An Eye-Tracking Study of Java Programmers and Application to Source Code Summarization

Paige Rodeghero, Cheng Liu, Paul W. McBurney, and Collin McMillan, *Member, IEEE*

Abstract—Source Code Summarization is an emerging technology for automatically generating brief descriptions of code. Current summarization techniques work by selecting a subset of the statements and keywords from the code, and then including information from those statements and keywords in the summary. The quality of the summary depends heavily on the process of selecting the subset: a high-quality selection would contain the same statements and keywords that a programmer would choose. Unfortunately, little evidence exists about the statements and keywords that programmers view as important when they summarize source code. In this paper, we present an eye-tracking study of 10 professional Java programmers in which the programmers read Java methods and wrote English summaries of those methods. We apply the findings to build a novel summarization tool. Then, we evaluate this tool. Finally, we further analyze the programmers' method summaries to explore specific keyword usage and provide evidence to support the development of source code summarization systems.

Index Terms—Source code summaries, program comprehension

1 Introduction

Programmers spend a large proportion of their time reading and navigating source code in order to comprehend it [1], [2], [3]. However, studies of program comprehension consistently find that programmers would prefer to focus on small sections of code during software maintenance [1], [3], [4], [5], and "try to avoid" [6] comprehending the entire system. The result is that programmers skim source code (e.g., read only method signatures or important keywords) to save time [7]. Skimming is valuable because it helps programmers quickly understand the underlying code, but the drawback is that the knowledge gained cannot easily be made available to other programmers.

An alternative to skimming code is to read a *summary* of the code. A summary consists of a few keywords, or a brief sentence, that highlight the most-important functionality of the code, for example "record wav files" or "xml data parsing." Summaries are typically written by programmers, such as in leading method comments for JavaDocs [8]. These summaries are popular, but have a tendency to be incomplete [9], [10] or outdated as code changes [11], [12].

As a result, automated source code summarization tools are emerging as viable techniques for generating summaries without human intervention [13], [14], [15], [16], [17], [18]. These approaches follow a common strategy: 1) choose a subset of keywords or statements from the code, and 2) build a summary from this subset. For example, Haiduc et al. described an approach based on automated text summarization using a vector space model (VSM) [19]. This approach selects the top-n keywords from Java methods according to a *term frequency / inverse document frequency* (*tf/idf*) formula. Taking a somewhat different approach, Sridhara et al. designed heuristics to choose statements from Java methods, and then used keywords from those statements to create a summary using sentence templates [20].

In this paper, we focus on improving the process of selecting the subset of keywords for summaries. The long-term goal is to have the automated selection process choose the same keywords that a programmer would when writing a summary. Future research in automated summarization could then be dedicated to the summary building phase.

Our contribution is four-fold. First, we conduct an eye-tracking study of 10 professional Java programmers. During the study, the programmers read Java methods and wrote summaries for those methods. In order to replicate our ideal problem situation, we specifically chose methods that were absolutely uncommented and seemingly isolated, as in many open source projects. Second, we analyzed eye movements and gaze fixations of the programmers to identify common keywords the programmers focused on when reviewing the code and writing the summaries. This analysis led us to different observations about the types of keywords programmers tended to view. We realized these observations were not sufficient enough to prove that only those types of keywords should be included in method summaries. We then designed a small tool that selects keywords from Java methods. Third, we compared the keyword selections from our tool to the keywords selected using a state-of-the-art approach [19]. We found that our tool improved over the state-of-the-art when compared to keyword lists written by human evaluators, showing that the conclusions we made from the eye-tracking results hold credibility. Last, we explore the professional programmers' method summaries to discover how the keywords being focused on in code are actually being used during summarization. We discovered that the keywords are being

• *The authors are with the Department of Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556. E-mail: {prodeghe, cliu7, pmcburne, cmc}@nd.edu.*

Manuscript received 19 June 2014; revised 28 Feb. 2015; accepted 1 May 2015. Date of publication 4 June 2015; date of current version 13 Nov. 2015. Recommended for acceptance by G.C. Murphy. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSE.2015.2442238

indirectly used in summaries through the use of English phrases that contain pieces of the original keyword.

2 The Problem

We address the following gap in the current program comprehension literature: there are no studies of how programmers read and understand source code specifically for the purpose of summarizing that source code. This gap presents a significant problem for designers of automated source code summarization tools. Without studies to provide evidence about the information that programmers use to create summaries, these designers must rely on intuitions and assumptions about what should be included in a summary. In one solution, Haiduc et al. proposed to adapt ideas from *text* summarization, and developed a tool that creates summaries by treating source code as blocks of natural language text [19]. Moreno et al. [15] and Eddy et al. [21] have built on this approach and verified that it can extract keywords relevant to the source code being summarized. Still, a consistent theme across all three of these studies is that different terms are relevant for different reasons, and that additional studies are necessary to understand what programmers prioritize when summarizing code.

Another strategy to creating summaries is to describe a high-level behavior of the code. The goal is to connect the summary to a feature or concept which a programmer would recognize. For example, Sridhara et al. create summaries by matching known patterns of features to Java methods [22]. In prior work, they had identified different heuristics for statements within Java methods, to describe the key functionality of these methods [20]. While these approaches are effective for certain types of methods or statements, they still rely on assumptions about what details the programmers need to see in the summaries. In both of these works, a significant focus was placed on evaluating the conciseness of the summaries—that is, whether the generated summaries described the appropriate information. Hence, in our view, existing summarization tools could be improved if there were more basic data about how programmers summarize code.

3 Related Work

This section will cover related work on program comprehension and eye-tracking studies in software engineering, as well as source code summarization techniques.

3.1 Studies of Program Comprehension

There is a rich body of studies on program comprehension. Ko et al. and Holmes and Walker have observed that many of these studies target the strategies followed by programmers and the knowledge that they seek [2], [23]. For example, during maintenance tasks, some programmers follow a "systemic" strategy aimed at understanding how different parts of the code interact [24], [25]. In contrast, an "opportunistic" strategy aims to find only the section of code that is needed for a particular change [4], [26], [27], [28]. In either strategy, studies suggest that programmers form a mental model of the software [1], specifically including the relationships between different sections of source code [12], [29], [30], [31]. Documentation and note-taking is widely viewed as important because programmers tend to forget the knowledge they have gained about the software [32], [33]. At the same time, programmers are often interrupted with requests for help with understanding source code [34], and written notes assist developers in answering these questions [35].

