Assessing the Effect of Programming Language and Task Type On Eye Movements of Computer Science Students

NILOOFAR MANSOOR, University of Nebraska - Lincoln, United States
COLE S. PETERSON, University of Nebraska - Lincoln, United States
MICHAEL D. DODD, University of Nebraska - Lincoln, United States
BONITA SHARIF, University of Nebraska - Lincoln, United States

Background and Context: Understanding how a student programmer solves different task types in different programming languages is essential to understanding how we can further improve teaching tools to support students to be industry-ready when they graduate. It also provides insight into students' thought processes in different task types and languages. Few (if any) studies investigate whether any differences exist between the reading and navigation behavior while completing different types of tasks in different programming languages.
Objectives: We investigate whether the use of a certain programming language (C++ vs. Python) and type of task (new feature vs. bug fixing) has an impact on performance and eye movement behavior in students exposed to both languages and task types.
Participants: Fourteen students were recruited from a Python course that taught Python as an introductory programming language.
Study Method: An eye tracker was used to track how student programmers navigate and view source code in different programming languages for different types of tasks. The students worked in the Geany IDE (used also in their course) while eye tracking data was collected behind the scenes making their working environment realistic compared to prior studies. Each task type had a Python and C++ version, albeit on different problems to avoid learning effects. Standard eye tracking metrics of fixation count and fixation durations were calculated on various areas of the screen and on source code lines. Normalized versions of these metrics were used to compare across languages and tasks.
Findings: We found that the participants had significantly longer average fixation duration and total fixation duration adjusted for source code length during bug fixing tasks than the feature addition tasks, indicating bug fixing is harder. Furthermore, participants looked at lines adjacent to the line containing the bug more often before looking at the buggy line itself. Participants who added a new feature correctly made their first edit earlier compared to those who failed to add the feature. Tasks in Python and C++ have similar overall fixation duration and counts when adjusted for character count. The participants spent more time fixating on the console output while doing Python tasks. Overall, task type has a bigger effect on the overall fixation duration and count compared to the programming language.
Conclusions: CS educators can better support students in debugging their code if they know what they typically look at while bug fixing. For new feature tasks, training students not to fear edits to learn about the code could also be actively taught and encouraged in the classroom. CS education researchers can benefit by building better IDE plugins and tools based on eye movements that guide novices in recognizing bugs and aid in adding features. These results will lead to updating prior theories on mental models in program comprehension of how developers read and understand source code. They will

Authors' addresses: Niloofar Mansoor, University of Nebraska - Lincoln, Lincoln, Nebraska, United States, niloofar@huskers.unl.edu; Cole S. Peterson, University of Nebraska - Lincoln, Lincoln, Nebraska, United States, cole.scott.peterson@huskers.unl.edu; Michael D. Dodd, University of Nebraska - Lincoln, Lincoln, Nebraska, United States, mdodd2@unl.edu; Bonita Sharif, University of Nebraska - Lincoln, Lincoln, Nebraska, United States, bsharif@unl.edu.

1946-6226/2023/11-ART $15.00
https://doi.org/10.1145/3632530

eventually help in designing better programming languages and better methods of teaching programming based on evidence on how developers use them.

CCS Concepts: • **Human-centered computing** → *Human computer interaction (HCI)*; • **Software and its engineering** → **Language types**; • **Applied computing** → **Education**; • **Social and professional topics** → *Software engineering education.*

Additional Key Words and Phrases: program comprehension, source code, C++, Python, bug fixing, new feature tasks, programming education, learning behavior, eye tracking study

1 INTRODUCTION

Software developers often use several different programming languages when implementing solutions to problems [74, 76]. Some problems are easier solved using features of one language compared to another. Tshukudu and Cutts offer a perspective on the mastering of several programming languages [74]. The choice of programming language has been a long debated topic with no clear empirical evidence of one faring better than another from the usability perspective of the developer [67]. In addition to the possibility of using different programming languages, a software developer typically completes various types of tasks when building software: implementing new features, fixing bugs, testing, or refactoring existing code [32]. These tasks require a developer to comprehend the code first before they make a change and modify it [16]. Similar to professional developers, computer science and software engineering students also learn and use different programming languages while learning computing concepts, and studying how they understand and work with these different languages provides valuable information for teaching and learning purposes. Empirical evidence on the effect of programming languages and various types of tasks can help build stronger theories on program comprehension and help with designing better methods for teaching programming.

Since the 1980s, there has been research published on mental models in program comprehension [12, 35, 50, 55, 69, 79]. Besides surveys and think aloud, another method to study program comprehension, which can be defined as the cognitive processes of understanding code to build a mental representation of the program [59], is to use eye tracking technology [25, 29, 65, 82] to understand what a person is paying attention to while working on a program. The eye movement data can be used to study a person's visual attention and make informed hypotheses about their thought processes and strategies used [14, 49]. Crosby et al. published one of the very first eye tracking studies on how students read a binary search algorithm [17] in 1990. However, eye tracking did not become popular as a method of data collection until after 2006 [8–10, 24, 46, 62]. Crosby found programmers to move between the code and comments instead of just reading the code linearly. A practical guide was recently published on how to properly conduct software engineering and program comprehension studies [61]. A prior eye tracking study by Abid et al. used eye movements to externalize the mental model of developers predicting whether top-down vs. bottom-up models [79] were used [1]. This study was done on the Java programming language on the task of summarizing methods. Another study by Turner et al. compared C++ and Python code shown as an image for bug localization tasks (where the buggy line needed to be spotted but not necessarily fixed) [75]. The first step in fixing a bug is to find it. This process of localizing the line where the bug is on is called bug localization. The next step is the actual fix where the edits are made. However, the study only used small code snippets, and the tasks were relatively easy. It determined the rate at which people looked at a buggy line of code between C++ and Python.

Students are often faced with many challenges with learning new programming languages [27, 58, 59]. In order to help students navigate the initial years of learning programming better, it is imperative to study their behaviors in different settings and use different modalities of data collection. The focus of this article is to understand program comprehension [12, 59, 72, 79] in CS students while they perform two task types: bug fixes and new feature additions in two different programming languages. Our study is rooted in the program comprehension literature in CS education [15, 59] and software engineering [72]. Student behavior is observed via tracking their gaze as well as edits as they solve the tasks. Standard measures such as fixation count and durations over selected chunks of code that act like beacons [12, 80] are analyzed both quantitatively and visually. To this end, an eye tracking study is presented that seeks to measure any differences in reading and navigation problem solving behavior in language (C++ vs. Python) and task type (bug fix vs. new feature). The main motivation behind this study was to determine empirically if there are inherently any differences in eye movement patterns or attention to specific programming constructs between two different programming languages (C++ and Python), and between solving two different software tasks (adding a new feature and bug fixing). As far as we are aware, this is the first study to compare differences in language and task using eye tracking equipment. This is important because all prior studies mainly focus on Java, short code snippets, and/or unrealistic environments that do not generalize to how users(students/experts) actually code in an IDE. [1] Another gap that this study bridges is studying comprehension of programs in different languages using eye trackers in distinct task types. Almost all tasks studied in the past are related to summarization, but as developers, we perform a variety of tasks [34, 42] to solve a problem. This paper provides a study environment setup that others can replicate to conduct more realistic eye tracking studies on various other tasks as well.

The study presented in this paper is fundamentally different from the Turner et al. paper [75]. Not only do we use an additional task type, but the study instrumentation, data collection, and processing are all uniquely different as well. The code snippets used in [75] were short (10-12 lines) and shown as images with no way of interacting with them, which makes the experience unrealistic. In addition, there is no prior eye tracking paper that investigates different task types done by the same user. There is also no eye tracking paper that we know of that investigates new feature addition. This is because of the inherent difficulty in conducting an eye tracking study that involves editing [22, 23].

To summarize, the study presented in this paper bridges many of the above mentioned gaps in empirical studies done in program comprehension by 1) using longer and more complex code snippets for C++ and Python 2) testing two different types of software tasks: new feature and bug localization 3) using a realistic IDE setting (namely the Geany IDE) where the student developer can compile, edit, and run the code while working on a task and 4) providing line-based analysis (derived from the program comprehension model literature [12, 79]) of eye movements both quantitatively and visually via scarfplots. The individual behavior is compared across the languages and tasks. Later as part of future work, we plan to evaluate different program complexities within each task type and do a comparative study.

The contributions of this paper are as follows:

- First eye tracking study comparing student behavior on different task types (new feature and bug fix) in two different programming languages (C++ and Python).
- A study design setup that makes use of a realistic IDE (Geany) where students interact with, scroll, edit, and modify the code freely (instead of images used in prior work). The code, requirements, and console output were all part of the tracking screen. This setup for study design would be more beneficial than just viewing the code.
- Usage of two unique analysis methods: a) tracking eye fixation durations and transitions on logically selected code lines for tracking navigation behavior during the task and b) using scarfplots to visualize these transitions across time.
- Insights into the student behaviors (reading, navigating, editing) for bug fixing and new feature tasks across languages. The evidence suggests that bug fixing is harder than new feature addition tasks (significantly longer average fixation duration and total fixation duration adjusted for source code length). Students looked at lines adjacent to the line containing the bug more often before looking at the buggy line itself.

[1]See Abid et al. [2] for an example where they replicated a short code snippet study with larger realistic programs showing that the results are different.

Participants who added a new feature correctly made their first edit earlier compared to those who failed to add the feature. Python and C++ have similar overall fixation duration and counts when adjusted for character count. Evidence suggests that task type has a bigger effect size on the overall fixation duration and count compared to the programming language. This is a strong indication that the task type is truly important and is the biggest factor in determining performance.
- A complete replication package of the eye tracking dataset collected, stimuli, scripts, and code in order to facilitate future replication with other tasks.

The paper is organized as follows. We formally state our research questions in Section 2. Related work is explored in Section 3. We describe our method in Section 4. Results are presented in Section 5. Section 6 presents the discussion and implications of our work to CS educators in the classroom. Section 7 concludes the paper highlighting the contributions and paving way for future work.

2 RESEARCH QUESTIONS

The four research questions this study seeks to address are as follows:

- RQ1: What are the differences between reading and navigation behavior in two programming languages: C++ and Python?
- RQ2: What are the differences between reading and navigation behavior between two task types: bug fixing and feature addition tasks?
- RQ3: What behaviors do developers engage in during a bug fixing task?
- RQ4: What behaviors do developers engage in during a new feature task?

The first research question (RQ1) seeks to understand how developers navigate between the various parts of the development environment, such as source code, output, and requirements, when C++ and Python are used. Investigating this could tell us how long developers spend debugging in different languages and how they navigate betwen the output console, code, and requirements. The second research question (RQ2) seeks to understand similar behavior differences as RQ1 but in the context of how developers read and navigate two types of tasks. The third research question (RQ3) tries to understand the behaviors developers use while trying to localize and fix a bug in both the C++ and Python languages. The fourth research question (RQ4) looks at developers' editing behaviors when given a set of requirements to implement in existing code in both C++ and Python. Since the nature of the two types of tasks is distinct and different, programmer behavior while working on the two types of tasks may vary. Their behavior may also vary when solving problems in each language.

3 RELATED WORK

In this section, we first present related computer science education work to emphasize the importance of program comprehension in relation to teaching and learning. We also present eye tracking related work from program comprehension and software engineering literature to show the importance of using eye tracking in studying attention and program comprehension. In addition, we provide a list of models and theoretical frameworks that are related to this line of work.

3.1 Computer Science Education - Learning to Program

Learning programming involves reading and comprehension, which in turn means that findings of programming comprehension studies can help computer science educators with shaping their course content and updating their teaching methods to enhance learning. There is a challenge, however, in relating the findings from empirical studies to teaching methods and learning. Izu et al. provide some examples of teaching methods and materials related to program comprehension [27]. In their critical review, Schulte et al. [59] analyze and compare the different programming comprehension models and provide some insights on how these models can be applied to teaching methods and provide students with better and more effective learning tasks. They conclude that the role of domain knowledge for program comprehension should be highlighted more in education, the instruments used in empirical studies might be useful to test learning outcomes, and the differences in expert and novice understanding of programs should also be discussed and investigated in programming education. They state that experts have a flexible and navigational mental representation (i.e., their representations are more than the sum of the elements from reading) of programs, which is in line with findings of Busjahn et al. [14] who found that novices have a more linear reading method when working on programs compared to experts. Additionally, There have been studies on challenges and barriers in learning how to program, and how the programming language affects learning. Stefik et al. [71] conducted a study on how novices learn syntax, and how learning varies across different programming languages. Their results showed the importance of syntax for novice programming, how variations in syntax affect the accuracy rate, and that some syntactic designs in languages were easier to comprehend for novices compared to others.

Due to the importance of learning programming, computer education researchers are interested in how students read and trace code, which is directly related to code comprehension. There are several works that have investigated the relationship between reading, tracing, and writing skills in programming students who have recently started to learn how to code [18, 36, 37, 78]. They all found direct relationships between tracing and reading code and code comprehension, and that students who write better code are better at tracing and explaining it as well. In this paper, we use eye tracking as a method to track student gazes on the code, which can give us insight into how they trace and understand code to solve specific problems.

There are several other approaches for analyzing the patterns of learning in students. Allevato et al. [4] analyzed the sequence of submitted assignments from students and allowed them to change their code so that it could pass the grading criteria and test cases. They realized that students who did better on assignments made more increasing changes and worked incrementally, compared to students who did poorly on assignments who made decreasing changes. Mansoor et al. [39] studied how students comprehend and learn the Alloy language, a specification language based on first-order logic. They created detailed tutorials for all participants, taught the language in some classes, and recruited some students from those classes. Additionally, they recruited non-novices who already knew the language, to compare the work patterns of novices and non-novices. They found a similar pattern of incremental changes when looking at the Alloy analyzer interaction logs, and that novice participants who made more edits and executed the models more often, had higher accuracy scores. Piech et al. [54] used another method to model how students learn, studying how they get to their final solution by capturing snapshots from compilations to analyze the changes between each compile. They present how their modeling can inform about the similarity and differences of learning patterns, and be a predictive model about each student's progress over the course of an assignment. We believe that using eye tracking while studying a participant's problem solving patterns, gives us more insight into how they make changes and why they make those changes on code, and learning these patterns can be very beneficial for educational purposes. If, as an instructor, you are able to see how your student is reading the code in real time, you can instruct them to correct their focus so they can get to the bug quicker.

Previous studies have also investigated the different approaches programmers employ to achieve program comprehension. When tasked with a fill-in-the-blank line in a program, programmers employed several different strategies to understand the program before filling in the line with the correct code [19]. The authors found that most programmers began trying to identify the subgoals of a program such as looping through an array or a maximum algorithm. If a failure to understand a subgoal occurs, additional strategies are employed to resolve the failure. Margulieux and Morrison et al. have also studied subgoal labels in Python and Java [41, 43].

3.2 Eye Tracking in Program Comprehension

In recent years, eye tracking studies have been performed to investigate program comprehension in novice and expert developers. In this section, we present some studies that are closely related. For further information on the state of eye tracking studies done on program comprehension, we direct the reader to prior systematic literature reviews [46, 62].

