A Tale of Two Comprehensions? Analyzing Student Programmer Attention during Code Summarization

ZACHARY KARAS, Computer Science, Vanderbilt University, Nashville, United States

AAKASH BANSAL, Computer Science and Engineering, University of Notre Dame, South Bend, United States

YIFAN ZHANG, Computer Science, Vanderbilt University, Nashville, United States

TOBY LI, Department of Computer Science and Engineering, University of Notre Dame, South Bend, United States

COLLIN MCMILLAN, Department of Computer Science and Engineering, University of Notre Dame, South Bend, United States

YU HUANG, Computer Science, Vanderbilt University, Nashville, United States

Code summarization is the task of creating short, natural language descriptions of source code. It is an important part of code comprehension and a powerful method of documentation. Previous work has made progress in identifying where programmers focus in code as they write their own summaries (i.e., Writing). However, there is currently a gap in studying programmers' attention as they read code with pre-written summaries (i.e., Reading). As a result, it is currently unknown how these two forms of code comprehension compare: Reading and Writing. Also, there is a limited understanding of programmer attention with respect to program semantics. We address these shortcomings with a human eye-tracking study (n = 27) comparing Reading and Writing. We examined programmers' attention with respect to fine-grained program semantics, including their attention sequences (i.e., scan paths). We find distinctions in programmer attention across the comprehension tasks, similarities in reading patterns between them, and differences mediated by demographic factors. This can help guide code comprehension in both computer science education and automated code summarization. Furthermore, we mapped programmers' gaze data onto the Abstract Syntax Tree to explore another representation of human attention. We find that visual behavior on this structure is not always consistent with that on source code.

CCS Concepts: • **Social and professional topics** → *Professional topics*; *Computing education*; *Computational science and engineering education;*

Additional Key Words and Phrases: Cognitive science, code summarization, eye-tracking, expertise, code comprehension

Authors' Contact Information: Zachary Karas, Computer Science, Vanderbilt University, Nashville, Tennessee, United States; e-mail: z.karas@vanderbilt.edu; Aakash Bansal, Computer Science and Engineering, University of Notre Dame, South Bend, Indiana, United States; e-mail: abansal1@nd.edu; Yifan Zhang, Computer Science, Vanderbilt University, Nashville, Tennessee, United States; e-mail: yifan.zhang.2@vanderbilt.edu; Toby Li, Department of Computer Science and Engineering, University of Notre Dame, South Bend, Indiana, United States; e-mail: toby.j.li@nd.edu; Collin McMillan, Department of Computer Science and Engineering, University of Notre Dame, South Bend, Indiana, United States; e-mail: cmc@nd.edu; Yu Huang, Computer Science, Vanderbilt University, Nashville, Tennessee, United States; e-mail: yu.huang@vanderbilt.edu.

1 INTRODUCTION

Documentation is critical to software maintenance and to software engineering at large [41, 61, 81]. Reading source code on its own is time-consuming, thus, software developers rely on natural language descriptions of code to both convey and understand its meaning [72]. Code summaries are one such example of natural language descriptions, in which the meaning of code is distilled into a short phrase [89]. For instance, 'removes an entry from the database' can help a developer grasp the purpose of a code snippet without reading each detail. From another perspective, writing a concise summary such as this demonstrates an insightful understanding of the code.

Accordingly, researchers have studied how humans interpret code and condense it into a summary, and have used *eye-tracking* to record programmers' gaze as they write summaries [5, 70]. Those studies have shed light on where programmers look in code as they *write* summaries, yet software developers typically read code with accompanying documentation [7, 35, 61]. Informally, if we consider summary writing to be an active, generative process, is this form of comprehension different from that when a summary is present? Research into code comprehension has been ongoing for over 40 years and has developed several models for how programmers understand source code, such as bottom-up and top-down comprehension [88]. However, there is limited research into whether programmers' *purpose* for reading source code has any influence on their comprehension strategies (i.e., reading code to generate documentation or reading code with the help of pre-written documentation). This has resulted in educational practices that may be imprecise around reading code in different contexts [20] and automated techniques trained on data from one type of comprehension for the role of another [70]. Uncovering nuances in code comprehension related to programmers' purpose will not only advance our foundational understanding [88] but can also provide guidance to educators teaching students to derive meaning from code [13] and can help advise programmers on how to write code summaries for those who will later read them [36, 43, 66]. This information can also help in developing methods for automated code summarization to better tailor their output for developers reading code with a summary [70, 89].

Previous studies in code summarization have attempted to investigate human attention patterns on the semantic level, but they have only considered four categories: the method declaration, the method body, control flow elements, and method calls [5, 70]. Those studies on student programmers and professional developers present slightly different interpretations from one another. However, by considering only these four categories, the finer details of programmer attention that reveal more nuance may have been overlooked. For instance, during code summarization tasks, it is currently unknown how programmers attend to variables, arguments, parameters, literals, or other semantic categories (see Table 2). Furthermore, those studies did not analyze semantics within the sequences of programmer attention (i.e., scan path) [69], which can be informative of deeper cognitive processes during code comprehension [19, 42].

In this article, we present results from a human study using eye-tracking from 27 undergraduate and graduate **computer science (CS)** students, totaling 35.68 hours of eye-tracking data for 1,657 Java summarization tasks and 6,848,501 eye-tracking data points. Participants read real-world Java methods under two conditions: **Reading** — participants were given pre-written summaries and asked to evaluate their quality using validated criteria [39] and **Writing** — participants generated their own code summaries. To understand programmer attention patterns in sharper detail, we

categorized each "word", or **token**, in the Java methods as one of 19 semantic categories [71]. We analyzed semantics in the ordered sequence (i.e., scan path) of what programmers read in the code for a deeper look into programmer cognition during code comprehension tasks. Finally, whereas current work in code comprehension has examined programmer attention in raw code [20, 63], we present, to the best of our knowledge, the first analysis of human attention on the **Abstract Syntax Tree (AST)**, which is an underlying structural representation of code. We hope this approach can inspire future work, especially considering artificial intelligence (AI) models that seek to automate code summarization using the AST [46, 51, 83]. In our analyses, we find that student programmers' focus can be mediated by expertise, other demographic factors, and the comprehension task (Reading or Writing). We also find that some broader reading patterns are stable between the two tasks, as well as some intriguing patterns from the AST mappings. These results have implications for our understanding of code comprehension and for CS Education, tool design, and automated code summarization methods.

Specifically, we found the following: (1) in the Writing condition, programmers focus more on *parameters*, *variable declarations*, and *method calls*; (2) programmers look more at the code when they are reading it to formulate their own summaries; (3) cognitive load increases as a function of code complexity; (4) novices consistently look more at variable declarations and conditional statements when summaries are provided but not when programmers write their own summaries; (5) regardless of the comprehension task, programmers consistently look between the same semantic categories (method declarations ⇄ variable declarations, loop bodies ⇄ conditional statements, method declarations ⇄ conditional statements); (6) programmers' attention patterns are significantly different on the raw code between Reading and Writing but not on the AST; (7) experts' and novices' attention patterns are significantly different on the raw code for Reading but not on the AST. For Writing, their patterns are significantly different on the raw code and even more so on the AST.

Our contributions are as follows.

— A controlled human study with 27 participants performing code summarization tasks: one in which they are given a summary, and one in which they generate their own summary.
— A semantic-level comparison and traditional eye-tracking comparison of programmer attention when they are given a pre-written summary and when they write their own.
— An analysis of the semantic categories that CS students commonly look between during code summarization tasks.
— A novel analysis mapping human attention to the AST and contrasting to human attention on raw code.
— A detailed comparison between experts and novices, including semantic categories, reading sequences, and AST mappings.
— Publicly available data and code **here**.

2   BACKGROUND AND RELATED WORK

In this section, we discuss prior literature on **eye-tracking** in software engineering and **code summarization**.

2.1   Eye-Tracking

Eye-tracking is a non-invasive technology that records visual attention and cognitive load [75]. The technology has its roots in the 1800s and has been used to study gaze patterns in marketing research [87], natural language reading [67], and even fields such as aviation [64]. Eye-tracking is particularly useful for software engineering, in which researchers can closely monitor

programmers' code reading patterns and behavior in realistic work conditions [40], especially with the development of such tools as iTrace [73]. Researchers have used other cognitive measures, such as neuroimaging, to gain insights into the cognitive processes of coding. For instance, researchers have compared patterns of brain activity and connectivity between coding and other cognitive skills, such as mental rotation and prose writing [45, 49, 50]. Those methods are expensive and are different from eye-tracking in that they have a lower temporal resolution and require researchers to make inferences about programmers' internal state. Eye-tracking, by contrast, measures humans' external visual behaviors at a high temporal resolution.

For eye-tracking, researchers typically rely on *fixation* data extracted from the raw data to measure cognition. A *fixation* is defined as a spatially stable eye gaze that lasts for 100 to 300 ms [76]. Most processing of visual information happens during fixations [48] and begins at the start of the fixation according to the immediacy assumption [48]. By calculating the amount of fixations, or the fixation *count*, and the time span of these fixations (i.e., fixation *duration*), researchers can roughly measure humans' cognitive effort and information processing [76]. Humans also make rapid eye movements or large jumps in their visual field, called *saccades*, which typically last for 40 to 50 ms. There is little cognitive processing that occurs during saccades [67]; thus, researchers use fixations to investigate cognitive load and visual attention patterns [75]. Humans will occasionally fixate on an area they have previously seen. These *regressions* occur when participants review prior information and also indicate higher cognitive effort [75].

There are also more complex eye-tracking metrics, such as the *scan path*, that offer insight into deeper cognitive patterns in humans. A scan path is simply an ordered sequences of fixations, but it reveals the order in which humans process information [76]. By nature, the scan path as an ordered sequence is suitable for interdisciplinary analyses. For instance, researchers have used an algorithm for comparing DNA sequences to study the similarities between scan paths [19, 26, 54]. In Software Engineering, researchers have applied depth-first search to scan paths from requirements comprehension [74] and have used edit distances to measure the similarity between programmer scan paths (i.e., reading strategies) [28]. In this article, we treat participants' scan paths like documents in the context of **Natural Language Processing (NLP)**, where there are contingencies between consecutive words. To study programmers' strategies for code summarization, we analyzed patterns of consecutive semantic categories (i.e., N-Grams) in participants' scan paths. In addition to scan paths, we analyzed fixation counts and durations as well as regressions to compare both facets of code summarization: reading code with a pre-written summary and reading code to generate one.

2.2 Code Summarization and Comprehension

Code summarization is a complex cognitive task in which programmers must synthesize distant pieces of code into a cohesive summary [70]. Previous research has studied humans as they write code summaries [4, 5, 69, 70], but the automation of this process is also an active area of research [89]. Automating this process is challenging because of the numerous (and fascinating) deviations of source code from natural language [70].

Researchers initially attempted to treat source code as natural language and applied text summarization techniques to code [38, 57]. However, there is not a one-to-one mapping between "words" in code and words in natural language. Researchers revised their approaches and attempted to find the key words or phrases within code that summaries should include [70, 82]. Rodeghero et al. used eye-tracking in this context to measure where 10 human developers looked in the code as they wrote summaries [70]. In studying where programmers focus, those researchers primarily considered *method declarations*, the *method body*, *control flow elements*, and *method calls*. They found

that programmers look more closely at the method header than the method body but do not focus more on method calls or control flow elements. Abid et al. conducted a similar eye-tracking study to Rodeghero et al., in which 18 student developers summarized Java methods in an integrated development environment (IDE) [5]. These researchers examined programmer attention on these same four categories, finding that programmers look more at the method body than the method declaration, focusing their attention on method calls and control flow elements. Based on these slightly different interpretations, it is unclear whether programmers attend more to the method declaration or the method body. In this study, we compared human attention between two different code summarization tasks and considered 19 semantic categories in an attempt to uncover more nuance.

To improve automated source code summarization, Rodeghero et al. designed their study to investigate where programmers focus as they summarize code and then incorporated this information into an automated model [68, 70]. We hope to similarly inform today's state-of-the-art methods for automated code summarization. Recently, Deep Learning techniques have proliferated to automate code summarization, with top-performing models using Transformers [8, 86]. These models typically perform well by incorporating structures of the code, such as call graphs [11] or ASTs [46, 51], into the training process. Thus, in this study, to further explore programmers' attention patterns and inspire future AI design for code summarization, we conducted an exploratory analysis by mapping the scan path onto the AST, and compared this with traditional metrics on the raw code (Section 5.2).

3 STUDY DESIGN

To investigate student programmers' attention patterns during both facets of code summarization, we designed our experiment to include two conditions: *Reading* and *Writing*.

Reading: Participants were given code with pre-written summaries, and asked to evaluate the summary quality.

Writing: Participants read Java methods and generated their own summaries.

Participants completed both conditions, all while their visual behavior was recorded using eye-tracking. In the remainder of this section, we discuss **participant recruitment**, the **study materials** used in the task (i.e., Java methods, eye-tracking), the **task design**, and the **experimental protocol**.

3.1 Participant Recruitment

Participants were recruited from Vanderbilt University and the University of Notre Dame to take part in the experiment. The two R1 universities are comparable in terms of CS curricula, and both teams obtained institutional review board approval at their respective institutions. The same recruitment procedure was followed in both locations, where students were invited to participate via in-class presentations, flyers, class forum posts, and mailing-list advertisements. To be included in the experiment, programmers needed to be at least 18 years old, have taken Data Structures or the equivalent, have at least 1 year of experience with Java, and no history of epileptic seizures [2]. In total, we recruited 29 undergraduate and graduate CS students across Vanderbilt University ($n = 19$) and the University of Notre Dame ($n = 10$). All participants were compensated $60 at both institutions. Due to protocol error in one case and a software malfunction in another, data from two participants was excluded, leaving 27 participants' data in the final dataset. Of the 27 participants, the average age was 23.81, 8 were women, 15 were graduate students, and 14 spoke English as a first language. Select demographic information for these 27 participants included in the final dataset can be found in Table 1.