Different studies have analyzed how programmers use the documentation and notes. One recent study found that programmers prefer face-to-face communication, but turn to documentation when it is not available [6]. The same study found that programmers read source code only as a last resort. The study confirms previous findings that, while reading source code is considered a reliable way of understanding a program, the effort required is often too high [7], [36]. These studies provide a strong motivation for automated summarization tools, but few clues about what the summaries should contain. Some studies recommend an incremental approach, such that documentation is updated as problems are found [37], [38]. Stylos and Myers recommend highlighting summaries of code with references to dependent source code [39]. These recommendations corroborate work by Lethbridge et al. that found that documentation should provide a high-level understanding of source code [40]. Taken together, these studies point to the type of information that should be in code summaries, but not to the details in the code which would convey this information.

3.2 Eye-Tracking in Software Engineering

Eye-tracking technology has been used in only a few studies in software engineering. Crosby and Stelovsky performed one early study, and concluded that the process of reading source code is different from the process of reading prose [41]. Programmers alternate between comments and source code, and fixate on important sections, rather than read the entire document at a steady pace [41]. Despite this difference, Uwano et al. found that the more time programmers take to scan source code before fixating on a particular section, the more likely they are to locate bugs in that code [42]. In two separate studies Bednarik and Tukiainen found that repetitively fixating on the same sections is a sign of low programming experience, while experienced programmers target the output of the code, such as evaluation expressions [43], [44]. In a study on requirements traceability, Ali et al. use eye tracking data to inform them on how developers' verify RT links. They then created a novel technique, similar to ours, to improve on the current *tf* / *idf* technique used in this area [45]. In this paper, we suggest that summaries should include the information from the areas of code on which programmers typically fixate. These previous eye-tracking studies suggest that the summaries may be especially helpful for novice programmers. Moreover, several of these findings have been independently verified. For example, Sharif et al. confirm that scan time and bug detection time are correlated [46]. Eye-tracking has been used in studies of identifier style [47], [48] and UML diagram layout [49], [50], showing reduced comprehension when the code is not in an understandable format.

3.3 Source Code Summarization

The findings in this paper can benefit several summarization tools. Some work has summarized software by showing connections between high- and low-level artifacts, but did

not produce natural language descriptions [51]. Work that creates these descriptions includes work by Sridhara et al. [17], [20], Haiduc et al. [19], and Moreno et al. [15] as mentioned above. Earlier work includes approaches to explain failed tests [52], Java exceptions [53], change log messages [54], and systemic software evolution [55]. Studies of these techniques have shown that summarization is effective in comprehension [21] and traceability link recovery [56]. Nevertheless, no consensus has developed around what characterizes a "high quality" summary or what information should be included in these summaries.

3.4 Vector Space Model Summarization

One state-of-the-art technique for selecting keywords from source code for summarization is described by Haiduc et al.. Their approach selects keywords using a vector space model, which is a classic natural language understanding technique [19]. A VSM is a mathematical representation of text in which the text is modeled as a set of *documents* containing *terms*. For source code, Haiduc et al. treat methods as documents and keyword tokens (e.g., variable names, functions, and datatypes) as terms. Then the code is represented as a large vector space, in which each method is a vector. Each term is a potential direction along which the vectors may have a magnitude. If a term appears in a method, the magnitude of the vector along the "direction" of that term is assigned a non-zero value. For example, this magnitude may be the number of occurrences of a keyword in a method, or weighted based on where the keyword occurs in the method.

Haiduc et al. assign these magnitude values using a *term frequency | inverse document frequency* (*tf/idf*) metric. This metric ranks the keywords in a method body based on how specific those keywords are to that method. *Tf/idf* gives higher scores to keywords which are common within a particular method body, but otherwise rare throughout the rest of the source code. In the approach by Haiduc et al., the summary of each method consists of the top-n keywords based on these *tf/idf* scores. An independent study carried out by Eddy et al. has confirmed that these keywords can form an accurate summary of Java methods [21]. See Section 6.3 for an example of the output from this approach.

4 Eye-Tracking Study Design

This section describes our research questions (RQ), the methodology of our eye-tracking study, and details of the environment for the study.

4.1 Research Questions

The long-term goal of this study is to discover what keywords from source code should be included in the summary of that code. Towards this goal, we highlight four areas of code that previous studies have suggested as being useful for deriving keywords. We study these areas in the four research questions that we pose:

- RQ_1 To what degree do programmers focus on the keywords that the VSM *tf/idf* technique [19], [21] extracts?
- RQ_2 Do programmers focus on a method's *signature* more than the method's body?

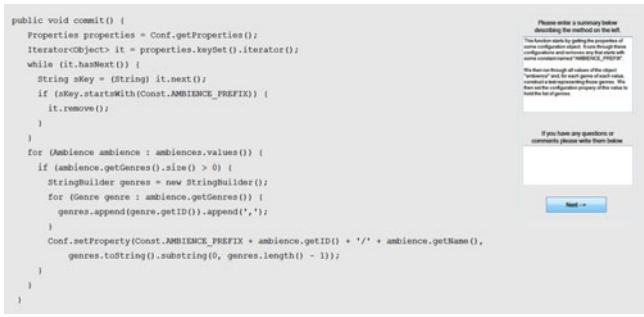


Fig. 1. Interface of the eye-tracking device.

- RQ_3 Do programmers focus on a method's *control flow* more than the method's other areas?
- RQ_4 Do programmers focus on a method's *invocations* more than the method's other areas?

The rationale behind RQ_1 is that two independent studies have confirmed that a VSM *tf/idf* approach to extracting keywords outperforms different alternatives [19], [21]. If programmers tend to read these words more than others, it would provide key evidence that the approach simulates how programmers summarize code. Similarly, both of these studies found that in select cases, a "lead" approach outperformed the VSM approach. The lead approach returns keywords from the method's signature. Therefore, in RQ_2, we study the degree to which programmers emphasize these signature terms when reading code. We pose RQ_3 in light of related work which suggests that programmers comprehend code by comprehending its control flow [2], [57], while contradictory evidence suggests that other regions may be more valuable [58]. We study the degree to which programmers focus on control flow when summarizing code, to provide guidance on how it should be prioritized in summaries. Finally, the rationale behind RQ_4 is that method invocations are repeatedly suggested as key elements in program comprehension [2], [12], [39], [59]. Keywords from method calls may be useful in summaries if programmers focusing on them when summarizing code.

4.2 Methodology

We designed a research methodology based on the related studies of eye-tracking in program comprehension (see Section 3.2). Participants were individually tested in a one hour session consisting of reading, comprehending, and summarizing Java methods. Methods were presented one at a time and eye gaze was captured with a commercial eye-tracker. Fig. 1 shows the system interface. The method was shown on the left, and the participant was free to spend as much time as he or she desired to read and understand the method. The method itself was in 10pt Lucida Console font, was not editable, and had no syntax highlighting. The participant would then write a summary of the method in the text box on the top-right. The bottom-right box was an optional field for comments. The participant clicked on the button to the bottom-right to move to the next method.