Given the importance of reading in code comprehension tasks, it logically follows that important insights can be gained from tracking eye movements while participants work on code. Busjahn et al. [15] present eye tracking as a tool to complement the methods used in computer science education research. Eye movements are an objective resource when it comes to studying a programmer's mental model and reading patterns [1, 50]. Eye movements are a proxy for attention, which provide insight into what information people are considering and in what order they do specific tasks. Given these properties and what can be learned from eye tracking data, it adds a lot of value to studies that want to explore comprehension through analyzing reading patterns.

Busjahn et al. conducted a study to look into the differences in how individuals read code versus how they read words, with an additional focus on programmer expertise [14]. Fourteen novices and nine professional software developers had their eye movements tracked while they read Java code. They found that novices looked at code in the same linear fashion that is observed when individuals read text (approximately 80% of the time). Experts, on the other hand, read code in a much less linear fashion. Since this study's focus is on comparing reading patterns between experts and novices, they did not ask the participants to work on various types of tasks and read code in different languages. Our paper tries to compare comprehension patterns in different types of tasks and languages instead.

We summarize a few relevant studies done using eye tracking in the program comprehension field. Peterson et al. examine lines developers familiar with open source systems view during summarization and try to correlate line length with the total duration of time spent on the line [51]. One of their findings is that smaller methods tend to have shorter overall fixation durations but have significantly longer durations per line. In another study the authors also investigated the information seeking behavior via eye movements of developers on Stack Overflow, which showed the importance of code snippets in the questions and answers, and showed that participants did not look at the title of a post, tags, or votes compared to the rest of the text [53]. Saddler et al. examine developer reading behavior on Stack Overflow while they search for information related to fixing bugs and building new features [57]. However, here their focus was more on the reading patterns on Stack Overflow instead of the code itself. Kevic et al. conducted one of the first eye tracking studies on bug fixing in open source software in the Eclipse IDE using an early prototype of the iTrace framework [33]. Their study investigated how developers navigate change tasks, and they found that developers focus on a few methods while working on the tasks, and read small parts of the code within those methods to complete the tasks. Jbara et al. conducted an eye tracking study to measure the time and effort spent reading and understanding regular code [28]. They define regular code as code that includes repetitions of the same basic pattern and is considered to be significantly longer than a non-regular version. They point out that initial code segments are read more than the later ones in regular code and also that code reading was far from being linear, as is also pointed out by Busjahn et al. [14]. Obaidellah et al. [47] look at novice programmer gaze patterns on pseudocode using eye tracking on 51 undergraduate CS students showing that as difficulty increases, the regressions between areas of interest also tend to increase. Hu et al. demonstrate that high-performing students had long fixation durations for analytical problems (more structured) and the problem-solving stage, whereas shorter fixations at the problem exploration stage of interactive problems (less structured) [26]. This study also uses images and short code snippets. Titus et al. showed via an eye tracking study that CS students found reading error messages equally hard compared to source code [7]. Abid et al. conducted an eye tracking study analysis of the use of top-down vs. bottom-up models used during code summarization tasks [1]. They found that, on average, experts and novices read methods using the bottom-up (more focused) mental model than using top-down (bouncing around), and on average, novices spent longer gaze time during the bottom-up process than experts. Aschwanden and Crosby show that beacons are usually present in code when the longest fixation duration is over a thousand milliseconds [6]. This study shows that beacons can be based on the code content and domain of study.

Turner et al. [75] conducted an eye tracking study comparing the accuracy and speed of both bug fixing tasks and overview tasks written in Python and C++. They found that there were no significant differences in accuracy or timing between the tasks based on the language they were written in, but they did find that there was a significant difference in the fixation rate on buggy lines of code between Python and C++. This is the only previous study we are aware of that compares two programming languages for bug localization and program overview tasks. Our paper, in addition to using C++ and Python, also looks at different types of tasks that possibly require different behaviors to perform them correctly, as by nature, a programmer will approach a bug fix very differently from a feature addition. In addition, it is a more realistic study that covers realistic tasks that are more than just a few lines long. Our study is fundamentally different in data collection and instrumentation as well. Moreover, a more comprehensive visualization of fixation transitions between the lines of code is presented via scarfplots.

Recently, Kather et al. [30] studied code composition and planning while programming and they investigated the effects of composition strategies and familiarity with code on program comprehension in an eye tracking study with students. Using eye tracking data and retrospective interviews, students' reading patterns were analyzed, and their mental models were studied. They found that familiarity with the template of the program makes it easier to create schemata. This study also uses images for the stimuli and excessively large areas of interest to analyze the data, which might miss some intricate details of how students navigate between chunks of code. It also does not allow the students to interact realistically with the code snippets.

To the best of our knowledge, we are not aware of any studies that compare eye movements on C++ and Python with respect to different types of software tasks in realistic scenarios, such as using an IDE. We bridge this gap in the literature and add to the empirical evidence by discussing the differences and similarities of comprehension behaviors of student programmers who have worked on these different tasks.

3.3 Models in Program Comprehension

In this section, we review various models and theoretical frameworks in the field of program comprehension. Program comprehension is a sub-field of software engineering that deals with building a mental representation (albeit subjective) of the code while solving a task. Storey et al. provide a consolidated review of all the theories, methods, and tools developed in the software engineering space for program comprehension [72].

Schulte et al. compare and contrast different program comprehension models (from an educational perspective) and discuss how a block model [58] for program comprehension is mapped to various other prior models [59]. Several theories were proposed in the early 1980s. Brooks introduced the concept of top-down comprehension [12], driven mainly by a hypothesis and beacons [80] in the code. Soloway and Ehrlich used a similar model using programming plans or rules of discourse that are used to form a mental representation [70]. Schneiderman et al. present a bottom-up comprehension model where programmers start with individual code items to get to higher level abstractions of what the code does [66]. Pennington et al. discuss a framework where two models, program/control flow and data flow, evolve simultaneously [50]. Letovsky provides a more opportunistic model approach where programmers use and switch between top-down and bottom-up models as needed [35]. Von Mayrhauser et al. build on previous models to introduce an integrated metamodel that consists of a top-down model, a program model, and a situation model [79] where programmers switch between these and build them simultaneously.

With respect to determining program complexity from a cognitive perspective, Duran et al. use Cognitive Load Theory and the Model of Hierarchical Complexity that extends Soloway's plan-based analysis of programs to a finer granularity [21]. Ajami et al. also look at code complexity and how syntax, predicates, and idioms could have an effect on it [3]. They found for loops to be significantly harder than ifs and that counting down is harder than counting up. However, they point out that there could be other factors besides the use of known idioms and syntactic structures that could affect code complexity, and more empirical evidence is required. Katzmarski and Koschke provide a programmer centric view of complexity and show that this does not coincide with complexity metrics rankings [31]. They point out that data-flow metrics align better with programmer viewpoints than control-flow metrics but even that is loosely correlated. Yu et al. provide a survey on software complexity metrics that could be used to determine task variability in program comprehension studies [83].

Izu et al. identify learning activities that address key components of program comprehension and provide a theoretical learning trajectory to guide teachers in selecting further activities in CS courses [27]. Tshukudu and Cutts propose a model describing how student novices are affected while learning different programming languages [74]. They studied students transitioning from Python to Java and vice versa and proposed ways to ease the transition process. Teague et al. use a neo-Piagetian framework that describes cognitive development stages that students go through to study simple programming concepts [73]. They show that students make many mistakes and focus on superficial aspects of the task until they reach the operational stage, at which decentration of focus occurs. That is the point where the cognitive ability to reason about abstractions and adapt skills to tasks that are closely related is formed. Clear et al. have published a report on the BRACElet project that has contributed key findings on how novices learn to program [16]. They also provide guidelines for programming problems for novices. Cunningham et al. provide support for sketching program traces on paper for code reading that correlates with greater success on code reading problems that involve loops, arrays, and conditionals [18]. They used this concept of sketching on new task types such as code writing, code ordering, and code fixing and found that different types of sketching were used for these tasks, not always with increased performance.

Not related directly to program comprehension, but generally to CS education, Nelson and Ko discuss that although theory can be helpful in interpreting designs and results, sometimes it can also inhibit progress [45]. We need to pay special attention to this observation, especially with respect to eye tracking studies, because we have just scratched the surface when it comes to using eye tracking as a means to learn how students and experts work. There aren't many studies that are conducted using eye tracking where one can do a meta analysis to come up with a theory on how students work. We may develop a working theory on how eye movements occur in different tasks but we still need more empirical evidence to validate such behaviors.

4 METHOD

The objective of this study is to assess how a student programmer approaches understanding two different programming languages: C++ and Python, in two different task types: bug fixing and new feature addition. Each student saw both C++ and Python code for the tasks. Eye movements were recorded during the entire study to objectively determine what students were looking at as they performed the tasks. The tasks themselves are not directly comparable as we wanted to avoid any learning effects however, they do use similar, semantic constructs as shown in Section 4.3. In this section, we present the participant demographics, sampling procedures, tasks, stimuli in the experimental design, eye tracking hardware used, the terminology used, and the tools we used to collect measures to answer our research questions. We followed the practical guide on conducting eye tracking experiments while designing the experiment [61].

Table 1. Participant Demographic Information

Demographic Categories	**Choices**	*n*[†]
Gender	Male	11
	Female	3
Age	18-24	8
	25-34	2
	34-44	1
	>45	3
Student Level	Not a student	4
	Undergraduate	6
	Graduate	4
Industry Employment	No	9
	Yes	5
IDE [‡]	Netbeans	6
	Eclipse	5
	Visual Studio	4
Design Skills	Average	8
	Above Average/Good	5
	Excellent	1
Programming Skills	Average	6
	Above Average/Good	6
	Excellent	2

Demographic Categories	**Choices**	*n*[†]
C++ Skills	Beginner	5
	Intermediate	5
	Average	1
	Advanced	3
Years of Programming in C++	Between 1 and 2	5
	Between 3 and 5	3
	Between 6 and 10	3
	More than 10	2
	None	1
Python Skills	I don't know Python	5
	Beginner	1
	Intermediate	5
	Advanced	3
Years of programming in Python	None	5
	Between 1 and 2	5
	Between 3 and 5	3
	Between 6 and 10	1
Programming Languages [‡]	Java	9
	C++	9
	C	8
	Python	5

[†] Number of participants who chose the corresponding option in the row.
[‡] Picking multiple answers was allowed.

4.1 Participant Characteristics

The participants were mainly students from a large Midwestern university in the United States. Fourteen volunteers participated in the study.

Table 1 shows a summary of the demographic information collected from the participants. Eleven participants were male and three of them were female. Eight participants were between 18 and 24 years old, two participants were between 25-34 years old, one was between 34 and 44 years, and three were over 45 years old. There were six undergraduate students, four graduate students, and four non-students (who had just graduated) among the participants. Nine participants did not have any industry employment and experience, and five participants indicated that they had industry experience. Netbeans was the most used IDE among the participants, with six participants choosing it as one of the IDEs they use for programming. Eclipse and Visual Studio were the next popular choices, appearing in the participants' answers five and four times, respectively.

We asked the participants to self-report their programming skills and experience levels. Siegmund et al. [68] state that self estimation is a reliable measurement of programming skills and experience. Eight participants rated their design skills as average, five rated them as above average/good, and one rated them as excellent. Six participants rated their programming skills as average, six rated them as above average/good, and two rated their

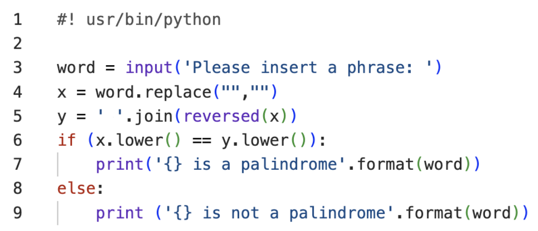
Table 2. List of Tasks/Stimuli Used in the Study

Stimulus	**Language**	**Type**	**LOC**	**Character Count**	**Description**	**Constructs Present**
Stimulus 1	Python	Bug Fixing	9	238	Creates the Palindrome of a string	Input/Output Built-in String Functions (join, reverse, ...) Conditionals
Stimulus 2	C++	Bug Fixing	26	431	Creates the reverse of a word or a phrase	Input/Output Pointers While loops Arrays Conditionals
Stimulus 3	Python	Feature Addition	46	1052	Prints the position of a number in an array, or that it was not found	Input/Output While Loops Conditionals Class/Functions
Stimulus 4	C++	Feature Addition	31	546	A class defining a Stack and all its related functions	Input/Output Arrays For Loops Conditionals Class/Functions

skills as excellent. As for programming language specific questions, five participants ranked their C++ skills as beginner level, five ranked their skills as intermediate, one ranked their skills as average, and finally, three participants ranked their skills as advanced. Five participants had between 1 and 2 years of experience in C++ programming, three participants had between 3 and 5 years of experience, three participants had between 6 and 10 years of experience, two participants had more than 10 years of experience, and finally, one participant had no experience in C++ programming. Subsequent questions were about the participants' skills in Python. Five participants said that they did not know Python. One ranked their skills as beginner level, five ranked their skills as intermediate, and three ranked their skills as advanced. Five participants had no experience in Python programming, five participants had between 1 and 2 years of experience, three participants had between 3 and 5 years of experience, and one participant had between 6 and 10 years of experience. Finally, the participants were asked to list the languages they could program in. Java was mentioned in the answers nine times, with C++, C and Python coming as the next most mentioned answers, respectively.

4.2 Sampling Procedures

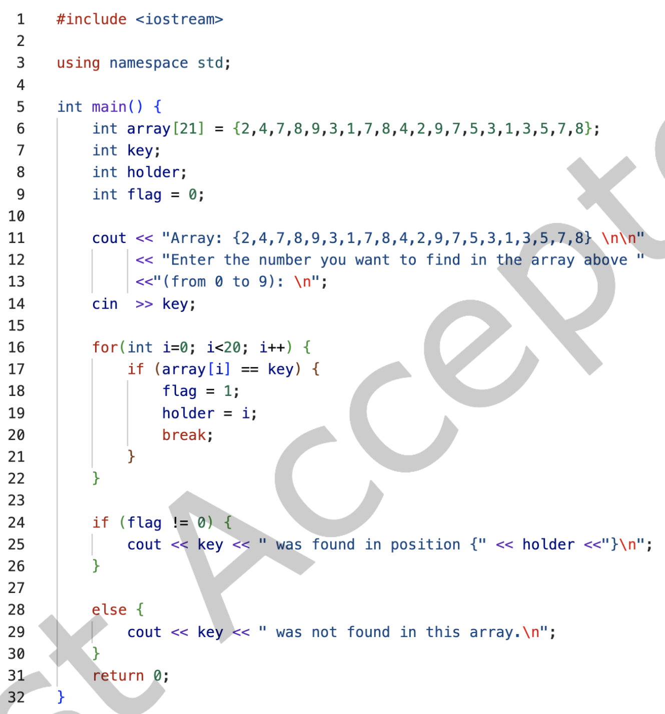
The students were recruited from a class that taught Python as an introductory programming language. All of the students were also knowledgeable in C++. There were no incentives for their participation. They were all



```python
#! usr/bin/python

word = input('Please insert a phrase: ')
x = word.replace("","")
y = ' '.join(reversed(x))
if (x.lower() == y.lower()):
    print('{} is a palindrome'.format(word))
else:
    print ('{} is not a palindrome'.format(word))
```

(a) Stimulus 1 (Python Bug Fixing)



```cpp
#include <iostream>

using namespace std;

int main() {
    int array[21] = {2,4,7,8,9,3,1,7,8,4,2,9,7,5,3,1,3,5,7,8};
    int key;
    int holder;
    int flag = 0;

    cout << "Array: {2,4,7,8,9,3,1,7,8,4,2,9,7,5,3,1,3,5,7,8} \n\n"
         << "Enter the number you want to find in the array above "
         <<"(from 0 to 9): \n";
    cin  >> key;

    for(int i=0; i<20; i++) {
        if (array[i] == key) {
            flag = 1;
            holder = i;
            break;
        }
    }

    if (flag != 0) {
        cout << key << " was found in position {" << holder <<"}\n";
    }

    else {
        cout << key << " was not found in this array.\n";
    }
    return 0;
}
```

(b) Stimulus 4 (C++ Feature Addition)

Fig. 1. The Python Bug Fixing task and the C++ Feature Addition task used in the study.

in the CS program. None of them were students of the authors. The authors did not know any of the students personally. All participation was voluntary and done via an announcement. The study took place in a quiet eye tracking lab where only the moderator and the participant were present without any outside distractions. The moderator was there to ensure the participant was seated at the correct distance from the eye tracker and to perform the calibration. They did not interact with the participants during the experiment. The University's Institutional Review Board approved the study prior to its implementation.

