needed, and if any problems occurred during the test.

CHAPTER 4

STUDY RESULTS

We present the results in terms of our two research questions. We first describe how the data was processed followed by comprehension task and timing results. Threats to validity and a discussion finally close the chapter.

4.1 Processing the Data for Analysis

After the study was complete, the data was exported from Tobii Studio to flat files. Before we run the statistical tests on the data, we needed to map the fixations on source code elements. We use eyeCode (M. Hansen, 2015) for this purpose. eyeCode is able to automatically determine lines and words given an image stimulus. These lines and words form the areas of interest (AOI). In our case the image stimulus is the NT and SC tasks. Then, it also maps the fixations on corresponding words so we are able to determine which fixation falls on which word in natural text or source code element.

Figure 3 and Figure 4 show us the pictorial description of how eyeCode segments the lines and words. The fixations are not shown in the figures but they are exported to a file.

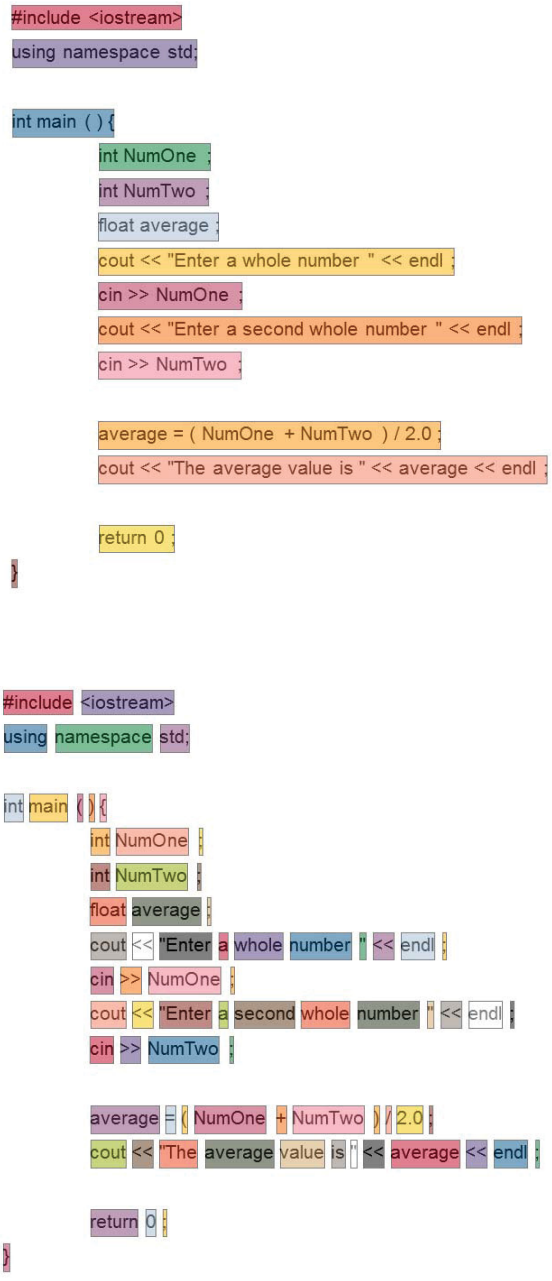

Figure 3. Source code SC2 (CalculateAverage) with line (top) and word (below) AOIs

Figure 4. Natural Language NT1 with line (top) and word (below) AOIs

Before we proceed to answer the research questions, we first report on the comprehension and timing results of the study. We use the Mann-Whitney test to compare novices and non-novices. We use the Wilcoxon paired test to compare within each group

i.e., within novices or within non-novices. These are non-parametric tests as our sample is not large enough for parametric tests.

4.2 Comprehension Scores

We use the abbreviation NT for natural language texts and SC for source code. See Figure 5 and Figure 6 for comprehension scores. We observe that overall non-novices scored higher than novices. In Figure 5, we can also see the gap between novices and non-novices per task. The gaps are larger for difficult programs like SignChecker and PrintPattern. The other programs that fell into the medium and easy difficulty category had less of a gap between novices and non-novices. This indicates that novices had a hard time giving a correct answer for difficult problems.

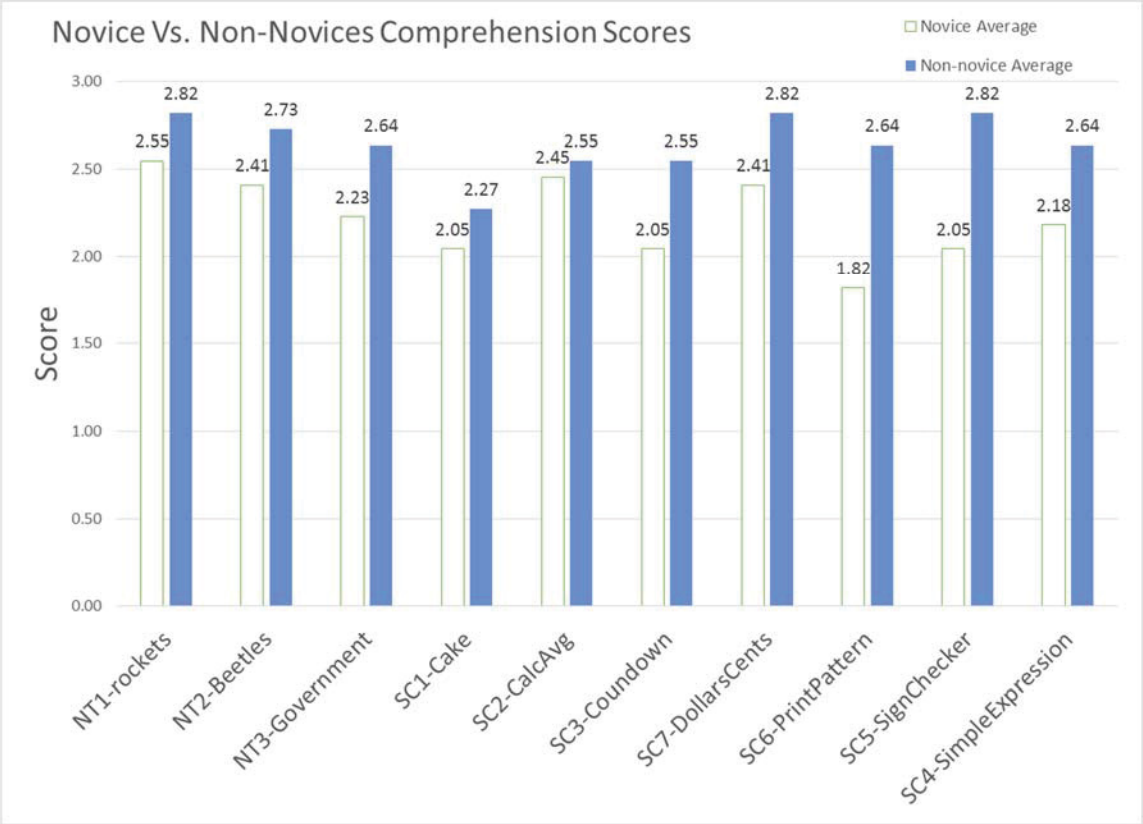

Figure 5. Average Comprehension Score for Novices and Non-Novices within each task