4.2.1 Data Collection

The eye-tracker logged the time and location (on the screen) of the participants' eye gaze. The interface also logged the

methods and keywords displayed at these locations. We define "keywords" in our study to include Java-specific keywords, as well as developer-created identifiers, which can contain several sub-words. We do not include any punctuation or operators in our definition of keywords. From these logs, we mapped each keyword to its position and calculated three types of eye-movement behavior to answer our research questions: gaze time, fixations, and regressions. We define gaze time as the total number of milliseconds spent on a region of interest (ROI-e.g., a method, a keyword). We define fixations as any locations that a participant viewed for more than 100 milliseconds. We define regressions as locations that the participant read once, then read again after reading other locations. To be more precise, the number of regressions on a keyword is one less than the total number of fixations on that keyword during a single reading. These three types of eye movements are widely used in the eye-tracking literature [41], [42], [60]. Taken together, they provide a model for understanding how the participants read different sections of the source code. Section 5 details how we use these data to answer each of our research questions.

4.2.2 Subject Applications

We selected a total of 67 Java methods from six different applications: NanoXML, Siena, JTopas, Jajuk, JEdit, and JHotdraw. These applications were all open-source and varied in domain, including XML parsing, text editing, and multimedia. The applications ranged in size from 5 to 117 KLOC and 318 to 7,161 methods. We randomly selected a total of 67 methods from these applications. While the selection was random, we filtered the methods based on two criteria. First, we removed trivial methods, such as *get*, *set*, and empty methods. Second, we removed methods greater than 22 LOC because they could not be displayed on the eye tracking screen without scrolling, which creates complications in interpreting eye movement (see Section 4.2.6). Methods were presented to the participants in a random order. However, to ensure some overlap for comparison, we showed all participants the five largest methods first. These methods were always shown in the same order.

4.2.3 Participants

We recruited 10 programmers to participate in our study. These programmers were employees of the center for research computing (CRC) at the University of Notre Dame. Note that developers at the CRC are not students, they are professional programmers engaged in projects for different departments at Notre Dame. Their overall programming experience ranged from 6 to 27 years, averaging 13.3 years. Each programmer had some recent experience with Java programming, with three of them currently working on projects that required daily Java use. Their specific overall Java experience ranged from 1 to 6 years, averaging 1.8 years. We also made sure that each participant had no detrimental eye impairments and were native English speakers.

4.2.4 Statistical Tests

We compared gaze time, fixation, and regression counts using the Wilcoxon signed-rank test [61]. This test is non-parametric and paired, and does not assume a normal distribution. It is suitable for our study because we compare the gaze times for different parts of methods (paired for each method) and because our data may not be normally distributed.

4.2.5 Equipment

We used a Tobii TX300 Eye-Tracker device for our study. The device has a resolution of 1,920 $\times$ 1,080 and a 300 Hz sampling rate. The monitor uses a 24" widescreen display with a DPI setting of 96. A technician was available at all times to monitor the equipment. The sampling rate was lowered to 120 Hz to reduce computational load; such a sample rate is acceptable for simple ROI analyses like the ones conducted here. The tool used to capture the eye gazes was OGAMA.[1]

4.2.6 Threats to Validity

Our methodology avoids different biases and technical limitations. We assume that all participants write comparable summaries for Java methods in order to reduce the need to filter certain summaries out. We also made the assumptions that the summaries written were acceptable summaries for the given methods. We realize that these aspects are not completely realistic, but we believe the risk is minimized since all participants were professionals. We showed the method as black-on-beige text to prevent potential bias from syntax highlighting preferences and distractions. However, this may introduce a bias if the programmers read the code outside of a familiar environment, such as an IDE. The isolated form of summarization (i.e., the inability to look at other related methods at the same time) may have reduced bias, as we intended, but may have also caused the summarization process to be different than normal for certain participants. To avoid fatigue effects, we ended the study after one hour, regardless of the number of methods summarized. The text boxes were shown on screen, rather than collected on paper, because of eye tracking complications, such as losing track of the eyes or ruining calibration, when the participant repeatedly looks away from the screen. We also realize that this means that the programmer could continue scanning the method while typing the summary, which could have an unknown affect compared to forcing the creation of a hand-written copy. We were also limited in the size of the Java methods: the interface did not support scrolling or navigation, and accuracy decreases as the font becomes smaller. We now know of an environment created to correct this problem [62], but this environment was unavailable to us at the time of the study. These limitations forced us to choose methods which were at most 22 lines long and 93 characters across. Therefore, we cannot claim that our results are generalizable to methods of an arbitrary size. We defined keywords to be of any length. We realize that there have been studies [63] showing that identifier length can affect the view of the identifier and, therefore, could change our results; however, we feel that adding this level of complexity to our study is not needed at this time.

1. http://www.ogama.net/

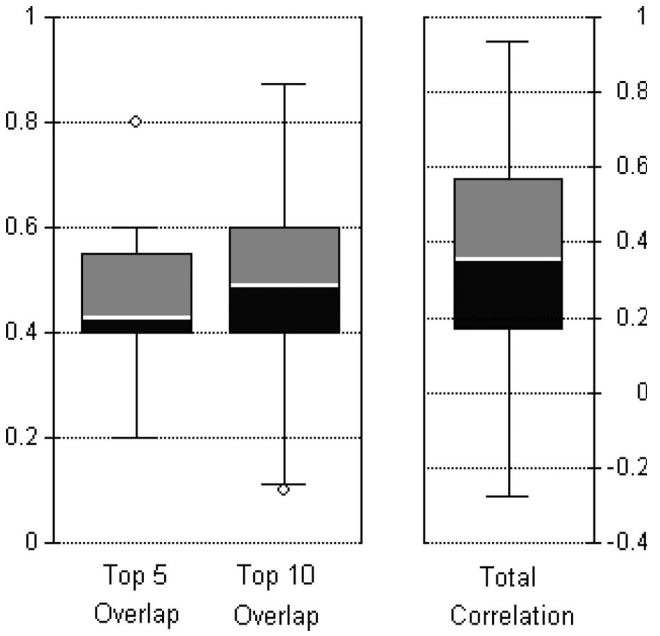


Fig. 2. Data for RQ_1. Plots to the left show percentage overlap of VSM tf/idf top keywords to the top most-read keywords in terms of gaze time. Plot to the right shows Pearson correlation between VSM tf/idf values and gaze time for all keywords. The white line is the mean. The black box is the lower quartile and the gray box is the upper quartile. The thin line extends from the minimum to the maximum value, excluding outliers.

4.2.7 Reproducibility

For the purposes of reproducibility and independent study, we have made all data available via an online appendix: http://www3.nd.edu/~prodeghe/projects/eyesum/

5 Eye-Tracking Study Results

In this section, we present our answer to each research question, as well as our data, rationale, and interpretation of the answers. These answers are the basis for the keyword selection approach we present in Section 6.