4.3 Conditions and Design

The four different combinations of programming language and task types used in this experiment are listed in Table 2. A high level description of the programs and the programming constructs that are present in the program are also listed. Two tasks were presented in Python, and two were presented in C++. From each language category, one task was a bug fixing task, and the other was a feature addition task.

For the bug fixing tasks, we asked the participants to find the bug located in the program, write the line number they thought contained the bug, and attempt to fix the bug. They were also given the expected input and output of the program. For the feature addition tasks, we gave the participants a description of the program's current capabilities and a description of an additional feature that they had to implement. Figure 1 shows Stimulus 1 and Stimulus 4. A complete replication package with all the tasks, programs, and eye movement data is available at [40].

Participants were given all four tasks in randomly generated order. They had access to the source code in Geany[2], the console output of the program, and the requirements of the task. Requirements included the expected input and output for the bug fixing tasks and the additional feature that needed to be implemented for the feature addition tasks. Figure 2 shows an image of the screen setup and these three areas. There was a trial task given to familiarize participants with the IDE setup so they could ask questions. We did not collect eye tracking data for the trial task.

We now provide some rationale for why we chose these two types of tasks (new feature and bug fix). As developers, we perform a variety of tasks on a daily basis, such as bug fixing, feature addition, refactoring, code review, testing, reading requirements, reading to comprehend code, summarizing code, and many more. Almost all of the eye tracking studies in program comprehension are on tasks that involve participants summarizing Java code, and very few are on fixing bugs. There are none on adding new features. Moreover, all studies (except for a few) are on short code snippets and all on Java. Besides Turner et al. [75] there are no published studies looking into eye movements on Python that we are aware of. It has been shown by Abid et al. that results derived from short code snippets are not always consistent with when you use realistic programs within an IDE to test developers [2]. To bridge this gap, we chose two of the activities we believe developers spend a lot of time on i.e., fixing bugs, and adding new features. In the future, we will add more task categories as provided by Murphy et al. [44].

The goal of this paper was to see how participants fare on different types of tasks. The tasks themselves are different categories and should not be considered comparable. The goal was to see how the same individual's eye movements differed between the different types of tasks. The two tasks chosen are representative of what software developers typically do i.e., fix bugs and implement new features, as also evidenced by many issue tracker systems in open source projects.

Our underlying assumption (based on theoretical frameworks such as [18] that looked at different tasks albeit without eye tracking) is that bug fixing and new feature tasks would require different levels of comprehension and problem solving skills. For bug fixes, developers generally start with the bug report and/or expected input/output and try to figure out which line the bug is on by tracing backward to find the line via stack traces or some other tracing method. With new feature tasks, developers do not do as much tracing since they are implementing forward based on the requirements they read and what the expected feature should do. Because of these reasons, we believe that solving these tasks would generate different user behaviors.

For the bug fixing tasks, the requirements of the task were somewhat comparable. One task reverses a word or phrase (C++) and the other creates a palindrome (Python). The new feature tasks, however, were slightly different, albeit they used similar constructs listed in Table 2. Since the study design is within subjects, giving very similar programs across languages would cause learning effects that we wanted to avoid. Since we recruited

[2]Geany IDE: https://www.geany.org/

our participants from a Python class, we also wanted to make sure we chose stimuli with concepts already taught in the class. We asked the instructor for their syllabus and weekly schedule to ensure we used programs and concepts that was known to the students. We did not use verbatim any code from the class itself. We were not instructors for the course.

Note that the goal of this paper was not to do a side by side comparison of the same task in C++ vs. Python. Instead, it was to see how each participant understood C++ vs. Python in two task categories. In order to account for the difference in lines of code in the tasks, we make sure we normalize our fixations per character because otherwise, longer programs will always have more fixations as there is more to read (see Section 4.6 for more details on normalization). For future work, we plan to evaluate different program complexities [3, 21] within each task category. However, task complexity was not the scope of this paper.

After each task, we asked the participants to rate the difficulty of the tasks, with the options: "Easy", "Average", and "Difficult". For the statistical analysis, we assigned the numbers 1, 2, and 3 to these choices, respectively. We also asked the participants to rate their confidence level about each task, with the options "Not Confident", "Somewhat Not Confident", "Somewhat Confident", and "Very Confident". For the statistical analysis, we mapped these choices to the numbers 1, 2, 3, 4, respectively.

4.4 Terminology

We provide definitions for basic terminology we use throughout the paper to help provide the reader with context for our study.

Program comprehension is a sub-field of software engineering/computer education that deals with a user building a mental representation (albeit subjective) of the code while solving a task.

Task Type refers to the various possible types of tasks a developer (in this case, a student) may engage in. Possibilities could be bug fixing, new feature addition, refactoring, code review, testing, etc. In this paper, we only evaluate two task types (bug fix and new feature addition).

Task refers to the actual set of artifacts that falls into the specific task category. For a bug fix task, this would be the code in the IDE, the program requirements, and expected input and output of the program. For the new feature task it would be the starter code in the IDE, current description of the program, and a description of the additional feature to be implemented. In both cases, the console output was also available to the participant. The participant is expected to engage with these artifacts to produce a result. In the case of the bug fixing task, the result would be the line that had the bug and a fix for the bug. For the new feature addition tasks, the result would be the newly written code that implements the new feature.

Bug localization (in our study) refers to the time when the participants read the line containing the bug, prior to any edits made, but do not fix the bug.

Stimuli is eye tracking terminology and simply means anything that is tracked on the screen by the eye tracker. In our case, the Geany IDE was the main stimulus that contained within it all the artifacts that the participant saw.

Chunks refer to a line or set of contiguous lines of code with a specific logical and semantic meaning. We also refer to them as beacons [6, 80].

Areas of Interest (AOI) refer to parts of the stimulus on which eye tracking metrics are recorded. Examples could be chunks in the code editor, the requirements area of the IDE, or the console output. The AOI is usually defined by the researcher.

Fixation is the stabilization of the eyes on an object of interest for a certain period of time. Fixations are made up of multiple raw gazes and have a duration associated with them which we refer to as the fixation duration. Most processing happens during fixations which is why they are a standard measure in most eye tracking studies.

Scanpath refers to the directed path formed by saccades between fixations. It determines how the eye navigates across the stimuli.

Reading behavior refers to the percentages of fixations that appear on the various AOIs in question.

Navigation behavior refers to the scanpath on source code over time.

Editing refers to the act of modifying the code in order to fix a bug or implement a new feature.

4.5 Procedure

We present the study procedure, including experimental setup, eye tracking hardware, and discuss the steps for pre-processing the eye tracking data to produce fixations on parts of the stimuli.

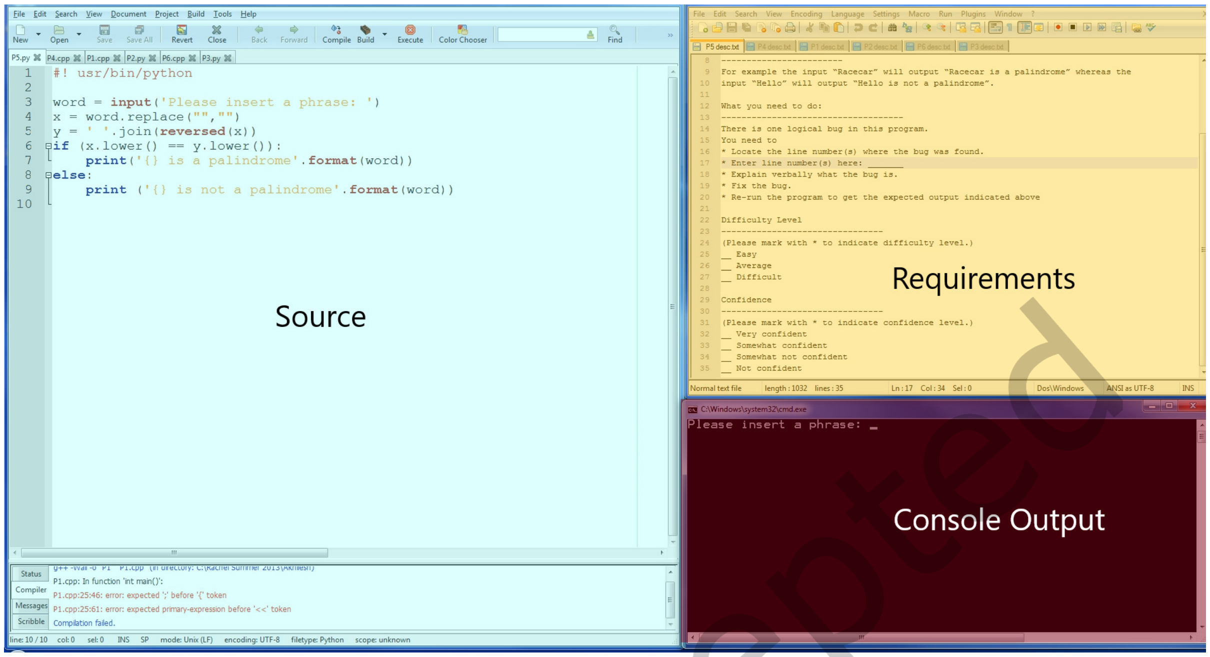
4.5.1 Experimental Setup - Study Environment. The experimental suite Tobii Studio was used to record all the eye tracking data. We set up Tobii Studio to record the computer's desktop so everything that appeared on the desktop during the study was recorded. This way, when we opened the Geany IDE, all eye tracking data was collected on the Geany IDE. The stimuli given were not images. Rather, the entire screen was a stimulus. Thus, anything looked at on the screen was recorded. Note that Tobii Studio is limited in processing eye movements with scrolling and context switching on desktop stimuli. In order to overcome this limitation we did a manual post processing step to detect scrolling and appropriately used keyframing available in Tobii Studio to detect the correct element that was looked at in the presence of scrolling. This was a manual time-consuming process. An example of how the desktop looked like is shown in Figure 2. The left part of the image shows the Geany IDE containing the source code. The top right part of the image shows a text document containing the requirements, input, and expected output. The bottom right part of the image shows the console output.

4.5.2 Eye Tracking Apparatus. The Tobii X60 eye tracker was used for the data collection and recording gaze data. It is a remote eye tracker with a 60Hz sample rate and an accuracy of 0.5 degrees. A nine point calibration was used prior to starting the study for each participant. The monitor used was a 24-inch LCD monitor at a 1280*1024 resolution.

The IV-T fixation filter [5] was run on the raw gazes and exported out of Tobii Studio for analysis. An interpolation to fill in missing gazes of up to 75 *ms* was used. A velocity window of 20 *ms* and a velocity threshold of 30 degrees per second were used to calculate the initial fixations. Adjacent fixations separated by less than 75 *ms* and 0.5 degrees are merged and fixations less than 60 *ms* are discarded.

4.5.3 Areas of Interest (AOI). In order to make sense of the eye tracking data, one first needs to define an area of interest (AOI) it falls under. Areas of interest are typically parts of the stimuli one is interested in observing. Areas of Interest (AOI) are created in the form of rectangles over the screen recording of participants completing the tasks. Tobii Studio was used to create these AOIs and map participant's fixations to the correct AOI. Two levels of AOIs were used. The top level category of AOIs is the three different sections shown in Figure 2. The three AOIs represent `Source` (the window that contains the source code), `Requirements` (the window that contains the requirements for the task), and `Console Output` (the output window used when running the program).

In addition to the top level AOIs listed above, there were some additional AOIs based on the source code. These AOIs are mostly defined as a single line of code. However, several AOIs that contain multiple lines of related code in a single chunk (also referred to as a beacon). As these are related to the code, these AOIs will differ between each stimulus. In addition, several of our programs required scrolling to view the entire program. In order to ensure that fixations were correctly mapped to the right AOI even when scrolling occurred, the AOIs were manually mapped in a post processing step onto the lines of code during scrolling so the fixation mapping would be correct. Note that the eye tracker is not aware that the items on the screen moved during a scroll (all it keeps track of is the x,y coordinate in pixels on the screen that the user is looking at) and does not automatically map gaze to the moved line, which is why we did this manually. The keyframing feature in Tobii Studio was used


File Edit Search View Document Project Build Tools Help
New Open Save Save All Revert Close Back Forward Compile Build Execute Color Chooser Find
P5.py P4.cpp P1.cpp P2.py P6.cpp P3.py
Source
Status Compiler Messages Scribble
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
P5 desc.txt P4 desc.txt P1 desc.txt P2 desc.txt P6 desc.txt P3 desc.txt
Requirements
C:\Windows\system32\cmd.exe
Console Output

```python
#! usr/bin/python

word = input('Please insert a phrase: ')
x = word.replace("","")
y = ' '.join(reversed(x))
if (x.lower() == y.lower()):
     print('{} is a palindrome'.format(word))
else:
     print ('{} is not a palindrome'.format(word))
```

```
g++ -Wall -o "P1" "P1.cpp" (in directory: C:\Rachel Summer 2013\Akhilesh)
P1.cpp: In function 'int main()':
P1.cpp:25:46: error: expected ';' before '{' token
P1.cpp:25:61: error: expected primary-expression before '<<' token
Compilation failed.
```

line: 10 / 10 col: 0 sel: 0 INS SP mode: Unix (LF) encoding: UTF-8 filetype: Python scope: unknown

```
------------------------
For example the input "Racecar" will output "Racecar is a palindrome" whereas the
input "Hello" will output "Hello is not a palindrome".

What you need to do:
------------------------------------
There is one logical bug in this program.
You need to
* Locate the line number(s) where the bug was found.
* Enter line number(s) here: _______
* Explain verbally what the bug is.
* Fix the bug.
* Re-run the program to get the expected output indicated above

Difficulty Level
--------------------------------
(Please mark with * to indicate difficulty level.)
__ Easy
__ Average
__ Difficult

Confidence
--------------------------------
(Please mark with * to indicate confidence level.)
__ Very confident
__ Somewhat confident
__ Somewhat not confident
__ Not confident
```

Normal text file length : 1032 lines : 35 Ln : 17 Col : 34 Sel : 0 Dos\Windows ANSI as UTF-8 INS

```
Please insert a phrase: _
```

Fig. 2. The three top level AOIs: Source, Requirements, Console Output

to keep track of where on the screen the scroll happened and the corresponding AOI was moved accordingly so the gaze context is maintained. This process is not completely automatic and took considerable time for all four tasks for each participant. Another student thoroughly spot checked the post processing keyframing to ensure they were done correctly. In addition, each line visible in the Geany IDE was mapped in a manual post processing step (after the keyframing was done) via our custom scripts. This was the best option to get line-level data from the editor.

