2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Programmers' Visual Attention on Function Call Graphs During Code Summarization

Samantha McLoughlin*, Zachary Karas*, Robert Wallace†, Aakash Bansal‡, Collin McMillan†, Yu Huang*,
*Vanderbilt University: {samantha.m.mcloughlin, z.karas, yu.huang}@vanderbilt.edu
†University of Notre Dame {rwallac1, cmc}@nd.edu
‡Louisiana State University abansal@lsu.edu

***Abstract*—This paper studies programmer visual attention on code as it relates to underlying function call graphs during code summarization. Programmer visual attention refers to where people look when performing a software engineering task, and code summarization is the task of writing a natural language description about a section of source code. Prior work has studied programmers' visual attention during code summarization, with the vast majority of research effort placed on details in single functional units of code. There have not been any techniques developed to understand code comprehension at the project level due to the difficulty of this task, despite the nature of most real-world methods as embedded within complex project context. This paper focuses on the visual attention paid to the call graph context in which a method sits. We analyze visual attention coverage of call graphs with graph-based metrics, such as the depth that programmers traverse or the amount of coverage they attain. We use these metrics, among other means, to reevaluate an existing dataset from a previous eye-tracking study of programmers ($n = 10$) that considered basic properties of programmer visual attention in a project context. We then created a new dataset ($n = 12$) using the same procedures specifically for this paper, resulting in a total of 88 hours of recorded visual behavior on source code. We used our proposed metrics to analyze how participants' visual strategies correlated with their code summary quality, and confidence in their summaries. Interestingly, we found that higher coverage of the call graph was associated with *decreases* in both summary quality and participants' confidence.**

I. Introduction

Code summarization is the task of writing a natural language description of code. Given the importance of effective documentation for software maintenance and code comprehension [1, 2], an improved understanding of effective human strategies for summary writing can facilitate efforts to advance code summarization techniques, including automated code summarization, developer tools, and education. To study how programmers discover and use relevant information during code summarization tasks, we consider programmers' *visual attention*, which refers to where a programmer looks while completing an SE task.

We study programmers' visual attention using *eye tracking*, which is a technology that has illuminated programmers' strategies and mental models for comprehending and summarizing source code [3, 4]. The vast majority of code comprehension research has focused on isolated functional units of code [5, 6, 7, 8], and existing metrics used in SE to study code comprehension were developed for studying visual attention on isolated code units [9]. Our understanding of code comprehension at the larger project level is limited partly because we lack meaningful eye-tracking metrics to characterize programmers' code comprehension behaviors at this systematic level. Previous work that has visualized and described large projects using the *function call graph* [10, 11, 12], which is a directed graph representing call relations between functional units in code. Using this structure, we propose to fill the gap by exploring programmers' visual attention on code as it relates to underlying call graphs, thereby deepening our foundational understanding of cognitive processes of code comprehension at the project level. We hypothesize and demonstrate that the function call graph is a promising structure for analyzing how programmers gather contextual information about underlying connections between functional units of code in larger projects.

Studying how programmers develop a high-level understanding of code within a project also has strong implications for automated code summarization. In particular, prior research has found that deep learning approaches for code summarization have substantially benefited from both including the call graph to provide models with context [13], and incorporating human eye-tracking data to indicate connections in code that are important from a human perspective [14]. Thus, integrating human eye-tracking data on code with underlying call graphs has potential for guiding deep learning models with human patterns of code comprehension within a larger project context.

In this study, we analyzed programmers' visual attention on the underlying call graph as they summarized methods in full open source Java projects. Specifically, we first developed metrics that integrate eye-tracking data with the call graph, and deployed those metrics to perform novel analyses of programmers' summarization strategies on an existing dataset of human eye-tracking data collected from the University of Notre Dame ($n_1 = 10$) [15]. That dataset explored basic measures of programmers' visual attention on code in a project level context, but overlooked programmers' visual behaviors on call graphs. We then tested the replicability of our design and findings with a new human study ($n_2 = 12$). Specifically, we collected a new dataset for this study by following the same experimental procedures and recruiting participants from Vanderbilt University, which is a comparable institution. In both datasets, each participant could take part in the study for five approximately 90 minute sessions to summarize methods in five separate projects. As a result, we applied our analyses on a total of 88 hours of programmers' recorded visual behavior. For both datasets, we tested how our metrics of call

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00089

graph coverage were correlated with downstream summary quality, as well as participants' confidence in their summaries. By examining participants' confidence, we could explore whether certain visual strategies might lead programmers to overestimate the quality of their summaries, and potentially contribute inadequate documentation.

Our study is guided by three research questions:

- RQ1. When summarizing a method in context, how do programmers interact with the method's call graph?
- RQ2. How do strategies for traversing a method's call graph affect the quality of summaries?
- RQ3. How do strategies for traversing a method's call graph affect participants' confidence in their summaries?

To first validate a crucial hypothesis for this study that programmers comprehend code at the project level by relying on the call graph, we initially analyzed the eye-tracking data and found that participants spent as much as 79.2% of their total time fixating on target methods and their call graphs, suggesting that programmers do preferentially use this structure as a comprehension tool for code situated within larger projects. Interestingly, as the first study to investigate programmers' visual attention on call graphs in complete projects, we found that higher call graph coverage was correlated with both lower-quality summaries and lower confidence in ones' summaries. However, we found nuance when participants traversed either up (callers) or down (callees) in the call graph; the former contains high-level usage examples, while the latter contains low-level implementation details.

The contributions of this study are summarized as follows:

- A new human eye-tracking study for replication of the proposed attention analysis on call graphs following the same study procedures as a previous study.
- An implementation of our study design with two datasets, which together provide further validation and facilitate the analysis of project-level code comprehension.
- Novel metrics to integrate eye-tracking data with call graphs.
- Findings that increased coverage of the call graph was correlated with *decreases* in summary quality and participant confidence.
- Publicly available data and analysis code here[1]

II. Related Work

In this section, we describe how our work relates to research on call graphs, eye-tracking, and code comprehension.

Call Graphs and Automated Code Summarization. The call graph is a directed graph that represents a software system with nodes representing a software unit (e.g., a method or function) and edges representing a direct relationship, such as a function call [12]. For a given method, its callees are the units that the method calls, while its callers are the units that call the method. As such, the **callee graph** represents low-level methods used in the implementation of the given method, and the **caller graph** represents high-level methods that use the given method. Previous work has created developer tools based on the call graph, including 2D visualizations, interactive visualizations, and clustering-based approaches [10, 11, 12]. Researchers have also leveraged context from the call graph to advance automated code summarization work [13, 16, 17]. Specifically, Bansal et al. introduced a Graph Neural Network that was trained on target methods' call graphs to improve the model's contextual understanding [18]. In this study, we integrate human eye-tracking data with the call graph to understand programmers' project-level comprehension patterns.