5.1 RQ_1: VSM tf/idf Comparison

We found evidence that the VSM *tf/idf* approach extracts a list of keywords that approximates the list of keywords that programmers read during summarization. Two types of data supporting this finding are shown in Fig. 2. First, we calculated the *overlap* between the top keywords from the VSM approach and the top keywords that programmers read during the study according to gaze time. For example, for the method `getLongestMatch`, programmers gaze time was highest for the keywords `currentMatch`, `iter`, `currentMax`, `getLongestMatch`, and `hasNext`. Ideal overalp would result in 100 percent. The overlap for the top five was 60 percent because VSM approach returned `currentMatch`, `retprop`, `currentmax`, `iter`, and `len`. Second, we computed the Pearson correlation between the gaze time and the VSM *tf/idf* values for the keywords in all methods. Full data for these calculations is available via our online appendix.

On average, five of the top 10 keywords selected by the VSM *tf/idf* approach overlapped with the top 10 keywords that programmers read during the study. Similarly, between two and three of the top five keywords overlapped. The correlation between the gaze time and VSM value was 0.35 on average, but varied between -0.28 and 0.94. The correlation was negative for only seven of 67 methods. These results indicate that when the VSM *tf/idf* value for a keyword is high, the gaze time for that keyword is also likely to be high. But, only about half of the keywords in a method's top five or 10 list from VSM are likely to match the keywords that programmers read and view as important. While the VSM *tf/idf* approach selects many appropriate keywords from a method, further improvements are necessary to choose the keywords that programmers read while summarizing code.

5.2 RQ_2: Method Signatures

We found statistically-significant evidence that, during summarization, programmers read a method's signature more-heavily than the method's body. The programmers read the signatures in a greater proportion than the signatures' sizes. On average, the programmers spent 18 percent of their gaze time reading signatures, even though the signatures only averaged 12 percent of the methods. The pattern suggested by this average is consistent across the different methods and participants in our study.

The following describes the procedure we followed to draw this conclusion: Consider the statistical data in Table 1. We compared the *adjusted* gaze time, fixation, and regression count for the keywords in the signatures to the keywords in the method bodies. To compute the gaze time percentage, we calculated the amount of time that the programmer spent reading the signature keywords for a given method. Then we adjusted that percentage based on the size of the signature. For example, if a programmer spent 30 percent of his or her time reading a method's signature, and the signature contains 3 of the 20 keywords (15 percent) in a method, then the adjusted gaze time metric would be $30/15 = 2$. For the fixation and regression metrics, we computed the percentage of fixations on and regressions to the signature or body, and adjusted these values for size in the same manner as gaze time. We then posed three hypotheses (H_1, H_2, and H_3) as follows:

H_n The difference between the adjusted [gaze time / fixation / regression] metric for method signatures and method bodies is not statistically-significant.

We tested these hypotheses using a Wilcoxon test (see Section 4.2.4). We reject a hypothesis only when $|Z|$ is greater than Z_{crit} for a p is less than 0.05. For RQ_2, we rejected two of these three null hypotheses (H_1 and H_2 in Table 1). This indicates that the programmers spent more gaze time, and fixated more often on, the method signatures than the method bodies, when adjusted for the size of the signatures. We did not find a statistical difference in the regression time, indicating that the programmers did not re-read the signatures more than the methods' body keywords.

5.3 RQ_3: Control Flow

We found statistically significant evidence that programmers tended to read control flow keywords less than the keywords from other parts of the method. On average, programmers spent 31 percent of their time reading control flow keywords, even though these keywords averaged 37 percent of the keywords in the methods. To determine the significance of the

TABLE 1
Statistical Summary of the Results for RQ_2, RQ_3, and RQ_4

RQ	H	Metric	Method Area	Samples	$\tilde{x}$	μ	Vari.	U	U_{expt}	U_{vari}	Z	Z_{crit}	p
	H_1	Gaze	Signature	95	1.061	1.784	4.237	2,808	2,280	72,580	1.96	1.65	0.025
			Non-Sig.	95	0.996	0.933	0.054						
RQ_2	H_2	Fixation	Signature	95	1.150	1.834	4.451	3,008	2,280	72,580	2.70	1.65	0.003
			Non-Sig.	95	0.984	0.926	0.050						
	H_3	Regress.	Signature	95	0.830	1.436	3.607	2,307	2,280	72,580	0.10	1.65	0.459
			Non-Sig.	95	1.014	0.978	0.032						
	H_4	Gaze	Ctrl. Flow	111	0.781	0.924	0.392	1,956	3,108	115,514	-3.389	1.96	0.001
			Non-Ctrl.	111	1.116	1.134	0.145						
RQ_3	H_5	Fixation	Ctrl. Flow	111	0.834	0.938	0.274	2,140	3,108	115,514	-2.848	1.96	0.004
			Non-Ctrl.	111	1.071	1.122	0.130						
	H_6	Regress.	Ctrl. Flow	111	0.684	0.813	0.269	1,463	3,108	115,514	-4.840	1.96	$<$1e-3
			Non-Ctrl.	111	1.132	1.199	0.165						
	H_7	Gaze	Invocations	106	0.968	1.069	0.778	2,586	2,836	100,660	-0.786	1.96	0.432
			Non-Invc.	106	1.021	1.027	0.086						
RQ_4	H_8	Fixation	Invocations	106	1.003	1.048	0.385	2,720	2,835	100,660	-0.364	1.96	0.716
			Non-Invc.	106	0.998	1.020	0.064						
	H_9	Regress.	Invocations	106	1.028	1.045	0.065	2,391	2,835	100,660	-1.399	1.96	0.162
			Non-Invc.	106	1.028	1.045	0.065						

Wilcoxon test values are U, U_{expt}, and U_{vari}. Decision criteria are Z, Z_{crit}, and p. A "Sample" is one programmer for one method.

results, we followed the same procedure outlined in Section 5.2: we computed the adjusted gaze time, fixation, and regression count for the control flow keywords versus all other keywords. A "control flow" keyword included any keyword inside of a control flow statement. For example, for the line `if(area < maxArea)`, the control flow keywords are "area" and "maxArea." We then posed H_4, H_5, and H_6:

H_n The difference between the adjusted [gaze time / fixation / regression] metric for control flow keywords and all other keywords is not statistically-significant.

Using the Wilcoxon test, we rejected all three null hypotheses (see RQ_3 in Table 1). These results indicate that the programmers did not read the control flow keywords as heavily as other keywords in the code.