Once a participant made an edit to the source code, we stopped mapping the source code AOIs as what the participants would be looking at in the AOIs may not correspond to the original source code. The new feature tasks involve a lot of editing. This means adding/removing code at different points in time as the task progresses. Tracking what a developer looks at while editing code is not a trivial problem. Currently, the state of the art does not support tracking gaze while editing in a clean manner to accurately tell what the person is looking at as the code is constantly being changed. This is simply because of how eye trackers work. Most studies done even in psychology, where eye tracking is very prevalent, only focus on static images and videos with large areas that are relatively unchanged. Because of this limitation, we chose not to report fixations on partial tokens of code (as they are written). We do not believe this data would be useful for interpretation. Instead, we report on the lines added and time to first edit, which we believe is a better metric for the new feature addition task. We are aware of only one community eye tracking infrastructure iTrace [23] that supports editing via an Atom extension [22] but even that is limited. This study was not done using iTrace or Atom. For this reason, we opted for the more traditional editing measures when looking at behaviors while performing the new feature tasks. We report on details on overall fixation count and durations for the new feature tasks without the editing involved in RQ1 (differences in programming language) and RQ2 (differences in task type).

Table 3. Measures used for each of the Research Questions

Research Question	Metric	Definition
RQ1, RQ2	Accuracy	Accuracy of a task
RQ1, RQ2	Time	The time participants took to complete a task
RQ1, RQ2	Total Fixation Count	The total amount of fixations for a given task
RQ1, RQ2	Total Fixation Count Per Character	The fixation count adjusted for the total characters of code in the stimulus
RQ1, RQ2	{AOI} Fixation Count	The total fixation count for the specified AOI
RQ1, RQ2	Total Fixation Duration	The sum of fixation durations for a given task
RQ1, RQ2	Total Fixation Duration Per Character	The fixation duration adjusted for the total characters of code in the stimulus
RQ1, RQ2	{AOI} Fixation Duration	The total fixation duration for the specified AOI
RQ1, RQ2	Average Fixation Duration	The average fixation duration for a given task
RQ3	{AOI} Fixation Duration Percentage	The {AOI} fixation duration as a percentage of the total fixation duration on the stimulus before edits
RQ3	Alpscarf Plots	Visualizes gaze transitions on specified AOI across time before edits
RQ4	Time Till First Edit	The amount of time until a participant begins to edit the source code
RQ4	Time Till First Edit Percentage	The amount of time until a participant begins to edit the source code in percentage of total time
RQ4	Lines Added	The amount of additional lines added to the source code during a feature addition task

4.6 Measures

The measures used in this experiment are based on best practices guidelines reported in the field of program comprehension, software engineering, and eye tracking [61]. We direct the reader to Duchowski et al. [20] for a detailed theoretical description of all eye tracking measures. Table 3 describes the metrics used to compare participants' behavior while working on the four tasks. We specify the research questions, the metrics used to answer the questions, and the definition of the metrics. We chose metrics based on fixations, a group of metrics used in eye tracking studies in software engineering [60, 61] that are used to measure visual effort. In prior studies, areas of interest with higher fixation count and duration are believed to have attracted more visual attention or that understanding them required more effort [61, 63, 64]. We calculated the total fixation count and duration over the given tasks, per character and specific AOIs. Furthermore, we calculated the mean fixation duration during each specific task as well. The fixation count and duration serve as a measure of visual effort when it comes to solving the different types of tasks (bug fixing and feature addition) and comparing the different programming languages (Python and C++) in our research questions.

Next, we explain why we use the fixation count per character as a metric. In order to compare eye tracking data across the different programming languages, we first need to normalize the data. The programming languages C++ and Python have very different semantic structures and different lines of code. There might not be a direct equivalent construct between the languages. In order to account for the difference in lines of code in the tasks, we make sure we normalize our fixations because indeed longer programs will have more fixations. We account for this in our analysis by normalizing by character. To do this we divide eye movement duration over a token. So if the total fixation duration is 400 *ms* on a token of 4 characters, the normalized total time is 100 *ms*. This has been done in prior work as well by Madi et al. [38] and Abid et al. [2].

5 EXPERIMENTAL RESULTS

In this section, we first present the participants' confidence levels for each task, and then present our findings for each research question. The accuracy of the bug fixing tasks was graded as correct/incorrect, by determining whether the participants found and fixed the bug correctly. The accuracy of the new feature tasks was based on whether or not the feature was correctly implemented. The time on task was measured via the eye tracking software by determining the start and end time markers in the eye tracking data for each task. On average, the participants spent 37.3 (± 19.8) minutes working on all tasks.

5.1 Confidence Levels

Given that some of our participants stated that they did not know Python, we looked at their confidence levels in their answers and understanding of each task. This information showed us that even though these participants did not consider themselves knowledgeable in Python, they mostly had a good understanding of the tasks, and we believe that due to this fact, we can include them in the analysis for answering the research questions. The following is the description of confidence levels for each task. Since all the participants were recruited from the same Python course, they were all learning Python. We asked the students to “Rate your Python programming skills" and they had the following choices: “I don't know Python", “Beginner", “Intermediate", and “Advanced". We also asked them to “Select years of experience in programming with Python", and the choices were: “None", “Between 1 and 2", "Between 3 and 5", “Between 6 and 10", “More than 10". We believe some of the students misunderstood these questions as asking about experiences prior to taking the course.

5.1.1 Bug Fixing Task in Python. The participants generally had high confidence levels about their answers for this task. Eleven participants were very confident about their answers, including three who stated that they did not know Python. Two participants were somewhat confident about their answers. Only one participant, one who did not know Python, was not confident in their answer. The participants who stated they were very confident or somewhat confident about the task answered correctly. In contrast, the only participant who answered the task incorrectly was the one who was not confident in their answer.

5.1.2 Bug Fixing Task in C++. The participants were mainly very confident about their answers to the bug fixing task in C++. All the participants stated that they knew C++. Of the 14 participants, nine were Very Confident about their answers to this task. One participant was Somewhat Confident in their answer, whereas one participant was Somewhat Not Confident in their answer. And finally, three participants were Not Confident in their answers. Additionally, every participant with high confidence in their answer and understanding of the task answered correctly.

5.1.3 Feature Addition Task in Python. The participants were mainly confident about this task as well. Two participants, who stated that they did not know Python, did not try to solve this task and had no answer to the confidence level question. Nine participants stated that they were very confident about their answers. Two of these participants did not know Python, but they were very confident in their answers to the task. One person was somewhat confident about their answer, while two people, one of whom did not know Python, were somewhat not confident about their answer. And finally, one person who stated that they did not know Python was not confident about their answer. The results show that out of the five participants who said that they did not know Python, three tried to solve the task. Two of them were very confident, and one was somewhat not confident. The confidence level and the score did not show a clear relationship. All seven participants who felt they answered the task correctly were very confident about their answers. In contrast, the participants who did not answer the task correctly had varying confidence levels in their answers.

5.1.4 Feature Addition Task in C++. Finally, we observed that the participants were mostly confident about the feature addition in C++ task. Three participants said that they were not confident in their answers. None of these three participants added any lines to the program. Two of these three participants did not try the Python feature addition task either. Eight participants were very confident in their answers, and three were somewhat confident. The participants who were either somewhat or very confident answered the task correctly, and the other three participants did not get a score because they did not try solving the task.

5.1.5 Observations. Our observations from the Python bug fixing task indicate that only one of the participants who claimed that they did not know Python was not confident about their answer and understanding of the program, and that person did not answer the task correctly. In contrast, the other four participants who claimed no Python knowledge stated that they had high levels of confidence about their understanding and answer to this task, and they answered the task correctly. Interestingly, we had more participants who did not feel confident about the bug fixing task in C++, even though all participants had stated that they knew the C++ language and had experience with it.

Furthermore, the confidence level of the feature addition tasks shows that three out of four participants who were not confident (either not confident or somewhat not confident) in the Python task were also not confident in the C++ task. This can imply that these participants had trouble with the feature addition tasks in general, and their claimed lack of knowledge in Python might not have been the most critical issue.

Based on these observations, we believe that it is more beneficial to keep the study data from the participants who claimed that they didn't have experience with Python, as their lack of experience did not affect their performance in Python tasks drastically. In addition, they were recruited from a class that taught Python.

5.2 RQ1: Reading differences between C++ and Python tasks

Research question 1 asks about the reading and navigation differences between C++ and Python tasks. The null and alternate hypotheses for this question are as follows.

LD_0 The programming language used for the tasks does not affect the visual effort of the participants working on those tasks.

LD_A The programming language used for the tasks affects the visual effort of the participants working on those tasks.

To test our hypothesis, we calculated accuracy, time, overall fixation count and duration, and AOI fixation count and duration for the tasks. Note that we use fixation metrics as a proxy for visual effort as stated in Section 4.6. We then compared these metrics in different languages. Table 4 summarizes the metrics for this research question and shows the statistical tests for language based differences in reading and code navigation.

5.2.1 Accuracy and Time. The first metrics we investigate are the time participants took to complete a task and the accuracy of the task. These two metrics together can provide insight into the difficulty of the tasks based on the programming language. Overall, we found that a single task took 586.1 seconds on average to complete, and

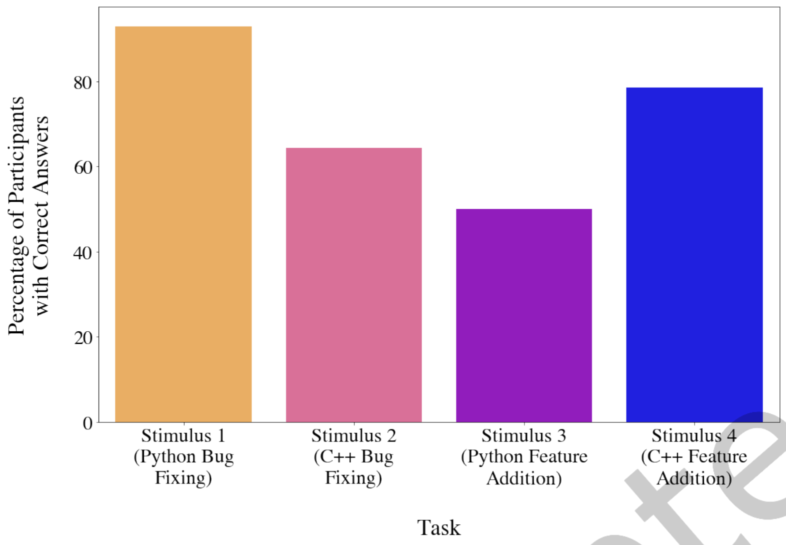


Fig. 3. Accuracy of Tasks By Stimulus

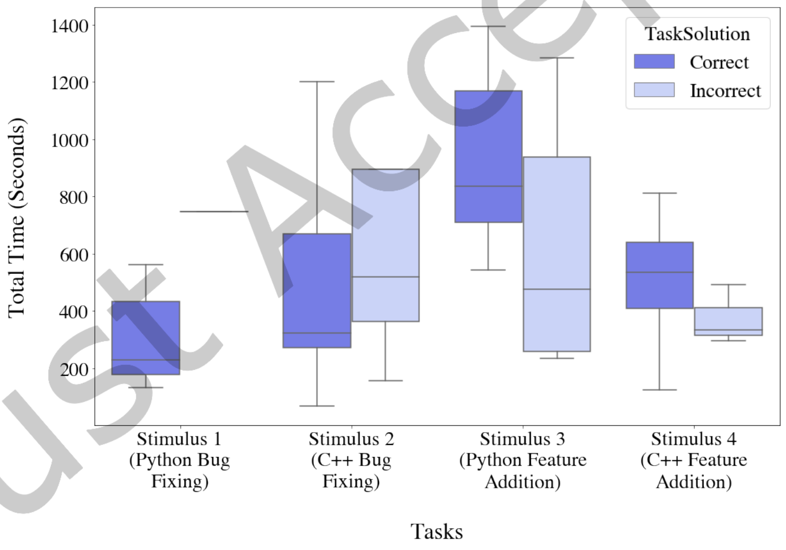


Fig. 4. Time To Complete Tasks By Stimulus (Outliers removed for scale)

there was an overall task accuracy of 73.33%. We also found that tasks written in Python took 543.7 seconds to complete, while tasks written in C++ took 628.4 seconds to complete. We also found that both Python and C++ tasks had an overall accuracy of 71.43%. Figure 3 is the bar chart showing the percentage of accurate answers from the participants for tasks from each language, and Figure 4 is the boxplot showing the total time taken

Table 4. Language-based Differences

Metric	AOI (for eye tracking metrics)	Python Tasks	C++ Tasks	p-value[‡]	Effect Size[‡]
Time (Seconds)[†]	N/A	543.7	628.4	0.8302	0.0333
Accuracy (%)[†]	N/A	71.43	71.43	-	-
Total Fixation Count	Overall (normalized per character)	2.52	2.44	0.8665	0.0178
	Source Code	955.21	859.82	0.7793	0.0255
	Requirement	170.93	185.32	0.2104	0.0484
	Output Console	170.39	90.82	0.0476*	0.2997
Total Fixation Duration (Seconds)	Overall (normalized per character)	0.612	0.570	0.9019	0.0280
	Source Code	212.00	206.91	0.7793	0.0051
	Requirement	38.63	39.63	0.6295	0.0230
	Output Console	34.08	19.08	0.0402*	0.3023
Average Fixation Duration (Seconds)	Overall	0.219	0.217	0.7282	0.0102

[†] Time and Accuracy are not eye tracking metrics, and AOI is not applicable.
‡ p-values are calculated by the Mann-Whitney test, and Effect Size is Cliff's Delta.
* $p < 0.05$

to complete each type of task among the participants. The outliers are removed by Python's plotting function, which uses the Interquartile Range Rule to detect outliers.