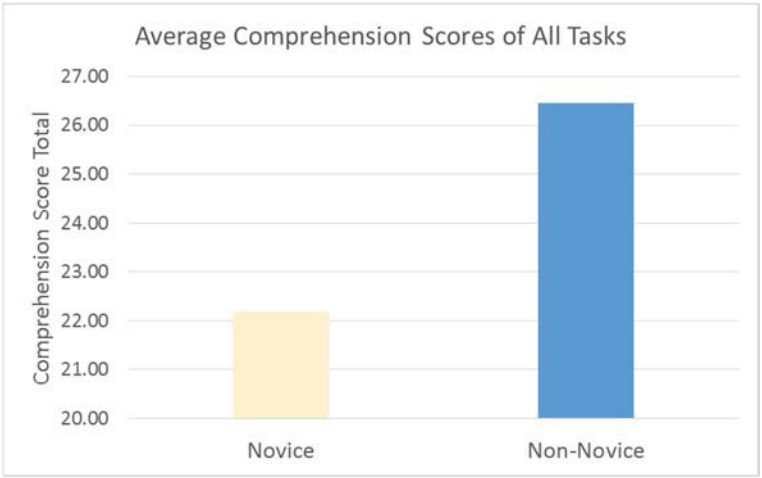

Figure 6. Average Comprehension Score for Novices and Non-Novices across all tasks

The Mann-Whitney test on the total scores between novices and non-novices showed significant differences ($p=0.00019$, $U=213.500$). The non-novices did significantly better i.e. higher scores, than the novices.

4.3 Time Taken

Figure 7 through Figure 10 shows the time taken on each task and over all tasks combined. It also reports on time differences between novices and non-novices. Both novices and non-novices spent the least amount of time in SC4 which was a small and simple program. Similar to comprehension scores, the gaps between novices and non-novices are much more apparent in difficult programs. Non-novices also tend to take the study more seriously compared to novices and so they spend more time on the tasks. A Mann-Whitney test reveals no significant differences in time between novices and non-novices ($p=0.866$, $U=126$).