Code Comprehension. Efforts to incorporate human attention patterns into automated code summarization methods rely on an understanding of human code comprehension. Previous work has developed mental models of how programmers comprehend and navigate codebases, which seek to explain how programmers search for and identify relevant sources of information in code. For example, many models are based on theories of "information foraging," where programmers search through relevant areas of code to maximize "information per unit cost" [19]. From an information foraging perspective, a call relation (e.g., a call graph edge) may suggest that two areas of code are mutually relevant [20]. In this study, we consider how traversing call relations during code summarization can provide a window into information foraging strategies.

Eye-tracking in Software Engineering. To understand programmers' code comprehension strategies, researchers often use eye tracking, which allows researchers to analyze data on human **fixations** (gazes that remain stable for 100-300ms) and **saccades** (rapid eye movements lasting around 40-50ms) [21]. While little cognitive processing typically happens during saccades [22], humans are theorized to process visual information during fixations, which can provide a window into human cognition [21]. Researchers also consider **scan paths**, which are defined as the sequential ordering of fixations [21], and give insight into contingencies between fixations. Eye-tracking technology has been applied to various code comprehension tasks, including debugging and code review [23, 24]. Previous work has also applied eye-tracking analysis to code summarization tasks, identifying key components of code, mapping gaze patterns onto the abstract syntax tree, and testing mental cognition models (e.g., "top-down" or "bottom-up") [7, 25, 26]. Wallace et al. presented a dataset of eye-tracking data gathered from code summarization tasks in project context, and presented baseline metrics and conclusions from their study [15]. We build off their work by considering eye-tracking metrics that relate eye-tracking data to the call graph.

III. Methods

This section outlines our methodology for eye-tracking data collection and analysis, as well as our call graph analyses.

A. *Study Design.*

To collect eye-tracking data from programmers summarizing Java methods in the context of a larger project, we conducted a human study where programmers were given five open source Java projects and asked to summarize eight

[1] https://github.com/samanthamcloughlin/call-graph-visual-attention

methods from each. Each programmer could participate in five sessions to summarize methods from each of the projects (taking into consideration of the length for each session to avoid fatigue). This design follows that of Wallace et al. [15], which includes additional study details. In this study, we applied novel analysis techniques mapping human gaze data onto method call graphs, both to the data from the original study from Wallace et al. (Study 1), and to a new dataset collected at Vanderbilt University (Study 2). The two individual studies allow us to test the replicability of findings between Study 1 and Study 2. Since these studies follow the same design, we also report results from combining the datasets, which grants more statistical power to potentially uncover important findings on programmers' visual attention on call graphs.

Participants. For our new study (Study 2), we collected over 48 total study hours, which amounted to 45 hours of time-on-task eye-tracking data across 12 participants with at least one year of Java experience each. Participants were recruited via in-class presentations as well as email, and compensated 60 USD per session. We obtained ethics approval (IRB: #220604), and collected informed consent which clearly indicated participants could leave at any time. As indicated in Table I, participants had an average of 4.6 years of programming experience, two participants identified as female, and ten identified as male. Three identified as non-native fluent English speakers, and nine as native English speakers. Across both studies (Study 1 and Study 2), there were 22 participants, over 100 total study hours, and over 88 time-on-task hours.

TABLE I: Participant demographic information for both studies separately and combined.

Study	Male	Female	Native English	Non-Native English	Avg. Experience
Study 1	6	4	4	6	5.0 years
Study 2	10	2	9	3	4.6 years
Combined	16	6	13	9	4.8 years

Tasks. For each session, we presented participants with eight methods from one of five open source Java projects ranging from games to machine learning toolboxes. The projects included Scrimage (image manipulation library), MLTK (collection of machine learning algorithms), OpenAudible (desktop application for audiobooks), MALLET (statistical package for natural language processing), and FreeCol (strategy game). We used the same projects and methods as those selected by Wallace et al., who aimed for sample diversity and generalizability [15]. Those researchers chose open source projects with 90% of their source code in Java (due to its wide usage in industry [27]). To keep participants' navigation and comprehension self-contained, projects were excluded that depended heavily on other codebases. The chosen projects ranged in age by 18 years and were between 5k and 128k lines of code. Methods were chosen that were non-trivial (i.e., excluded getters and setters), were called at least twice, and performed some core functionality in the project.

For our analyses, we excluded methods without a relevant call graph. There were 11 methods with no callees, and 0 methods with no callers. Therefore, our analysis included 29 methods for callee graph analysis, and 40 methods for caller graph analysis. Data from 57 method summaries that were incomplete due to incomplete eye-tracking data or other technical errors were also excluded. The amount of data we analyzed in the study (88 hours total, 45 hours for the new study) was calculated after excluding this data.

Procedure. Following procedures from Study 1 [15], participants were given 90 minutes for each session and instructed to generate three sentence summaries for a given method within a project. The first described the purpose of the method, the second described the method's specific functionalities, and the third described its use within the overall project. Participants were instructed to rate their confidence in each summary on a scale from 1 to 5. Experiments were run in an office room with natural lighting.

Eye Tracking. To record eye-tracking data, we used a Tobii Pro Fusion eye tracker mounted at the bottom of a Spectre 24-inch monitor with a resolution of 1920x1080 and 60Hz screen refresh rate. The eye tracker records data at a frequency of 120Hz, and has an accuracy of 0.03° and precision of 0.04° in optimal conditions. Projects were presented to participants in Eclipse, and we used the iTrace suite to interface with the eye tracker. Specifically, we used (1) iTrace Core, which manages core eye-tracking and screen recording functionality, (2) the iTrace Eclipse plugin, which connects data from iTrace Core to the IDE and specific file information, and (3) the iTrace toolkit, which processes eye-tracking data [28, 29, 30].

Qualitative Annotation. In this study, we analyzed how visual patterns with respect to the call graph affected downstream summary quality, so we rated participants' summaries to obtain quality scores. Specifically, each summary was scored by two authors independently on four subscore categories (rated out of 5): accuracy, conciseness, completeness, and clarity [15]. For every question, the researchers deducted a point for each instance of inaccuracy, incompleteness, and so on. Each subscore category was assessed independently of other factors, and a total quality score was calculated as the sum of the subscores (rated out of 20). We used the original authors' ratings of participants' summaries for those in Study 1, and two authors in this study with 7 and 10 years of coding experience rated participants' summaries for Study 2. In both Study 1 and Study 2, graders met to discuss and resolve all conflicting scores, and agreed upon all final scores.