5.4 RQ_4: Method Invocations

We found no evidence that programmers read keywords from method invocations more than keywords from other parts of the methods. Programmers read the invocations in the same proportion as they occurred in the methods. We defined invocation keywords as the keywords from the invoked method's name and parameters. For example, for the line `double ca = getArea(circle, 3)`, the invocation keywords would be "getarea" and "circle," but not "double" or "ca." We then posed three hypotheses (H_5, H_6, and H_7):

H_n The difference between the adjusted [gaze time / fixation / regression] metric for invocation keywords and all other keywords is not statistically-significant.

As shown in Table 1, the Wilcoxon test results were not conclusive for RQ_4. These results indicate that the programmers read the invocations in approximately the same proportion as other keywords.

5.5 Summary of the Eye-Tracking Results

We derive two main interpretations of our eye-tracking study results. First, the VSM *tf/idf* approach roughly approximates the list of keywords that programmers read during summarization, with about half of the top 10 keywords from VSM matching those most-read by programmers. Second, programmers prioritize method signatures above invocation keywords, and invocation keywords above control flow keywords. We base our interpretation on the finding that signature keywords were read more than other keywords, invocations were read about the same, and control flow keywords were read less than other keywords. In addition, the adjusted gaze time for method signatures (H_1) averaged 1.784, versus 1.069 for invocations (H_7) and 0.924 for control flow (H_4). An adjusted value of 1.0 for an area of code means that the programmers read that area's keywords in a proportion equal to the proportion of keywords in the method that were in that area. In our study, the adjusted gaze times were greater than 1.0 for signatures and invocations, but not for control flow keywords. Our conclusion is that the programmers needed the control flow keywords less for summarization than the invocations, and the invocations less than the signature keywords.

6 Our Approach

In this section, we describe our approach for extracting keywords for summarization. Generally speaking, we improve the VSM *tf/idf* approach we studied in RQ_1 using the eye-tracking results from answering RQ_2, RQ_3, and RQ_4.

6.1 Key Idea

The key idea behind our approach is to modify the weights we assign to different keywords, based on how programmers read those keywords. In the VSM *tf/idf* approach, all occurrences of terms are treated equally: the *term frequency* is the count of the number of occurrences of that term in a method (see Section 3.4). In our approach, we weight the terms based on where they occur. Specifically, in light of our eye-tracking results, we weight keywords differently if they occur in method signatures, control flow, or invocations.

TABLE 2
The Weight Given to Terms Based on the Area of Code Where the Term Occurs

Code Area	VSM_{def}	Eye_A	Eye_B	Eye_C
Method Signature	1.0	1.8	2.6	4.2
Method Invocation	1.0	1.1	1.2	1.4
Control Flow	1.0	0.9	0.8	0.6
All Other Areas	1.0	1.0	1.0	1.0

6.2 Extracting Keywords

Table 2 shows four different sets of weights. Each set corresponds to different counts for keywords from each code area. For the default VSM approach [19], denoted VSM_{def}, all occurrences of terms are weighted equally. In one configuration of our approach, labeled Eye_A, keywords from the signature are counted as 1.8 occurrences, a keyword is counted as 1.1 if it occurs in the method invocation, and 0.9 if in a control flow statement. Because we wanted to label each keyword with only one section, if a keyword occurs in both a control flow and invocation area, we count it as only in control flow. These weights correspond to the different weights we found for these areas in the eye-tracking study (see Section 5.5). Eye_B and Eye_C work similarly, except with progressively magnified differences in the weights

These changes in the weights mean that keywords appearing in certain code areas are inflated, allowing those keywords to be weighted higher than other keywords with the same number of occurrences, but in less important areas. After creating the vector space for these methods and keywords, we score each method's keywords using *tf/idf*, where term frequency of each term is defined by its own weighted score, rather than the raw number of occurrences.

6.3 Example

In this section, we give an example of the keywords that our approach and the default VSM *tf/idf* approach generate using the source code in Fig. 3. In this example, where VSM *tf/idf* increments each weight a fixed amount of each occurrence of a term, we increment by our modified weights depending on contextual information. Consider the keyword list below:

Keywords Extracted by Default VSM Approach
"textarray, text, match, offset, touppercase"

The term "textArray" occurs in 2 of 6,902 different methods in the project. But it occurs twice in `regionMatches()`, and therefore the default VSM *tf/idf* approach places it at the top of the list. Likewise, "text" occurs in 125 different methods, but four times in this method. But other keywords, such as "ignoreCase", which occur in the signature and control flow areas, may provide better clues about the method than general terms such as "text", even though the general terms appear often. Consider the list below:

Keywords Extracted by Our Approach
"match, regionmatches, text, ignorecase, offset"

The term "match" is ranked at the top of the list in our approach, moving from position three in the default approach. Two keywords, "regionMatches" and "ignoreCase", that appear in our list do not appear in the list from the default approach. By contrast, the previous approach favors "toUpperCase" over "ignoreCase" because "toUpperCase" occurs in 22 methods, even though both occur twice in this method. These differences are important because it allows our approach to return terms which programmers are likely to read (according to our eye-tracking study), even if those terms may occur slightly more often across all methods.

```
public static boolean regionMatches(boolean ignoreCase,
    Segment text, int offset, char[] match) {
    int length = offset + match.length;
    if(length > text.offset + text.count)
        return false;
    char[] textArray = text.array;
    for(int i = offset, j = 0; i < length; i++, j++)
    {
        char c1 = textArray[i];
        char c2 = match[j];
        if(ignoreCase)
        {
            c1 = Character.toUpperCase(c1);
            c2 = Character.toUpperCase(c2);
        }
        if(c1 != c2)
            return false;
    }
    return true;
}
```

Fig. 3. Source code for example.

7 Evaluation of Our Approach

This evaluation compares the keyword lists extracted by our approach to the keyword lists extracted by the state-of-the-art VSM *tf/idf* approach [19]. In this section, we describe the user study we conducted, including our methodology, research subjects, and evaluation metrics.

7.1 Research Questions

Our objective is to determine the degree to which our approach and the state-of-the-art approach approximate the list of keywords that human experts would choose for summarization. Hence, we pose the following two questions:

RQ_5 To what degree do the top-n keywords from our approach and the standard approach match the keywords chosen by human experts?

RQ_6 To what degree does the *order* of the top-n keywords from our approach and the standard approach match the order chosen by human experts?

The rationale behind RQ_5 is that our approach should extract the same set of keywords that a human expert would select to summarize a method. Note that human experts rate keywords subjectively, so we do not expect the human experts in our study to agree on every keyword, and the experts may not normally limit themselves to keywords within one method. Nevertheless, a stated goal of our approach is to improve over the state-of-the-art approach (e.g., VSM *tf/idf* [19]), so we measure both approaches against multiple human experts. In addition to extracting the same set of keywords as the experts, our approach should extract the keywords *in the same order*. The order of the keywords is important because a summarization tool may only choose a small number of the top keywords that are most-relevant to the method. Therefore, we pose RQ_6 to study this order.