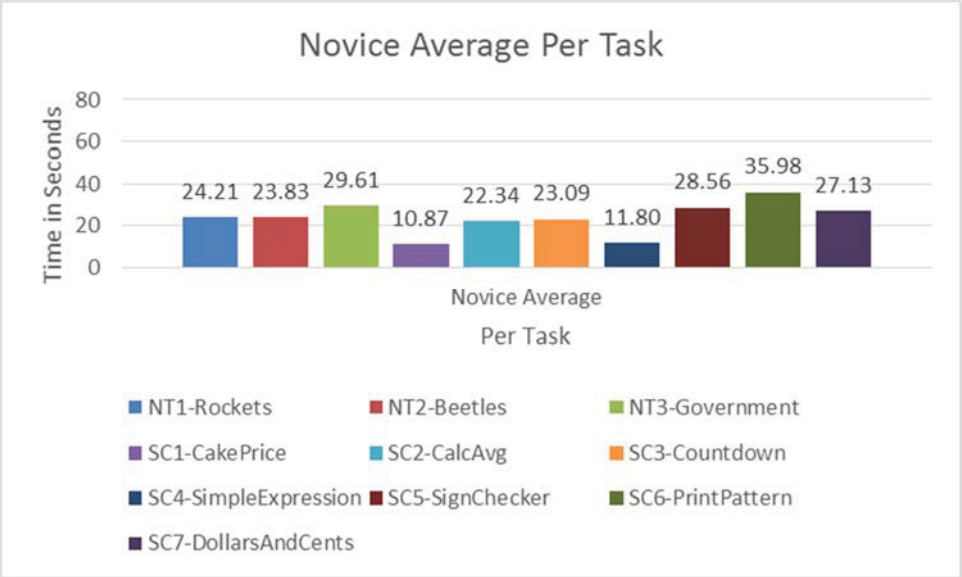

Figure 7. Average total time spent by novices on each task

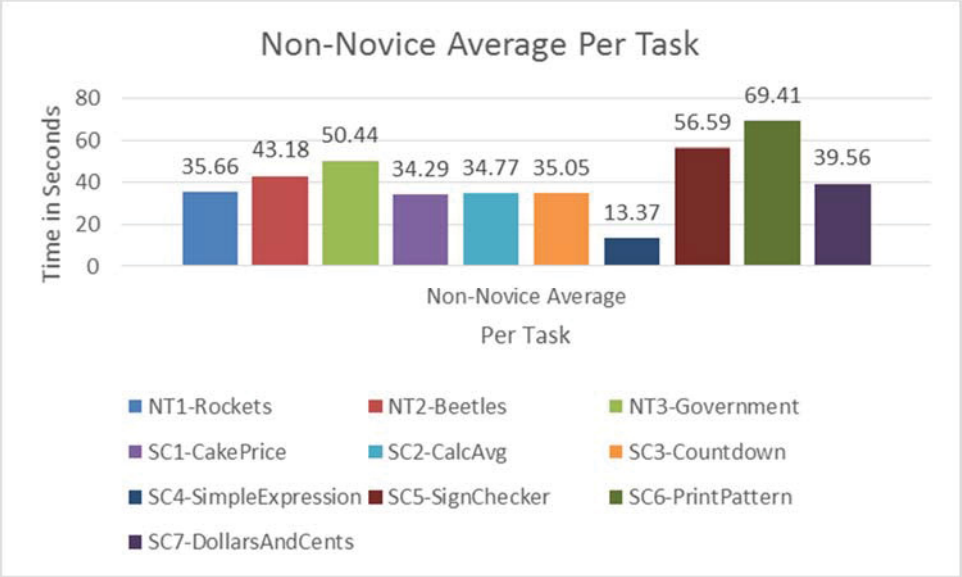

Figure 8. Average total time spent by non-novices on each task

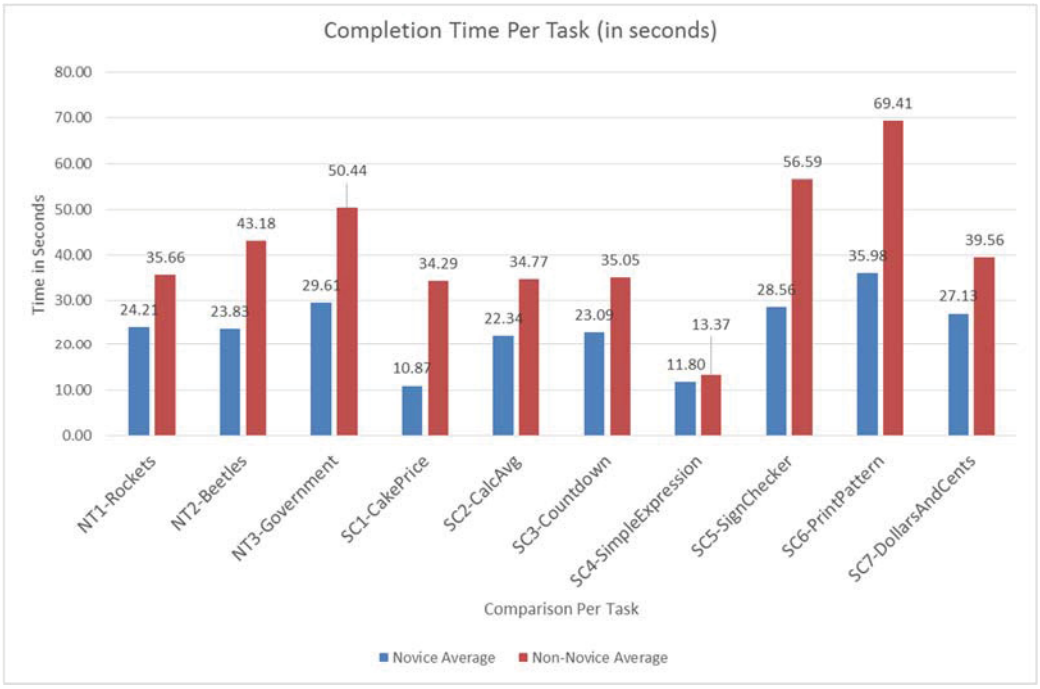

Figure 9. Average total time spent on each task between novices and non-novices.

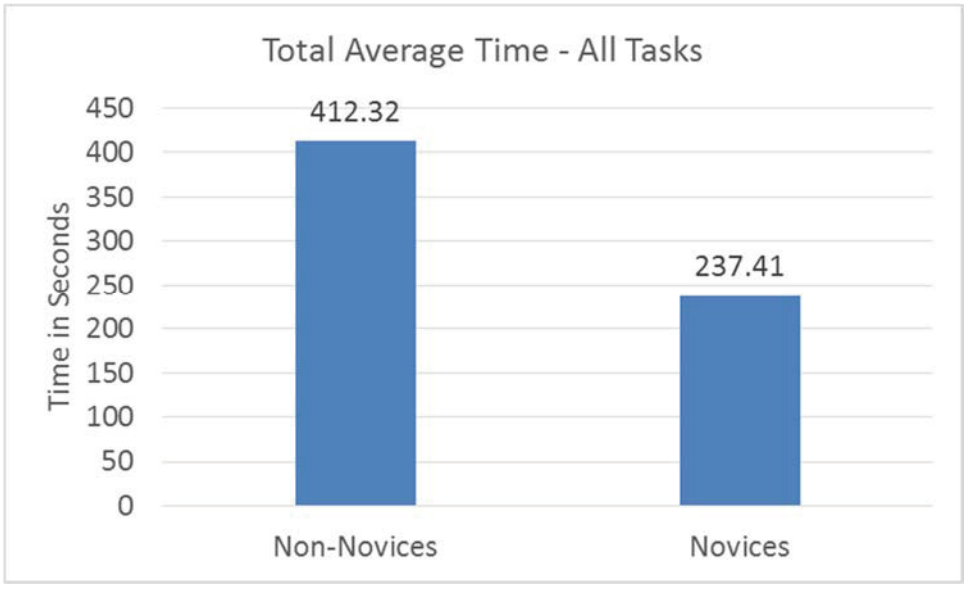

Figure 10. Average total time spent on all tasks.

4.4 RQ1: Is there an inherent difference in the way novice programmers read natural language text compared to source code?

In order to answer RQ1, we report on the linearity metrics we introduced in Section 3.2. Figure 11 shows the five linearity metrics for NT and SC. Notice that the measures Vertical Next Text, Vertical Later Text, and Horizontal Later Text are all higher for NT compared to SC. These are all linearity measures indicating that NT is read more linearly than SC. The regression measures deal with non-linear reading. The line regression rates were higher in NT than SC i.e., participants went back to read a line more often in NT than SC.

We ran the Wilcoxon paired test within novices to see if the above measures were significantly different. Table 4 shows the results of the Wilcoxon test. We see that the measures Vertical Next Text, Vertical Later Text, and Horizontal Later Text and Line Regression Rate are all significantly different between NT and SC. The Regression Rate which involves regressions of any length is not. This means that there is significant more line regressions in SC vs. NT reading. This also indicates that SC is harder for novices to read than NT even though the reading is less linear.

Figure 11. Average linearity measures for all novices.

Table 4 Wilcoxon signed ranked test for NT vs. SC for Novices

Linearity Measure	Sum of Positive Signed Ranks	*p*
Vertical Next Text	253	$p < 0.001$ *
Vertical Later Text	253	$p < 0.001$ *
Horizontal Later Text	253	$p < 0.001$ *
Regression Rate	136	$p = 0.775$
Line Regression Rate	246	$p < 0.001$ *

In **Figure** 12, we see that the length of the saccade is higher for NT than SC. This means that in NT they jumped a few lines more than in the SC. In **Figure** 13 we present the element coverage. This means that 31.7% of NT in novices were looked at and 34% of elements were looked at for SC. This measure simply indicates how many elements on the stimulus were looked at. We also discuss this point in the threats to validity section.