Table 1. Demographic Information of the Participants in
Our Final Data Sample

Demographic	Number of Participants	
Gender	Men	19
	Women	8
Expertise	<= 4 years	10
	5 – 6 years	8
	>= 7 years	9
Program	Undergraduate	12
	Graduate	15

During data analysis, participants were separated into three
groups based on their programming experience in years.

3.2 Study Materials

In this section, we discuss specifics related to the origin of the Java methods as well as the eye-tracking hardware, software, and setup.

Java Methods The Java methods and associated summaries used in this study originate from the publicly available FunCom dataset of 2.1 million Java methods collected from open-source projects [11, 52]. This dataset has been used, filtered, and refined in previous research [10, 39]; we continue this lineage using a sample of 205 methods employed in prior human studies [12, 39]. For this study, we indented and formatted these 205 methods according to Java coding conventions [58]. To fit the screen constraints, we omitted methods that either exceeded 26 lines of code or contained lines of code that wrapped onto the next line. The final dataset after cleaning based on these characteristics consisted of 162 Java methods. Before filtering, the average method length in the dataset was 12.37 lines of code (σ = 4.72), with an average line length of 27.38 characters (σ = 27.25). In terms of cyclomatic complexity, methods in this pre-filtered dataset had an average complexity of 2.53 (σ = 1.59), where each count is the number of linearly independent paths through a method.

After filtering, the average method length in the dataset was 11.72 lines of code (σ = 4.26), with an average line length of 26.52 characters(σ = 26.31). The average method complexity in our final sample was 2.59, with a standard deviation of 1.56. The shortest method was 5 lines of code, whereas the longest was 26 lines of code. The simplest methods had a complexity of 1, and the most complex method in our sample had a complexity of 11. Method summaries ranged from 3 to 13 words long (i.e., "refresh tree panel"), with an average of 8.29 and a standard deviation of 2.78 words. These methods were randomly assigned to either the Reading condition or Writing condition, ensuring that each method could only be used for one condition (i.e., methods used in the Reading condition were not reused in the Writing condition). With data collection for eye-tracking in mind, we ensured that the selection of Java methods follows the best practices in Software Engineering for participant fatigue as well as constraints for hardware and software [76].

Eye-tracking Eye-tracking data was recorded using a Tobii Pro Fusion eye-tracker mounted on a 24" computer monitor (1920 × 1080 resolution) with a refresh rate of 60 Hz [2]. The eye-tracking model is accurate down to 0.1 to 0.2 in on the screen (0.26–0.53 cm). We developed a task interface to run locally using Python Flask that presented Java methods and recorded participant input. An example of this can be seen for both conditions in Figure 1. To record eye-tracking data, we integrated the Tobii-Pro Software Developer Kit into the task [1]. The Java methods were presented at font size 14, without syntax highlighting. To improve data quality for eye-tracking, participants were asked to wear contact lenses instead of glasses when possible. We ensured that the materials

(a) Example Reading Stimulus

(b) Example Writing Stimulus

Fig. 1. Example stimuli used in the task. In both conditions, the code was displayed on the left, and the summaries, pre-written or participant generated, were located in the top right. In the Reading condition, Likert-scale questions for assessing summary quality were presented on the right below the pre-written summary.

and methodology were consistent across both institutions and followed a script when interacting with participants.

3.3 Task Design

We used a within-subjects experimental design: each participant completed both the Reading and the Writing conditions, but whether a participant started in Reading or Writing was randomized. The entire pool of 162 Java methods was randomly split between Reading and Writing so that participants would see a given method in only one context. For each experimental session (i.e., for each participant), 65 Java methods of the 162 were randomly selected and presented. Methods were presented as *stimuli*; each *stimulus* consisted of one method and a summarization task specific to that condition (i.e., writing a summary, rating a pre-written summary). Of these 65 stimuli, 40 were presented in Reading and 25 in Writing.

To maximize both the variety and amount of eye-tracking data collected with respect to our stimuli, we purposefully ensured that 60% of the stimuli were seen among all participants, whereas the other 40% was taken from the larger pool (reserved for that condition). Before we began collecting

eye-tracking data, we randomized which stimuli comprised the 60% seen by all participants and which made up the larger pool from which the remaining 40% was sampled for each experimental session. During the experimental sessions, the order in which the stimuli were presented was also randomized. In total, 89 Java methods were covered in the Reading condition, of which 24 were seen by everyone. In the Writing condition, 67 Java methods were covered, with 15 of those being seen by every participant. Thus, 156 of the 162 methods were covered during data collection.

Three breaks were built into the task, both for participants to rest, and for the researchers to recalibrate the eye-tracker (for data quality). There was no time limit for breaks. Participants were notified of breaks via "rest" slides built into the interface. These were placed in the following locations: one halfway through the Reading condition, one in between the two conditions, and one halfway through the Writing condition. For example, if participants started with the Writing condition, they would first complete 13 stimuli, take a break, then finish the remaining 12. Before starting the Reading condition, they would take a second break. Now in Reading condition, participants would finish 20 stimuli, then take their third break. They would then finish the remaining 20 stimuli of the Reading condition.

In the Reading condition, participants were shown Java methods on the left side of the screen and the corresponding summary in the upper-right. Likert-scale questions were located below the pre-written summary. For Writing, a text box for participants' summaries was located in the upper right of the screen. Example stimuli can be seen in Figure 1. Pre-written summaries in the Reading condition were either human written and from the original dataset of Java methods from open-source projects [52] or generated via Deep Learning [11, 39]. We discuss quality control for these summaries below. To ensure that this summarization task was treated as a code comprehension task, participants were given four Likert-scale questions for each stimulus requiring them to closely read both the summary and the code. Questions 1 to 3 were previously validated [39], whereas the fourth was added for the purposes of this current study. The questions were on a scale of 1 to 5, ranging from Strongly Disagree to Strongly Agree, and based on the following criteria:

(1) Summary accuracy
(2) Whether the summary is missing information
(3) Whether the summary contains unnecessary information
(4) Summary readability

Quality Control The quality of pre-written summaries could potentially influence programmer attention on the code. Thus, we removed data associated with egregiously low-quality summaries. The summaries were previously validated as well [11], but we implemented further checks to bolster the data quality for the current study. Specifically, using a 1 to 5 scale, we excluded data from summaries that had an average score of 4 or greater for questions (2) and (3) above, and an average score of 2 or below on questions (1) and (4). In other words, we removed data associated with pre-written summaries if, on average, participants *agreed* it was missing information and contained unnecessary information and *disagreed* that it was accurate and readable. In total, data associated with 4 pre-written summaries was removed.

Likewise, participant-written summaries that do not match the code indicate poor comprehension. Here, we assume that participants formed mental models of the code based on information they saw. If participants' understanding of a method was malformed, this may be reflected in their eye-tracking data as well. While we wanted a variety of comprehension types and skill levels within the dataset, we also sought to ensure that the eye-tracking data represents a base level of comprehension. For example, we excluded eye-tracking data from a summary that included "to be honest, I am not entirely sure what this function is doing."Therefore, two of the authors of this article rated and subsequently filtered participant summaries using the same Likert-scale questions

mentioned above [25]. The two researchers (8 years of Java experience, 5 years of Java experience) first rated participants' summaries independently. The researchers then resolved any rating conflicts together (i.e., a valence mismatch: Agree/Disagree, Strongly Disagree/Agree, Neutral/Agree), obtaining an **Inter-Rater Reliability (IRR)** of 1 [53]. Using these ratings, eye-tracking data associated with 5 participant summaries was excluded. Informally, if we consider one participant's eye-tracking data for one stimulus as a single data point, the final dataset contained 996 samples for the Reading condition and 661 samples for the Writing condition. We discuss how this eye-tracking data was analyzed to compare both forms of code comprehension in Section 4.

Statistical Power Using the effect sizes of results in previous research as a guideline [78], we evaluated the statistical power of data collected in this study. Sharif et al. conducted a study with 15 participants, comprised of undergraduate and graduate students as well as two faculty members. That study used a within-subjects study design, and reported small to moderate Cohen's d effect sizes, with a minimum of 0.15, a maximum of 0.57, and an average effect size of 0.27. In this study, we used a within-subjects design for comparing between Reading and Writing, and a between-subjects design for comparing based on different demographics. Using G-Power, we calculated that we would need 150 total data points in comparing Reading and Writing to obtain sufficient statistical power to detect the average effect size of 0.27 with an alpha of 0.05 [32]. In other words, we would need at least 75 samples in both conditions. As previously mentioned, we obtained 996 samples for the Reading condition and 661 samples in the Writing condition.

For analyzing differences between groups based on demographic factors, we would need between 68 samples (d = 0.15) and 963 samples (d = 0.57) in both groups, or 298 samples in both for an effect size of 0.027. In this study, we compared participants based on years of experience, gender, and native language. For expertise, we did not include all participants' data in our analyses, and instead split the participants into three groups, only comparing between participants with the lowest amount of experience with those with the highest. Based on our sample of students, we considered participants to be novices if they had 4 years of experience or less (n = 10), and experts if they had more than 7 years of experience (n = 9). We excluded data from the middle group in our comparison to yield a starker contrast between experts and novices. For the Writing condition, we obtained 215 and 242 samples from experts and novices, respectively. In the Reading condition, we collected 336 and 370 samples from experts and novices, respectively.

Our sample is sufficiently powered for comparing based on gender and native language but is not ideal due to other characteristics of the dataset. For gender, only one woman in our sample is in the expert group and, with respect to native language, only two native English speakers are in the expert group. To ensure that the samples were not biased towards experts, we excluded *all* experts in our analyses, comparing student programmers based on gender and native language. We collected 270 samples from women (n = 7) in the Reading condition and 174 samples in the Writing condition. From men (n = 11), we collected 390 samples in the Reading condition and 272 samples in the Writing condition. Next, we compared native English speakers (n = 12) and non-native English speakers (n = 6). From the former, we collected 460 samples for Reading and 295 for Writing. From the latter, we collected 200 samples for Reading and 151 for Writing. As an additional consideration, the remaining non-native English speakers represent 5 other languages, which may introduce additional variables in the comparison. We nonetheless analyzed these factors to explore potential differences and understand their influence on the larger dataset. We present *preliminary* analyses based on gender and native language in Section 5.4.

3.4 Experimental Protocol

Programmers were recruited to take part in the experiment via in-class presentations, flyers, forum posts, and mailing list advertisements. Individuals who contacted the researchers completed the

consenting and prescreening processes electronically (the experimental session was in person). After programmers gave their consent, they completed a prescreening questionnaire to ensure that they were eligible for the study. Individuals were eligible for the study if they were at least 18 years old, had taken Data Structures or the equivalent, had at least 1 year of Java experience, and no history of epileptic seizures [2]. In addition, we gave programmers a prescreening question to test their basic Java understanding, following previous work [11]. We asked them to briefly describe the purpose of an obfuscated Java method (i.e., in-order tree traversal). All participants included in the current study met our eligibility criteria and wrote accurate descriptions.

Participants completed the summarization tasks in person, in an office with natural lighting. At the beginning of each experimental session, participants completed a pre-task survey containing questions related to age, gender, native language, and classes taken. The pre-task survey was limited in scope to reduce any priming effects [56]. The researcher would then give participants scripted instructions for the experiment and calibrate the eye-tracker using the Tobii Pro Eye Tracker Manager [2]. The eye-tracker itself may have introduced observer bias, as participants might change their behavior knowing their gaze was being recorded [84]. While a certain amount of experimental bias is unavoidable [21], we as researchers attempted to minimize observer bias by leaving the room while participants completed the task. Participants were instructed to alert the researcher once they reached the breaks (Section 3.3). After each break, the researcher recalibrated the eye-tracker.

After finishing the experimental session, participants completed a post-task survey. The post-task survey asked participants about their preferred coding languages, coding experience, and personal criteria for high- and low-quality summaries. Experimental sessions lasted about 90 minutes. The summarization tasks alone (i.e., Reading, Writing, breaks) lasted roughly 75 minutes.

4   DATA ANALYSIS

In this section, we discuss the methodology used for data preprocessing and analysis. Our overarching goal through this process was to extract the semantics of what programmers read during code summarization tasks. To accomplish this, we broke down the problem into the following steps:

(1) Map participants' eye-tracking data to the Java code on the monitor. We achieved this via **bounding boxes** around each token in the methods.
(2) Assign all tokens in the Java methods to **semantic categories**. We developed a strategy for this based on prior literature and the semantics of Java, and used **ASTs**.
(3) Calculate **eye-tracking metrics** with respect to these semantic categories and the ASTs.

Bounding Boxes Raw eye-tracking data consists of x and y coordinates on the screen as well as timestamps for these gaze points. However, this does not indicate *what* participants actually read at a given moment. For our purposes, we sought to map these x and y gaze coordinates to tokens in the Java methods. We calculated the pixel-coordinate boundaries, or *bounding boxes*, of each token on the screen [37]. To generate these bounding boxes, we first captured screenshots of each of the Java methods, then used the opencv-python library to compute the contours and coordinates of the tokens within the images. Because these were *images*, information about the actual code within the bounding boxes was lost at this stage. To recover information about the code, we used the easyocr library to identify the characters within each bounding box. The output from optical character recognition did not always align with the original code. Thus, we used the difflib library to match predicted strings with the closest token in the original code. Finally, the researchers did a manual pass to ensure that strings associated with bounding boxes matched tokens in the original code.

Fig. 2. For a Java method in our dataset, assigning tokens to semantic categories based on their context in the Abstract Syntax Tree. All semantic categories can be found in Table 2. Punctuation was not analyzed.

In many cases, the same tokens appear multiple times in the same method. For instance, consider the line of code 'int n = n + 1'. Here, we need a means of differentiating the first occurrence of 'n' from the second, both to specify which token a programmer was examining and to assign the appropriate semantic categories. In these cases, we notated repeated tokens with incremented numbers. Based on our implementation, the tokens in the above example would be notated as `int.0`, **`n.0`**, `=.0`, **`n.1`**, `+.0`, `1.0`. After this process with the bounding boxes was complete, we could localize participants' gaze coordinates to tokens on the screen, but we still lacked semantic information for the code they read.