B. Eye Tracking Metrics

For this study mapping visual behavior onto method call graphs, we use standard measures from eye-tracking research of fixations, saccades, and scan paths [21]. To distinguish fixations from saccades, we used the I-VT velocity filter provided by the iTrace toolkit [28]. This filter identified a fixation as consecutive gazes with velocities below 50 pixels/ms and an overall duration above 80 ms. Otherwise, an eye movement was identified as a saccade. Following standard procedures, we computed the fixation counts and durations. We use consecutive fixations to compute scan paths to analyze programmers' strategies for traversing method call graphs.

C. *Call Graph Analysis*

Call Graphs. We define the **callee graph** of a target method as that which contains *methods that the target method calls*, and the **caller graph** of a method as that which contains *methods that call the target method*. We represent each method within both types of call graphs as a node, and each call as an edge. We define a method's depth within the call graph as the number of calls between it and the target method being summarized. If the same method appears at multiple depths within a given call graph, we distinguished these nodes based on their depths.

Call Graph Generation. To generate callee and caller graphs for each target method summarized by participants, we used IntelliJ's "Call Hierarchy" tool. To reduce extreme variation in call graph size, in our analyses we only included methods within five calls of the target method. Participants did not look five or more calls deep in 98.5% of trials for the callee graph, and 97.3% of trials for the caller graph.

Method Annotation. To map eye-tracking data onto call graphs, we annotated each fixation with either the signature of the method being fixated on, or a label indicating that the fixation corresponded to a non-method. We defined the bounds of a method as starting from the row and column of its declaration, and ending at its closing bracket. We considered all coordinates between these two points as belonging to that method. For fixations on nested methods, we annotated the fixation with the inner method.

D. *Call Graph Eye Tracking Metrics*

In this section, we describe how we adapted established eye tracking metrics to the purpose of call graph analysis. Broadly, we used these metrics to perform pairwise statistical tests, and as predictor variables in mixed effects linear models.

Fixation Proportions. To understand how participants interacted with the call graph, we calculated the proportion of time that they spent fixating on the following code categories: target method, call graph methods, non-call graph methods, and non methods. We used these fixation proportions to indicate whether programmers use call relationships (i.e. edges of the call graph) as meaningful avenues for potentially relevant information. The extent to which programmers' visual attention adheres to methods in the call graph can indicate whether the call graph is a significant underlying cognitive tool for programmers (e.g., for theories such as information foraging). We define fixation proportion as the sum of all participants' fixation durations for a code category divided by the sum of all fixation durations.

Call Graph Scan Paths. Using method annotations for each fixation, we calculated participants' scan paths with respect to call graphs as successive fixations in the same call graph. This also provided an indication whether participants traversed edges defined by the call graph, or "created" their own edges. We also considered programmers' visual strategies by calculating the time participants spent fixating on each node, and the number of times an edge was traversed. Figure 1 presents a visualization of this mapping.

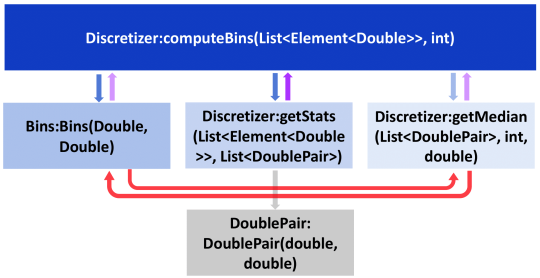

Discretizer:computeBins(List<Element<Double>>, int)
Bins:Bins(Double, Double)
Discretizer:getStats (List<Element<Double >>, List<DoublePair>)
Discretizer:getMedian (List<DoublePair>, int, double)
DoublePair: DoublePair(double, double)

Fig. 1: A visualization of one participant's scan path mapped onto a call graph. Colored edges and nodes are visited by the participant, while gray edges and nodes are not. Darker shades represent greater total fixation duration for nodes, and a greater number of total traversals for edges. Blue arrows represent edges in the call graph. Purple arrows represent edges "created" by participants that reverse the direction of call graph edges, while red arrows represent created edges that are not part of the call graph.

Depth. We calculated how "deep" into the call graph participants looked, meaning how many calls away from the target method they visited. In reporting our findings on depth, we include mean (μ) and the 90^{th} percentile (P_{90}) to illustrate average and extreme behavior, respectively. For pairwise t-tests, we report the t-statistic (t), p-value (p), and Cohen's d effect size (d). For cases when we ran multiple t-tests, we report Bonferroni corrected p-values (q) instead of uncorrected p-values (p). We denote statistics for Study 1, Study 2, and the combined dataset with subscripts 1, 2, and c, respectively.

E. *Call Graph Coverage Metrics*

To quantify participants' traversal of call graphs, we calculated the following metrics:

Maximum Depth. We define maximum depth as the maximum depth that a participant visited in a given call graph. We defined a method as *visited* by participants if they fixated upon it at least once.

Node Coverage. We define node coverage as the number of nodes in a call graph that participants visited at least once.

Edge Coverage. We define edge coverage as the number of edges in a call graph that participants traversed at least once. A traversal of an edge between two nodes is defined as a fixation on the first node directly followed by a fixation on the second node, with no fixations in between.

Weighted Node Coverage. We calculated weighted node coverage as the weighted proportion of the nodes in the call graph that were visited at least once, where nodes more calls away from the target method are given successively smaller weights. We used this weighted metric because, when using an unweighted proportion of nodes covered, participants consistently exhibit low coverage scores on methods with extensive call graphs. As such, unweighted coverage metrics may become a measure of (inverse) call graph size, rather than an informative model of call graph coverage. Furthermore, validated by programmers' gaze patterns (see Section IV-A), which demonstrates how attention tends to fade for methods further away from the target method in the graph, this weighted measures of call graph coverage aligns with the observation that methods far away from the target method of a call graph (i.e. a higher depth) are less central to understanding the target method than methods close to the target method (i.e. a lower depth). This reflects the premise that attention spreads over the callee and caller graph from the target method (the base node of attention). We calculated weighted node coverage separately for the callee and caller graphs. We use the target method node as the base node of attention for both graphs, with a weight of 1 ($w_0 = 1$). Each outgoing edge from a node n propagates a weight of w_n/e_n to its adjacent nodes (callees for the callee graph, and callers for the caller graph), where w_n is the weight of node n and e_n is the number of outgoing edges from node n. Therefore, weighted node coverage is defined by the following weighted arithmetic mean:

$$\frac{\sum_{n=0}^{n=N} w_n v_n}{\sum_{n=0}^{n=N} w_n}$$

where N is the number of nodes in the call graph, w_n is the weight of node n. The symbol $v_n = 1$ if n was visited at least once, $v_n = 0$ otherwise.