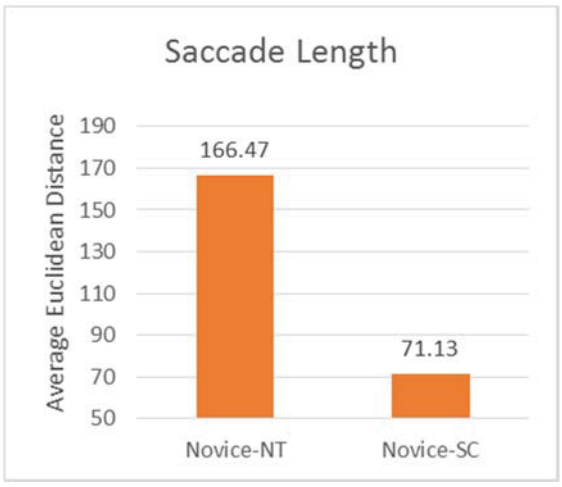

Figure 12. Saccade length for all novices in NT and SC.

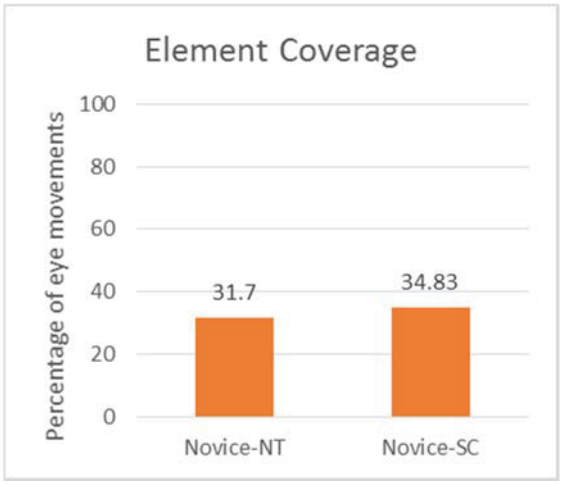

Figure 13. Element coverage for all novices in NT and SC.

Both saccade length ($p < 0.001$) and element coverage ($p = 0.03$) are significantly different between NT and SC. This is another indication that NT and SC are quite different in terms of their cognitive load.

We now discuss the alignment of NT and SC to Story Order and show the results of the Needleman-Wunch (N-W algorithm) in Table 5. Story order is basically reading the stimulus one line at a time from top to bottom (typically the way we read natural language text). The N-W algorithm is used as a string matching algorithm to determine story order. It has also been used by Cristino et al. (Cristino et al., 2010) in earlier work on eye movement research. The algorithm gives a similarity score where a high score indicates that two sequences are close to each other. The difference between naïve and dynamic scores is whether repetitions through the code are counted. So if we care about how many times the person read through the code, we keep repeating the string matching with the story order and the eye gaze movements to get a dynamic score.

Table 5 Needleman-Wunch Results comparing the story order for NT and SC for novices.

Story Order	NT	SC	p
Naïve N-W Score	-8.27	-21.16	<0.001
Dynamic N-W Score	18.71	-4.06	<0.001
Repetitions	3.42	2.6	<0.001

These scores are not close to one another. This means that even novices do not start with an approach that is very top down and left to right (contrary to what (Busjahn et al., 2015) found). The results indicate that both natural language text and source code were both read multiple times. The NT was read 3.42 times compared to SC which was read 2.6 times. The more read-throughs the higher the N-W alignment score. We found a significant difference between NT and SC's story order for novices.

4.5 Non-novice programmers: Natural language text vs. source code

See Figure 14 for a similar set of linearity metrics for non-novices (similar to what we showed for novices in the previous section. In Table 6, we do an equivalent analysis on non-novices by running the Wilcoxon test to detect for differences in how non-novices read NT vs. SC. It follows the same general trend as novices with one exception. The regression rate is also significantly different for experts between NT and SC.

Figure 14. Average linearity measures for all non-novices.

Table 6 Wilcoxon signed ranked test for NT vs. SC for non-novices

Linearity Measure	Sum of Positive Signed Ranks	*p*
Vertical Next Text	253	$p = 0.001$ *
Vertical Later Text	253	$p = 0.001$ *
Horizontal Later Text	253	$p = 0.001$ *
Regression Rate	136	$p = 0.01$ *
Line Regression Rate	246	$p = 0.002$ *

Figure 15 and Figure 16 show the saccade length and element coverage respectively for non-novices. Both the saccade length (p = 0.001) and coverage (p = 0.01) were significantly different for NT and SC in the non-novices group as well.

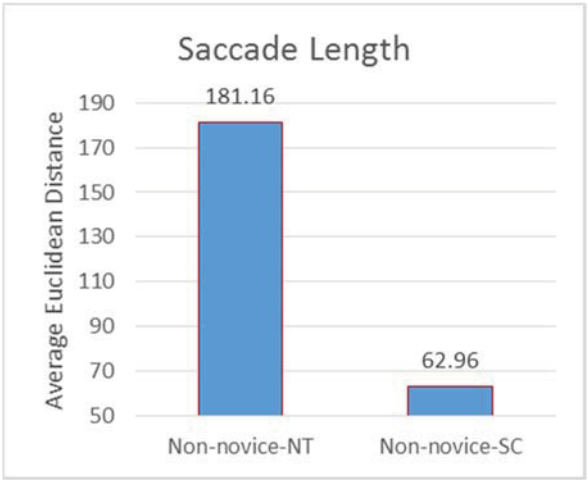

Figure 15. Saccade length for all non-novices in NT and SC.

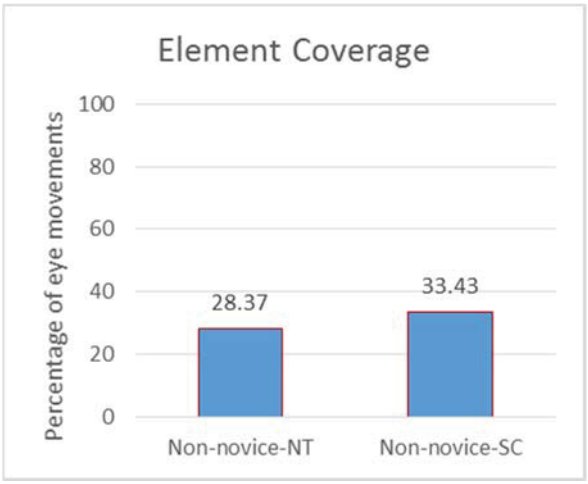

Figure 16. Element coverage for all non-novices in NT and SC.

4.6 RQ2: Does expertise play a role in the reading behavior of programmers, in particular, with respect to linear reading?

We ran the Mann-Whitney test on all novices vs. non-novices. We did not find any significant differences in novices vs. non-novices here as was found in (Busjahn et al., 2015). It could be because our non-novices were not really experts in industry. They were just slightly more advanced than novices. Table 7 shows the results of the Mann Whitney test.