Semantic Categories and Abstract Syntax Trees Using bounding boxes, we could determine that programmers read Token A more than Token B. However, informally, is Token A a variable? Is Token B involved in exception handling? To draw broader conclusions about where programmers focus during code summarization tasks, we still needed generalized semantic information for tokens in the Java methods (Section 3.2). To obtain this information, we *abstracted* tokens according to their AST context and assigned them to **semantic categories**. Specifically, we used srcML to parse Java methods into ASTs [22], then recursively walked through the trees to derive tokens' structural context. For example, each of the four tokens in the line `String s = 'hello world'` would be labeled 'variable declaration' because these are children of 'declaration statement' and 'initialization' nodes according to srcML's parsing. Initially, each token could be classified as one of 25 categories, which are listed in Table 2 (we discuss how this was reduced to 19 below and further filtered in Section 5.1). We formulated these categories by referring to Chapters 1 to 6 of a standard Java textbook [71] and show an example of these semantic categories in Figure 2.

In some cases, multiple labels might apply to a single token. For example, consider the first line of code from Figure 2:

`public void genSql() throws PositionedError`. In this line, 'void' is part of the method declaration, but it is also the return type. Similarly, 'genSql' is in the method declaration, but it is also the method *name*. To resolve these label conflicts, we created an order of *precedence*, as shown in Table 2. In these conflicts, a token would be given the abstract label with the higher precedence (i.e., method declaration for `void, genSql` based on our criteria). Informally, we emphasize that this list is not absolute and may be refined in future work. It was designed for our purposes in the current study based on the semantics of Java [71].

This order of precedence was decided based on prior literature [5, 14, 70] to achieve high granularity while preserving meaningful semantics. For example, the pair of tokens 'ref.getFile()' in Line 4 of Figure 2 consists of an externally defined variable followed by a method call, but it is also an argument. We contend that 'argument' is a more meaningful semantic category for this pair of

Table 2. List of Categories Considered for Each Token in the Code, with a
Count of their Occurrences within Our Dataset

Precedence	Category	Amount in Dataset
1	Comment	78
2	Method Declaration	465
3	Parameter	349
4	Return	362
5	Conditional Statement	807
6	Exception Handling	197
7	Loop Body	650
8	Conditional Body	461
9	Variable Declaration	937
10	Argument	601
11	Variable Name	126
12	Method Call	277
13	External Class	59
14	External Variable/Method	216
15	Assignment	74
16	Operation	16
17	Literal	9
18	Operator	16
19	Index Operation	1
NA	'This' Keyword	0
NA	Assignment	0
NA	Type	0
NA	Return Type	0
NA	Method Name	0
NA	Constructor Call	0

Multiple labels may apply to one token. For these conflicts, the token was given
the label with higher *precedence* (i.e., higher priority based on previous research).
We originally started with 25 categories, but only 19 remained after labeling
conflicts were settled. For example, 'Return Type' and 'Method Name' were always
superseded by 'Method Declaration.'

tokens considering their context within the method [59]. Previous code summarization research examines the method declaration, control flow elements, and method calls. Thus, these semantic categories were given higher priority in our list of precedence [5, 70]. We also note that higher precedence does not always mean higher importance. For instance, 'Comments' are given a high position on the list, which may be counterintuitive. However, whereas labeling a token 'Loop Body' or 'Variable Name' may be debatable, a comment is unambiguously a comment. In addition, the list is relative in that some categories do not conflict with one another (e.g., 'Return' and 'Method Declaration').

We inspected return statements and values because prior code comprehension research demonstrates the relevance of program output for programmers' attention [14]. Furthermore, parameters, arguments, and variables were given higher precedence to gain insight into how the output evolves throughout the method [15]. To summarize, we originally considered 25 semantic categories for each token, but after settling label conflicts using this order of precedence, only 19 unique labels remained. We further filtered these categories based on where participants focused most and

provide more details in Section 5.1. After this process was complete, we obtained generalized semantic information for tokens in the Java code. At this stage, we could both localize programmers' gaze to tokens in the code and link this to broader semantic information about the code. The next step in the analysis was calculating eye-tracking metrics with respect to these semantic categories.

Eye-tracking Metrics To compare human attention during both facets of code summarization, we analyzed eye-tracking data with respect to the semantic categories detailed above. We use the following metrics in our comparison.

— **Fixations:** Researchers use fixations as a proxy to measure cognition [76], but fixations first need to be distinguished from saccades within the eye-tracking data. Fixations are spatially stable eye-gazes, whereas saccades are rapid eye movements during which little cognitive processing happens [67]. Current algorithms for discerning fixations from saccades rely on the *velocity* of the eye-movement [60]. If an eye-movement exceeds a certain threshold, it is considered a saccade. Following best practices in our implementation of a **Velocity-Threshold Identification (I-VT)** algorithm [16], we classified a gaze point as a saccade if it exceeded 400 px/100 ms. We both *counted* the gaze points identified as fixations and calculated their *average durations*. Our purpose for using these **fixation counts** and **fixation durations** was to measure differences in programmer attention between the Reading and Writing conditions. To this end, we compared cumulative fixation counts and durations between both conditions and compared fine-grained differences related to the semantic categories. We performed statistical tests, namely, Welch's t-tests corrected for multiple comparisons, between fixation data from both conditions and between experts and novices. These results are detailed in Sections 5.1 and 5.4.

— **Scan Path:** The scan path, or the ordered sequence of fixations, is informative of deeper cognitive processes [75, 76]. To develop our understanding of human attention and cognition during code summarization tasks, we replaced raw tokens in the scan path with their semantic categories, giving us *abstract scan paths*. For example, the raw scan path in Figure 3 contains the sequence printCUnit → SqlcPrettyPrinter → spp. The corresponding *abstract* scan path would be Method Call → External Class → Variable Declaration. Thus, by creating abstract scan paths, we could examine generalized patterns of programmer attention during code summarization tasks. We explored these patterns using analyses inspired by NLP, treating the abstract scan paths as *documents*. Specifically, we calculated the most common sequences within participants' abstract scan paths using N-Gram analyses, which we describe in Section 5.2.

   We also leveraged the scan path to explore another novel representation of programmer attention during code comprehension. To provide some background, previous research has measured the distances between programmers' consecutive fixations [19], noting that experts typically look farther. In the current study, we implemented a similar distance metric to measure the clustering of programmer attention during code summarization tasks. However, whereas prior research measured these distances in the raw gaze coordinates, we measured these as distances between nodes on the AST. We then compared these distances with those on the raw code. Previous research used Euclidean distance, but our measure on raw code is based on the order of the tokens within methods to make an equivalent comparison with the AST distances. For example, consider again the first line of code from Figures 2 and 3, with notations of the token order:

(1) public (2) void (3) genSql (4) throws (5) PositionedError.

   In this example, the absolute distance in raw code between '(1) public' and '(4) throws' would be 3. The Java methods in our dataset were not uniform in length; thus, the distances between consecutive fixations in larger methods may be farther than those in shorter
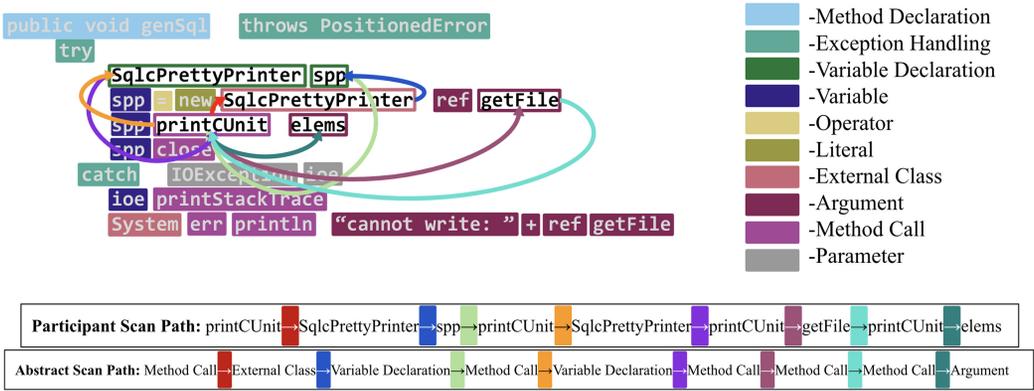
Fig. 3. An illustration of one participant's scan path through the `genSql` method. The actual scan path is shown below the code, with colors matching those of the arrows above. The *abstract* scan path is also shown, which illustrates the participant's flow of attention between semantic categories. The `genSql` method is the same method depicted in Figure 2.

methods. Likewise, a far jump in a shorter method may appear relatively minor in a larger method. For this reason, we normalized scan path distances based on the farthest possible distance per method and separately for the raw code and ASTs. We used these normalized distance metrics to compare code reading patterns between both facets of code summarization and between experts and novices, which we describe in more detail in Sections 5.3 and 5.4.

— **Regression:** Prior research defines regressions as backwards movements in the text [19]. We follow the same definition in the current study with respect to code. That is, if participants look from Token A to Token B and Token B is located earlier in the method, we consider that a regression. Regressions in eye-tracking data are informative of code reading patterns [19] and indicative of greater cognitive effort [76]. We use this metric to explore both cognitive effort and code reading patterns related to code summarization tasks in the current study. We report details about regressions in Section 5.2.

5 EXPERIMENTAL RESULTS

In this study, we analyzed differences between two code comprehension tasks using both facets of code summarization: reading code with a pre-written summary, and reading code to generate one. Accordingly, we frame our investigations with the following research questions:

— **RQ1.** How do attention patterns compare between code comprehension tasks — reading code with a pre-written summary and reading code to generate one?
— **RQ2.** Does the comprehension task influence programmers' sequences of attention between semantic categories?
— **RQ3.** What can we learn about code comprehension by mapping gaze data onto another representation of code (i.e., the AST)?
— **RQ4.** Do experience and other demographic factors mediate programmers' attention patterns during code comprehension tasks?

5.1 RQ 1: Comparison Among Summarization Tasks

In the current study, student programmers completed an experiment testing both facets of code summarization: reading code with a pre-written summary and reading code to generate one. We used eye-tracking to compare these two forms of code comprehension, first considering

cumulative differences in time taken and eye-tracking metrics. Second, we calculated differences in programmer attention on **semantic categories**.

5.1.1 Cumulative Differences. When we looked at the time these programmers spent reading just the code in the two conditions, we found that participants read each method for longer in the Writing condition compared with the Reading condition ($p < 0.0001$), averaging 26.54 seconds and 11.64 seconds per method, respectively. Moreover, we found that participants had higher fixation counts when writing summaries compared with when they were given summaries ($p < 0.0001$), averaging 94.92 fixations and 38.99 fixations on the code in each method, respectively. This suggests that undergraduate and graduate programmers invest more time and effort in understanding the code when they generate their own documentation. We found this trend for fixation durations as well, where each fixation on the code was longer, on average, when participants wrote summaries (0.11s) compared with when they were given a summary (0.1s). However, the difference did not reach statistical significance ($p = 0.06$).

5.1.2 Trends in Gaze Behavior. Total fixation counts and durations are informative, but we next sought to determine whether similar trends could be seen between participants based on the method characteristics. To explore this question, we used linear regression to analyze how eye-tracking metrics are affected by cyclomatic complexity. That is, do fixation counts or durations increase, decrease, or stay the same as code complexity increases? For each method, we calculated both the mean fixation counts for all participants who encountered it in the task and their mean average fixation durations for that method. To then compare results between Reading and Writing, and between fixation counts and fixation durations, we normalized the eye-tracking metrics to be within 0 and 1. The results are shown in Figure 4, but we can see in all cases that the slope (m) is positive, meaning that fixation counts and durations increase as a function of code complexity. Participants' average fixation durations increased more drastically than the counts for both Reading ($m = 0.008$) and Writing ($m = 0.028$), suggesting that the more complex methods elicit more focus and cognitive load from student programmers. In addition to the steeper slope for fixation durations in Writing, we can also see that the y-intercepts (b) are higher for Writing ($b = 0.36$, $b = 0.293$) compared with those in Reading ($b = 0.266$, $b = 0.257$). This suggests that the baseline level of cognitive load in the Writing condition is higher than that in the Reading condition. These results are intuitive and lay a foundation for more detailed comparisons we conducted between conditions and demographics.

5.1.3 Semantic Categories. Next, to form a fine-grained understanding of programmer attention in both summarization tasks, we considered *fixation counts* and *average fixation durations* on the semantic categories. Notably, we had 19 semantic categories for the tokens, but not every method contained all 19 categories. We thus concentrated our analyses on the categories in which programmers focused the most in the Reading and Writing conditions.