5.2.2 Fixation Count. Next, we investigate the number of fixations from participants during the tasks. Overall, participants had 1255.07 fixations across all tasks, including all fixations on the `Source Code`, the `Requirements`, and the `Output Console`. However, to see the effect of programming languages on this metric, we must compare the programming languages. First, we found that tasks written in Python had, on average, 1346.64 fixations, while tasks written in C++ had, on average, 1163.50 fixations. Source code length can play a role in fixation count. The longer the source code (or any type of written text) is the more fixations are required to read through and understand it, so it is important to control for fixation count as a function of the total character count of code in the stimulus. We show the total fixation count normalized by character count metric in Table 4, as well as Mann-Whitney tests comparing the metric overall. The table also reports the overall and AOI specific number of Total Fixation Count and Mann-Whitney U tests, looking at the differences between the metrics across different programming languages. We found significant differences in the Total Fixation Count on the `Output Console` AOI between the two languages (Mann-Whitney U $p = 0.0476$, small Cliff's Delta ($d = 0.2997$)).

5.2.3 Fixation Duration. Next, we look at the Total Fixation Duration in seconds overall and over the different AOIs. Looking at the total fixation duration during the tasks, we can see that the overall fixation duration for a task is, on average, 288.29 seconds. However, as with the fixation count metric, we must compare programming languages. First, we found that tasks written in Python had a total fixation duration of 304.7 seconds on average, while tasks written in C++ had a total fixation duration of 271.9 seconds. We report the normalized per character total fixation duration, the total fixation duration over different AOIs, and Mann-Whitney U test results comparing

Table 5. Task-based Differences

Metric	**AOI (for eye tracking metrics)**	**Bug Fixing**	**Feature Addition**	***p*-value** [‡]	**Effect Size** [‡]
Time (Seconds) [†]	N/A	474.1	690.8	0.0286*	0.3304
Accuracy (%) [†]	N/A	79.31	67.74	0.3200	0.1157
Total Fixation Count	Overall (normalized per character)	3.05	1.91	0.0298*	0.306
	Source Code	748.5	1066.54	0.0635	0.2385
	Requirement	130.42	225.82	< 0.001*	0.471
	Console Output	98.96	162.25	0.1299	0.1632
Total Fixation Duration (Seconds)	Overall (normalized per character)	0.757	0.452	0.0098*	0.306
	Source Code	185.8	330.9	0.1315	0.1403
	Requirement	30.69	47.63	0.0009*	0.355
	Console Output	22.67	30.77	0.2741	0.0663
Average Fixation Duration (Seconds)	Overall	0.227	0.209	0.0002*	0.1939

[†] Time and Accuracy are not eye tracking metrics, and AOI is not applicable.
‡ p-values are calculated by the Mann-Whitney test, and the Effect Size is Cliff's Delta.
* $p < 0.05$

the two languages in Table 4. The tables show that there are significant differences between the total fixation duration on the Output Console between the tasks in different languages (Mann-Whitney U $p = 0.0402$, small Cliff's Delta ($d = 0.3023$)). Finally, we did not see any significant differences between the average fixation duration throughout the tasks in different languages.

RQ1 Finding: The results show that the participants fixated more and longer on the `Console Output` AOI while working on Python tasks, and the difference is statistically significant. Based on the results, we can reject the null hypothesis LD_0.

5.3 RQ2: Reading differences between bug fixing and feature addition tasks

We present the null and alternate hypotheses for the research question on task type differences.

TD_0 The task type (bug fixing vs feature addition) does not affect the visual effort of the participants working on those tasks.

TD_A The task type (bug fixing vs feature addition) affects the visual effort of the participants working on those tasks.

Once again, we calculated accuracy, time, overall fixation count and duration, and AOI fixation count and duration for the tasks for testing our hypothesis. Table 5 summarizes the metrics for this research question and shows the statistical tests for task-based differences in reading and code navigation.

5.3.1 Accuracy and Time. We found that, on average, bug fixing tasks took significantly less time to complete. We report the accuracy and time in Table 5. The Mann-Whitney U test shows that the difference in the time working on the two types of tasks is statistically significant ($p = 0.0286$) with a medium effect size according to its Cliff's Delta ($d = 0.3304$).

5.3.2 Fixation Count. We investigate the effect of task type on fixation count. We found that bug fixing tasks, on average, had 1005.79 fixations, whereas the feature addition tasks had, on average, 1504.36 fixations throughout the task. We could not find a statistical significance in the differences. After normalizing these fixation count with the character count of the stimuli, we found significant differences between the two task types after running a Mann-Whitney test ($p = 0.0298$) with a medium effect size according to its Cliff's Delta ($d = 0.306$). We looked at the Total Fixation Count between the different AOIs in different types of tasks, shown in Table 5. We found that the only significant difference in fixation count is between the fixations on the `Requirement` AOI (Mann-Whitney U $p < 0.001$, medium Cliff's Delta ($d = 0.471$))).

We can see that while feature addition tasks had significantly more fixations, after controlling for the stimulus length, these tasks had significantly fewer fixations than bug fixing tasks. This indicates that although participants did not spend as much time and had fewer overall fixations on the bug fixing tasks, the bug fixing tasks were read more thoroughly than the feature addition tasks.

5.3.3 Fixation Duration. We also report on the effect of task type on fixation duration in Table 5. We found that bug fixing tasks, on average, had a total fixation duration of 245.7 seconds, while feature addition tasks had, on average, a total fixation duration of 330.9 seconds. We report the normalized per character fixation duration in Table 5, and we see that there is a significant difference in the metric between the two types of tasks (Mann-Whitney U $p = 0.0098$, small Cliff's Delta ($d = 0.306$)). Investigating the Total Fixation Duration over the different AOIs, we only saw a significant difference in the metric on the `Requirement` AOI (Mann-Whitney U $p = 0.0009$, medium Cliff's Delta ($d = 0.355$)). Finally, results show that there is a significant difference between the Average Fixation Duration in the two types of tasks (Mann Whitney U $p = 0.0002$, small Cliff's Delta ($d = 0.1939$)).

RQ2 Finding: Overall, the results show significant differences in the Total Fixation Count and Total Fixation Duration overall (normalized per character), indicating that participants had more frequent and longer normalized fixations overall in the bug fixing tasks. Furthermore, participants had significantly longer and more frequent fixations on the `Requirement` AOI in the feature addition tasks. There was also a significant difference between the Average Fixation Duration over all AOIs. These differences give us enough evidence to reject the null hypothesis (TD_0), showing that the different types of tasks affect the reading and navigation patterns.

5.4 RQ3: Problem solving behavior in bug fixing tasks

Next, we turn our analysis to the two bug fixing tasks. To address the third research question, we look at the scan patterns (scan paths) of the participants and the distribution of fixations on the buggy lines. This research question is exploratory in nature and does not have a formal hypothesis. Since we stop mapping gazes to lines when we detect edits (refer to Section 4.5.3 for more information), we report eye gaze distributions in this section until the first edit. We refer to this as the bug localization phase. Note that there were a lot fewer edits in the bug fix task since most bugs were limited to 1 line and required minor changes.

5.4.1 Visualization of Scan Patterns. To visualize the scan patterns of the participants while they were working on the bug fixing tasks, we used the augmented scarf plots generated by the Alpscarf [81] web application. Note that this visualization is similar but more rich in how it conveys eye transitions between the different lines and chunks of code compared to the scan patterns shown by Uwano et al. [77] and Sharif et al. [63]

Scarf plots are used in eye tracking research to visualize gaze transitions among areas of interest over time. They become less effective when there are a higher number of AOIs in a study, and Alpscarf presents a way to

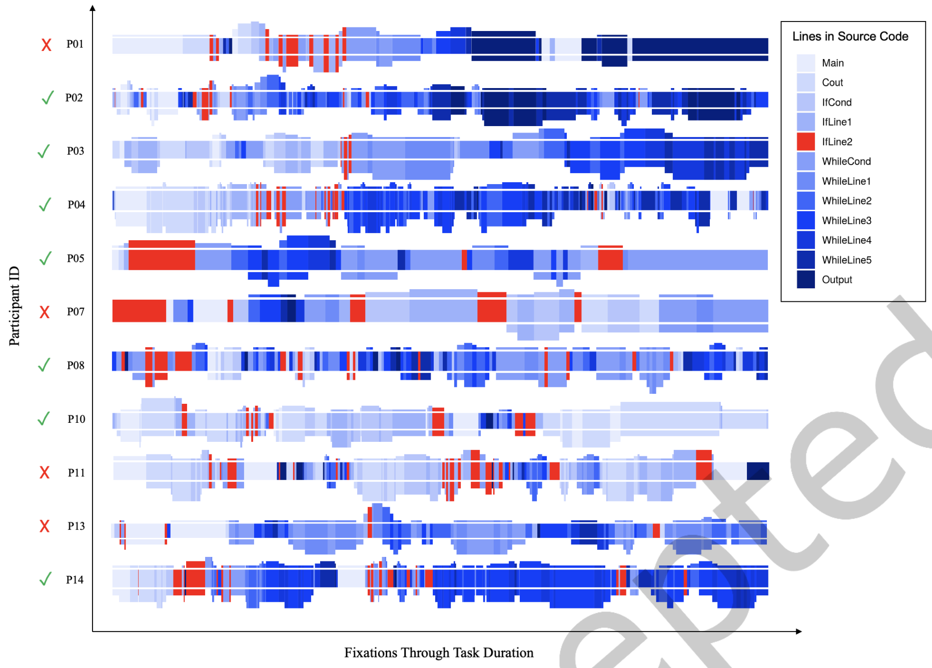


Fig. 5. Alpscarf Showing Scanpath and Proportional Fixation Duration on Lines for Each Participant During C++ Bug Localization

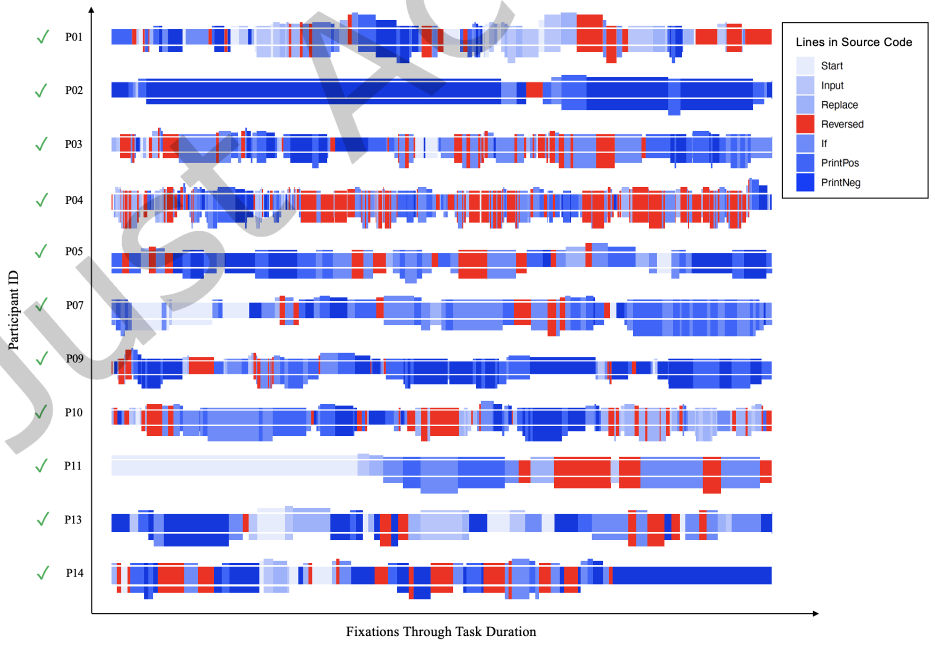
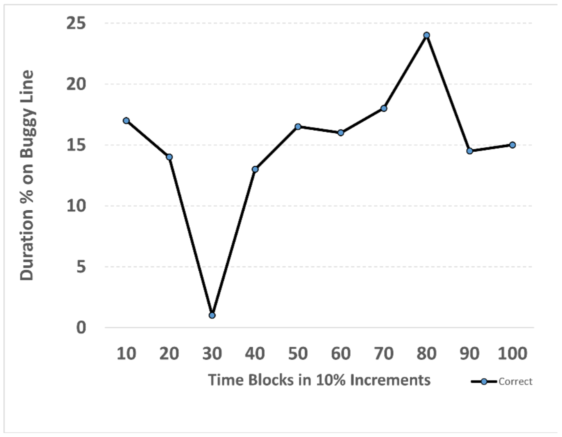
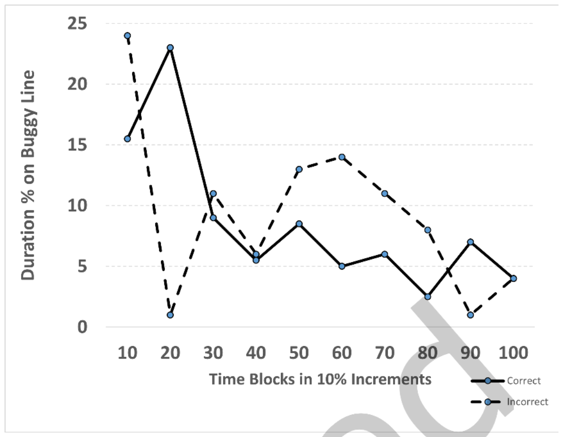


Fig. 6. Alpscarf Showing Scanpath and Proportional Fixation Duration on Lines for Each Participant During Python Bug Localization



(a) Fixation Duration Percentage of Line Containing Bug in Stimulus 1 (Python) (No incorrect answers)



(b) Fixation Duration Percentage of Line Containing Bug in Stimulus 2 (C++) (Correct and incorrect answers)

Fig. 7. Timeline of the percentage of time spent looking at buggy lines in Python and C++ Stimuli.

visualize the transitions and includes order conformity and revisits. We plotted the fixations on the different lines of the bug fixing tasks' source code (or groups of lines (chunks), as specified in Tables 8 and 6) over the entire duration of working on the task until finding the bug and before the first edit. The application gives us multiple options for visualization, and we chose the *Duration-focus* and *Normalized* plot. In the duration-focused plots, as seen in Figures 5 and 6, the width of each bar specifying a fixation is proportionate to the fixation duration. By using this option, we can see both the transitions the reading order, and the relative time spent on specific lines. We also chose the normalized view, which results in all the scarf plots being the same width despite the various number of transitions or the different fixation times and overall duration of the task. We chose this option for the better visibility of the data, as some participants spent a relatively longer time than others. The normalized view helps in comparing the Alpscarfs to better discover patterns. In the Alpscarfs, the mountains (the hills over the fixations) represent conforming to the expected fixation order (such as reading a program line by line), and the valleys represent the revisits over the AOIs.

Figures 5 and 6 show the Alpscarfs of the fixations of eleven people. The fixations on the line containing the bug are specified with the color red. A green checkmark is placed next to the participants who answered the tasks correctly, and a red X is placed next to those who answered incorrectly. As mentioned earlier, we do not have any source code fixations for P6 and P12. Thus, these participants are not included in the visualization. P12 completed both tasks incorrectly, and P6 completed both tasks correctly. Furthermore, for better visibility of the data visualization in the paper, we removed the Alpscarf of P9 in Figure 5, as that participant spent an unusually long time on the task. Despite the normalization of the width of the Alpscarfs, the scan pattern of P9 was not observable. We also removed P8 from Figure 6, as that participant only fixated on one area of interest (PrintNeg), and there was no pattern to be studied. The scan patterns that were removed from the paper to increase readability are included in the replication package [40].