Table 7 Mann Whitney results for novices vs. non-novices over all tasks

Linearity Measure	U	*p*
Vertical Next Text	546	$p = 0.406$
Vertical Later Text	539	$P = 0.461$
Horizontal Later Text	536	$p = 0.487$
Regression Rate	540	$p = 0.453$
Line Regression Rate	487	$p = 0.973$

We plan on conducting the study with real experts to see if a difference can be found for linearity metrics.

4.7 Post Questionnaire Results and Quotes

After the study, we asked the participants if they had enough time to complete the tasks which was unanimously agreed upon as a yes. Participants were also asked about the difficult of the study overall, less than 10% found it very easy, 34% found it easy, 41% considered it average, and 16% found it difficult. The last question asked was about any difficulty faced during the study, 7 participants agreed that it was difficult when it came to remember the stimuli and answer the comprehension question. The following table gives some insight as to what some participants thought of the study overall.

Table 8 Post-Questionnaire opinions

ID	Did you have sufficient time to complete the study?	How would you rate the overall difficulty of the tasks?	Describe any difficulty you faced during the study
P08	Yes	Difficult	I faced some difficulty with the programs because I did not understand parts of it.
P09	Yes	Easy	The hardest part was memorizing what the output was as opposed to figuring out what the program was doing.
P16	Yes	Difficult	understanding what it is asking and remembering it
P22	Yes	Average	Remembering what was on previous screens was difficult.
P23	Yes	Very easy	I didn't remember those readings.
P25	Yes	Easy	i may not have as much programming experience as i would like to have had to answer these questions
P27	Yes	Very easy	Short term memory loss
P28	Yes	Average	I had no real difficulty with the directions or how to complete the study. The only real problem I faced was the output of the incremental program. That was because of my own skill level not the program itself.

The results of the study indicate that novices and non-novices both read source code similarly however, non-novices read code less linearly than natural language. Also, the level of the programmer changes the way that they see a program. We asked participants about what was going on in their minds as they watched their eye movements for two different programs, PrintPatternTest and SignChecker. Table 9 show some of these quotes.

Table 9 Thought Process Quotes

Name	Skill Level	Program	Quote
P1	Novice	PrintPattern	*I was trying to do the loop, just go over it trying to make a tracing table*
		SignChecker	*We haven't gone over strings yet so I figured it was just like everything else. I kind of forgot to look at the output.*
P2	Non-novice	PrintPattern	*I noticed I'm skipping all over the place, I was reading real fast skimming, but kept bouncing up to the top to know what I was thinking*
		SignChecker	*I was sort of just reading what it was going to do and I ended up having to pop back up to the top to see how many i and j were because I wasn't exactly paying attention to variables at the start*
P3	Novice	PrintPattern	*Obviously I have no clue what the computer programming is and I knew I couldn't go back to it so a lot of it was trying to remember the little things.*
		SignChecker	*I had no idea, this was the first one I literally had no idea what was going on, absolutely nothing and I was just looking for patterns. That the number was 0 and what the number means. You can kind of understand it, most of it was just trying to remember if i got asked if the number was greater than 0 or less than 0 it's negative. I got what it meant really fast I just wanted to make sure, yeah.*

4.1 Threats to Validity

We now discuss the threats to validity to this study. To account for different control structures in source code as well as different word lengths in natural language text, we used three NT passages and seven source code passages. The fact that we did not find any differences between novices and non-novices indicates that they are possibly at the same level of reading skills. In order to find differences in linearity, it might be necessary to study expert programmers in industry who program on a daily basis and have been working in industry for more than 10 years.

One major threat to validity is the skewness that occurs in eye tracking data. The linearity metrics are directly dependent on how accurately the fixations are mapped to words or source code elements. We did not correct skewness for this study and hence the results should be taken with caution. We also discarded all trials with less than 60% mapping on source code elements and found the same significant results. This indicates that we might find even more strength and effect in our findings if all the data was corrected. We are currently working on an automated fixation-correction algorithm

Also, on examining the scan paths, we found that most of them were close to the word that they were looking at. We strongly believe that after correction, we should see even stronger significance. We have left this as an immediate future exercise.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

We conducted a study that characterized linearity in eye movements between natural language text and source code in C++. This study replicates an earlier study that looks into Java code. Our results show that both non-novices and novices read source code significantly different than natural language text, while most natural text is read left to right, top to bottom with few regressions, source code is read in a less linear manner with more regressions. While these findings are different, aggregate data shows that there is no difference overall between novices and non-novices in our study sample. We consider this study as a first of two studies done this year (2015) that quantifies linearity. Even though this seems to be quite intuitive, there was no proof that this was indeed the case.

As part of future work, we are currently conducting a second phase of this study with the same group of students. The purpose is to determine if the eye movements differ at the end of the semester indicating if any learning occurred. This phase of the study is being conducted during the last week of the semester. We are also taking a look at fixations and durations on specific areas (beacons) in the code provided and want to determine if the difficulty of a task makes a difference on how both novices and experts read the code. Beacons are places in the code that non-novices tend to focus on as one chunk of data. We believe that studies like this one help towards quantifying eye movements for specific tasks. In our case the task was to quantify linearity. More studies and replications need to be done to add to the body of knowledge and thereby advance the state of the art in experimentation.

APPENDIX A: Study Material Replication Package

A.1. Study Instructions

The purpose of this study is to understand how people comprehend code and natural languages. It is not to measure your skills at programming. There are two parts to this study. First, you will be asked to read three English passages and answer a question about them. The second part of the study will involve reading C++ code and answering a question about the code.
Three possible comprehension questions you could be asked are:

A multiple choice question about the algorithmic idea (or text passage for the passages)

OR

What is the output of the program? (some fact about the passages)

OR

Give a summary of the program (or passages). Here you will say what the program (passage) does and
what the algorithmic idea is.

- You may study the English text and code snippet for as long as you like. But as soon as you are done understanding what it does, we ask that you move on to answer the question following it.
- Please do not guess the answers.
- The source code given to you does not contain any bugs.
- I would like to encourage you to please think aloud. Verbalize your thoughts as this will help us understand your thought process.
- Please answer the questions from the perspective of a person trying to understand and comprehend the code.
- You will fill in your answers in a web form that will pop up automatically.
- For each question, you will be asked to rate the difficulty level you faced and your confidence in the answer you provided. Use the mouse to select the options.
- When you are done with the reading the code on the screen, click the LEFT mouse button only once to advance to the next screen.
- Important note: You will not be able to see the source code while you are answering the question.
- There is no possibility to go back. Please be sure of your choices before you advance to the next screen.