Inspired by the measurement of term importance in NLP, we calculated the weight ($\mathcal{W}_{sc}$) of each semantic category (sc) separately for the two conditions, based on participants' abstract scan paths and the frequency of semantic categories therein. Informally, the weights here represent how frequently a semantic category received attention from programmers. The weights were calculated as follows, where we first concatenated the scan paths for each method, M_i:

$$M_i \quad (i \in \{1, 2, \ldots, m\}), \tag{1}$$

where m is the number of methods for a condition (Reading ($m = 89$) or Writing ($m = 67$)), and from each participant, $P_{i,j}$, who engaged with method M_i in the experiment:

$$P_{i,j} \quad (j \in \{1, 2, \ldots, n_i\}), \tag{2}$$
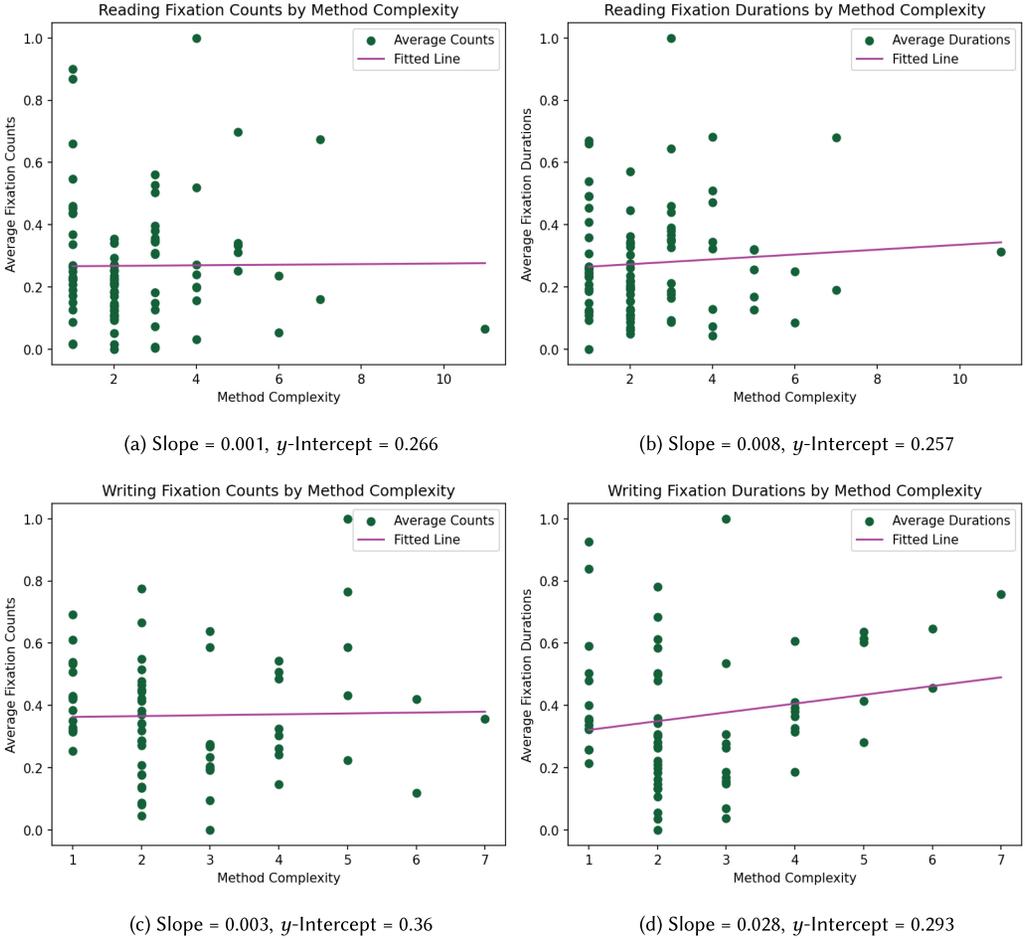
Fig. 4. Linear regression results calculated between eye-tracking metrics and method complexity. Specifically, we used cyclomatic complexity and considered its effect on fixation count in Reading and Writing, depicted in (a) and (c), and averaged between participants. In (b) and (d), we fit a line between cyclomatic complexity and average fixation durations, averaged between participants. In all subfigures, each datapoint represents averaged metrics across participants for one method. To compare slopes and intercepts between the two conditions and metrics, we normalized the fixation counts and durations to be within 0 and 1.

where n_i is the number of participants who engaged with method M_i. Because of task randomization, $n_i \in \{1, 2, \ldots, 27\}$. For method M_i and participant $P_{i,j}$, the symbol $sp_{i,j}$ refers to the participant's scan path for that method. From here, we collected all scan paths SP_i, which is a collection of all $sp_{i,j}$ specific to method M_i from all participants:

$$SP_i = \{sp_{i,1}, sp_{i,2}, \ldots, sp_{i,n_i}\}. \tag{3}$$

Thus, the total set of scan paths for all methods from all participants in either Reading or Writing can be denoted as SP:

$$SP = \{SP_1, SP_2, \ldots, SP_m\}. \tag{4}$$

For each method M_i, we calculated the weight $w_{sc,i}$ for each of the 19 semantic categories (sc) by first totaling the occurrences of *all* semantic categories (SC_i) in scan paths SP_i:

$$SC_i = \sum_{k=1}^{19} if \ sc_k \in SP_i. \tag{5}$$

For each semantic category sc, we calculated its weight ($w_{sc,i}$) within method M_i as a ratio, where the numerator is its occurrences (f_{sc}) in all scan paths for M_i (i.e., SP_i). The denominator is the occurrences of all semantic categories (SC_i) within method M_i. The log term measures the number of scan paths (SP_i) in which the semantic category sc is present, slightly scaling the first term based on the category's rarity:

$$w_{sc,i} = \frac{f_{sc}}{SC_i} \times \log\left(\frac{|SP|}{1 + |\{SP_i \in SP : sc \in SP_i\}|}\right) \tag{6}$$

This formula would give a scaled measure of semantic category sc's prevalence in scan paths SP_i for method M_i. The final weight for each semantic category $\mathcal{W}_{sc}$ was then obtained by averaging its weights across all methods:

$$\mathcal{W}_{sc} = \frac{1}{m} \sum_{i=1}^{m} w_{sc,i}. \tag{7}$$

To determine a subset of categories to consider for our analyses, we used the average weights $\mathcal{W}_{sc}$, reported in Table 3, and cross-referenced the categories with previous literature. We found that the top 8 semantic categories for both conditions include those reported by previous research, while also refining their classification. Specifically, prior code summarization research referenced the importance of the method declaration, method body, method calls, and control flow elements [5, 70]. Code comprehension research has noted the impact of elements that contribute to code complexity on neurological activity [62], suggesting that these might receive more attention.

In our list in Table 3, we see that these top 8 semantic categories for Reading and Writing contain those previously described in the literature, while also expanding upon them. We therefore used these top 8 semantic categories and considered the subset of Java methods containing these categories in subsequent analyses. The semantic categories we considered were the following: variable declaration, method declaration, conditional statement, parameter, method call, argument, conditional block, and loop body. Concentrating on this subset yielded eye-tracking data from 15 methods, totaling 2,850 eye-tracking data points. We note that choosing this subset involved a degree of subjectivity despite our best efforts and discuss this further in Section 7.

To compare fixation metrics between the conditions, Reading and Writing, we used the subset of methods containing these top semantic categories. Previous research highlights the impact of control flow complexity on attention and cognition [5, 62]; thus, we investigated conditional blocks and loop bodies separately. We scaled fixation counts based on the condition, Reading or Writing, because participants had higher fixation counts in the Writing condition. By contrast, we left the *average* fixation durations unaltered since this aggregate statistic is consistent across both conditions. We used Welch's t-test to compare fixation metrics because some samples had equal variance with their counterparts, while others did not [29]. We also performed **false delivery rate (FDR)** correction for multiple comparisons and report the corresponding q-values.

Comparing the two conditions then, we find that programmers writing a summary have higher fixation counts on parameters ($t = 3.09$, $d = 0.5$, $p < 0.01$, $q < 0.05$) and method calls ($t = 2.8$, $d = 0.46$, $p < 0.01$, $q < 0.05$). We find that, on average, programmers writing a summary fixate *longer* on **parameters ($t = 4.1$, $d = 0.66$, $p < 0.0001$, $q < 0.001$)**, variable declarations ($t = 3.23$, $d = 0.52$, $p < 0.01$, $q < 0.01$), and method calls ($t = 2.28$, $d = 0.37$, $p < 0.05$, $q < 0.05$). This illustrates that programmers focus significantly more on parameters, method calls, and variable declarations to

Table 3. Weights for Each Semantic Category Based on Where
Participants Focused Most (Frequency in Scan Paths)

Semantic Category	Reading	Writing
Variable Declaration	0.287	0.247
Method Declaration	0.264	0.251
Conditional Statement	0.226	0.236
Parameter	0.201	0.205
Argument	0.196	0.186
Method Call	0.170	0.167
Conditional Block	0.152	0.151
Loop Body	0.147	0.150
External Variable/Method	0.118	0.112
Return	0.106	0.136
Exception Handling	0.086	0.115
External Class	0.085	0.066
Variable	0.063	0.041
Comment	0.047	0.089
Assignment	0.030	0.035
Operator	0.025	0.007
Index Operation	0.007	NA
Operation	NA	0.018
Literal	NA	0.014

The frequency for each semantic category was scaled by its frequency in
scan paths from all methods. Not every method contained all 19 semantic
categories, so we examined the subset of methods containing the top
categories (those in bold).

understand the code when writing a summary. We can contextualize these findings using participants' post-task survey data. We asked participants the following: "When writing a code summary, what are 1 to 3 details of the code you think are the most important to write about?" Out of 27 participants, 18 mentioned either parameters or inputs, and 5 mentioned a description of how the inputs change. We see this pattern reflected in their eye-tracking data, where programmers pay particular attention to elements associated with the input. We further discuss the implications of this in Section 6.

Interestingly, when participants were given pre-written summaries, we find they had comparatively higher fixation counts but lower fixation durations on method declarations, variable declarations, conditional statements, and arguments. None of these differences rises to the level of statistical significance. However, this suggests that programmers do not expend as much effort to understand these semantic categories when documentation is present.

5.1.4 Loops. Next, we considered only the subset of methods within our larger subset that contain loops (9 methods, 1,482 data points). Intriguingly, we find that participants do not focus more on the body of the loop itself; rather, they focus more intently on other components of the code. Specifically, in the Writing condition, participants had even higher fixation counts on **parameters ($t = 4.48$, $d = 1.16$, $p < 0.0001$, $q < 0.001$)** and method calls ($t = 3.77$, $d = 0.98$, $p = 0.001$, $q < 0.01$). This trend is more pronounced for fixation durations, where participants again fixate longer on **parameters ($t = 5.4$, $d = 1.39$, $p < 0.0001$, $q < 0.0001$)** and **method calls ($t = 4.54$, $d = 1.11$, $p < 0.0001$, $q < 0.001$)**. All values for these comparisons and those in the previous section can be found in Table 4.

Table 4. Differences in Fixation Counts and Average Fixation Durations between
the Two Conditions, Reading and Writing

Category	Metric	Subset of Methods			Subset of Methods with Loops		
		Reading	Writing	Delta	Reading	Writing	Delta
Method Declaration	Fixation Count	**0.078**	0.052	0.026	**0.077**	0.063	0.014
	Fixation Duration	0.053	**0.060**	0.007	**0.057**	0.056	0.001
Parameter	**Fixation Count**	0.027	**0.053**	0.026 **	0.013	**0.064**	0.051 ****
	Fixation Duration	0.031	**0.066**	0.035 ****	0.015	**0.081**	0.066 ****
Variable Declaration	Fixation Count	**0.107**	0.095	0.012	**0.127**	0.112	0.015
	Fixation Duration	0.086	**0.140**	0.054 ****	0.120	**0.129**	0.009
Conditional Statement	Fixation Count	**0.065**	0.041	0.024	**0.068**	0.050	0.018
	Fixation Duration	0.052	**0.073**	0.021	0.052	**0.053**	0.001
Method Call	**Fixation Count**	0.026	**0.061**	0.035 **	0.017	0.097	0.080 ***
	Fixation Duration	0.026	**0.041**	0.015 *	0.015	**0.058**	0.043 ****
Argument	Fixation Count	**0.027**	0.020	0.007	0.036	**0.038**	0.002
	Fixation Duration	0.030	**0.035**	0.005	0.031	**0.054**	0.023
Loop Body	Fixation Count				0.081	**0.114**	0.033
	Fixation Duration				0.090	**0.125**	0.035

Fixation counts are normalized per condition. Comparisons are also shown for only methods that contain loops.
Average fixation durations are an aggregate metric, and are therefore not normalized. ($^*p < 0.05$, $^{**}q < 0.05$,
$^{***}q < 0.01$, $^{****}q < 0.001$).

Using fixations as a proxy for cognitive load, we see that student programmers devote more effort to understanding parameters and method calls when loops are present. This may be attributed to programmers tracing the program inputs through the methods and evaluating how they change. This appears to be more difficult when loops are present. As before, this hypothesis is further supported by participants' post-task survey data, where 18 out of 27 mention the importance of parameters and inputs, and 5 mention the importance of describing how inputs change.

5.1.5 Conditional Blocks. We next considered only methods of the larger subset that contain conditional blocks to study the impact of this control flow element on programmer attention (6 methods, 1,052 data points). Here, we find that participants have slightly higher fixation counts in the Writing condition on the conditional blocks themselves ($t = 2.19$, $d = 0.82$, $p = 0.041$, $q = 0.15$) and method calls ($t = 2.39$, $d = 1.018$, $p = 0.03$, $q = 0.15$). Looking at fixation durations when conditional blocks were present, we see that participants fixated longer in the Writing condition on variable declarations ($t = 3.43$, $d = 1.48$, $p < 0.01$, $q < 0.05$), conditional statements ($t = 2.33$, $d = 0.91$, $p < 0.05$, $q = 0.08$), and conditional blocks ($t = 2.25$, $d = 0.69$, $p < 0.05$, $q = 0.07$).

In calculating cyclomatic complexity, both loops and standard conditional blocks increase the program complexity. However, we find that these two semantic categories have divergent impacts on human attention; conditional blocks do not intensify programmer attention to the same degree as loops do. We discuss this in Section 6 as well, but we found this interesting in the context of a recent study into the neural correlates of code complexity. Specifically, Peitek et al. found no correlation between an increased cyclomatic complexity and brain activity [62]. Based on our results, it is possible that these two semantic categories have different effects on cognition, even though a loop and an if-statement have an equivalent impact on cyclomatic complexity.

> Student programmers focus more intently on **parameters**, **variable declarations**, and **method calls** when writing their own code summaries. In the post-task survey, 18 of 27 participants mention the importance of inputs or parameters in a good code summary. We also examined the influence of complexity on human attention, and find that loops intensify programmers' focus on parameters and method calls. Conditional blocks have a lesser impact.

5.2 RQ2: Patterns of Attention Sequences

From results in the previous research question, we have a better understanding of what semantic categories students focus on in both facets of code summarization: reading code with a pre-written summary and reading code to generate one. However, code comprehension is a complex cognitive task in which programmers relate disparate parts of code to one another [38]. We cannot explore sequences of attention by measuring only where programmers focus most.