7.2 Methodology

To answer our research questions, we conducted a user study in which human experts read Java methods and ranked the *top five* most-relevant keywords from each method. We chose five as a value for the top-*n* to strike a balance between two factors: First, we aimed to maximize the number of keywords that our approach can suggest to a summarization tool. However, a second factor is that, during pilot studies, fatigue became a major factor when human experts were asked to choose more than five keywords per method, after reading several methods. Because fatigue can lead to innaccurate results, we limited the keyword list size to five.

During the study, we showed the experts four Java methods from six different applications, for a total of 24 methods. We used the same six applications that were selected for the eye-tracking study in Section 4.2.2. Upon starting our study, each expert was shown four randomly-selected methods from a randomly-selected application. The expert read the first method, then read a list of the keywords in that method. The expert then chose five of those keywords that, in his or her opinion, were most-relevant to the tasks performed by the method. The expert also rank ordered those five keywords from most-relevant to least-relevant. After the expert finished this process for the four methods, we showed the expert four more methods from a different application, until he or she had ranked keyword lists for all 24 methods. For the purpose of reproducibility, we have made our evaluation interface available via our online appendix.

7.2.1 Participants

To increase generalizability of the results, the participants in this study were different than the participants in the eye-tracking study. We recruited nine human experts who were skilled Java programmers among graduate students in the Computer Science and Engineering department at the University of Notre Dame and other universities. These participants had an average of 6.2 years of Java experience, and 10.5 years of general programming experience.

7.2.2 Evaluation Metrics and Tests

To compare the top-*n* lists for RQ_5, we used one of the same keyword list comparison metrics we used in Section 5.1: overlap. For RQ_6, to compare the lists in terms of their order, we compute the *minimizing Kendall tau distance*, or K_{min}, between the lists. This metric has been proposed specifically for the task of comparing two ordered top-*n* lists [64], [65], and we follow the procedure recommended by Fagin et al. [64]: For each top-*n* list for a Java method from a human expert, we calculate the K_{min} between that list and the list extracted by our approach. We also calculate the K_{min} between the expert's list and the list from the state-of-the-art approach. We then compute the K_{min} value between the list from our approach and the list from the state-of-the-art approach.

The results of this procedure are three sets of K_{min} values for each configuration of our approach (Eye_A, Eye_B, and Eye_C): one between human experts and our approach, and one between our approach and the state-of-the-art approach. We also create one set of K_{min} values between human experts and the state-of-the-art approach. To compare these lists, we use a two-tailed Mann-Whitney statistical test [66]. The Mann-Whitney test is non-parametric, so it is appropriate for this comparison where we cannot guarantee that the distribution is normally distributed. The result of this test allows us to answer our research question by determining which differences are statistically-significant.

7.2.3 Threats to Validity

Our study carries threats to validity, similar to any evaluation. One threat is from the human experts we recruited. Human experts are susceptible to fatigue, stress, and errors. At the same time, differences in programming experience, opinions, and personal biases can all affect the answers given by the experts. We cannot rule out the possibility that our results would differ if these factors were eliminated. However, we minimize this threat in two ways: first, by recruiting nine experts rather than relying on a single expert, and second by using statistical testing to confirm the observed differences were significant.

Another key source of threat is in the Java source code that we selected for our study. It is possible that our results would change given a different set of Java methods for evaluation. We mitigated this threat by selecting the methods from six different applications in a wide range of domains and sizes. We also randomized the order in which we showed the applications to the study participants, and randomized the methods which we selected from those applications. The purpose of this randomization was to increase the variety of code read by the participants, and minimize the effects that any one method may cause in our results. We realize that we did not also have any randomized selection of keywords in addition to our intelligent selection of keywords in order to further introduce randomization and the possibility of doubt. However, we believe that this small addition of randomness would not have significantly affected our results. In addition, we released all data via our online appendix so that other researchers may reproduce our work.

8 Comparison Study Results

In this section, we present the results of the evaluation of our approach. We report our empirical evidence behind, and answers to, RQ_5 and RQ_6.

8.1 RQ_5: Overlap of Keyword Lists

Our approach outperformed the default VSM *tf/idf* approach in terms of overlap. The best-performing configuration of our approach was Eye_C. It extracted top-five keyword lists that contain, on average, 76 percent of the keywords that programmers selected during the study. In other words, almost four out of five of the keywords extracted by Eye_C were also selected by human experts. In contrast 67 percent of the keywords, just over three of five, from VSM_{def} were selected by the programmers. Table 3 shows overlap values for the remaining configurations of our approach. Values for columns marked "Users" are averages of the overlap percentages for all keyword lists from all participants for all methods. For other columns, the values are averages of the lists for each method, generated by a particular approach. For example, 94 percent of the

TABLE 3
Data for RQ_5

	Users	VSM_{def}	Eye_A	Eye_B	Eye_C
Users	1.00	0.67	0.72	0.75	0.76
VSM_{def}	0.67	1.00	0.90	0.84	0.78
Eye_A	0.72	0.90	1.00	0.94	0.88
Eye_B	0.75	0.84	0.94	1.00	0.94
Eye_C	0.76	0.78	0.88	0.94	1.00

Overlap for top-five lists.

TABLE 4
Data for RQ_6

	Users	VSM_{def}	Eye_A	Eye_B	Eye_C
Users	0.00	0.54	0.50	0.46	0.43
VSM_{def}	0.54	0.00	0.20	0.31	0.40
Eye_A	0.50	0.20	0.00	0.15	0.27
Eye_B	0.46	0.31	0.15	0.00	0.16
Eye_C	0.43	0.40	0.27	0.16	0.00

K_{min} for top-five lists.

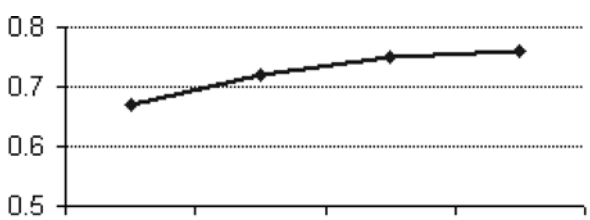
keywords in lists generated by Eye_B were also in lists generated by Eye_C.

To confirm the statistical significance of these results, we pose three hypotheses (H_{10}, H_{11}, and H_{12}) of the form.

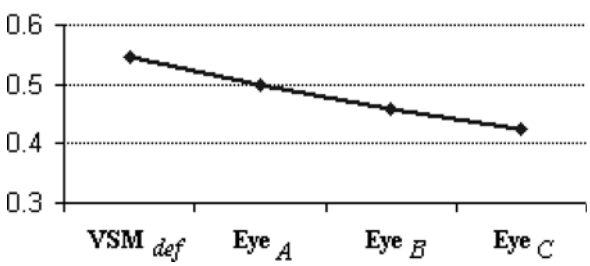
H_n The difference between the overlap values for keyword lists extracted by [Eye_A / Eye_B / Eye_C] to the programmer-created keyword lists, and the overlap values of lists extracted by VSM_{def} to the programmer-created lists is not statistically-significant.