- Please try to maintain your position in the chair while you do the study so that we do not lose the tracking of your eyes. Moving the chair back or moving yourself back in the chair will cause the eye tracker to stop tracking. Small head movements such as looking at the keyboard to type should be fine.
- Find a comfortable position so we can begin.
- We will first begin with calibrating your eyes. Look at the black dot in the center of the red circle and follow it around on the screen.
- Then, we will begin with a sample task followed by the actual study questions

We request that you please not let students who have not taken the study in your class know of the questions asked after you leave. Thank you for maintaining the integrity of the data.

We appreciate your time in taking this study.

A.2. Pre Questionnaire

ID *

What is your native language? *

Your English Level? *

- None
- Low
- Medium
- High

Your overall programming expertise? *

- None
- Low
- Medium
- High

Your expertise in C++? *

- None
- Low
- Medium
- High

Age *

- < 18 years
- 18 – 22 years
- 23 – 27 years
- > 27 years

Gender *

- Male
- Female

Years of programming experience in C++ *

- None
- < 1
- 1-2 years
- 3-5 years
- >5 years

How often do you program in C++? *

- less than 1 hour / month
- less than 1 hour / week
- less than 1 hour / day
- more than 1 hours/day

Years of programming experience in any language *

- None
- < 1
- 1-2 years
- 3-5 years
- >5 years

How often do you program in a language other than C++? *

- less than 1 hour / month
- less than 1 hour / week
- less than 1 hour / day
- more than 1 hours/day

When did you first start programming? (list the year) *

Which language did you first learn programming in? *

What programming language classes are you currently taking? *
State the course number. for e.g., CSIS 2610 and/or CSIS 3700

Which programming languages do you know? *
State the language and level of expertise (low, medium, high)

Do you want to say anything else about your programming background? If yes, you may enter it below.

A.3. The Stimuli - Natural Text (NT) and Source Code (SC)

NT1: Civilisation

The invention of rockets is linked inextricably with the invention of 'black powder'. Most historians of technology credit the Chinese with its discovery. They base their belief on studies of Chinese writings or on the notebooks of early Europeans who settled in or made long visits to China to study its history and civilisation. It is probable that, some time in the tenth century, black powder was first compounded from its basic ingredients of saltpetre, charcoal and sulphur.

NT2: Beetles

Introducing dung beetles into a pasture is a simple process: approximately 1,500 beetles are released, a handful at a time, into fresh cow pats in the cow pasture. The beetles immediately disappear beneath the pats digging and tunnelling and, if they successfully adapt to their new environment, soon become a permanent, self-sustaining part of the local ecology. In time they multiply and within three or four years the benefits to the pasture are obvious.

NT3: Government

The role of governments in environmental management is difficult but inescapable. Sometimes, the state tries to manage the resources it owns, and does so badly. Often, however, governments act in an even more harmful way. They actually subsidise the exploitation and consumption of natural resources. A whole range of policies, from farm-price support to protection for coal-mining, do environmental damage and (often) make no economic sense. Scrapping them offers a two-fold bonus: a cleaner environment and a more efficient economy.

Demo Source Code and Solution Shown

```
#include <iostream>
using namespace std;

int main() {
   int n = 6 ;

   if ( n % 2 == 0 ) {
      cout << n << " is even. " << endl ;
   }
   else {
      cout << n << " is odd. " << endl ;
   }

            return 0 ;
}
```

Sample Task - Solution

```
#include <iostream>
using namespace std;

int main() {
   int n = 6 ;

   if ( n % 2 == 0 ) {
      cout << n << " is even. " << endl ;
   }
   else {
      cout << n << " is odd. " << endl ;
   }

            return 0;
}
```

If you were asked
What is the output of the program?
Your answer would be

6 is even

If you were asked
Write a summary of the program?
Your answer would be

This program determines if the number 6 is odd or even. In this case, since 6 is even, it prints "6 is even". It uses the % (modulo) operator to check for no remainder in the case of even numbers.

If you were asked to choose from the following choices on what the program outputs,
You would choose choice A as your answer.

A. 6 is even
B. 6 is odd
C. 3is even
D. 3 is odd

Remember that you will only be asked one of the questions above.

SC1: Cake price pseudocode

```
cake prices are 1.0 and 2.0

for each item
        if cake price is even
            print "even"
        else
            print "uneven"
```

SC2: Calculate Average

```
#include <iostream>
using namespace std;

int main ( ) {
        int NumOne  ;
        int NumTwo  ;
        float average ;
        cout << "Enter a whole number " << endl ;
        cin >> NumOne  ;
        cout << "Enter a second whole number " << endl ;
        cin >> NumTwo  ;

        average = ( NumOne  + NumTwo  ) / 2.0 ;
        cout << "The average value is " << average << endl ;

        return 0 ;
}
```

SC3 : CountDown

```
#include <iostream>
using namespace std;

int main ( )
{
        int n = 3 ;
        while ( n > 1 ) {
            cout << n ;
            n = n - 1 ;
        }

        return 0 ;
}
```

SC4: Simple Expression

```
#include <iostream>
using namespace std;

int main ( ) {
        cout << " answer = " ;
        cout << ( 40 + 2 ) << endl ;
        return 0;
}
```

SC5: Sign Checker

```
#include <iostream>
#include <cstring>
using namespace std;

int main () {
        int number = 0 ;
        string theSign = " " ;

        if ( number > 0 ) {
                theSign = "positive" ;
        } else if ( number < 0 ) {
                theSign = "negative" ;
        } else {
                theSign = "null" ;
        }

        cout << "This is your number " << number
             << ". The sign is " << theSign  << endl ;

        return 0 ;
}
```

SC6: Dollars and Cents

```
#include <iostream>
using namespace std;

int main ( ) {
        int cents = 140 ;
        int dollars = cents / 100 ;
        int restCents = cents % 100 ;
        cout << "That is " << dollars << " dollar(s) and " ;
        cout << restCents << " cent(s) " ;

        return 0 ;
}
```

SC7: Print Pattern Test

```
#include <iostream>
using namespace std;

int main ( ) {
        for (int i = 1 ; i <= 3 ; i++ ) {
                for ( int j = 0 ; j < i ; j++ ) {
                        cout << " * " ;
                }
                cout << " .. " ;
        }

        return 0 ;
}
```

A.4. The Comprehension Questions for the NT and SC Stimuli

Before each text, one of these three instructions will be given:

Please read and comprehend the following text. When you are done, press the left mouse button.
Then you will be given a MULTIPLE CHOICE question about the content.

Please read and comprehend the following text. When you are done, press the left mouse button.
Then you will be asked about SOME FACT from the texts.