To study programmers' deeper cognitive patterns during code comprehension tasks, we analyzed programmers' abstract scan paths from both code summarization conditions, Reading and Writing. Scan paths can be quite long and, therefore, inconsistent between participants and difficult to interpret [76]. As an alternative means of studying scan paths, we used analyses inspired by NLP to calculate common sub-sequences within them. We treated participant scan paths as *documents*, and calculated the most common transitions (i.e., N-Grams) that participants made between semantic categories. Specifically, we calculated the most common bigrams and trigrams, composed of two and three consecutive semantic categories, respectively. The most common bigrams are depicted in Figure 5. To illustrate bigrams within an abstract scan path, we can consider the following sequence: Method Call → External Class → Variable Declaration. Example bigrams would be Method Call → External Class, and External Class → Variable Declaration. Based on our implementation of semantic categories, adjacent tokens on the same line of code may be classified as the same semantic category. We excluded bigrams and trigrams that contained repeat categories to better understand transitions between categories.

Whereas the fixation data accentuates differences between Reading and Writing, here we find high agreement between them. We also find a high rate of vacillation, where student programmers frequently look between the same two semantic categories. For instance, participants looked most frequently from **method declaration → variable declaration** (Reading: 1585, Writing: 2593). We then see the reverse with the second highest frequency: **variable declaration → method declaration** (Reading: 1530, Writing: 2533). This coupling pattern continues for the next two transitions on both tasks, where participants frequently looked from **conditional statement → loop body** (Reading: 802, Writing: 2179), then the opposite (Reading: 765, Writing: 2189). Next, in both conditions, participants frequently looked from **conditional statement → method declaration** (Reading: 717, Writing: 1615), then back (Reading: 714, Writing: 1588) (Figure 5). Supporting these results, we also find a high rate of *regressions* in participant scan paths. We counted a regression every time a participant looked backwards in the code at a token that came previously. In the Writing condition, 47% of transitions in the scan path were regressions. In the Reading condition, 45% of these transitions were regressions.

The fixation data alone suggests that attention patterns can be influenced by the summarization condition, but we see that some broader code reading strategies are remarkably consistent between the two conditions. Upon closer inspection of Table 4, we see that method declarations and variable declarations have the highest fixation counts and fixation durations in both conditions. However, the differences between conditions are not significant. Programmers may commonly look between these categories to understand the relationship between the method declaration and the method body. These results and the high rate of regressions may also clarify the differing interpretations reported in previous studies on students and professional developers [5, 70]. Specifically, it is unclear whether programmers focus more on the method declaration or the method body. In the current study, inspecting fine-grained semantic information in scan paths, we find that programmers commonly vacillate between them. Based on the current findings, it may be more meaningful to consider the connections *between* the method header and the method body. We also note the prevalence of conditional statements in the most common transitions. Again considering cyclomatic complexity, we find evidence that control flow elements act as a sink for programmers'

Writing: Most Common Transitions

(a) Writing Bigrams

Reading: Most Common Transitions

(b) Reading Bigrams

Fig. 5. Bigram frequency representing common gaze transitions between semantic categories during the Writing and Reading conditions.

attention as they read code. We note that our participant sample consists of undergraduate and graduate students, which may not generalize to professional developers. We discuss this further in Section 7.

We also calculated the most common trigrams, or transitions between three semantic categories, for the Reading and Writing conditions. Regardless of the condition, participants most frequently looked from **variable declaration→ method declaration→ parameter** (Reading: 109, Writing: 171). Participants diverged from here, depending on the condition. For the Reading condition, the next two most frequent trigrams were method declaration → parameter → variable declaration (90), and parameter → method declaration → variable declaration (67). In the Writing condition, the second most frequent trigram was method declaration→ conditional statement→ conditional

block (146), followed by parameter → method declaration → variable declaration (138). We again see the importance of variable declarations to programmer attention in both conditions, which was previously unreported. These results also suggest that programmers seek to relate variables within the method with the parameters entering the method. These trigrams also have lower frequencies compared with the bigrams, which suggests the high variability between participants' reading strategies.

> Comparing both facets of code summarization, student programmers consistently look between the same semantic categories, with a high rate of vacillation. Student programmers most commonly look between the **method declaration ⇄ variable declaration**, followed by **conditional statement ⇄ loop body** and **conditional statement ⇄ method declaration**.

5.3 RQ3: Human Attention on the Abstract Syntax Tree

In this study, we examined patterns of human attention to compare two forms of code comprehension. We, along with prior research, have only considered human attention on *raw* code so far. However, Java has a rich underlying structure, the Abstract Syntax Tree (AST). Exploring human attention on the AST offers another avenue through which we can study and understand code comprehension. While programmers may not interpret the AST directly, they may implicitly comprehend some of the information therein. For instance, 'parameter list' nodes in the AST specify the parameters and their types for the compiler. A human developer can infer the parameters from looking at contextual clues in the source code without looking at the AST. In this study, we considered the reading distance that programmers traveled from one token to the next. We used distance to measure the general clustering of programmer attention during both code summarization tasks. Using the previous example about parameter lists, if programmers focus intently on elements of a parameter list, the distances between consecutive tokens will be smaller than if they frequently switch their attention from parameters to return statements, for example.

Previous code comprehension research measured the Euclidean distance between consecutive fixations [19]. We implemented a similar distance metric, but analyzed consecutive tokens in the scan path. That is, we calculated the distance from one token to the next as two nodes on the AST. We also compared these *AST* distances to *raw code* distances, as described in Section 4. We measured the AST distances between consecutive tokens in the scan path using Breadth-First Search, and calculated the raw code distances using the tokens' order within the methods. As mentioned, the Java methods in our dataset were not uniform in length, which may influence both the AST distances and raw code distances. To generalize both of these distance metrics, we normalized each distance by the farthest possible path in its method, and separately for raw code and ASTs. We should note that in the previous research question, we analyzed abstract scan paths, but here we considered scan paths on raw tokens.

After calculating AST distances and raw code distances using scan paths, we first compared the two metrics with one another to explore the relationship between the two. In other words, using the same scan paths, are distances generally farther in the code, or farther in the AST? In total, we had 163,384 data points from raw code distances and AST distances. Combining both types of distances, there were 56,500 data points in the Reading condition and 106,884 data points in the Writing condition. The data was not normally distributed; thus, we used Mann-Whitney U-tests for our calculations. We did not correct the following statistical tests for multiple comparisons because they were individual isolated tests [3]. We find that AST distances are significantly farther than raw code distances ($U = 869, 986, 484.0$, $d = 1.72$, $p < 0.0001$), which may not be surprising considering the verbosity of AST representations.

Next, to compare patterns of attention clustering between the two forms of code summarization, we analyzed differences between the two conditions, Reading and Writing. Looking at raw code distances, we find that participants looked farther, on average, between consecutive tokens in the Writing condition ($U = 839{,}092{,}048.5$, $d = 0.1$, $p < 0.0001$). Surprisingly, when we compared both forms of comprehension using *AST* distances, we found **no significant difference ($U = 910{,}543{,}439.0$, $d = 0.004$, $p = 0.745$)**. This null result was unexpected and may demonstrate the potential disconnect between human attention on these two representations of the same code. As a possible explanation, we can re-examine the results from Section 5.2. To reiterate them here, we found that regardless of the condition, participants made consistent patterns of attention sequences between semantic categories. Perhaps student programmers look between equivalent semantic categories whose locations may vary in the raw code. However, it is possible their locations are more stable in the ASTs. Regardless, further study is needed to contextualize these results.

> We examined the distances between consecutive tokens in participants' scan paths using both AST and raw code distances. We find that AST distances are significantly farther than the latter. Comparing Reading and Writing, raw code distances are farther between consecutive tokens when programmers write summaries. However, there is no significant difference between the conditions in terms of AST distances.

5.4 RQ4: Differences Mediated by Experience and Other Demographics

Software Engineering research regularly aims to identify factors differentiating experts from novices [5, 33, 34, 47]. By isolating the effective behaviors of experts, we can ideally help guide novices toward these practices [79]. Accordingly, we conducted an in-depth comparison between experts and novices within our participant pool of undergraduate and graduate CS students. We also present preliminary analyses based on gender and native language. We considered the top third, or tercile, of our participants as experts ($n = 9$, $>= 7$ years coding experience), and roughly the bottom tercile as novices ($n = 10$, $<= 4$ years). We did not include the middle tercile of our participants in our comparisons to make a clearer distinction between experts and novices. Following a similar framework as previous sections, we first present a comparison between the two groups based on their **cumulative differences** in time and fixations. We then compare how the two groups focus on **semantic categories** for both conditions, Reading and Writing. We then examine differences between the groups' **attention on the AST** and conclude with **comparisons based on gender and native language**.

5.4.1 Cumulative Differences. To understand general differences between novices and experts, we first compared the amount of time each group spent reading the code in both forms of code comprehension. We found that experts spent an average of 7.66 seconds reading just the code in the Reading condition compared with novices, who spent an average of 14.3 seconds ($p < 0.0001$). In the Writing condition, we found that experts spent an average of 21.65 seconds reading just the code, whereas novices spent an average of 30.87 seconds ($p < 0.0001$). We find that novices spent almost double the amount of time as experts did when pre-written summaries were provided and about 40% more time when programmers wrote their own summaries.

Next, we compared the groups based on their fixation data, which followed the same trend. Novices had higher average fixation counts (55.59, $p < 0.0001$) and durations (0.1, $p < 0.0001$) on each method in the Reading condition compared with experts' average fixation counts (23.56) and durations (0.06s). We see this pattern in the Writing condition as well: novices had higher average fixation counts (106.61, $p < 0.01$) and durations (0.12s, $p < 0.0001$) on each method compared with experts' average fixation counts (81.69) and durations (0.09s). From these results alone, it appears

(a) Reading: Fixation Counts

(b) Reading: Fixation Durations

(c) Writing: Fixation Counts
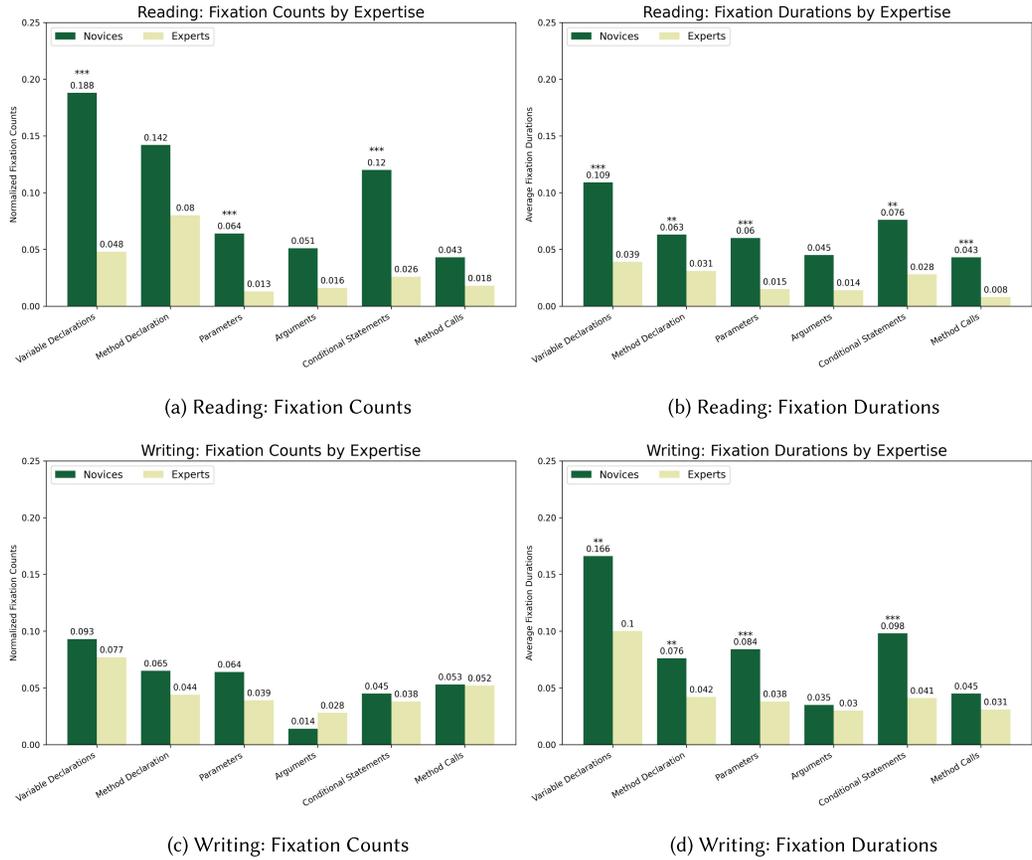
(d) Writing: Fixation Durations

Fig. 6. Comparison between experts' and novices' fixation counts (a, c), and fixation durations (b, d) during both conditions, Reading and Writing. We initially considered 19 semantic categories, but not every method contained every category. We therefore filtered the categories based on which participants focused the most (Section 5.1) and examined programmer attention with respect to these. Fixation counts were normalized based on the condition, Reading or Writing, whereas the average fixation durations were not ($^*p < 0.05$, $^{**}q < 0.05$, $^{***}q < 0.01$, $^{****}q < 0.001$).

that experts expend less time and effort to read the code regardless of whether they are given a pre-written summary or generating their own.

5.4.2 Semantic Categories. We next investigated where in the code the groups focus in both facets of code summarization. To this end, we compared experts' and novices' attention with respect to the semantic categories during the Reading and Writing conditions. As before, we had 19 semantic categories, but not all methods contained every category. We used the same calculations as those in Section 5.1 to focus on the semantic categories that received the most attention: variable declarations, method declarations, parameters, arguments, conditional statements, and method calls. This yielded data from 15 methods and 396 eye-tracking data points.