Weighted Edge Coverage. Similar to the definition of weighted node coverage, we calculated weighted edge coverage as the weighted proportion of the edges in the call graph that were traversed at least once, where edges more calls away from the target method are given successively smaller weights. Each edge outgoing from the base node of attention (the target method) has a weight of $1/e_{n=0}$ where $e_{n=0}$ is the number of outgoing edges from the target method. If a node n has incoming edge e, each outgoing edge from node n is propagated the weight w_e/e_n, where w_e is the weight of the incoming edge e and e_n is the number of outgoing edges from node n. Therefore, weighted edge coverage is defined by the following weighted arithmetic mean:

$$\frac{\sum_{e=0}^{e=E} w_e t_e}{\sum_{e=0}^{e=E} w_e}$$

where E is the number of edges in the call graph, w_e is the weight of edge e. The symbol $t_e = 1$ if e was traversed at least once, $t_e = 0$ otherwise.

F. *Absolute Confidence Difference*

To examine whether participants' confidence in their own summaries aligned with their actual summary quality, we calculated the differences between their confidence scores and summary quality scores. Formally, we define absolute confidence difference as the absolute value of the difference between the percentile of a participant's confidence and the percentile of a participant's summary quality score. Formally, absolute confidence difference for participant i on trial j is defined as $|P(c_{ij}) - P(s_{ij})|$ where $P(c_{ij})$ refers to the percentile of participant i's confidence c in their summary for trial j relative to all other confidence ratings, and $P(s_{ij})$ refers to the percentile of participant i's summary quality score s for trial j relative to all other scores. We calculated percentiles across all trials, and used percentiles to enable comparison between confidence and summary quality.

G. *Mixed Effects Model*

To analyze the relationship between our metrics and downstream outcomes of summary quality and confidence, we used mixed effects linear regression models, which allow us to control for unobserved variation across individuals. We included the participant as a random effect in all models, and the study number as a random effect for models using the combined dataset. We used summary quality, subscore measures, and confidence as predictor variables, and report results for node and edge coverage, weighted and unweighted. We performed modeling separately for the callee graph and the caller graph, and applied Bonferroni correction to account for multiple comparisons and report corrected p-values (q) with the regression coefficients.

Standardizing model coefficients can skew the results of mixed effects regression, so at the cost of interpretability, we do not standardize model coefficients. Thus, only the sign (positive or negative) of the coefficients reported should be meaningfully interpreted, not their relative magnitudes.

IV. Results

In this section, we present the results of applying these call graph analysis techniques to the datasets from Study 1 and Study 2.

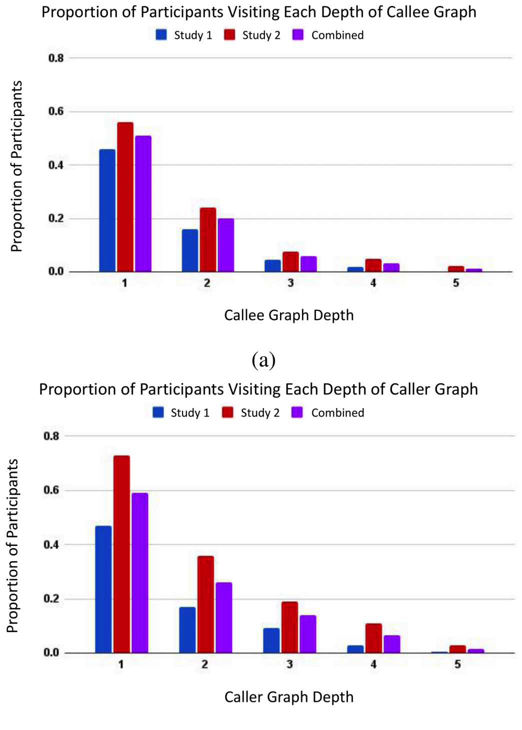
A. *RQ1: Call Graph Interaction*

To examine whether the call graph is a relevant mechanism for understanding code comprehension, we extracted trends in how participants' gaze patterns covered the call graph of the target method they were summarizing. We tested the proportion of time that participants spent examining the call graph and how deep into the call graph they searched. To determine the proportion of time that participants spent examining the call graph, we calculated the fixation proportions for the target method, call graph methods (both callee and caller), non-call graph methods, and non methods.

TABLE II: Proportion of time participants fixated on the target method, call graph methods, non-call graph methods, and non methods for Study 1, Study 2, and the combined dataset.

Study	Target Method	Call Graph Methods	Non-Call Graph Methods	Non Methods
Study 1	0.6675	0.1371	0.1313	0.0641
Study 2	0.5786	0.2026	0.1679	0.0508
Combined	0.6189	0.1729	0.1513	0.0568

As shown in Table II, participants spent the most time fixating on the target method (61.89% of total fixation duration time). Call graph methods represented the second highest proportion of total fixation duration time (17.29%), which is similar to the proportion of total fixation durations spent on non-call graph methods (15.13%). Initially, this evidence



(a)

(b)

Fig. 2: Proportion of participants that fixated at least once at each depth level of the (a) callee graph and (b) caller graph. Depth 0 is excluded since all participants looked at the target method, so the proportion is trivially 1.

might suggest that participants did not preferentially interact with call graph methods. However, call graph methods comprised, on average, approximately 1.57% of all methods in each project. If the call graph were irrelevant, we would expect the proportion of time spent on call graph methods to be lower than that of non-call graph methods. Additionally, the target method is a member of its own call graph, so 79.2% of the total fixation duration can be mapped onto the call graph. Therefore, these results align with our hypothesis that programmers do preferentially look to the call graph as a tool for comprehending methods in context. These habits of human programmers affirm that the call graph might provide helpful context for code summarization agents as well.

We also analyzed how "deep" into the call graph programmers traversed, which can signal where in the call graph programmers find relevant information [13]. We report the average (μ) and 90^{th} percentile (P_{90}) of participants' maximum fixation depths in Table III. Prior literature has assumed that most information programmers need to comprehend a function is within two calls of that function [13]. Our results support this assumption, as participants did not visit methods more than two calls away from the target method in 93.9% of trials for the callee graph and 86.0% of trials for the caller graph. Interestingly, participants looked farther than this "two calls" assumption for the caller graph more than twice as much (14.0%) compared to the callee graph (6.1%). This pattern was reflected for maximum depth as well, where participants tended to fixate at a greater maximum depth for the caller graph ($\mu = 1.077, P_{90} = 3.0$) compared to the callee graph ($\mu = 0.935, P_{90} = 2.0$). This difference, driven by participants in Study 2, suggests that programmers searching for relevant information may look further into the methods that call the target method (callers), as opposed to the methods that are called by the target method (callees). For automated code summarization, these trends indicate the extent of the caller and callee graphs that may be helpful to provide to an agent.