Please read and comprehend the following text. When you are done, press the left mouse button.
Then you will be asked to give a SUMMARY.

You will not know which of the above questions (multiple choice, some fact, or summary) you will be asked about.

You will not be able to view the passage while you answer the questions.

Say Begin when you are ready to start

1 - MC for NT1

Which statement describes the content of the text? *

- Black powder was probably discovered some time in the tenth century by the Chinese. It is essential for the invention of rockets.
- Historians believe that black powder was discovered by the Europeans after visits to China.
- Based on Chinese and European accounts, rockets were essential for the invention of black powder
- I'm not sure

« Back Submit

Never submit passwords through Google Forms.

NT1

* Required

2 - Some Fact for NT1

Where was black powder supposedly discovered? *

« Back Submit

Never submit passwords through Google Forms.

NT2

* Required

1 - MC for NT2

Which statement describes the content of the text? *

- Dung beetles dig themselves into pastures and have to be permanently maintained
- Dung beetles live on cow pats and have to be permanently maintained
- Dung beetles dig themselves into pastures and become a self-sustaining part of the ecosystem
- I'm not sure

« Back Submit

Never submit passwords through Google Forms.

NT2

* Required

2 - Some Fact for NT2

Can pastures benefit from dung beetles? *

« Back Submit

Never submit passwords through Google Forms.

NT3

* Required

1 - MC for NT3

Which statement describes the content of the text? *

- Economy sometimes mismanages resources with provisions that are meant to support the environment
- Governments sometimes mismanage resources with provisions that are meant to support the economy
- Governments sometimes mismanage resources with provisions that are meant to support the environment
- I'm not sure

« Back Submit

Never submit passwords through Google Forms.

NT3

2 - Some Fact for NT3

Discarding certain financial supports would be beneficial for economy and

« Back Submit

Never submit passwords through Google Forms.

CakePrice.txt

* Required

1- MC for CakePrice

What is the output? *

- even
- uneven
- even, uneven
- uneven, even
- I'm not sure

« Back Submit

Never submit passwords through Google Forms.

CakePrice.txt

* Required

3 - Output for CakePrice

What is the program's output? *
(If you don't know the answer, type '?'.)

« Back Submit

Never submit passwords through Google Forms.

CalculateAverage.cpp

* Required

1-MC for CalculateAverage

This program *

- ... gets two numbers from the user and prints their sum
- ... gets two numbers from the user and prints their average
- ... determines the average value of two
- ... prints that there is no average
- I don't know

« Back Submit

Never submit passwords through Google Forms.

CalculateAverage.cpp

* Required

3 - Output for CalculateAverage

What is the program's output if the program is executed with the numbers 2 and 8 as input. *
(If you don't know the answer, type '?'.

« Back Submit

Never submit passwords through Google Forms.

CountDown.cpp

* Required

1 - MC for CountDown

What is the output? *

- ○ 3
- ○ 3,2
- ○ 3, 2, 1
- ○ I don't know

« Back Submit

Never submit passwords through Google Forms.

CountDown.cpp

* Required

3 - Output for Countdown

What is the program's output? *
(if you don't know the answer, type '?'.)

« Back Submit

Never submit passwords through Google Forms.

SimpleExpression.cpp

* Required

1 - MC for SimpleExpression

What is the output? *

- ○ 42
- ○ The answer is 42
- ○ answer = 42
- ○ answer = 40 + 2
- ○ I don't know

« Back Submit

Never submit passwords through Google Forms.

SimpleExpression.cpp

* Required

3 - Output for SimpleExpression

What is the program's output? *

(If you don't know the answer, type '?'.)

« Back Submit

Never submit passwords through Google Forms.

SignChecker.cpp

* Required

1 - MC for SignChecker

What is the output? *

- ○ The sign is positive
- ○ This is your number 0. The sign is negative
- ○ This is your number 0. The sign is null
- ○ The sign is positive, negative, or null
- ○ I do not know

« Back Submit

Never submit passwords through Google Forms.

SignChecker.cpp

* Required

3 - Output for SignChecker

What is the program's output? *

(If you don't know the answer, type '?'.)

« Back Submit

Never submit passwords through Google Forms.

PrintPattern.cpp

* Required

1 - MC for PrintPattern.cpp

What is the output? *

- *,**,***
- *,**,****
- *..**..***
- *..**..****
- I don't know

« Back Submit

Never submit passwords through Google Forms.

PrintPattern.cpp

* Required

3 - Output for Print Pattern

What is the program's output? *

(If you don't know the answer, type '?'.)

« Back Submit

Never submit passwords through Google Forms.

A.5. Post Questionnaire

ID *

- This is a required question

Did you have sufficient time to complete the study? *

- Yes
- No
- This is a required question

How would you rate the overall difficulty of the tasks? *

- Very easy
- Easy
- Average
- Difficult
- Very Difficult

Describe any difficulty you faced during the study

Any other comments about the study are welcome

References

Ali, N., Sharafi, Z., Guéhéneuc, Y.-G., & Antoniol, G. (2014). An empirical study on the importance of source code entities for requirements traceability. *Empirical Software Engineering*, 1–37. http://doi.org/10.1007/s10664-014-9315-y

Banker, R. D., Davis, G. B., & Slaughter, S. A. (1998). Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study. *Management Science*, *44*(4), 433–450. http://doi.org/10.1287/mnsc.44.4.433

Bednarik, R., & Tukiainen, M. (2008). Temporal Eye-Tracking Data : Evolution of Debugging Strategies with Multiple Representations Case Study : Visual Attention During Debug-. In *2008 Symposium on Eye Tracking Research & Applications (Savannah, Georgia, March 26 - 28, 2008)* (Vol. 1, pp. 99–102). New York: ETRA '08. ACM.

Binkley, D., Davis, M., Lawrie, D., Maletic, J. I., Morrell, C., & Sharif, B. (2012). The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, *18*(2), 219–276. http://doi.org/10.1007/s10664-012-9201-4

Brooks, R. (1982). A Theoretical Analysis of the Role of Documentation in the Comprehension of Computer Programs (pp. 125–129). New York, NY, USA: ACM. http://doi.org/10.1145/800049.801768

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, *18*(6), 543–554. http://doi.org/10.1016/S0020-7373(83)80031-5

Busjahn, T., Bednarik, R., Begel, A., Crosby, M., Paterson, J., Schulte, C., … Tamm, S. (2015). Eye Movements in Code Reading: Relaxing the Linear Order. In *International Conference on Program Comprehension* (p. to appear).