The results are illustrated in Figure 6, and we also detail the notable findings here. Most strikingly, we see that when programmers were given a pre-written summary, novices had higher fixation counts and fixation durations for each of the categories we considered. However, when programmers wrote their own summaries, there were **no significant differences** between

experts' and novices' fixation counts on these semantic categories. In fact, when writing summaries, experts had higher fixation counts than novices did on arguments, but this did not rise to the level of statistical significance. We note that these null results appear to conflict with those reported above in Section 5.4.2. It is possible that the groups' attention differed on other categories outside of our subset, accounting for the significant differences above in Section 5.4.2. We discuss this in Section 7. With respect to the figure, we also observe a general pattern in which programmers had higher fixation *counts* when they were given pre-written summaries ($\mu_{Reading}$ = 0.067, $\mu_{Writing}$ = 0.051), but longer fixation *durations* when they were writing their *own* summaries ($\mu_{Reading}$ = 0.044, $\mu_{Writing}$ = 0.065). These differences do not rise to the level of significance (p = 0.36, p = 0.16, respectively) but are intuitive in that programmers may *frequently* look at certain semantic categories if a pre-written summary is provided. However, they may not need to invest the same amount of attention and cognitive effort as if they were writing their own summaries, as we can see from their fixation durations.

Looking at results from each condition in more detail, we see in the Reading condition (Figure 6) that novices had significantly higher fixation counts on variable declarations (t = 3.66, d = 0.98, p < 0.001, q < 0.01), parameters (t = 3.21, d = 0.86, p < 0.01, q < 0.01), and conditional statements (t = 3.01, d = 0.81, p < 0.01, q < 0.01). In the same condition, novices had significantly longer average fixation durations on variables declarations (t = 3.11, d = 0.83, p < 0.01, q < 0.01), method declarations (t = 2.59, d = 0.68, p < 0.05, q < 0.05), parameters (t = 3.01, d = 0.80, p < 0.01, q < 0.01), conditional statements (t = 2.81, d = 0.75, p < 0.01, q < 0.05), and method calls (t = 3.35, d = 0.91, p < 0.01, q < 0.01). In terms of where the two groups focused *most*, we see in the Reading condition that novices fixated the most and for longest on variable declarations. By contrast, experts fixated the most (i.e., fixation count) on method declarations, and the longest (i.e., fixation duration) on variable declarations. These results suggest that when a pre-written summary is provided, experts can sufficiently understand the code by focusing on the method declaration and variable declarations, perhaps focusing on *beacons* in the code that are key to comprehension [27]. Novices, however, appear to expend more effort to understand internal components of the methods. This is supported by our observation that novices fixated significantly more and for longer than experts did on variable declarations and conditional statements.

As previously mentioned, when participants wrote summaries, there were no significant differences between the groups' fixation counts on the semantic categories we considered (Figure 6). We do see that novices fixated for significantly longer on certain semantic categories: variable declarations (t = 2.45, d = 0.66, p < 0.05, q < 0.05), method declarations (t = 2.56, d = 0.72, p < 0.05, q < 0.05), parameters (t = 3.66, d = 0.97, p < 0.001, q < 0.01), and conditional statements (t = 3.61, d = 0.96, p < 0.001, q < 0.01). Interestingly, when participants wrote summaries, we see that both groups attended most to variable declarations. These results together indicate that when programmers are reading code to generate their own summaries, novices may still fixate for longer on some semantic categories, but regardless of their expertise, programmers appear to devote similar amounts of cognitive effort into reading the code.

5.4.3 Attention on the Abstract Syntax Tree. We next tested whether we could uncover differences between experts and novices when we map their attention onto another representation of code, the AST. Following the same procedure as above in Section 5.3, we calculated the distances between consecutive tokens in experts' and novices' scan paths. In total, we had 42,554 data points for the Reading condition and 71,474 data points for the Writing condition. Each distance measure was scaled per method using the longest path in the tree or farthest positional distance in the raw code. Because the data is not normally distributed, we used Mann-Whitney U-tests to compare between code distances and AST distances.

In the Reading condition and on raw code, novices looked slightly farther between consecutive tokens, on average, than experts did ($U = 55{,}473{,}559.5$, $d = 0.05$, $p < 0.001$). However, in AST distances, there was no significant difference between the groups ($U = 56{,}630{,}433.5$, $d = 0.02$, $p = 0.22$). In the Writing condition, this relationship was flipped in both respects. Experts looked slightly farther between consecutive tokens in the raw code ($U = 192{,}818{,}739.5$, $d = 0.001$, $p < 0.01$), and this difference was *magnified* for AST distances (**$U = 201{,}802{,}465.5$, $d = 0.13$, $p < 0.0001$**). We see that mapping programmer attention onto the AST does not always have the same outcome (i.e., more significant), and does not necessarily match attention patterns on raw code. In the Reading condition, we see that AST distances are more muted than raw code distances, but they are more pronounced in the Writing condition. These puzzling results may indicate that experts are more fixed and methodical in how they read code, whereas novices may be more haphazard.

Previous studies may provide some insights. Researchers have examined code reading patterns, in particular, whether programmers focus on *beacons* in the code or read code based on its *execution order* [19, 27]. A beacon is a feature in the code that is key for facilitating comprehension of the program. Crosby et al. performed an eye-tracking study and found that novices do not discriminate between different areas of the code, whereas experts tend to identify and focus on these beacons [19, 27]. Based on our study, it is possible that experts are drawn to these beacons regardless of the comprehension task. Researchers have also tested whether programmers read code from one token to the next in a linear fashion or whether they read the code based on its flow of *execution*. Researchers hypothesize that experts read code closer to its execution order. That study Busjahn et al. [19] reported that novices actually read code closer to its execution order but also notes that the code snippets read by novices in their experiment were more linear by nature, which may have been a confounding factor. Based on these theories for code comprehension, it is possible that more experienced programmers read code in a more structured way, which may be observable in the AST.

5.4.4 Comparisons Based on Gender and Native Language. Our participants in this study were diverse in terms of experience, gender, and native language. While our sample is suitable for comparison based on years of experience, which we detail above, it is not ideal for comparison based on gender and native language. We nonetheless conducted preliminary analyses based on these factors to understand their influence on the dataset. These results may lack generalizability and warrant thorough validation in future study. For gender, only one woman is in our expert group, meaning the men's data is biased towards the experts. Similarly, 7 of the 9 experts are in the non-native English speakers group. With an imbalanced sample, any significant differences between the groups may be attributable to the differences in expertise and not the variable in question [31]. We therefore excluded *all* experts in comparing these groups.

After excluding experts, there was no significant difference between men ($n = 11$) and women ($n = 7$) in their years of experience ($\mu = 4.5$ years, $\mu = 3.5$ years, respectively). Similarly, there was no significant difference between native English speakers ($n = 12$) and non-native English speakers ($n = 6$) in their years of coding experience after excluding experts from both groups ($\mu = 4.25$ years, $\mu = 4.17$ years, respectively). As an additional consideration, non-native English speakers in our sample are heterogeneous in that they do not all share the same native language: these 6 participants represent 5 other languages. Furthermore, in comparisons based on native language, all participants are students at R1 universities in the United States, where admission is contingent on English speaking and comprehension ability, and classes are taught in English. For both gender and native language, we present preliminary results on cumulative fixations, fixations with respect to semantic categories, and distances in raw code and on the AST.

Gender Analyzing differences between men and women students then, we first considered cumulative fixation differences. We found that women fixated with significantly higher frequency

(52.35, $p < 0.01$) and longer (0.11 s, $p < 0.01$) on code in the Reading condition, compared with men's fixation counts (43.02) and average durations (0.09 s). In the Writing condition, by contrast, we see that men had significantly higher fixation counts (111.06, $p < 0.01$) compared with those of women (90.5). Women had slightly higher fixation durations (0.123 s) for the Writing condition compared with those of men (0.117 s), but this did not rise to the level of statistical significance. These results on cumulative fixations suggest differences in how men and women read code, implying that women invest more attention and effort in understanding the code when there is accompanying documentation. This is supported by previous eye-tracking research in Software Engineering, in which women were found to focus on all answer options in a study on identifier style [77]. Here, our results suggest that women spend more time and effort checking whether code matches accompanying documentation.

This is not necessarily supported by our results comparing fixations on the top semantic categories, where we found no significant differences between men and women. This was true for both Reading and Writing, and the subset of methods containing loops, and the subset containing conditional blocks. It may be possible that differences between men's and women's fixations relate to semantic categories we did not analyze in this study. However, considering the potential influence of demographics on the larger dataset, these results suggest that participants in both groups are similar in how they fixate on semantic categories of the code. Interestingly, when we compared men and women based on the code and AST distances, men looked significantly farther for Reading on both the raw code ($U = 57,422,118.5$, $d = 0.046$, $p < 0.001$) and the AST ($U = 57640685.0$, $d = 0.046$, $p < 0.001$). By contrast, women looked significantly farther for Writing on both the raw code ($U = 169,771,038.5$, $d = 0.057$, $p < 0.001$) and the AST ($U = 170,753,340.0$, $d = 0.055$, $p < 0.0001$). Here, we can see that the distances in both conditions align with the fixation data, where women had higher fixation counts and durations in the Reading condition, and shorter distances looked from one token to the next. In the Writing condition, men had higher fixation counts and looked shorter distances from one token to the next compared with women. This suggests that women may read code more thoroughly when documentation is present, whereas men may read the code more closely when generating their own summaries. This gender difference has not been previously reported in eye-tracking research within Software Engineering [44, 77] and warrants further study to understand the basis for these findings. Our preliminary results suggest that men and women comprehend code to different degrees depending on the circumstances.

Native Language In our analyses comparing native English speakers with non-native English speakers, we first considered cumulative differences. In the Reading condition, we found that native English speakers' average fixations were significantly longer (0.11 s, $p < 0.01$) compared to those of non-native English speakers (0.09 s). That being said, there was no significant difference in the groups' fixation *counts* in Reading. In the Writing condition, native English speakers had significantly higher fixation counts (113.27, $p < 0.0001$), on average, compared with those of non-native English speakers (83.06), but there was no significant difference between the groups in their fixation durations in this condition. Based on these results, we see a trend in which native English speakers in the sample invested more effort to read the code in both the Reading and Writing conditions. This is also supported by results below related to distances looked on the raw code and on the AST. Similar to the comparison between men and women, we also found no significant differences in where the two groups focus based on the top semantic categories. This was also true for the subsets of methods with loops and those with conditional blocks. These null results also suggest that native language does not influence student programmers' attention with respect to the semantic categories.

While we did not find significant differences in terms of fixations on semantic categories, we found significant differences between these two groups in their average distances looked on raw

code and on the AST. In the Reading condition, we found that non-native English speakers looked significantly farther than native English speakers did on both the raw code ($U = 48,280,575.5$, $d = 0.041$, $p < 0.0001$), and on the AST ($U = 45,746,143.0$, $d = 0.175$, $p < 0.0001$). This was also the case in the Writing condition for distances on both the raw code ($U = 138,773,451.5$, $d = 0.062$, $p < 0.0001$) and the AST ($U = 139,679,166.0$, $d = 0.067$, $p < 0.0001$). From the cumulative fixation results, we see that native English speakers tend to fixate more on the code, and the distances they look from one token to the next are subsequently shorter. More research would be necessary to contextualize these preliminary findings, but since code primarily uses English key words, it is possible that native English speakers read code more linearly than non-native English speakers. This may contribute to the higher fixations and the shorter distances from one token to the next. If non-native English speakers are less familiar with the language, perhaps they are less restricted in their reading patterns.

Novices spend more time and effort focusing on the code in both forms of code comprehension, attending the most to variable declarations, method declarations, and conditional statements. We find one caveat: when experts and novices write their own summaries, there are no significant differences in their fixation counts, at least on the categories we considered. Mapping human attention onto the AST, novices look farther in the raw code when they are given a summary, but this is not significant in terms of AST distances. Experts look farther in the raw code in the Writing condition, which is even more pronounced in the AST distances. Preliminary results suggest that women read code more thoroughly when a summary is present, whereas men read code more thoroughly when generating their own documentation. In addition, native English speakers fixate more on the code in both conditions. Preliminary results suggest that demographic factors do not influence attention with respect to semantic categories.

---

6 DISCUSSION

Based on the results from analyzing the eye-tracking data, we present the following interpretations and future directions. Specifically, we discuss and contextualize the results **comparing these two forms of code comprehension** (i.e., Reading and Writing), implications for **eye-tracking methodology**, and findings from mapping **human attention onto the AST**.

6.1 Comparing Two Forms of Comprehension

We set out to understand the differences between code comprehension tasks by looking at both facets of code summarization, Reading and Writing. In this context, reading code to write a summary can be considered a more active, generative process. Based on the fixation data, we consistently find that writing a summary demands more attention to the code from student programmers. We also see that writing a summary somewhat equalizes experts and novices, where we find no significant differences in their fixation counts in the Writing condition on the categories we considered. Comparing Reading and Writing more generally, we see that student programmers writing summaries will focus comparatively more on parameters ($q < 0.001$), variable declarations ($q < 0.001$), and method calls ($q < 0.001$). To consider why these semantic categories differentiate the two forms of comprehension, we can look at what the programmers said themselves. Over 60% of our participants mentioned the importance of inputs for understanding the overall purpose of a method. A smaller percentage mentioned the importance of describing how inputs change throughout the method. We then see this trend reflected in the eye-tracking data, where programmers appear to focus on the input and how it changes. If a pre-written summary is provided, our results suggest that programmers do not need to follow these elements as intently, and instead follow the high-level features.

Considering differences between experts and novices more explicitly, we see from Figure 6 that these groups generally align in where they focus (i.e., method declaration, variable declarations). Novices simply tend to focus more on these categories. However, we consistently see across the conditions that novices pay attention significantly more to conditional statements ($q < 0.001$). Perhaps experts are quicker to discern the meaning of conditional statements, but it is also possible that a deep understanding of the conditions is not crucial for these comprehension tasks. We note that the same would not be true if we were testing debugging tasks. In support of this, we asked our participants in the post-task survey "What are 1 to 3 examples of unimportant details in the code that don't need to be mentioned in a code summary?" Out of 27 participants, 5 mentioned the logic or conditions specifically. For code summarization, these results together suggest that conditional statements are an area where novices could save time and effort. Furthermore, our results have implications for tool design [6, 55], where static analysis might give novices more context for the semantic categories where they focus most: variable declarations ($q < 0.001$), parameters ($q < 0.05$), conditional statements ($q < 0.001$), and the method declaration ($p < 0.01$).