For brevity, we only test hypotheses for overlap values that are compared to human-written keyword lists. The results of these tests are in Table 5. We rejected all three hypotheses because the Z value exceeded Z_{crit} for p less than 0.05. Therefore, our approach's improvement in overlap, as compared to VSM_{def}, is statistically-significant. We answer RQ_5 by finding that overlap increases by approximately 9 percent from the default VSM *tf/idf* approach (67 percent) to our best-performing approach, Eye_C (76 percent).

Fig. 4a depicts a pattern we observed in overlap with the expert-created lists: Eye_A, Eye_B, and Eye_C progressively increase. This pattern reflects the progressively-magnified differences in weights for the three configurations of our approach (see Section 6). As the weight differences are increased, the approach returns keyword lists that more-closely match the keyword lists written by human experts.



(a) Overlap.



(b) K_{min}.

Fig. 4. Overlap and K_{min} values for the default approach and three configurations of our approach. For overlap, *higher* values indicate higher similarity to the lists created by participants in our study. For K_{min}, *lower* values indicate higher similarity. Eye_C has the best performance for both metrics.

This finding is strong evidence that some areas of code should be prioritized over other areas for summarization. We expand on the implications of these findings in Section 11.

8.2 RQ_6: Keyword List Order

Eye_C was the best-performing approach in terms of the *order* of the keyword lists. We found statistically-significant improvement by the approach over the default VSM *tf/idf* approach in terms of K_{min}, which we use to measure similarity of list order (see Section 7.2.2). Table 4 presents the K_{min} values of VSM_{def}, Eye_A, Eye_B, and Eye_C compared to the human-written values, and compared to each other. The K_{min} distance between the lists from Eye_C was 0.43 on average. This distance compares to 0.54 for VSM_{def}. Configurations with similar weights return similar keyword lists; the K_{min} distance between Eye_A and Eye_B is 0.15. Likewise, VSM_{def} returns lists most-similar to Eye_A (0.20 distance), which has the least-exaggerated weights. Eye_C returned the lists most like those written by the human experts.

The differences in K_{min} between our approach and the default approach are statistically-significant. We tested the statistical-significance using the same procedure as in the previous section. We posed three hypotheses (H_{13}, H_{14}, and H_{15}) of the form:

H_n The difference between the K_{min} values for keyword lists extracted by [Eye_A / Eye_B / Eye_C] to the programmer-created keyword lists, and the K_{min} values of lists extracted by VSM_{def} to the programmer-created lists is not statistically-significant.

We rejected all three hypotheses based on the values in Table 5. Therefore, our approach improved over the default approach in terms of K_{min} by a significant margin. The interpretation of this finding is that the order of the keyword lists returned by our approach more-closely matched the order of the keyword lists written by programmers in our study, than the order of the lists from VSM_{def}. Our answer to RQ_6 is that the best-performing approach, Eye_C, improves over VSM_{def} by approximately 11 percent in terms of K_{min} $(0.54 - 0.43)$.

We observed a similar pattern in our analysis of RQ_6 as for RQ_5. As Fig. 4b illustrates, the K_{min} values decrease progressively for Eye_A, Eye_B, and Eye_C. As the weights increase for keywords in different areas of code, the order of the keyword lists more-closely matches the order of the lists written by programmers. In other words, the quality of the keyword lists improves if those lists contain keywords from some areas of code instead of others. Our approach emphasizes keywords from areas of code that programmers view as important. This emphasis lead to a statistically-significant improvement. Eye_C had the most-aggressive set of weights

TABLE 5
Statistical Summary of the Results for RQ_5 and RQ_6

RQ	Metric	H	Approach	Samples	$\tilde{x}$	μ	Vari.	U	U_{expt}	U_{vari}	Z	Z_{crit}	p
RQ_5	Overlap	H_{10}	Eye_A	170	0.800	0.719	0.030	6,725	3,698	234,964	6.246	1.96	$<$1e-3
			VSM_{def}	170	0.600	0.671	0.027						
		H_{11}	Eye_B	170	0.800	0.749	0.029	9,092	5,175	331,605	6.802	1.96	$<$1e-3
			VSM_{def}	170	0.600	0.671	0.027						
		H_{12}	Eye_C	170	0.800	0.761	0.033	9,628	5,727	360,607	6.496	1.96	$<$1e-3
			VSM_{def}	170	0.600	0.671	0.027						
RQ_6	K_{min}	H_{13}	Eye_A	170	0.489	0.498	0.036	3,519	6,878	406,567	-5.268	1.96	$<$1e-3
			VSM_{def}	170	0.533	0.545	0.032						
		H_{14}	Eye_B	170	0.467	0.460	0.044	3,028	7,222	412,696	-6.529	1.96	$<$1e-3
			VSM_{def}	170	0.533	0.545	0.032						
		H_{15}	Eye_C	170	0.444	0.425	0.244	2,971	7,254	412,950	-6.664	1.96	$<$1e-3
			VSM_{def}	170	0.533	0.545	0.032						

Wilcoxon test values are U, U_{expt}, and U_{vari}. Decision criteria are Z, Z_{crit}, and p. A "sample" is a list chosen by a particpant. Testing procedure is identical to Table 1.

for keywords based on code area; it also experienced the highest level of performance of any approach tested here.

9 "Stubborn" Keywords

In this section we study a phenomenon we observed in our study called "stubborn" keywords, which are keywords that would seem to be keywords that programmers would use in summaries, but do not. Technically, we define stubborn keywords to be words that have a high *tf/idf* score as well as relatively-long gaze time, but are not included in the summaries written by programmers. Our definition is rooted in the idea that our tool, the state-of-the-art tool, and the programmers' own eye movements tend to agree on predictions of which keywords that programmers report as relevant (see Section 7), but a small subset of these keywords do not appear in programmer-written summaries even when all three indicators agree.

Stubborn keywords are not unique to our approach, and have been reported in related literature on software traceability [67]. Gibiec et al. and Zou et al. identified the cause as words that appear synonymous to human readers based on personal experience, but do not appear in official lists of synonyms and are difficult to detect using automated tools [67], [68]. A possible unfortunate result of stubborn keywords for our work is that our approach performs poorly on certain methods. Evidence that this is occurring is visible as a substantial tail in the boxplots in Fig. 5: in a small number of cases, not a single keyword selected by our approach or the default VSM *tf/idf* approach matched a keyword selected by a programmer. This section explores the extent of the problem and attempts to identify the source of the stubborn keywords.