For the bug fixing tasks, a bug was introduced on a single line of the source code. Knowing when participants identified the bug is essential to understanding the problem-solving behavior of the participants. To investigate the timing of when participants looked the buggy line while working on the task, we segmented the data into ten sections. We computed the percentage of the time spent fixating on the line containing the bug out of the time spent fixating on any line in the program. We then compared these percentages. These bug localization fixation duration distributions can be seen in Figure 7a for Stimulus 1 (Python) and in Figure 7b for Stimulus 2 (C++).

We see a pattern of initial decrease of the fixations on the line containing the bug in the Stimulus 1 bug localization timeline. It seems to indicate that participants read past the bug in the program, potentially while reading for comprehension, and returned to the line containing the bug after reading other parts of the program. Figure 6 confirms that most participants spent time looking at the line containing the bug at the beginning of the task. The Alpscarf shows participants looking at the buggy line more and longer at the beginning stages of the task and coming back to it again later on (Line `Reversed`). This is expected since, in these two stimuli, the bugs are located in a similar relative position in the code: line 5 of 9 in Stimulus 1 and line 15 of 26 in Stimulus 2. As such, the initial reading of a program should look similar if participants read past the line containing the bug.

We analyzed Stimulus 2's bug localization timeline to compare the participants who correctly found and corrected the bug and those who failed to find and correct the bug. However, due to the limitations of source code AOI mapping mentioned in Section 4.5.3, we could not find fixations on any source code AOIs for two participants, P6 and P12. P12 completed both tasks incorrectly, while P6 completed both tasks correctly. All the other participants working on Stimulus 1 completed the task correctly. As for Stimulus 2, aside from P6 and P12, eight participants submitted the correct solutions to the task and four participants gave incorrect answers to the task.

Looking at overall patterns in the bug localization timelines for Stimulus 2 (Figure 7b), we see that the segment with the highest percentage of duration time spent on the line containing a bug occurs within the first 20% of the timeline. This indicates that participants spent significant time at the beginning of the bug localization task looking at the line containing the bug. This is also confirmed by Figure 5, in which we can observe that most participants have fixated on line `IfLine2` containing the bug early in the task.

Comparing the bug localization timeline between correct and incorrect solutions can lend insight into the participants' different behaviors. We notice in Figure 5 that the participants with incorrect answers (P01, P07, P11, P13) had a higher percentage of fixations on the line containing the bug in the middle of the task. This can also be seen in the timeline data (Figure 7b). However, the participants who answered the task correctly, mostly looked at the line containing the bug at the beginning of the task.

5.4.2 Context of Fixations on Line Containing Bug. The bug localization timeline focused on identifying when participants looked at a line containing a bug, but it did not help us understand the context of those fixations. We wanted to understand whether the participants were reading the program in a linear order or if they were relating another line to the line containing the bug. Investigating this context allows us to get a better sense of the participant's strategy to locate the bug.

To learn the context of what was looked at before fixations on the line containing the bug, we specified a criterion and filtered the fixations based on that criterion. Our criteria were "fixations within five fixations of a fixation on the line containing the bug." Some of the fixations that fall into this criteria are fixations on the line that contains the bug, but we do not include those fixations. This windowed fixation dataset can then allow us to understand the context of the fixations on the line containing the bug by contrasting it with the overall distribution of fixations on the stimuli.

We first analyze the distributions of Stimulus 2 (C++) shown in Table 6. The distribution of fixation duration on the source code AOIs is reported on the overall dataset and the windowed fixation dataset. We can see that there exist several differences between these two distributions. First, the line containing the bug, `IfLine2`, makes up a larger percentage of fixation duration in the windowed context. In addition, the line AOIs immediately before and after `IfLine2` also make up a large fixation duration percentage. This shows that participants looked at the lines closer to the bug before reading the line containing the bug.

Table 6. Overall Source Code AOI Distribution Vs Windowed Source Code AOI Distribution For Stimulus 2 (C++)
* indicating where the bug is located

Line	**Code**	**Overall Duration**	**Windowed Duration**

Main	`#include <iostream>`
	`#include <string.h>`
	`#define MAX_SIZE 256`
	`using namespace std;`
	`int main()`
		7.98%	9.40%

Cout	`char word[MAX_SIZE];`
	`cout <<"Please enter a phrase to be translated: ";`
	`cin.get(word, MAX_SIZE);`
		9.84%	8.45%

IfCond	`if(strlen(word) >0)`	6.58%	7.26%
IfLine1	`char* first = &word[0];`	6.26%	16.17%
IfLine2*	`char* last = &word[strlen(word)];`	8.37%	22.83%
WhileCond	`while(first <last)`	15.33%	21.91%
WhileLine1	`char tmp = *first;`	8.36%	7.68%
WhileLine2	`*first = *last;`	8.64%	7.93%
WhileLine3	`*last = tmp;`	10.85%	8.33%
WhileLine4	`++first;`	6.88%	3.99%
WhileLine5	`-last;`	6.14%	3.75%
Output	-	7.00%	2.74%

Comparing participants with a correct solution and incorrect solutions in Table 7, we can see that differences in the windowed fixation datasets exist between these participants. First, participants with an incorrect solution looked at the source code AOIs above `IfLine2` for a longer percentage of time than participants with a correct solution. The high percentage of fixation duration on the first AOI, `Main`, indicates that they started from the top of the program and read to the line containing the bug multiple times throughout the task. This pattern can be observed in Figure 5 in participants who answered the task incorrectly. In addition, we see that the AOI immediately after `IfLine2` is looked at with a lower percentage of fixation duration than participants who correctly found the bug, indicating that they did not regress to the line containing the bug from the `WhileCond` AOI as often.

Overall, it seems that participants with a correct solution seem to have fixations that are focused on lines that are physically close to the bug before looking at the line containing the bug. We can also observe this in Figure 5, which shows us that the participants with correct answers spend significant time looking at the line `WhileCond`. This is in line with the findings of Peterson et al. [52], stating that participants view related lines together. The replication package contains the entire source code mapping of lines with the line mnemonic label we use here (i.e., `WhileCond` and such).

Table 7. Windowed Source Code AOI Fixation Duration Distribution For Correct vs Incorrect Solutions For Stimulus 2 (C++) * indicating where the bug is located

Line	Code	Correct Solution	Incorrect Solution
Main	`#include <iostream>` `#include <string.h>` `#define MAX_SIZE 256` `using namespace std;` `int main()`	3.03%	15.14%
Cout	`char word[MAX_SIZE];` `cout <<"Please enter a phrase to be translated: ";` `cin.get(word, MAX_SIZE);`	8.26%	9.20%
IfCond	`if(strlen(word) >0)`	5.24%	11.43%
IfLine1	`char* first = &word[0];`	14.00%	21.28%
IfLine2*	`char* last = &word[strlen(word)];`	24.14%	20.05%
WhileCond	`while(first <last)`	23.52%	13.00%
WhileLine1	`char tmp = *first;`	8.87%	12.27%
WhileLine2	`*first = *last;`	7.34%	10.90%
WhileLine3	`*last = tmp;`	10.33%	3.49%
WhileLine4	`++first;`	4.47%	0.69%
WhileLine5	`-last;`	4.16%	2.09%
Output	-	2.25%	4.74%

Table 8. Overall Source Code AOI Distribution Vs Windowed Source Code AOI Distribution For Stimulus 1 (Python)

Line	Code	Overall Duration	Windowed Duration
Start	`#! usr/bin/python`	8.71%	3.40%
Input	`word = input('Please inser a phrase: ')`	4.87%	4.49%
Replace	`x = word.replace("","")`	7.54%	9.10%
Reversed*	`y = ' '.join(reversed(x))`	17.17%	28.72%
If	`if (x.lower() == y.lower()):`	21.12%	29.41%
PrintPos	`print('{} is a palindrome'.format(word))`	16.63%	13.74%
PrintNeg	`else:` `print ('{} is not a palindrome'.format(word))`	34.96	23.93

Next, we analyze the distributions of Stimulus 1 (Python) shown in Table 8. First, we see that the line containing the bug, `Reversed`, was looked at for a higher percentage of time in the windowed context. While the AOIs immediately before and after the line containing the bug are looked at with a higher percentage of fixation duration in the windowed dataset, the difference is not as pronounced as in the line immediately following it. However, we still see that the fixation duration of the fixations before looking at the line containing the bug is higher on the adjacent lines than the overall duration during the task.

5.4.3 Observations from the Alpscarfs. Looking closer into the participants' scan patterns in Figure 5 and 6, in which the AOI of the line containing the bug is in the color red, we can see that the line contained the bug was more frequently fixated on in the Python bug fixing task. Figure 6 shows that P01, P03, P04, P05, P09, P10, and P14 all looked at the line containing the bug at the beginning of the Python bug fixing task, and they frequently revisited that AOI until the end of the task, indicating the importance of the line to the readers. The Alpscarf also shows that the participants did not necessarily read the Python code in order, and they went back and forth between the different lines many times.

In the C++ bug fixing task, as seen in Figure 5, we can see different patterns in participants who did not answer the task correctly. We can see a more chronological reading pattern in some participants reading the C++ bug fixing code (P01, P02, P03, P4, P14) compared to Python. We do not see a similar pattern in all the participants who solved the task correctly, as some visited the lines in the order they were written, and some did not. Out of the participants who answered incorrectly, P01 and P13 fixated on the line containing the bug a few times at the beginning of the task, but the fixations were very short. In particular, P01 did not go back to the line containing the bug at all. Both P07 and P11 revisited the line containing the bug multiple times, but they did not provide the correct answer to the task.

Finally, we compared the scan patterns of some of the individual participants across the bug fixing tasks in the two languages. As an example, P04 worked on the C++ task in a very chronological manner, starting from the beginning of the source code and reading it to the end, only to come back later and re-read the source again. The same participant did not follow such a method while working on the Python task. They went back and forth between the AOIs in the Python code and read the code multiple times, resulting in many fixations over the buggy line. The participant answered both questions correctly, indicating that either the type of bug required longer fixations in Python or that both methods of code reading work well for this particular participant. As another example, P13, who answered the C++ task incorrectly and the Python task correctly, read the C++ code lines more in the order they were written but chose another approach for reading the Python code. We can also see longer fixations from P13 on the buggy line in the Python task but infrequent and shorter fixations in the buggy line in the C++ task.

Overall, even though we could not find a very clear pattern in the scan patterns of the participants, comparing the patterns still provided some insight into the individuals' choices and reading patterns. It is possible that these observations account for the individual differences that occur in each person as they are building the mental model for the programs.

RQ3 Finding: The results show us that the participants pay the most attention to the lines surrounding the buggy line. Most participants did not read the code linearly and they kept going back to the buggy line and the lines surrounding it. We also observe that each participant did not necessarily follow the same reading patterns for both Python and C++ tasks.

5.5 RQ4: Problem solving behavior in new feature tasks

Since new feature tasks, by definition, require the students to change the code, we investigate editing behavior to address this research question. We explain in Section 4.5.3 why eye tracking measures are not used during editing as they are not reliably mapped to edited code and no vendor based software supports this to date. Research prototypes such as [22] have some support for editing (only in Atom), however, our study was not done with their framework. We plan on using the iTrace framework for future studies as it significantly simplifies the mapping of gaze data on tokens [23, 84]. Similar to RQ3, this research question is exploratory in nature and does not have a formal hypothesis. Table 9 shows the different metrics used in RQ4 and the results of the statistical tests comparing them.

Table 9. Metrics Used In RQ4

Language	Metric	Correct	Incorrect	p-value	Effect Size
Python	Percent Time Till First Edit	13.44%	44.10%	0.0146*	-0.7460
	Total Time Till First Edit	107.09 sec.	177.71 sec.	0.0195*	-0.7143
	Lines Added	9.2222	4.2857	0.0186*	0.7143
	Confidence Level	4.0000	2.4286	0.0045*	0.7143
	Difficulty	1.7778	2.4286	0.1005	-0.4762
	Participant Count	7	7		
C++	Percent Time Till First Edit	23.65%	91.75%	0.0114*	-1.0000
	Total Time Till First Edit	122.35 sec.	333.77 sec.	0.0115*	-1.0000
	Lines Added	7.8333	0.0000	0.0108*	1.0000
	Confidence Level	3.6667	1.0000	0.0054*	1.0000
	Difficulty	1.9167	2.0000	0.9367	-0.0556
	Participant Count	11	3		

* $p < 0.05$

5.5.1 Time to First Edit. To investigate the editing behavior of the participants in the feature addition task, the first metric considered is the time until the first edit of the source code. Before participants can add a feature, they must understand and comprehend the program. While they don't need to understand the entirety of the program to add a feature, they must have familiarity with the source code and know what code needs to be added and where it needs to be added. For Stimulus 3 using Python, we found that participants who correctly completed the feature addition task waited 107.09 seconds on average before making their first edit, while the participants who failed to correctly complete the feature addition task waited 177.71 seconds on average. This difference was statistically significant according to a Wilcoxon test ($p = 0.0195$) with a large effect size according to its Cliff's Delta ($d = 0.7143$).

For Stimulus 4 using C++, we found that participants who correctly completed the feature addition task waited 122.35.09 seconds on average before making their first edit, while participants who failed to correctly complete the feature addition task waited 333.77 seconds on average. This difference was statistically significant according to a Wilcoxon test ($p = 0.0115$) with a large effect size according to its Cliff's Delta ($d = 1.00$).

In order to adjust for the total time that the feature addition task took, we also compared the time until the first edit in terms of percentage of total time (TTFE Percentage). For Stimulus 3 (Python), we found that, on average, participants who correctly completed the feature addition had 13.44% of the total time pass before making the first edit, and participants with an incorrect solution had 44.10% of the total time pass before making the first edit. This difference was statistically significant according to a Wilcoxon test ($p = 0.0146$) with a large effect size according to its Cliff's Delta (d = 0.7460). For Stimulus 4 (C++), we found that, on average, participants who correctly completed the feature addition had 23.65% of the total time passed before making the first edit, and participants with an incorrect solution had 91.75% of the total time passed before making the first edit. This difference was statistically significant according to a Wilcoxon test (p = 0.0114) with a large effect size according to its Cliff's Delta (d = 1.0000).

5.5.2 Added Lines. Another aspect of feature addition to investigate is the number of lines added to the source code. We found that the number of lines added to the source code differed between participants with a correct solution and participants with an incorrect solution. For Stimulus 3 (Python), participants with a correctly implemented solution added an average of 9.22 lines to the source code, while participants with an incorrectly implemented solution added an average of 4.28 lines to the source code. This difference was statistically significant according to a Wilcoxon test (p = 0.0186) with a large effect size according to its Cliff's Delta (d = 0.714). Of the seven trials that resulted in an incorrectly implemented feature addition task, three solutions did not add any lines to the source code, one added 12 lines, while the remaining three added six lines.

For Stimulus 4 (C++), participants with a correctly implemented solution added an average of 7.83 lines to the source code, while participants with an incorrectly implemented solution did not add any lines to the source code. This difference was statistically significant according to a Wilcoxon test (p = 0.0108) with a large effect size according to its Cliff's Delta (d = 1.0000). Only three participants failed to complete the task, two of whom did not make any attempts to change the source code.