The results based on fixations illustrate the differences between these forms of code comprehension, but looking at more complex patterns of attention reveals notable parallels between them. We see that regardless of the condition, student programmers commonly vacillate between the same semantic categories. That is, in both conditions, students commonly look between method declaration $\rightleftarrows$ variable declarations, loop bodies $\rightleftarrows$ conditional statements, method declarations $\rightleftarrows$ conditional statements. It is somewhat unexpected that variable declarations are so prominent in student programmers' attention, especially since this semantic category has not yet been reported in code summarization literature [5, 70]. Previous studies presented different interpretations as to whether programmers focus more on the method declaration or the method body. Our results provide another interpretation, in which programmers in our sample commonly make connections *between* the method body and the method declaration. This suggests that the link between them is critical. Very informally, if we think of the method name as a book title, the method body would be the story, and variables would be the main characters. In this analogy, it is important to understand the role of the variables with respect to the method's purpose. Thus, can we determine whether these forms of code comprehension are distinct? It appears that there are nuances in where student programmers' focus but stable consistencies in how they read the code.

Practical Implications In light of these findings, how can we benefit from a more nuanced understanding of code comprehension? The implications can extend from CS education to model training for automated code summarization and tool design for IDEs. In this study, we see differences in how students read source code depending on their purpose for doing so and specific strategies employed by more expert programmers. We see an overarching story forming from the fixation data where students focus on how program inputs become outputs, from parameters to variable declarations and method calls. This pattern is more pronounced when students are summarizing source code themselves, which provides concrete evidence for educators to guide new programmers' attention to these components of methods on tasks requiring a deep understanding of the code. By contrast, when students are reading code through the lens of a pre-written summary, our fixation and bigram results suggest that they pay particular attention to method and variable declarations, perhaps just "skimming" or "checking" the high-level components of the code.

This can be informative for educators and tool design for IDEs alike, where programmers can be specifically directed to these elements if they are forming a high-level understanding of documented code. These results suggest that variable declarations and method declarations in particular should be contextualized for student programmers reading code with additional documentation. For deep learning models, it is possible that the (publicly available) gaze data from this study can

be directly used to improve automated methods for code summarization using human attention. Here, we also see an opportunity for novices, where experts fixate significantly less on conditional statements during these summarization tasks. If programmers need to form a rapid understanding of a method, results from our experts suggest that a more cursory look at conditional statements may be sufficient. Though it does not rise to level of statistical significance, we also see that experts in our sample fixate comparatively more on arguments when they are writing a summary, suggesting that the information conveyed by arguments can be helpful for comprehension. Overall, results from this study offer a precise look at code comprehension behaviors in student programmers and can be informative in situations in which code comprehension is taught, analyzed, and facilitated.

6.2   Eye-Tracking Methodology

In this experiment, we relied upon the scan path in particular to study reading patterns during code summarization tasks. We leveraged analyses inspired by NLP to expose patterns within this ordered sequence of attention. Fixation data alone is informative and can reveal details about where humans focus. However, the scan path as a sequence of these fixations can elucidate deeper cognitive patterns [75]. Despite its potential, there are implicit challenges that come with analyzing the scan path. First and foremost, scan paths can become quite long and can vary widely between participants, becoming unwieldy and difficult to interpret [76]. Previous studies have tackled this problem by comparing the similarity between scan paths as a whole [19, 26], visualizing scan paths [68], and encoding the behavior of the developer at different time points in the scan path (i.e., debugging, coding) [5]. In this study, we attempted to dissect and analyze the scan path by taking advantage of the semantic categories.

By creating *abstract scan paths* based on the tokens' semantic categories, we could examine generalized patterns of attention within the scan path. The insight then came from treating these abstract scan paths as *documents*, which then frames the analyses as Information Retrieval tasks in the context of NLP [80]. First, we used this insight to calculate the categories on which programmers focus most in Section 5.1. We examined the categories' frequency in scan paths for one method and compared this to their prevalence in scan paths from all other methods. This may have been accomplished using raw fixation data, but an analysis for this purpose is already typical in NLP [9, 65]. Second, in Section 5.2, we used N-Gram analyses to enumerate common transitions programmers made between semantic categories. By using bigrams and trigrams, we could identify the predominant clusters of student programmers' reading patterns. While N-Gram analyses are usually an early step in NLP tasks [18], we used them as a final step to uncover sub-sequences within scan paths. Using these foundational NLP metrics, we demonstrate the efficacy of treating scan paths as documents for Information Retrieval. Moreover, *because* these metrics are foundational, this suggests exciting new directions for analyzing the scan path using more advanced NLP techniques.

6.3   Human Attention on the Abstract Syntax Tree

In this study, we also explored human attention on an alternative representation of code, the AST. Human programmers may not interact with the AST directly but implicitly derive the same information (i.e., variable types, arguments). Therefore, we can gather detailed information about programmer focus by considering their visual attention in the context of the AST. We conceptualized this type of mapping as analogous to Fourier transformation for audio, in which a complex signal is disassembled and compressed into meaningful features. We find exciting and puzzling results, suggesting the potential for this type of mapping in future research. More specifically, we find inconsistent patterns between how far programmers look in the raw code and how far they look in the AST. First, in the Writing condition, students look significantly farther between consecutive tokens in raw code but not significantly farther in terms of AST distances. Second,

comparing experts and novices in our sample, we find in the Reading condition that novices look farther between consecutive tokens in the raw code but not significantly so in the AST. The opposite is true for Writing, in which experts look significantly farther in the raw code and even more so in the AST. We see that AST distances do not always match raw code distances, which suggests that these two types of information are revealing, yet not necessarily equivalent.

Previous research has examined programmers' code-reading order [19] as well as comprehension strategies: bottom-up versus top-down [63, 85]. Those studies reported that experts read code less linearly and may be better with *top-down* comprehension [19, 63], in which they form a preconceived understanding of the code based on higher-level features [63]. There is a related vein of research into *beacons* in the code, which are features that are key to comprehension [27]. Crosby et al. [27] conducted an eye-tracking study. They found that experts are able to recognize these features and subsequently focus more attention on them. By contrast, that study found that novices' attention is more distributed over the program. Based on those studies' findings, it is possible that experts read code in a more structured way, which may be observable on the AST. If experts do read code more deliberately, this could perhaps explain why their AST and raw code distances agree in the Writing condition and why novice distances do not demonstrate an enduring pattern in the Reading condition.

Furthermore, prior research examined human attention during code summarization to inform and improve methods for automated code summarization [70]. We continue this trend with the current study by mapping human attention onto the AST. More specifically, current top performing models for automated code summarization include structural information from the AST during training [46, 51, 83, 89]. In their study, Rodeghero et al. [70] aimed to align the output of automated methods with human priorities. In the current study, we aim to extend this by measuring the paths programmers take through the AST as they perform code summarization tasks. We measured the distance between consecutive tokens in the scan path to explore the clustering of programmers' attention. Subsequently, we find intriguing initial results from comparing experts and novices and code comprehension types. That being said, the raw AST may be limited as a vehicle for studying code comprehension. Using a previous example, if programmers glance from a parameter to a return value, there will likely be intermediate information in the tree that is unnecessary for understanding this attention switch from parameter to return. Here, the programmers' attention may represent a "short cut" in the AST, which warrants more refined metrics for studying code comprehension on the AST. Informative and perhaps limited, our results suggest fertile ground for future research to explore other metrics of attention on the AST to better understand code comprehension.

7 THREATS TO VALIDITY

In this section, we consider potential threats to validity in our study. We primarily group these into two categories. First, we discuss the possible limits to the **generalizability** of our findings, such as the use of Java and our participants' level of experience. Second, we discuss conceivable sources of **noise and random effects**, such as running the study in two locations and the quality of pre-written summaries.

Generalizability There are a few factors that may limit the applicability of our findings more broadly. Here, we discuss our sample size, participants' demographics, and the selection of the Java methods and their summaries. First, we draw conclusions about student programmers at large based on the data from 27 participants. While this sample may not be entirely representative, our participants are diverse in age, gender, native language and experience. We also collected data from students at two universities. Furthermore, the quantity of data we collected has sufficient statistical power to confidently detect differences between the comprehension tasks and between

experts and novices (relative within our sample). Related to this, we classified programmers in our study as novices if they had 4 years of programming experience or fewer (bottom tercile) and experts if they had 7 or more years of experience (top tercile). Based on these criteria, our experts may still be relatively inexperienced. Nonetheless, experts in our study were all graduate students, and we excluded the middle third of participants from our comparison to give a sharper contrast between the two groups. The significant differences found between experts and novices in our sample might extend to even larger experience gaps in industry. However, further study would be needed to validate the hypothesis. We also conducted preliminary analyses based on gender and native language, but our sample was not ideally suited for this task due to the uneven distribution of experts within these groups. We excluded experts from these analyses so that any significant differences were not due to imbalances in expertise. We sought to explore the influence of other demographics on our data and recommend future validation of these results.

Lastly, the selection of Java as our target language and of the particular Java methods and associated summaries we used may inherently affect the generalizability of our findings. Specifically, the analyses on Java's semantic categories may not scale to other programming languages. By nature, the semantics of Java are similar to other object-oriented programming languages, and we first selected Java as our target language because of its prevalence in CS education and real-world software projects [17, 23, 24]. However, further study would be required to truly test language-specific visual attention. In addition, the summaries used in this study may not be a representative sample of all summaries, which may have influenced participants' attention to the code. This was a consideration for previous research [12, 39, 52], which sourced the methods from open-source projects and refined them for research purposes, including automated code summarization and human studies. Next, our interface was designed for the current study without elements such as scrolling or syntax highlighting; therefore, they may lack realism. These decisions were consciously made for the benefit of other study factors, such as eye-tracking data quality. Nonetheless, we attempted to improve our task's generalizability by using the FunCom dataset, which consists of real-world Java methods [52]. We sought to further increase the robustness of our collected data by presenting a wide variety of randomized methods to our participants. Lastly, we asked participants in the Reading condition to rate the quality of code summaries using Likert-scale questions, which would not likely be asked in a real-world scenario. Even though programmers outside of an experimental setting may not explicitly assign values to the accuracy and readability of a summary, for instance, they may be making these judgements implicitly.

Noise and Random Effects Next, we consider factors that may have had an unintended influence on the study. Here, we discuss the two study locations, the content of the pre-written summaries, participants' self-reported experience, and decisions made during data analysis. For data collection, the human studies were run in two separate locations, which may have led to slight differences in how participants completed the tasks. Both institutions are similarly sized private universities and have comparable CS curricula, but we attempted to further limit this possibility by synchronizing our experimental procedure and equipment. We used a script during experimental sessions to ensure that all participants received the same information in the same order. Researchers also reduced observer effects by leaving the experimental room while participants completed the task, though some effect may have persisted due to the presence of the eye-tracker.

Next, the quality of pre-written summaries in the Reading condition may influence how programmers subsequently read the code. The summaries were previously used in human studies [12, 39] and in studies involving automated code summarization [11, 52]. However, to further mitigate this risk, we implemented quality control checks and removed data associated with egregiously low-quality summaries. We implemented similar measures of quality control for participant summaries, in which data associated with low-quality participant summaries was excluded,

as this demonstrates poor comprehension and may be reflected in the gaze data. In addition, our results may have been influenced by decisions made during the analysis stage. For instance, we concentrated our analyses on only a subset of the 19 semantic categories that we originally considered (Section 5.1), which could have caused a streetlight effect [30]. In other words, we may have focused our search to the point at which we ignored other possibilities.

The consequence of this may be present in Section 5.4, where we found that experts and novices had significantly different *cumulative* fixation counts but not with respect to the subset of categories we considered. Previous research has found that novices' attention is more distributed, which may have influenced the cumulative differences in this study [27]. To curb the impact of this decision, however, we combined prior research with a data-driven approach. Specifically, we ranked the semantic categories based on their frequency in programmers' scan paths and cross-referenced the top resultant categories with those examined by previous code summarization [5, 70] and code comprehension research [62]. Finally, we rely on participants' self-reported measures of their experience in making our comparison between experts and novices. We have no reason to believe that participants were untruthful, but slight inaccuracies in reported expertise could conceivably add noise or outliers to our novice or expert groups. This possibility was mitigated by only comparing the top and bottom terciles of participants and excluding the middle tercile from this stage of our analyses.

8 CONCLUSION

In this study, we used eye-tracking to compare two forms of code comprehension: reading code with a pre-written summary and reading code to generate one. To form a better understanding of code summarization in both contexts, we analyzed the fine-grained semantics of where participants focused using 19 categories based on the semantics of Java. We also examined common attention sequences between these semantic categories using the scan path. Inspecting fixation data, we found that writing a code summary influences where programmers focus in the code and for how long. Using scan paths, we found parallels between the two conditions in terms of programmers' attention sequences between semantic categories. Furthermore, to explore human attention on another representation of code, we mapped participants' gaze data onto the AST. We found that programmers' visual behavior on raw code does not always match that on the AST. This disconnect between human attention on raw code and the AST suggests the potential for further research into mapping human attention onto the AST. Lastly, we found numerous differences between novices and experts in their visual behavior during these code comprehension tasks: novices generally fixate more and for longer on the code, with some notable exceptions. By analyzing human attention using fine-grained semantic information and the AST, we find both consistencies and discrepancies between two forms of code comprehension.

REFERENCES

[1] 2019. Python - SDK Reference Guide - Tobii Pro SDK Documentation. Retrieved from https://developer.tobiipro.com/python/python-sdk-reference-guide.html

[2] 2023. Tobi Pro Fusion User Manual. Retrieved from https://go.tobii.com/tobii-pro-fusion-user-manual

[3] Hervé Abdi et al. 2007. Bonferroni and Šidák corrections for multiple comparisons. *Encyclopedia of Measurement and Statistics* 3, 01 (2007), 2007.

[4] 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*. 1–9.

[5] 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.

[6] Marjan Adeli, Nicholas Nelson, Souti Chattopadhyay, Hayden Coffey, Austin Henley, and Anita Sarma. 2020. Supporting code comprehension via annotations: Right information at the right time and place. In *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 1–10.