TABLE III: Maximum depth of the call graph fixated on at least once for Study 1, Study 2, and the combined dataset and reported separately for callee and caller graphs.

	Callee Depth		**Caller Depth**	
Study	**Mean**	90^{th} **Percentile**	**Mean**	90^{th} **Percentile**
Study 1	0.795	2	0.767	2
Study 2	1.095	2	1.422	4
Combined	0.937	2	1.080	3

We further examined this difference by looking at the proportion of participants who fixated at least once at each depth. We report these distributions for the callee graph and caller graphs in Figure 2.

While participants viewed progressively fewer methods at greater depths for both the caller and callee graphs, participants traversed deeper, on average, into the caller graph. Applying t-tests confirmed that this difference was statistically significant for both the maximum depth ($t = 2.206, p < 0.05, d = 0.125$) and the average depth ($t = 2.988, p < 0.01, d = 0.170$) of the combined dataset. These results suggest that human programmers may search further into the caller graph than into the callee graph to find information when summarizing code. This suggests that code in the callee graph diminishes in utility for human programmers faster than that in the caller graph. Callee graph methods are likely to contain implementation details of methods called by the target method. By contrast, caller graph methods are likely to contain examples of the target method being used, and the context in which the target method (or its callers) are called.

Finally, we analyzed how much of the callee and caller graphs participants covered using the metrics of node coverage, weighted node coverage, edge coverage, and weighted edge coverage. As seen in Table IV, we find no strong evidence of a difference between callee and caller graphs in terms of coverage trends. Applying t-tests to the combined dataset, the difference between callee and caller graph was significant for weighted node coverage ($t = 2.798, q < 0.05, d = 0.159$), indicating that participants, on average, covered a greater weighted proportion of methods in the caller graph than in the callee graph. Participants may cover a similar number and proportion of methods in both the callee and caller graphs, but their coverage may tend to occur at greater depths for the caller graph.

TABLE IV: ***Callee and Caller Graph Coverage.*** Mean values for node coverage, weighted node coverage, edge coverage, and weighted edge coverage for both callee and caller graphs across both studies and the combined dataset.

	Node Coverage		Weighted Node Coverage		Edge Coverage		Weighted Edge Coverage	
Study	**Callee**	**Caller**	**Callee**	**Caller**	**Callee**	**Caller**	**Callee**	**Caller**
Study 1	2.082	1.953	0.468	0.411	0.736	0.584	0.133	0.129
Study 2	2.629	3.039	0.512	0.509	1.155	1.189	0.189	0.223
Combined	2.342	2.472	0.489	0.458	0.935	0.873	0.160	0.174

TABLE V: ***Call Graph Coverage and Summary Quality.*** Results of mixed effects linear regressions with summary score as the outcome variable. We modeled each predictor separately, and report the resulting coefficients and statistical significance.

Graph	Study	Node Coverage	Weighted Node Coverage	Edge Coverage	Weighted Edge Coverage
Caller	Study 1	0.001	-1.051	-0.024	-0.365
	Study 2	0.006	-0.572	-0.055	-0.678
	Combined	0.010	-0.794	-0.030	-0.497
Callee	Study 1	***-0.249	-1.405	**-0.335	-0.566
	Study 2	-0.137	-1.570	-0.242	-1.142
	Combined	***-0.183	*-1.396	**-0.269	-0.760

$*q < 0.05 ** q < 0.01, ***q < 0.001.$

Participants preferentially fixated on methods from the call graph, where 61.89% of fixations were spent on the target method and 17.29% on call graph methods for a total of 79.2%. Participants traversed deeper into the caller graph than the callee graph on average ($p < 0.01$), and at the maximum ($p < 0.05$).

B. *RQ2: Call Graphs and Summary Quality*

Next, we tested whether increased call graph coverage (as defined by the aforementioned metrics) is associated with a significant difference in summary quality. For this purpose, we used mixed effects linear regression modeling.

Callee Graph Coverage and Summary Quality. To examine how callee graph coverage relates to summary quality, we used multiple measures of callee graph coverage: weighted node coverage, weighted edge coverage, node coverage, and edge coverage. We included both weighted and unweighted metrics to test whether our results were consistent despite possible variance in call graph size. More broadly, node coverage indicates how much of the call graph was covered, while edge coverage indicates whether participants traversed call relations defined within the graphs, or "created" their own edges in traversing the call graph.

Results are reported in Table V, but we found that increased coverage of the callee graph was significantly negatively associated with summary quality. Specifically, in Study 1 and the combined dataset, weighted node coverage ($q_c < 0.001$) and edge coverage ($q_1 < 0.01$, $q_c < 0.01$) significantly predicted lower summary quality scores. Node coverage significantly predicted lower quality scores in the combined dataset ($q_c < 0.001$), whereas weighted edge coverage did not significantly predict summary quality. These unexpected results do not align with our hypotheses, but may suggest that information foraging techniques that focus on the callee graph may be ineffective. The code found in the callee graph may be less informative than alternate sources of information (e.g., the target method, similar methods, or project context).

Callee Graph and Summary Subscores. We also examined the relationship between callee graph coverage metrics and each subscore (accuracy, conciseness, clarity, and completeness). We expect the effects on individual subscores to be weaker than the effects on total scores, so here we analyzed the combined dataset for increased statistical power and report results in Table VI.

We found evidence suggesting that increased call graph coverage had no significant effect on completeness, but significant negative effects on conciseness and clarity. Specifically, we found that node coverage was significantly negatively associated with clarity ($q_c < 0.05$), and weighted edge coverage was significantly negatively associated with conciseness ($q_c < 0.05$). We also found that edge coverage had negative effects on both conciseness ($q_c < 0.01$) and clarity ($q_c < 0.001$). These results suggest that greater coverage of the callee graph is associated with less concise and clear summaries. These results may provide insight into the negative relationship between callee graph coverage and summary quality. Moreover, callee graph coverage does not affect completeness, which reasonably suggests that programmers who thoroughly investigate the call graph will be thorough in their summary.