Busjahn, T., & Schulte, C. (2013a). The Use of Code Reading in Teaching Programming (pp. 3–11). New York, NY, USA: ACM. http://doi.org/10.1145/2526968.2526969

Busjahn, T., & Schulte, C. (2013b). The Use of Code Reading in Teaching Programming (pp. 3–11). New York, NY, USA: ACM. http://doi.org/10.1145/2526968.2526969

Busjahn, T., Schulte, C., & Busjahn, A. (2011). Analysis of code reading to gain more insight in program comprehension. In *Proceedings of the 11th Koli Calling International Conference on Computing Education Research - Koli Calling '11* (pp. 1–9). New York, New York, USA: ACM Press. http://doi.org/10.1145/2094131.2094133

Busjahn, T., Schulte, C., Sharif, B., Simon, Begel, A., Hansen, M., … Antropova, M. (2014). Eye Tracking in Computing Education (pp. 3–10). New York, NY, USA: ACM. http://doi.org/10.1145/2632320.2632344

Cristino, F., Mathôt, S., Theeuwes, J., & Gilchrist, I. D. (2010). ScanMatch: A novel method for comparing fixation sequences. *Behavior Research Methods*, *42*(3), 692–700. http://doi.org/10.3758/BRM.42.3.692

Crosby, M. (1986). *Natural Versus Computer Languages: A Reading Comparison*. University of Hawaii.

Crosby, M. E., Scholtz, J., & Wiedenbeck, S. (2002). The Roles Beacons Play in Comprehension for Novice and Expert Programmers. In *14th Workshop of the Psychology of Programming Interest Group* (pp. 58–73). Citeseer.

Fan, Q. (2010). *The effects of beacons, comments, and tasks on program comprehension process in software maintenance*. University of Maryland at Baltimore County, Catonsville, {MD}, {USA}.

Fritz, T., Begel, A., Müller, S. C., Yigit-Elliott, S., & Züger, M. (2014). Using Psycho-physiological Measures to Assess Task Difficulty in Software Development. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 402–413). New York, NY, USA: ACM.

Gellenbeck, E. M., & Cook, C. R. (1991). *An Investigation of Procedure and Variable Names As Beacons During Program Comprehension*. Corvallis, OR, USA: Oregon State University.

Hansen, M. (2015). EyeCode. Retrieved April 30, 2015, from

Hansen, M. E., Goldstone, R. L., & Lumsdaine, A. (2013). What Makes Code Hard to Understand? *CoRR*, *abs/1304.5257*. Retrieved from http://arxiv.org/abs/1304.5257

Jetty, G. H. (2013, May). *An Empirical Study Assessing the Impact of SeeIT3D on Comprehension*. Youngstown State University, Youngstown.

Kitchenham, B. A., Dyba, T., & Jorgensen, M. (2004). Evidence-Based Software Engineering. In *Proceedings of the 26th International Conference on Software Engineering* (pp. 273–281). Washington, DC, USA: IEEE Computer Society. Retrieved from http://dl.acm.org/citation.cfm?id=998675.999432

Liblit, B., Begel, A., & Sweetser, E. (2006). Cognitive Perspectives on the Role of Naming in Computer Programs. In *Proceedings of the 18th Annual Psychology of Programming Workshop*. Brighton, England, United Kingdom.

Lientz, B. P., Swanson, E. B., & Tompkins, G. E. (1978). Characteristics of application software maintenance. *Communications of the ACM*, *21*(6), 466–471. http://doi.org/10.1145/359511.359522

Lister, R., Seppälä, O., Simon, B., Thomas, L., Adams, E. S., Fitzgerald, S., … Sanders, K. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, *36*(4), 119. http://doi.org/10.1145/1041624.1041673

Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, (19), 295–341.

Rayner, K., Chace, K. H., Slattery, T. J., & Ashby, J. (2006). Eye Movements as Reflections of Comprehension Processes in Reading. *Scientific Studies of Reading*, *10*(3), 241–255. http://doi.org/10.1207/s1532799xssr1003_3

Rodeghero, P., McMillan, C., McBurney, P. W., Bosch, N., & D'Mello, S. (2014). Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th International Conference on Software Engineering* (pp. 390–401). ACM. http://doi.org/10.1145/2568225.2568247

Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., & Paterson, J. H. (2010). An introduction to program comprehension for computer science educators (p. 65). ACM Press. http://doi.org/10.1145/1971681.1971687

Sharif, B., Falcone, M., & Maletic, J. I. (2012). 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* (pp. 381–384). New York, New York, USA: ACM Press. http://doi.org/10.1145/2168556.2168642

Sharif, B., & Kagdi, H. (2011). On the Use of Eye Tracking in Software Traceability (pp. 67–70). New York, NY, USA: ACM. http://doi.org/10.1145/1987856.1987872

Sharif, B., & Maletic, J. I. (2010). An eye tracking study on camelCase and under_score identifier styles. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension* (pp. 196–205). IEEE. http://doi.org/10.1109/ICPC.2010.41

Siegmund, J., Kästner, C., Apel, S., Parnin, C., Bethmann, A., Leich, T., … Brechmann, A. (2014). Understanding understanding source code with functional magnetic resonance imaging. *Proceedings of the 36th International Conference on Software Engineering*, 378–389. http://doi.org/10.1145/2568225.2568252

Soloway, E., Lampert, R., Letovsky, S., Littman, D., & Pinto, J. (1988). Designing Documentation to Compensate for Delocalized Plans. *Commun. ACM*, *31*(11), 1259–1267. http://doi.org/10.1145/50087.50088

Turner, R., Falcone, M., Sharif, B., & Lazar, A. (2014). An eye-tracking study assessing the comprehension of c++ and Python source code. In *Proceedings of the Symposium on Eye Tracking Research and Applications* (pp. 231–234). Safety Harbor, Florida: ACM. http://doi.org/10.1145/2578153.2578218

Uwano, H., Nakamura, M., Monden, A., & Matsumoto, K. (2006). Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement (pp. 133–140). New York, NY, USA: ACM. http://doi.org/10.1145/1117309.1117357

Walters, B., Falcone, M., Shibble, A., & Sharif, B. (2013). Towards an eye-tracking enabled IDE for software traceability tasks. In *Proceedings of the 2013 International Workshop on Traceability in Emerging Forms of Software Engineering* (pp. 51–54). IEEE. http://doi.org/10.1109/TEFSE.2013.6620154

Walters, B., Shaffer, T., Sharif, B., & Kagdi, H. (2014). Capturing software traceability links from developers' eye gazes. In *Proceedings of the 22nd International Conference on Program Comprehension* (pp. 201–204). ACM. http://doi.org/10.1145/2597008.2597795