[7] Emad Aghajani, Csaba Nagy, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, Michele Lanza, and David C. Shepherd. 2020. Software documentation: The practitioners' perspective. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 590–601.

[8] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653* (2020).

[9] Akiko Aizawa. 2003. An information-theoretic perspective of TF-IDF measures. *Information Processing & Management* 39, 1 (2003), 45–65.

[10] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740* (2017).

[11] Aakash Bansal, Zachary Eberhart, Zachary Karas, Yu Huang, and Collin McMillan. 2023. Function call graph context encoding for neural source code summarization. *IEEE Transactions on Software Engineering* (2023).

[12] Aakash Bansal, Chia-Yi Su, Zachary Karas, Yifan Zhang, Yu Huang, Toby Jia-Jun Li, and Collin McMillan. 2023. Modeling programmer attention as scanpath prediction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1732–1736.

[13] Roman Bednarik, Carsten Schulte, Lea Budde, Birte Heinemann, and Hana Vrzakova. 2018. Eye-movement modeling examples in source code comprehension: A classroom study. In *Proceedings of the 18th Koli Calling International Conference on Computing Education Research*. 1–8.

[14] Roman Bednarik and Markku Tukiainen. 2008. Temporal eye-tracking data: Evolution of debugging strategies with multiple representations. In *Proceedings of the 2008 Symposium on Eye Tracking Research & Applications*. 99–102.

[15] Jean-Francois Bergeretti and Bernard A. Carré. 1985. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 7, 1 (1985), 37–61.

[16] Birtukan Birawo and Pawel Kasprowski. 2022. Review and evaluation of eye movement event detection algorithms. *Sensors* 22, 22 (2022), 8810.

[17] Neil C. C. Brown, Pierre Weill-Tessier, Maksymilian Sekula, Alexandra-Lucia Costache, and Michael Kölling. 2022. Novice use of the Java programming language. *ACM Transactions on Computing Education* 23, 1 (2022), 1–24.

[18] Peter F. Brown, Vincent J. Della Pietra, Peter V. Desouza, Jennifer C. Lai, and Robert L. Mercer. 1992. Class-based n-gram models of natural language. *Computational Linguistics* 18, 4 (1992), 467–480.

[19] 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.

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

[21] Joseph Cesario. 2022. What can experimental studies of bias tell us about real-world group disparities? *Behavioral and Brain Sciences* 45 (2022), e66.

[22] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. 2013. SRCML: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 516–519.

[23] Robert Cordingly, Hanfei Yu, Varik Hoang, David Perez, David Foster, Zohreh Sadeghi, Rashad Hatchett, and Wes J. Lloyd. 2020. Implications of programming language selection for serverless data processing pipelines. In *DASC/PiCom/CBDCom/CyberSciTech*. IEEE, 704–711.

[24] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. 2017. Empirical study of usage and performance of Java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 389–400.

[25] Richard Craggs and Mary McGee Wood. 2005. Evaluating discourse and dialogue coding schemes. *Computational Linguistics* 31, 3 (2005), 289–296.

[26] Filipe Cristino, Sebastiaan Mathôt, Jan Theeuwes, and Iain D. Gilchrist. 2010. ScanMatch: A novel method for comparing fixation sequences. *Behavior Research Methods* 42 (2010), 692–700.

[27] Martha E. Crosby, Jean Scholtz, and Susan Wiedenbeck. 2002. The roles beacons play in comprehension for novice and expert programmers. In *PPIG*. 5.

[28] Benoît De Smet, Lorent Lempereur, Zohreh Sharafi, Yann-Gaël Guéhéneuc, Giuliano Antoniol, and Naji Habra. 2014. Taupe: Visualizing and analyzing eye-tracking data. *Science of Computer Programming* 79 (2014), 260–278.

[29] Marie Delacre, Daniël Lakens, and Christophe Leys. 2017. Why psychologists should by default use Welch's t-test instead of Student's t-test. *International Review of Social Psychology* 30, 1 (2017), 92–101.

[30] David Demirdjian, Leonid Taycher, Gregory Shakhnarovich, Kristen Grauman, and Trevor Darrell. 2005. Avoiding the "streetlight effect": Tracking by exploring likelihood modes. In *10th IEEE International Conference on Computer Vision (ICCV'05) Volume 1*, Vol. 1. IEEE, 357–364.

[31] Alan Ewert and Jim Sibthorp. 2009. Creating outcomes through experiential education: The challenge of confounding variables. *Journal of Experiential Education* 31, 3 (2009), 376–389.

[32] Franz Faul, Edgar Erdfelder, Albert-Georg Lang, and Axel Buchner. 2007. G* Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior Research Methods* 39, 2 (2007), 175–191.

[33] Benjamin Floyd, Tyler Santander, and Westley Weimer. 2017. Decoding the representation of code in the brain: An fMRI study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 175–186.

[34] Davide Fucci, Daniela Girardi, Nicole Novielli, Luigi Quaranta, and Filippo Lanubile. 2019. A replication study on code comprehension and expertise using lightweight biometric sensors. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 311–322.

[35] Golara Garousi, Vahid Garousi, Mahmoud Moussavi, Guenther Ruhe, and Brian Smith. 2013. Evaluating usage and quality of technical software documentation: an empirical study. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*. 24–35.

[36] Michael Gnatz, Leonid Kof, Franz Prilmeier, and Tilman Seifert. 2003. A practical approach of teaching software engineering. In *Proceedings of the 16th Conference on Software Engineering Education and Training, 2003 (CSEE&T 2003)*. IEEE, 120–128.

[37] Jaekyu Ha, Robert M. Haralick, and Ihsin T. Phillips. 1995. Document page decomposition by the bounding-box project. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Vol. 2. IEEE, 1119–1122.

[38] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. 223–226.

[39] Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. 2022. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. 36–47.

[40] Florian Hauser, Jürgen Mottok, and Hans Gruber. 2018. Eye tracking metrics in software engineering. In *Proceedings of the 3rd European Conference of Software Engineering Education*. 39–44.

[41] Mary Hegarty, Richard E. Mayer, and Carolyn E. Green. 1992. Comprehension of arithmetic word problems: Evidence from students' eye fixations. *Journal of Educational Psychology* 84, 1 (1992), 76.

[42] Prateek Hejmady and N. Hari Narayanan. 2012. Visual attention patterns during program debugging with an IDE. In *Proceedings of the Symposium on Eye Tracking Research and Applications*. 197–200.

[43] Xing Hu, Xin Xia, David Lo, Zhiyuan Wan, Qiuyuan Chen, and Thomas Zimmermann. 2022. Practitioners' expectations on automated code comment generation. In *Proceedings of the 44th International Conference on Software Engineering*. 1693–1705.

[44] Yu Huang, Kevin Leach, Zohreh Sharafi, Nicholas McKay, Tyler Santander, and Westley Weimer. 2020. Biases and differences in code review using medical imaging and eye-tracking: Genders, humans, and machines. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 456–468.

[45] Yu Huang, Xinyu Liu, Ryan Krueger, Tyler Santander, Xiaosu Hu, Kevin Leach, and Westley Weimer. 2019. Distilling neural representations of data structure manipulation using fMRI and fNIRS. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 396–407.

[46] Yasir Hussain, Zhiqiu Huang, Yu Zhou, and Senzhang Wang. 2020. CodeGRU: Context-aware deep learning with gated recurrent unit for source code modeling. *Information and Software Technology* 125 (2020), 106309.

[47] Sarah Jessup, Sasha M. Willis, Gene Alarcon, and Michael Lee. 2021. Using eye-tracking data to compare differences in code comprehension and code perceptions between expert and novice programmers. (2021).

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

[49] Zachary Karas, Andrew Jahn, Westley Weimer, and Yu Huang. 2021. Connecting the dots: Rethinking the relationship between code and prose writing with functional connectivity. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 767–779.

[50] Ryan Krueger, Yu Huang, Xinyu Liu, Tyler Santander, Westley Weimer, and Kevin Leach. 2020. Neurological divide: An fMRI study of prose and code writing. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 678–690.

[51] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. 2020. Improved code summarization via a graph neural network. In *Proceedings of the 28th International Conference on Program Comprehension*. 184–195.

[52] Alexander LeClair and Collin McMillan. 2019. Recommendations for datasets for source code summarization. *arXiv preprint arXiv:1904.02660* (2019).

[53] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. 2019. Reliability and inter-rater reliability in qualitative research: Norms and guidelines for CSCW and HCI practice. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–23.

[54] Nora A. McIntyre and Tom Foulsham. 2018. Scanpath analysis of expertise and culture in teacher gaze in real-world classrooms. *Instructional Science* 46 (2018), 435–455.

[55] Mónika Mészáros, Máté Cserép, and Anett Fekete. 2019. Delivering comprehension features into source code editors through LSP. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, 1581–1586.

[56] Daniel C. Molden. 2014. Understanding priming effects in social psychology: An overview and integration. *Social Cognition* 32, Supplement (2014), 243–249.

[57] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K. Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[58] Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel. 1999. Code conventions for the Java programming language. *Sun Microsystems* (1999).

[59] Patrick Niemeyer and Jonathan Knudsen. 2005. *Learning Java*. O'Reilly Media, Inc.

[60] Anneli Olsen. 2012. The Tobii I-VT fixation filter. *Tobii Technology* 21 (2012), 4–19.

[61] David Lorge Parnas. 2010. Precise documentation: The key to better software. In *The Future of Software Engineering*. Springer, 125–148.

[62] Norman Peitek, Sven Apel, Chris Parnin, André Brechmann, and Janet Siegmund. 2021. Program comprehension and code complexity metrics: An fMRI study. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 524–536.

[63] Norman Peitek, Janet Siegmund, and Sven Apel. 2020. What drives the reading order of programmers? An eye tracking study. In *Proceedings of the 28th International Conference on Program Comprehension*. 342–353.

[64] Sylvia Peißl, Christopher D. Wickens, and Rithi Baruah. 2018. Eye-tracking measures in aviation: A selective literature review. *The International Journal of Aerospace Psychology* 28, 3-4 (2018), 98–112. https://doi.org/10.1080/24721840.2018.1514978 arXiv:https://doi.org/10.1080/24721840.2018.1514978

[65] Juan Ramos et al. 2003. Using TF-IDF to determine word relevance in document queries. In *Proceedings of the 1st Instructional Conference on Machine Learning*, Vol. 242. Citeseer, 29–48.

[66] Pooja Rani, Arianna Blasi, Nataliia Stulova, Sebastiano Panichella, Alessandra Gorla, and Oscar Nierstrasz. 2023. A decade of code comment quality assessment: A systematic literature review. *Journal of Systems and Software* 195 (2023), 111515.

[67] Keith Rayner. 1998. Eye movements in reading and information processing: 20 years of research. *Psychological Bulletin* 124, 3 (1998), 372.

[68] Paige Rodeghero, Cheng Liu, Paul W. McBurney, and Collin McMillan. 2015. An eye-tracking study of Java programmers and application to source code summarization. *IEEE Transactions on Software Engineering* 41, 11 (2015), 1038–1054.

[69] Paige Rodeghero and Collin McMillan. 2015. An empirical study on the patterns of eye movement during summarization tasks. In *2015 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–10.

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

[71] Herbert Schildt. 2014. *Java: The Complete Reference*. McGraw-Hill Education Group.

[72] Hugo H. Schoonewille, Werner Heijstek, Michel R. V. Chaudron, and Thomas Kühne. 2011. A cognitive perspective on developer comprehension of software design documentation. In *Proceedings of the 29th ACM International Conference on Design of Communication*. 211–218.

[73] Timothy R. Shaffer, Jenna L. Wise, Braden M. Walters, Sebastian C. Müller, Michael Falcone, and Bonita Sharif. 2015. iTrace: Enabling eye tracking on software artifacts within the IDE to support software engineering tasks. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 954–957.

[74] Zohreh Sharafi, Alessandro Marchetto, Angelo Susi, Giuliano Antoniol, and Yann-Gaël Guéhéneuc. 2013. An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 33–42.

[75] 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.

[76] Zohreh Sharafi, Bonita Sharif, Yann-Gaël Guéhéneuc, Andrew Begel, Roman Bednarik, and Martha Crosby. 2020. A practical guide on conducting eye tracking studies in software engineering. *Empirical Software Engineering* 25 (2020), 3128–3174.

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

[78] 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.

[79] Susan Elliott Sim, Sukanya Ratanotayanon, Oluwatosin Aiyelokun, and Erin Morris. 2006. An initial study to develop an empirical test for software engineering expertise. *Institute for Software Research, University of California, Irvine, CA, USA, Technical Report# UCI-ISR-06-6* (2006).

[80] Amit Singhal et al. 2001. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.* 24, 4 (2001), 35–43.

[81] Ian Sommerville. 2001. Software documentation. *Software Engineering* 2 (2001), 143–154.

[82] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. 43–52.

[83] Ze Tang, Chuanyi Li, Jidong Ge, Xiaoyu Shen, Zheling Zhu, and Bin Luo. 2021. AST-Transformer: Encoding abstract syntax trees efficiently for code summarization. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1193–1195.

[84] William C. Thompson. 2016. Observer effects. *A Guide to Forensic DNA Profiling* (2016), 171–173.

[85] Anneliese Von Mayrhauser and A. Marie Vans. 1995. Program comprehension during software maintenance and evolution. *Computer* 28, 8 (1995), 44–55.

[86] Ruyun Wang, Hanwen Zhang, Guoliang Lu, Lei Lyu, and Chen Lyu. 2020. Fret: Functional reinforced transformer with BERT for code summarization. *IEEE Access* 8 (2020), 135591–135604.

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

[88] Marvin Wyrich, Justus Bogner, and Stefan Wagner. 2022. 40 years of designing code comprehension experiments: A systematic mapping study. *arXiv preprint arXiv:2206.11102* (2022).

[89] Chunyan Zhang, Junchao Wang, Qinglei Zhou, Ting Xu, Ke Tang, Hairen Gui, and Fudong Liu. 2022. A survey of automatic source code summarization. *Symmetry* 14, 3 (2022), 471.