Furthermore, we found that the effects do not uniformly impact the subscores. Consider the linear regression coefficients for edge coverage as an example. Completeness demonstrates a mellow downward slope ($\beta = -0.011$), but the subscores of conciseness ($\beta = 0.094$) and clarity ($\beta = 0.088$) were more steeply influenced by edge coverage. This suggests that higher edge coverage within the callee graph is associated with stronger detrimental effects on conciseness and clarity compared to completeness and accuracy.

Caller Graph Coverage and Summary Quality.

Next, we examined whether traversing the caller graph had a different influence on summary quality compared to the

TABLE VI: ***Callee Graph Coverage and Summary Quality Subscores.*** Results of mixed effects linear regressions with each subscore (accuracy, conciseness, completeness, clarity) as the outcome variable. We modeled each predictor separately, and report the resulting coefficients and statistical significance.

Subscore	**Node Coverage**	**Weighted Node Coverage**	**Edge Coverage**	**Weighted Edge Coverage**
Accuracy	-0.047	-0.340	-0.068	-0.179
Conciseness	-0.050	-0.436	****-0.094**	***-0.428**
Completeness	-0.029	-0.259	-0.011	0.078
Clarity	***-0.052**	-0.326	****-0.088**	-0.197

$*q < 0.05 ** q < 0.01, ***q < 0.001.$

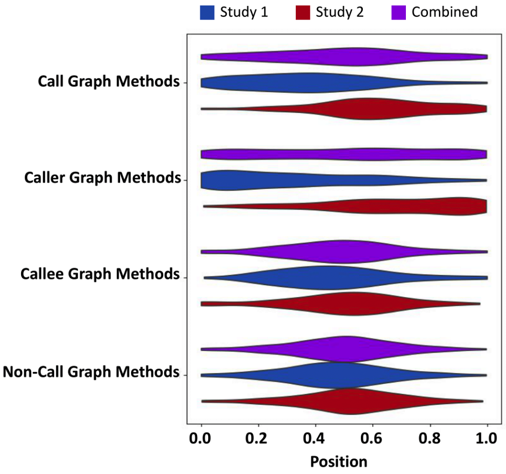


Fig. 3: Distribution of average position of fixations on code categories during each trial.

callee graph. Results are reported in Table V, but we found no significant correlations between metrics for caller graph coverage and summary quality. These findings support the idea that traversing the callee graph has a much stronger negative influence on summary quality. One possible explanation is that the callee graph is more likely to contain granular information that explains only a small portion of the target method, while the caller graph contains contextual examples of how the target method is used. Alternatively, programmers may perceive the callee graph as more useful than the caller graph if they are struggling to understand the target method.

To test these two possibilities, we conducted a follow-up analysis examining the relative order in which participants fixated on the call graph. If turning to the callee graph is a reaction to confusion (as opposed to the cause), we would expect to see callee graph fixations occur later in the trial. To this end, we calculated the average positions of fixations relative to each trial for each code category (e.g., callee graph methods, target method). We define the position of a fixation as f_{prev}/f_{trial} where f_{prev} refers to the total duration of fixations occurring previously, and f_{trial} refers to the total fixation duration of the trial. We find little evidence based on the combined dataset suggesting that fixations on the callee graph occur later in the trial. Instead, the distributions for all categories center around 0.5, as shown in Figure 3. Interestingly, we find variations between Study 1 and Study 2 with respect to caller graph methods, suggesting that participants in Study 2 examined caller graph methods later during the trials, on average. Nonetheless, our results suggest that callee graph coverage may not be a reaction to participants' confusion. If callee graph coverage does hinder code comprehension and result in lower quality summaries, programmers may wish to err away from the callee graph while writing summaries. Additionally, in the context of automated code summarization, researchers may also have reason to err away from training agents to rely heavily on callee graph context.

> Higher callee graph coverage was significantly correlated with decreases in summary quality, as measured by weighted and unweighted metrics of node and edge coverage. Higher caller graph coverage had no significant effect on summary quality.

C. *RQ3: Call Graphs and Confidence*

To understand how call graph coverage is related to participants' confidence in the quality of their summaries, we analyzed the relationships between call graph coverage and both confidence and absolute confidence difference. By examining programmers' confidence (confidence) and the "accuracy" of this confidence (absolute confidence difference), we can assess how programmers' strategies for traversing the call graph influence their self-estimation. Informally, if examining the caller graph leads programmers to overestimate their understanding, programmers may contribute documentation that is detrimental or counterproductive. Alternatively, if certain strategies lead programmers to accurately assess their summary quality, they may be more likely to consult outside resources when necessary. Here we used mixed effects modeling again, but with participants' own confidence ratings as our outcome variable.

Callee Graph Coverage. First, we tested how callee graph metrics predict confidence with a mixed effects regression model. We report results in Table VII, but found that increased coverage of the callee graph was significantly negatively associated with summary confidence. These results were consistent across both studies, and across measures of call graph coverage. We found that decreased confidence was significantly predicted by increased node coverage ($q_1 < 0.01$, $q_c < 0.01$), weighted node coverage ($q_1 < 0.05$, $q_2 < 0.01$, $q_c < 0.001$), and weighted edge coverage ($q_c < 0.05$). Edge coverage significantly predicted decreased confidence in Study

TABLE VII: ***Call Graph Coverage and Confidence.*** Results of mixed effects linear regressions with confidence as the outcome variable. We modeled each predictor separately, and report the resulting coefficients and statistical significance.

Graph	Study	Node Coverage	Weighted Node Coverage	Edge Coverage	Weighted Edge Coverage
Caller	Study 1	-0.041	**-0.780	-0.075	-0.464
	Study 2	-0.035	-0.487	-0.075	-0.339
	Combined	-0.038	***-0.641	-0.076	*-0.402
Callee	Study 1	**-0.103	*-0.890	***-0.204	-0.490
	Study 2	-0.072	**-1.056	-0.090	-0.568
	Combined	**-0.087	***-0.961	***-0.136	*-0.521

$*q < 0.05$ $**q < 0.01$, $***q < 0.001$.

TABLE VIII: ***Call Graph Coverage and Absolute Confidence Difference.*** Results of mixed effects linear regressions with absolute confidence difference as the outcome variable and node coverage, weighted node coverage, edge coverage, and weighted edge coverage of the caller graph as predictors. Each predictor is modeled separately, and its resulting coefficient and p value are reported.