RQ4 Finding: Results show significant differences between the editing related metrics between the two groups of participants with correct and incorrect answers to the feature addition task. We observe that the participants who make an edit in the earlier stages of working on the task are more likely to answer the question correctly. The participants with correct answers also add more lines to the code.

5.6 Threats to Validity

We discuss the possible threats to validity for internal, external, construct, and conclusion and state how we tried to mitigate them.

Internal validity: While we do compare tasks based on the language and task type to determine if language and task type have an effect on eye movement patterns of programmers, it is possible for a different but comparable task to have a different difficulty level to a certain programmer, meaning that even if we present equally difficult but different tasks some programmers may find one task to be more difficult for them to complete. In addition, we tried to have bug fixing tasks and feature addition tasks have a similar level of difficulty. However, we do not claim that these are representative of all bug fix and feature addition tasks.

External validity: For external validity, the small size of the programs used may limit the generalizability of our results. In addition, our participants were mainly students. Because of this, our results may not generalize to a larger population of programmers, including professionals.

Construct validity: Addressing construct validity, we used some thresholds in our analysis. For example, in RQ3, we looked at the first five fixations before any fixation on the line containing the bug. We also chose to use ten segments in the bug localization timeline. Increasing the segment count will provide additional granularity, but the small sample count may cause gaps in the dataset to appear. We believe that ten segments balance these two goals for our purposes. In RQ4, we used the time until the first edit as a proxy for when participants were done understanding the program and began to add a feature. However, a participant can begin adding a feature and continue reading the program for comprehension after editing has begun. In RQ3, we looked at the distribution of fixation duration over the AOIs of the source code. These AOIs are mostly line based AOIs, but some closely related lines were grouped together as a single chunk. Chunk based analysis has been conducted in previous studies [56], but the decision over which lines to group together can influence the analysis. We mitigate this by only directly comparing the distribution of fixation durations over AOIs of the same task. These source code AOIs are also only tracked before an edit occurs. To mitigate this, we only used the source code AOIs for the bug localization part of the bug fixing tasks which should occur in its entirety before they attempt to fix the bug.

Conclusion validity: Finally, for conclusion validity, we used the appropriate statistical tests for our inferential statistics.

6 DISCUSSION AND IMPLICATIONS

In this paper, we looked at the differences in eye movement behaviors in C++ and Python in task types of fixing a bug and adding a new feature.

We found that while working on bug fixing tasks participants had significantly more fixations per character count of code than when they worked on feature addition tasks. However, regarding the absolute number of fixations, the bug fixing tasks had significantly fewer fixations as the tasks were completed in a shorter amount of time. We also found that bug fixing tasks had an average fixation duration significantly longer than the average fixation duration for feature addition tasks. We found no significant differences in the total fixation count or duration when adjusted for the character counts in the stimuli between tasks written in Python and tasks written in C++. This shows that for the overall fixation metrics that we measured, the task type is more important for determining these metrics than the language the task is written in.

For the bug fixing tasks, we found several similarities in the navigation behavior during the bug localization phase of the task. First, we found that after the first 20% of the bug localization phase, a decrease in the percentage of time spent fixating on the line containing the bug was observed for C++. There are a few possible explanations for this behavior. The first is that a large portion of the time was not spent locating the bug, but instead, it was spent on understanding the behavior of the program. After they understand what the program is supposed to do, the bug becomes easier to spot and they spend little time looking for the bug. The second explanation is that they located the potential bug early in the bug localization task and spent the remaining time verifying that the line was indeed a bug by reading the rest of the code. In addition to this behavior, we also saw that participants often looked at lines that were physically close to the line containing the bug before looking at the bug and quite often regressed back to the line containing the bug from the AOI after it. In Python, however, there were more fixations on the buggy line during the middle and latter part of the session for a majority of the participants. This is in direct contrast to what was observed in C++. This indicates that choice of language plays a role in how students read the code looking for the buggy line.

For the feature addition tasks, we found that participants who correctly implemented the task added more lines of code to the source code and were quicker to make their first edit to the source code than participants who incorrectly implemented it. While the number of lines added is biased against the participants with incorrect solutions, as several incorrect solutions added no additional lines to the code, the time till the first edit is still a clear divider. It seems to indicate that the participants who completed the feature addition task were able to identify where to start adding the feature quicker or iterate over potential solutions quicker. This difference was seen in both the Python and C++ tasks. We also found that feature addition tasks had significantly more fixations and fixation time spent inside the `Requirements` AOI. Since participants needed to refer to the requirements located inside this AOI to correctly implement the new feature, it makes sense that more fixations and fixation time were needed for these feature addition tasks.

6.1 Relation to Prior Work

With respect to RQ1, which tried to determine differences in programming languages, we did indeed see a difference based on the normalized total fixation count between C++ and Python. Students spent more time looking at the Console output when performing Python tasks. We do not believe this is due to noise in the data because both Python and C++ had I/O operations (See Table 4.3). We believe that it might be possible that the Python programs were easier to change and the students got immediate feedback in the console. The data seems to support this assumption. This result seems to align with Tshukudu and Cutts [74], where different models were needed to transition between programming languages. Murphy et al. also point to the Console view being selected the most during their collection of interaction data from Java developers in Eclipse [44].

With respect to RQ2-RQ4, which tried to determine differences in task type, the results show significant differences in the Total Fixation Count and Total Fixation Duration overall (normalized per character), indicating that participants had more frequent and longer normalized fixations overall in the bug fixing tasks. These results align with what was reported by Cunningham et al. [18], where they found the behaviors to change when different task types are used. One possible reason why the fixation count might be higher for bug fixing is because the reading strategy when looking for bugs is very different from reading code just to understand what it is doing. When finding and fixing bugs, one zeroes into certain parts and traces and re-reads them.

With respect to RQ3 and bug fixing behavior, fixations are found closer to the line containing the bug right before they look at buggy lines. This behavior is also found in prior work by Peterson et al. [52], indicating that participants view related lines together and using chunking as a mechanism to map eye gazes is replicable in other studies and tasks as well.

Finally, with RQ4, we found that for both C++ and Python, participants could identify where to start adding a feature (noted by the time to first edit) quicker and potentially iterate over solutions to solve the task correctly. Brown et al. collected five years of programmer activity data in a Java IDE, namely BlueJ [13]. Part of this data includes edit sequences of novices. However, they state that no study has made use of the code execution and code editing sequences as of yet. It would be interesting to see how their results relate to what we found in our study with respect to the time to first edit and task performance. This is left as a future exercise. The individual differences we see in our study are also reported in Jbara et al. [28].

6.2 Implications for CS Education Researchers

This is one of the first eye tracking studies that look at the same individual performing two task types in two programming languages. None of the prior work used the visualization plots shown in Figures 5 and 6. This was a new form of data visualization and analysis that is richer than what is presented in Uwano et al. [77] and Sharif et al. [63]. In the future, researchers looking at scan patterns can compare not just lines but chunks of lines across time. This is important because sometimes, a programming plan [55] or beacon [12, 35] is not necessarily encapsulated in just one line. We believe this method of comparison opens up new avenues of research for comparing studies with each other in a more scalable way. When analyzing eye tracking studies, it is also important to account for individual differences that are quite common. We see this in our analysis of the scarfplots in our study but also in other studies in the literature, such as in Jbara et al. [28]. CS education researchers can also benefit from these results by building better tools that guide novices in recognizing bugs, thereby advancing the state of the art of teaching novices.

The fact that we have noticed differences in task type within the same individual tells us that the type of task is extremely important, and as education researchers, we should be studying all the different types of tasks that developers perform on a daily basis. We name a few, such as refactoring, summarization, bug fixing, new feature addition, testing, and code review. Eye tracking has only mainly studied summarization, with a few papers looking into bug localization and only one on code review [11]. In addition, most work is done on the Java programming language. It is time to start branching out to other languages, using more realistic tasks, and also multiple task types with varying complexities [3, 21]. Especially now that we have eye tracking frameworks such as iTrace [23, 84] that make the running and mapping of gazes to tokens relatively easy and straightforward.

Kersten and Murphy provide a task context model to help with developer productivity [32]. The context is created by monitoring a programmer's activity and extracting structural relationships between program artifacts. CS education researchers can consider doing something similar with eye tracking data where when enabled, the eyes are tracked while the programmer is fixing a bug. Later, these scan paths can eventually be used to replay the thought process to the same or another developer via visualizations. Such future tools would help recommendation systems as well, where eye gaze history could be used to recommend areas a student should look at, based on how they have viewed it in the past so as to keep their mental model in sync with their prior debugging session. Given the advancement in eye tracking infrastructure [23, 84] and the affordability of research-grade trackers, this is not a far fetched goal.

6.3 Implications for CS Educators - Teaching

Our findings show a more substantial difference in eye movement patterns in different task types than in various programming languages. This finding indicates that the comprehension patterns differ regarding the goal of the task at hand and suggests a need for finding different teaching techniques for solving various types of programming tasks, no matter what the programming language taught to the students is. Further investigation into differences of programming languages versus differences of tasks on learning and comprehension, can offer some insight into "Which programming languages should we teach to students?" [48] and can help in determining what factors other than language are the most important in program comprehension.

Due to the comprehension pattern differences between tasks, we also suggest that instructors try to include different tasks in programming homework (e.g. bug fixing, adding features, and summarization) to improve different program comprehension skills in the students, instead of only focusing on full implementation of specific problems. Students need more practice reading code that is not written by them so they can practice their program comprehension skills and learn when to switch back and forth between different models of comprehension [12, 35, 50, 58, 79]. It will also prove to be more useful in their future careers, as software developers do incremental work on partial code similar to subgoals [43] instead of always writing code from scratch.

Finally, CS educators can better support student debugging if they know what novice students typically look at during various types of tasks. They can also actively teach students not to fear editing the code early on, because we see a correlation between time to edit and accuracy in feature addition task performance.

7 CONCLUSIONS AND FUTURE WORK

The paper presents an eye tracking study on how the type of task (bug fix and new feature addition) and language (C++ vs. Python) affect student programmer behavior. We found that the participants had significantly longer average fixation duration and total fixation duration adjusted for source code length during bug fixing tasks compared to the feature addition tasks. We also find that the total fixation duration adjusted for source code length was significantly higher during tasks done in Python than in C++, but the effect was not as pronounced. We found that during the bug fixing task in C++ many participants read the line containing the bug early in the task and then continued to other parts of the code before ultimately returning to the line containing the bug. In Python however, they read the buggy line many times in the middle or later in the session. We also found that participants looked at lines next to the line containing the bug before looking at the line containing the bug more often than the overall distribution and that they often regressed back to the line containing the bug from the lines following it. Finally, we found that participants who successfully completed a feature addition task took significantly less time to make the first edit to the source code.

As part of future work, we plan on conducting a study using modern eye-tracking frameworks such as iTrace [23, 84] in order to see how participants traverse through larger and more realistic open source systems written in Python, Java, and C++. This would allow us to see if our results scale to a much larger realistic setting. We would also like to vary additional factors like task complexity within each task type.

8 ACKNOWLEDGMENTS

This work is supported in part by the US National Science Foundation under Grant Numbers CNS 18-55753 and CCF 18-55756. We thank all the participants of this study for their time. We also thank the reviewers for the valuable feedback received.

REFERENCES

[1] Nahla J. Abid, Jonathan I. Maletic, and Bonita Sharif. 2019. Using Developer Eye Movements to Externalize the Mental Model Used in Code Summarization Tasks. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications (ETRA '19)*. ACM, New York, NY, USA, Article 13, 9 pages. https://doi.org/10.1145/3314111.3319834

[2] Nahla J Abid, Bonita Sharif, Natalia Dragan, Hend Alrasheed, and Jonathan I Maletic. 2019. Developer Reading Behavior While Summarizing Java Methods: Size and Context Matters. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 384–395. https://doi.org/10.1109/ICSE.2019.00052

[3] Shulamyt Ajami, Yonatan Woodbridge, and Dror G Feitelson. 2019. Syntax, predicates, idioms—what really affects code complexity? *Empirical Software Engineering* 24 (2019), 287–328.

[4] Anthony Allevato and Stephen H Edwards. 2010. Discovering patterns in student activity on programming assignments. In *ASEE Southeastern Section Annual Conference and Meeting*.

[5] Richard Andersson, Linnea Larsson, Kenneth Holmqvist, Martin Stridh, and Marcus Nyström. 2017. One algorithm to rule them all? An evaluation and discussion of ten eye movement event-detection algorithms. *Behavior Research Methods* 49, 2 (2017), 616–637. https://doi.org/10.3758/s13428-016-0738-9

[6] Christoph Aschwanden and Martha Crosby. 2006. Code scanning patterns in program comprehension. In *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06)*.

[7] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson R. Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 575–585. https://doi.org/10.1109/ICSE.2017.59

[8] Roman Bednarik. 2007. *Methods to analyze visual attention strategies: Applications in the studies of programming*. University of Joensuu.

[9] Roman Bednarik and Justus Randolph. 2008. Studying cognitive processes in computer program comprehension. *Passive Eye Monitoring* (2008), 373–386.

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

[11] Andrew Begel and Hana Vrzakova. 2018. Eye Movements in Code Review. In *Proceedings of the Workshop on Eye Movements in Programming (EMIP '18)*. ACM, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/3216723.3216727

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

[13] Neil C. C. Brown, Amjad AlTadmri, Sue Sentance, and Michael Kölling. 2018. Blackbox, Five Years On: An Evaluation of a Large-scale Programming Data Collection Project. In *Proceedings of the 2018 ACM Conference on International Computing Education Research, ICER 2018, Espoo, Finland, August 13-15, 2018*, Lauri Malmi, Ari Korhonen, Robert McCartney, and Andrew Petersen (Eds.). ACM, 196–204. https://doi.org/10.1145/3230977.3230991

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

[15] Teresa Busjahn, Carsten Schulte, Bonita Sharif, Andrew Begel, Michael Hansen, Roman Bednarik, Paul Orlov, Petri Ihantola, Galina Shchekotova, and Maria Antropova. 2014. Eye tracking in computing education. In *Proceedings of the tenth annual conference on International computing education research*. 3–10.

[16] Tony Clear, JL Whalley, Phil Robbins, Anne Philpott, Anna Eckerdal, and Mikko-Jussi Laakso. 2011. Report on the final bracelet workshop: Auckland university of technology, september 2010. (2011).

[17] Martha E Crosby and Jan Stelovsky. 1990. How do we read algorithms? A case study. *Computer* 23, 1 (Jan. 1990), 24–35.

[18] Kathryn Cunningham, Sarah Blanchard, Barbara Ericson, and Mark Guzdial. 2017. Using tracing and sketching to solve programming problems: Replicating and extending an analysis of what students draw. In *Proceedings of the 2017 ACM Conference on international computing education research*. 164–172.

[19] Françoise Détienne and Elliot Soloway. 1990. An empirically-derived control structure for the process of program understanding. *International Journal of Man-Machine Studies* 33, 3 (1990), 323–342.