Graph	Study	Node Coverage	Weighted Node Coverage	Edge Coverage	Weighted Edge Coverage
Caller	Study 1	-1.055	-3.762	-0.899	-1.303
	Study 2	-1.081	-21.298	*-4.149	-15.761
	Combined	-1.103	-11.158	-2.670	-7.493
Callee	Study 1	-1.311	8.654	-1.386	7.420
	Study 2	-0.384	-8.558	-0.289	0.172
	Combined	-0.877	0.156	-0.770	3.620

$*q < 0.05$ $**q < 0.01$, $***q < 0.001$.

1 ($q_1 < 0.001$) and the combined dataset ($q_c < 0.001$), but not Study 2. These results present strong evidence suggesting that greater coverage of the callee graph is associated with lower confidence in the resulting summary. perhaps due to low-level implementation details present in the callee graph.

Caller Graph Coverage. Next, we examined how caller graph metrics predict confidence. We report results for regressions in Table VII, and found in the combined dataset that lower confidence was significantly predicted by greater weighted edge coverage ($q_c < 0.01$) of the caller graph. Greater weighted node coverage significantly predicted lower confidence in Study 1 ($q_1 < 0.01$) and the combined dataset ($q_c < 0.001$). Interestingly, we found evidence suggesting that the unweighted measures of edge and node coverage were significantly associated with a difference in confidence. This difference between weighted and un-weighted metrics perhaps suggests that the *proportion* of the caller graph that programmers traverse may have an influence on programmers' confidence, where a higher amount of coverage is again associated with lower confidence. We next analyze whether this decrease in confidence is justified by accounting for the quality of participants' summaries.

Callee Graph and Absolute Confidence Difference. To further investigate whether the association with low confidence is due to better self-assessment, we analyzed how callee graph coverage predicts absolute confidence difference. We report results for regressions in Table VIII, but found no association between absolute confidence difference and callee graph coverage metrics. With neither a positive nor a negative association between callee graph coverage metrics and absolute confidence difference, we did not find evidence to suggest that programmers are more or less accurate in their self-estimation when traversing the callee graph for contextual information. Therefore, programmers' decreased confidence is likely attributable to the lower quality of their summaries, rather than more accurate self-estimation.

Caller Graph and Absolute Confidence Difference. Next, we analyzed how caller graph metrics predict absolute confidence difference. We report results for regressions in Table VIII, but found evidence to suggest that edge coverage in the caller graph may be associated with lower absolute confidence differences (e.g., better self-estimation). Specifically, in Study 2, lower absolute confidence difference was associated with greater edge coverage ($q_2 < 0.01$). These results were not replicated between the two studies, but perhaps suggest that programmers may have a more accurate self-estimation of their summary quality when they traverse edges of the caller graph. We are hesitant to draw strong conclusions, but compared to callee graph coverage, this evidence may reasonably suggest that programmers develop a more complete understanding of a target method's functionalities when traversing the caller graph. These findings also align with our results showing no significant negative relationship between caller graph coverage and summary quality. Nonetheless, further research should be conducted to confirm this effects.

Higher caller and callee graph coverage were significantly correlated with decreases in participants' confidence, but the effects were more consistent for callee graph coverage metrics. Coverage of the caller graph was significantly correlated with better self-estimation, mostly for Study 2.

V. Discussion

We next interpret our results in terms of understanding how the call graph relates to code comprehension strategies, and explore the implications of our findings.

Interpretation. In this study, we mapped eye-tracking data onto the call graph to examine code comprehension in larger projects. Contrary to our hypotheses, greater callee graph coverage was linked to lower-quality summaries, particularly in clarity and conciseness, which might suggest that low-level implementation details may hinder summarization. While caller graph coverage showed no strong association with summary quality, a slight negative correlation hints that "less is more" when using the call graph for context. Increased coverage of both graphs was also associated with reduced confidence, though only caller graph coverage appeared to improve the accuracy of self-assessments. This might suggest that traversing the caller graph may help programmers better gauge their performance by enhancing task understanding.

Implications. Our results may impact automated code summarization, tool design, and computer science education. First, the data from this study may be directly used to improve automated code summarization techniques. Previous research using deep learning has found that model performance improved when researchers included the call graph during training [13], and included human eye-tracking data during training [14]. Here, we mapped human eye-tracking data onto the call graph, which may help improve automated code summarization by encoding the nodes and edges that human developers deem important. Our work can also provide guidance for which call graph elements are valuable as context. First, we found that most participants stayed within two callee calls and three caller calls, suggesting that researchers can focus on context within these limits. Additionally, our findings suggest that code summarization agents might benefit from focusing on caller context over callee context for multiple reasons. First, our results suggest that participants traverse deeper into the caller graph than the callee graph. Second, we found a consistent and negative relationship between callee graph coverage and summary quality, which was not present for the caller graph. Third, our results indicate that caller graph coverage may allow for better self-estimation, perhaps by providing a better perspective on the target method's usage.

Our results also have implications for developer tools and education; since our findings suggest that most programmers only look two callee calls and three caller calls away from the target method, IDE tools can be designed to focus on methods within this range. Developers could also build visualizations of these portions of the call graph, or make them more prominent via pop-ups or other documentation elements. Moreover, students could be encouraged to search for context at these depths. Our findings also validate the importance of the work that has been done to visualize and describe call graphs so far [10, 11, 12], as our results suggest that programmers meaningfully use the call graph in their comprehension strategies. Additionally, our findings can help guide programmers in industry and educational settings towards both writing high quality summaries and accurately assessing their own summary quality. Specifically, our results suggest that when writing documentation, programmers should potentially be wary of over-reliance on the callee graph. When contributing code summaries, our results suggest that programmers should similarly be cautious about referring to the callee graph to avoid perpetuating low quality documentation, and may instead benefit from consulting the caller graph.

VI. Threats to Validity

Generalizability. Since our participants were students, the findings may not fully generalize to professional settings, where developers may use more efficient or different strategies. Classroom-taught approaches may also differ from industry practices. To address this, we used two datasets and included participants with varied experience levels. Furthermore, all code was in Java, which may limit generalizability to other languages. To improve robustness, the original authors selected diverse projects [15] and designed the experiment so participants viewed multiple codebases.

Replicability. While some findings were confirmed across both studies, others—particularly those involving subscores—were only observed in the combined dataset and lack independent replication. Validating these would require a larger, separate study. Subjectivity in summary quality ratings also limits replicability, though we mitigated this using predefined metrics and independent ratings with conflict resolution. Additionally, the link between confidence difference and call graph coverage was not completely consistent across studies, suggesting further investigation might be needed. Findings replicated in both studies should be considered more robust than those that were not.

VII. Conclusion

We integrated programmers' eye-tracking data with function call graphs from open source Java projects to understand programmers' code comprehension patterns in a larger project context. Using graph-based metrics, we studied how programmers' code comprehension strategies correlated with downstream code summary quality and participants' confidence in their summaries (to assess whether they overestimated their summary quality). To this end, we applied our metrics to a previously collected eye-tracking dataset, then validated the findings on a new dataset collected for the purposes of this study. We found that call graph coverage was significantly correlated with decreases in summary quality and confidence, but the effects were less pronounced for the caller graph compared to the callee graph.

VIII. Acknowledgments

This research was supported in part by NSF grant CCF-2211429. We thank the participants who contributed their time, our research groups at Vanderbilt University and the University of Notre Dame, and the Vanderbilt Learning Innovation Incubator.

References

[1] A. Gupta and S. Sharma, "Software maintenance: Challenges and issues," *International Journal of Computer Science Engineering*, vol. 4, pp. 23–25, 2015.

[2] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: a survey," in *Proceedings of the 2002 ACM Symposium on Document Engineering*, ser. DocEng '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 26–33. [Online]. Available: https://doi.org/10.1145/585058.585065

[3] J. A. Silva Da Costa and R. Gheyi, "Evaluating the code comprehension of novices with eye tracking," in *Proceedings of the XXII Brazilian Symposium on Software Quality*, ser. SBQS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 332–341. [Online]. Available: https://doi.org/10.1145/3629479.3629490

[4] U. Obaidellah, M. Al Haek, and P. C.-H. Cheng, "A survey on the usage of eye-tracking in computer programming," *ACM Comput. Surv.*, vol. 51, no. 1, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3145904

[5] N. Peitek, S. Apel, C. Parnin, A. Brechmann, and J. Siegmund, "Program comprehension and code complexity metrics: An fmri study," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 524–536.

[6] M. Wyrich, J. Bogner, and S. Wagner, "40 years of designing code comprehension experiments: A systematic mapping study," *ACM Comput. Surv.*, vol. 56, no. 4, Nov. 2023. [Online]. Available: https://doi.org/10.1145/3626522

[7] P. Rodeghero and C. McMillan, "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)*, 2015, pp. 1–10.

[8] T. Busjahn, C. Schulte, and A. Busjahn, "Analysis of code reading to gain more insight in program comprehension," in *Proceedings of the 11th Koli Calling International Conference on Computing Education Research*, 2011, pp. 1–9.

[9] Z. Sharafi, T. Shaffer, B. Sharif, and Y.-G. Guéhéneuc, "Eye-tracking metrics in software engineering," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 96–103.

[10] M. Alnabhan, A. Hammouri, M. Hammad, M. Atoum, and O. Al-Thnebat, "2d visualization for object-oriented software systems," in *2018 International Conference on Intelligent Systems and Computer Vision (ISCV)*, 2018, pp. 1–6.

[11] M. D. Shah and S. Z. Guyer, "An interactive microarray call-graph visualization," in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, 2016, pp. 86–90.

[12] R. Alanazi, G. Gharibi, and Y. Lee, "Facilitating program comprehension with call graph multilevel hierarchical abstractions," *Journal of Systems and Software*, vol. 176, p. 110945, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016412122100042X

[13] A. Bansal, B. Sharif, and C. McMillan, "Towards modeling human attention from eye movements for neural source code summarization," *Proc. ACM Hum.-Comput. Interact.*, vol. 7, no. ETRA, May 2023. [Online]. Available: https://doi.org/10.1145/3591136

[14] Y. Zhang, J. Li, Z. Karas, A. Bansal, T. J.-J. Li, C. McMillan, K. Leach, and Y. Huang, "Eyetrans: Merging human and machine attention for neural code summarization," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 115–136, 2024.

[15] R. Wallace, A. Bansal, Z. Karas, N. Tang, Y. Huang, T. J.-J. Li, and C. McMillan, " Programmer Visual Attention During Context-Aware Code Summarization ," *IEEE Transactions on Software Engineering*, no. 01, pp. 1–13, Mar. 5555. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/TSE.2025.3554990

[16] B. Liu, T. Wang, X. Zhang, Q. Fan, G. Yin, and J. Deng, "A neural-network based code summarization approach by using source code and its call dependencies," in *Proceedings of the 11th Asia-Pacific Symposium on Internetware*, ser. Internetware '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3361242.3362774

[17] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 390–401. [Online]. Available: https://doi.org/10.1145/2568225.2568247

[18] A. Bansal, Z. Eberhart, Z. Karas, Y. Huang, and C. McMillan, "Function call graph context encoding for neural source code summarization," *IEEE Transactions on Software Engineering*, vol. 49, no. 9, pp. 4268–4281, 2023.

[19] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, 2006.

[20] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013.

[21] Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "A systematic literature review on the usage of eye-tracking in soft-

ware engineering," *Information and Software Technology*, vol. 67, pp. 79–107, 2015.

[22] K. Rayner, "Eye movements in reading and information processing : 20 years of research," *Psychological bulletin*, 1998.

[23] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proceedings of the Symposium on Eye Tracking Research and Applications*, ser. ETRA '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 381–384. [Online]. Available: https://doi.org/10.1145/2168556.2168642

[24] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications*, ser. ETRA '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 133–140. [Online]. Available: https://doi.org/10.1145/1117309.1117357

[25] Z. Karas, A. Bansal, Y. Zhang, T. Li, C. McMillan, and Y. Huang, "A tale of two comprehensions? analyzing student programmer attention during code summarization," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 7, Aug. 2024. [Online]. Available: https://doi.org/10.1145/3664808

[26] N. J. Abid, J. I. Maletic, and B. Sharif, "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*, ser. ETRA '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3314111.3319834

[27] R. A. Hamaamin, O. M. A. Ali, and S. W. Kareem, "Java programming language: time permanence comparison with other languages: a review," in *ITM Web of Conferences*, vol. 64. EDP Sciences, 2024, p. 01012.

[28] J. Behler, P. Weston, D. T. Guarnera, B. Sharif, and J. I. Maletic, "itrace-toolkit: A pipeline for analyzing eye-tracking data of software engineering studies," in *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2023, pp. 46–50.

[29] T. R. Shaffer, J. L. Wise, B. M. Walters, S. C. Müller, M. Falcone, and B. Sharif, "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*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 954–957. [Online]. Available: https://doi.org/10.1145/2786805.2803188

[30] D. T. Guarnera, C. A. Bryant, A. Mishra, J. I. Maletic, and B. Sharif, "itrace: eye tracking infrastructure for development environments," in *Proceedings of the 2018 ACM Symposium on Eye Tracking Research & Applications*, ser. ETRA '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3204493.3208343