[20] Andrew Duchowski. 2007. *Eye Tracking Methodology: Theory and Practice.* https://doi.org/10.1007/978-1-84628-609-4

[21] Rodrigo Duran, Juha Sorva, and Sofia Leite. 2018. Towards an analysis of program complexity from a cognitive perspective. In *Proceedings of the 2018 ACM Conference on International Computing Education Research*. 21–30.

[22] Sarah Fakhoury, Devjeet Roy, Harry Pines, Tyler Cleveland, Cole S. Peterson, Venera Arnaoudova, Bonita Sharif, and Jonathan Maletic. 2021. gazel: Supporting Source Code Edits in Eye-Tracking Studies. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 69–72. https://doi.org/10.1109/ICSE-Companion52605.2021.00038

[23] Drew T Guarnera, Corey A Bryant, Ashwin Mishra, Jonathan I Maletic, and Bonita Sharif. 2018. itrace: Eye tracking infrastructure for development environments. In *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*. ACM, 105.

[24] Yann-Gaël Guéhéneuc. 2006. TAUPE: Towards Understanding Program Comprehension. In *Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research (CASCON '06)*. IBM Corp., Riverton, NJ, USA, Article 1. https://doi.org/10.1145/1188966.1188968

[25] Dan Witzner Hansen and Qiang Ji. 2009. In the eye of the beholder: A survey of models for eyes and gaze. *IEEE transactions on pattern analysis and machine intelligence* 32, 3 (2009), 478–500.

[26] Yiling Hu, Bian Wu, and Xiaoqing Gu. 2017. An eye tracking study of high-and low-performing students in solving interactive and analytical problems. *Journal of Educational Technology & Society* 20, 4 (2017), 300–311.

[27] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. 2019. Fostering program comprehension in novice programmers-learning activities and learning trajectories. In *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 27–52.

[28] Ahmad Jbara and Dror G Feitelson. 2017. How programmers read regular code: a controlled experiment using eye tracking. *Empirical software engineering* 22, 3 (2017), 1440–1477.

[29] Marcel A Just and Patricia A Carpenter. 1980. A theory of reading: from eye fixations to comprehension. *Psychological review* 87, 4 (1980), 329.

[30] Philipp Kather, Rodrigo Duran, and Jan Vahrenhold. 2021. Through (Tracking) Their Eyes: Abstraction and Complexity in Program Comprehension. *ACM Trans. Comput. Educ.* 22, 2, Article 17 (Nov. 2021), 33 pages. https://doi.org/10.1145/3480171

[31] Bernhard Katzmarski and Rainer Koschke. 2012. Program complexity metrics and programmer opinions. In *2012 20th IEEE international conference on program comprehension (ICPC)*. IEEE, 17–26.

[32] Mik Kersten and Gail C. Murphy. 2006. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2006, Portland, Oregon, USA, November 5-11, 2006*, Michal Young and Premkumar T. Devanbu (Eds.). ACM, 1–11. https://doi.org/10.1145/1181775.1181777

[33] Katja Kevic, Braden Walters, Timothy Shaffer, Bonita Sharif, David C. Shepherd, and Thomas Fritz. 2017. Eye gaze and interaction contexts for change tasks - Observations and potential. *J. Syst. Softw.* 128 (2017), 252–266. https://doi.org/10.1016/j.jss.2016.03.030

[34] Amy J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Trans. Software Eng.* 32, 12 (2006), 971–987. https://doi.org/10.1109/TSE.2006.116

[35] Stanley Letovsky. 1987. Cognitive processes in program comprehension. *Journal of Systems and Software* 7, 4 (1987), 325–339. https://doi.org/10.1016/0164-1212(87)90032-X

[36] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. *Acm sigcse bulletin* 41, 3 (2009), 161–165.

[37] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research*. 101–112.

[38] Naser Al Madi, Cole S. Peterson, Bonita Sharif, and Jonathan I. Maletic. 2021. From Novice to Expert: Analysis of Token Level Effects in a Longitudinal Eye Tracking Study. In *29th IEEE/ACM International Conference on Program Comprehension, ICPC 2021, Madrid, Spain, May 20-21, 2021*. IEEE, 172–183. https://doi.org/10.1109/ICPC52881.2021.00025

[39] Niloofar Mansoor, Hamid Bagheri, Eunsuk Kang, and Bonita Sharif. 2023. An Empirical Study Assessing Software Modeling in Alloy. In *2023 IEEE/ACM 11th International Conference on Formal Methods in Software Engineering (FormaliSE)*. IEEE, 44–54.

[40] Niloofar Mansoor, Bonita Sharif, and Cole S Peterson. 2023. Assessing the Effect of Programming Language and Task On Eye Movements - Replication Package. https://osf.io/uh95r

[41] Lauren E. Margulieux, Briana B. Morrison, Baker Franke, and Harivololona Ramilison. 2020. Effect of Implementing Subgoals in Code.org's Intro to Programming Unit in Computer Science Principles. *ACM Trans. Comput. Educ.* 20, 4 (2020), 26:1–26:24. https://doi.org/10.1145/3415594

[42] Roberto Minelli, Andrea Mocci, and Michele Lanza. 2015. I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time. In *2015 IEEE 23rd International Conference on Program Comprehension*. 25–35. https://doi.org/10.1109/ICPC.2015.12

[43] Briana B. Morrison, Adrienne Decker, Lauren E. Margulieux, and Austin Cory Bart. 2022. Subgoals for CS1 in Python. In *ICER 2022: ACM Conference on International Computing Education Research, Lugano and Virtual Event Switzerland, August 7 - 11, 2022, Volume 2*, Jan Vahrenhold, Kathi Fisler, Matthias Hauswirth, and Diana Franklin (Eds.). ACM, 44–45. https://doi.org/10.1145/3501709.3544283

[44] Gail C. Murphy, Mik Kersten, and Leah Findlater. 2006. How Are Java Software Developers Using the Eclipse IDE? *IEEE Software* 23, 4 (July 2006), 76–83. https://doi.org/10.1109/MS.2006.105

[45] Greg L. Nelson and Amy J. Ko. 2018. On Use of Theory in Computing Education Research. In *Proceedings of the 2018 ACM Conference on International Computing Education Research, ICER 2018, Espoo, Finland, August 13-15, 2018*, Lauri Malmi, Ari Korhonen, Robert McCartney, and Andrew Petersen (Eds.). ACM, 31–39. https://doi.org/10.1145/3230977.3230992

[46] Unaizah Obaidellah, Mohammed Al Haek, and Peter C-H Cheng. 2018. A survey on the usage of eye-tracking in computer programming. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 5.

[47] Unaizah Obaidellah, Tanja Blascheck, Drew T Guarnera, and Jonathan Maletic. 2020. A fine-grained assessment on novice programmers' gaze patterns on pseudocode problems. In *ACM Symposium on Eye Tracking Research and Applications*. 1–5.

[48] Arnold Pears, Stephen Seidman, Lauri Malmi, Linda Mannila, Elizabeth Adams, Jens Bennedsen, Marie Devlin, and James Paterson. 2007. A survey of literature on the teaching of introductory programming. *Working group reports on ITiCSE on Innovation and technology in computer science education* (2007), 204–223.

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

[50] Nancy Pennington. 1987. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology* 19, 3 (1987), 295–341. https://doi.org/10.1016/0010-0285(87)90007-7

[51] Cole S. Peterson, Nahla J. Abid, Corey A. Bryant, Jonathan I. Maletic, and Bonita Sharif. 2019. Factors influencing dwell time during source code reading: a large-scale replication experiment. In *Proceedings of the 11th ACM Symposium on Eye Tracking Research & Applications, ETRA 2019, Denver , CO, USA, June 25-28, 2019*, Krzysztof Krejtz and Bonita Sharif (Eds.). ACM, 38:1–38:4. https://doi.org/10.1145/3314111.3319833

[52] Cole S. Peterson, Jonathan A. Saddler, Tanja Blascheck, and Bonita Sharif. 2019. Visually Analyzing Students' Gaze on C++ Code Snippets. In *Proceedings of the 6th International Workshop on Eye Movements in Programming (EMIP '19)*. IEEE Press, Piscataway, NJ, USA, 18–25. https://doi.org/10.1109/EMIP.2019.00011

[53] Cole S Peterson, Jonathan A Saddler, Natalie M Halavick, and Bonita Sharif. 2019. A Gaze-Based Exploratory Study on the Information Seeking Behavior of Developers on Stack Overflow. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems*. ACM, LBW2510.

[54] Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 153–160.

[55] Robert S. Rist. 1986. Plans in Programming: Definition, Demonstration, and Development. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*. Ablex Publishing Corp., USA, 28–47.

[56] Jonathan A Saddler, Cole S Peterson, Patrick Peachock, and Bonita Sharif. 2019. Reading Behavior and Comprehension of C++ Source Code-A Classroom Study. In *International Conference on Human-Computer Interaction*. Springer, 597–616.

[57] Jonathan A Saddler, Cole S Peterson, Sanjana Sama, Shruthi Nagaraj, Olga Baysal, Latifa Guerrouj, and Bonita Sharif. 2020. Studying developer reading behavior on stack overflow during api summarization tasks. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 195–205.

[58] Carsten Schulte. 2008. Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching. In *Proceedings of the Fourth international Workshop on Computing Education Research*. ACM, 149–160.

[59] Carsten Schulte, Tony Clear, Ahmad Taherkhani, Teresa Busjahn, and James H Paterson. 2010. An introduction to program comprehension for computer science educators. In *Proceedings of the 2010 ITiCSE working group reports*. ACM, 65–86.

[60] Zohreh Sharafi, Timothy Shaffer, Bonita Sharif, and Yann-Gaël Guéhéneuc. 2015. Eye-tracking metrics in software engineering. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 96–103.

[61] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha E. Crosby. 2020. A practical guide on conducting eye tracking studies in software engineering. *Empir. Softw. Eng.* 25, 5 (2020), 3128–3174. https://doi.org/10.1007/s10664-020-09829-4

[62] Zohreh Sharafi, Zéphyrin Soh, and Yann-Gaël Guéhéneuc. 2015. A systematic literature review on the usage of eye-tracking in software engineering. *Information and Software Technology* 67 (2015), 79–107.

[63] Bonita Sharif, Michael Falcone, and Jonathan I Maletic. 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*. 381–384.

[64] Bonita Sharif and Jonathan I Maletic. 2010. An eye tracking study on camelcase and under_score identifier styles. In *2010 IEEE 18th International Conference on Program Comprehension*. IEEE, 196–205.

[65] Bonita Sharif and Timothy Shaffer. 2015. The Use of Eye Tracking in Software Development. In *Foundations of Augmented Cognition - 9th International Conference, AC 2015, Held as Part of HCI International 2015, Los Angeles, CA, USA, August 2-7, 2015, Proceedings (Lecture Notes in Computer Science)*, Dylan Schmorrow and Cali M. Fidopiastis (Eds.), Vol. 9183. Springer, 807–816. https://doi.org/10.1007/978-3-319-20816-9_77

[66] Ben Shneiderman and Richard E. Mayer. 1979. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *Int. J. Parallel Program.* 8, 3 (1979), 219–238. https://doi.org/10.1007/BF00977789

[67] Nischal Shrestha, Colton Botta, Titus Barik, and Chris Parnin. 2020. Here we go again: why is it difficult for developers to learn another programming language?. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 691–701. https://doi.org/10.1145/3377811.3380352

[68] Janet Siegmund, Christian Kästner, Jörg Liebig, Sven Apel, and Stefan Hanenberg. 2014. Measuring and modeling programming experience. *Empirical Software Engineering* 19, 5 (2014), 1299–1334.

[69] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *Software Engineering, IEEE Transactions on* SE-10 (10 1984), 595 – 609. https://doi.org/10.1109/TSE.1984.5010283

[70] Elliot Soloway and Kate Ehrlich. 1984. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* SE-10, 5 (1984), 595–609. https://doi.org/10.1109/TSE.1984.5010283

[71] Andreas Stefik and Susanna Siebert. 2013. An Empirical Investigation into Programming Language Syntax. *ACM Transactions on Computing Education* 13, 4, Article 19 (nov 2013), 40 pages. https://doi.org/10.1145/2534973

[72] Margaret-Anne D. Storey. 2006. Theories, tools and research methods in program comprehension: past, present and future. *Softw. Qual. J.* 14, 3 (2006), 187–208. https://doi.org/10.1007/s11219-006-9216-4

[73] Donna Teague and Raymond Lister. 2014. Manifestations of Preoperational Reasoning on Similar Programming Tasks. In *Sixteenth Australasian Computing Education Conference, ACE 2014, Auckland, New Zealand, January 2014 (CRPIT)*, Jacqueline L. Whalley and Daryl J. D'Souza (Eds.), Vol. 148. Australian Computer Society, 65–74. http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV148Teague.html

[74] Ethel Tshukudu and Quintin Cutts. 2020. Understanding conceptual transfer for students learning new programming languages. In *Proceedings of the 2020 ACM conference on international computing education research*. 227–237.

[75] Rachel Turner, Michael Falcone, Bonita Sharif, and Alina Lazar. 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*. ACM, 231–234.

[76] Phillip Merlin Uesbeck, Cole S. Peterson, Bonita Sharif, and Andreas Stefik. 2020. A randomized controlled trial on the effects of embedded computer language switching. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 410–420. https://doi.org/10.1145/3368089.3409701

[77] Hidetake Uwano, Masahide Nakamura, Akito Monden, and Ken-ichi Matsumoto. 2006. Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. In *Proceedings of the 2006 Symposium on Eye Tracking Research and Applications (ETRA '06)*. Association for Computing Machinery, New York, NY, USA, 133–140. https://doi.org/10.1145/1117309.1117357

[78] Anne Venables, Grace Tan, and Raymond Lister. 2009. A Closer Look at Tracing, Explaining and Code Writing Skills in the Novice Programmer. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop (ICER '09)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1584322.1584336

[79] A. Von Mayrhauser and A.M. Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55. https://doi.org/10.1109/2.402076

[80] Susan Wiedenbeck. 1986. Beacons in Computer Program Comprehension. *Int. J. Man Mach. Stud.* 25, 6 (1986), 697–709. https://doi.org/10.1016/S0020-7373(86)80083-9

[81] Chia-Kai Yang and Chat Wacharamanotham. 2018. Alpscarf: Augmenting scarf plots for exploring temporal gaze patterns. In *Extended abstracts of the 2018 CHI conference on human factors in computing systems*. 1–6.

[82] Alfred L Yarbus. 1967. *Eye Movements During Perception of Complex Objects*. Springer US, Boston, MA, 171–211. https://doi.org/10.1007/978-1-4899-5379-7_8

[83] Sheng Yu and Shijie Zhou. 2010. A survey on metric of software complexity. In *2010 2nd IEEE International conference on information management and engineering*. IEEE, 352–356.

[84] Vlas Zyrianov, Drew T. Guarnera, Cole S. Peterson, Bonita Sharif, and Jonathan I. Maletic. 2020. Automated Recording and Semantics-Aware Replaying of High-Speed Eye Tracking and Interaction Data to Support Cognitive Studies of Software Engineering Tasks. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*. IEEE, 464–475. https://doi.org/10.1109/ICSME46990.2020.00051