9.1 Research Questions

The goal of this study is to investigate the degree of the effect of stubborn keywords on our approach. To this end, we propose the following four Research Questions:

RQ_7 What is the degree of correlation between the time that programmers spend reading a keyword and the likelihood that they will use that keyword in a summary?

RQ_8 What is the prevalence of keywords that have both high *tf/idf* scores and long gaze time, but are not used in summaries (e.g., the prevalence of stubborn keywords)?

RQ_9 Do stubborn keywords tend to be program keywords such as API call names?

RQ_{10} Do stubborn keywords tend to be the longest keywords?

The rationale behind RQ_7 is that programmers spend more time reading some keywords than others (see Section 5), and a common assumption in eye-tracking literature is that the words that programmers spend more time reading are more important for comprehension [41], [44]. However, an open question is whether programmers are more likely to use a keyword in a summary if they spend more time reading that keyword in the code. It is important to answer this question for automatic documentation generation because it would provide further evidence that the data we collect about programmer eye movements is useful for predicting which keywords should be included in summaries. In other words, if programmers read a type of keyword in code more heavily than other types, then automatic documentation generation tools should use those keywords in summaries.

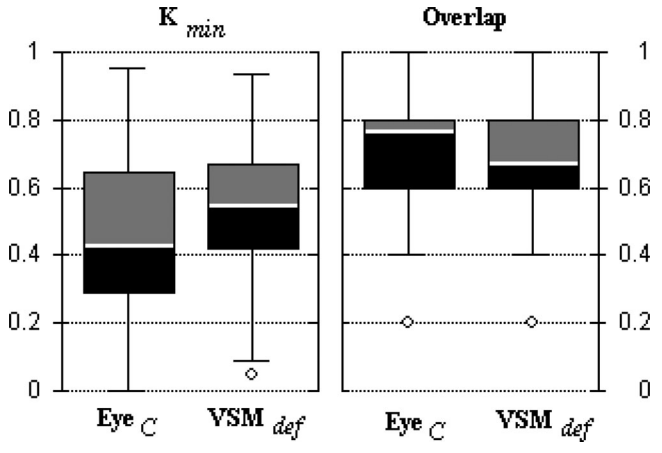


Fig. 5. Boxplots comparing VSM_{def} with Eye_C, the best performing configuration of our approach. Each datapoint in each boxplot represents one K_{min} or Overlap value. That value is calculated between two keyword lists: one generated by the given approach for a method, and one written by a participant in our study for the same method. The points for K_{min} for Eye_C are concentrated at lower values, and the points for Overlap at higher values, suggesting better performance than VSM_{def}.

Likewise, the rationale behind RQ_8 is that, even if a correlation exists between gaze time and keyword usage in summaries, a certain subset of keywords (e.g., the "stubborn" keywords) may violate the correlation. We found limited evidence that stubborn keywords exist in Section 7, but not enough evidence to draw a conclusion about the extent of stubborn keywords. It is important to know the extent because it represents a performance limitation for both our tool and for previous state-of-the-art tools.

We pose RQ_9 to explore one possible explanation of the stubborn keywords, in light of related work that suggests API calls to be important keywords for code comprehension [69], [70], [71]. Therefore, it is possible that these keywords could be the "synonymous" keywords suggested as stubborn keywords by Zou et al. [68]. If API calls are a major source of the stubborn keywords, it may be possible to improve the performance of automatic documentation generators by filtering API calls from summaries.

Additionally, we pose RQ_{10} to explore another possible explanation of the stubborn keywords, considering related work suggesting that the length of identifiers affects the overall visibility and usage of that identifier [63]. Thus, these keywords could be viewed for extended periods of time simply due to their length. Another possibility is that these complex keywords are multi-word identifiers that are being used in summaries in their separated form—splitting may not detect all of these, for example "primaryctrlbtn" to refer to "primary control button." If long keywords are generally not used in summaries, then automatic documentation generators could be improved by running some preprocessing such as abbreviation detection on these long keywords before building summaries.

9.2 Methodology

This section describes the methodology we follow to study RQ_7, RQ_8, RQ_9, and RQ_{10}. First we collect the summaries written by programmers in our eye-tracking study. Then, according to the data collection procedure in the next section, we pre-process the keywords from those summaries and the keywords from source code.

For RQ_7, we divide the keywords into two groups: words that are "longer viewed" and words that are viewed close to the average amount. Section 9.2.2 explains our procedure for determining which words are "longer viewed", but the idea is that some keywords are viewed well above the average amount of time. We would expect these words to be important for comprehension, and therefore appear in the summaries, according to related eye-tracking literature (see Section 9.1). For each Java method, we extract all keywords. Then, we compute the percentage of those keywords that were "longer viewed" and which were not. Then, we compute the percentage of "longer viewed" words that were used in summaries, and the percentage of the remaining words that were used in summaries. The result is a set of two percentages for each Java method. If the percentage is higher for "longer viewed" keywords, it implies that those keywords are more likely to be used in summaries. To determine if the percentage is higher by a significant margin, we use a statistical test as described in Section 9.2.3.

For RQ_8, we extracted keywords which were selected by the VSM *tf/idf* approach for summaries in our earlier studies, and we selected from these keywords those words which were also "longer viewed." In other words, we computed the intersection of the set of keywords with high *tf/idf* scores and high gaze time. Then, for each Java method, we calculated the percent of those keywords that the programmers did not use in a summary for that method. The result was a set of the "stubborn keywords" for each method and a percent of each method consisting of those stubborn keywords. We then analyze and report descriptive statistics of those percentages for the Java methods.

For RQ_9, we obtained a list of keywords from method names in the official Java API. Then we calculated the size of the intersection of these "API keywords" and the stubborn keywords for each Java method, and computed descriptive statistical data.

For RQ_{10}, we computed the average keyword lengths in both the source code and the programmer summaries and compared the lengths using a statistical hypothesis test. The keywords from source code only included keywords with long gaze times.

9.2.1 Data Collection

We collected the summaries of Java methods that programmers wrote during our study in Section 4. We also use the Java methods from that study and apply the same preprocessing to extract keywords from both the summaries and from the Java methods. We obtained the list of Java API method names from the official Java documentation processed in related literature [72].

9.2.2 Definition of "Longer-Viewed"

The "longer-viewed" keywords are the keywords in source code that programmers spend the most time reading. We measure the time programmers read each keyword as gaze time, and "longer-viewed" keywords are those words with higher gaze time. To determine which keywords belong to the longer-viewed group, we defined a cutoff point of 3,000 milliseconds. If programmers, on average, read a keyword for longer than 3,000 milliseconds, we include that keyword as longer-viewed.

We decided on this cutoff point using a K-medoids clustering algorithm. We set up the algorithm to create two clusters based on the gaze time data: one cluster for longer-viewed words and one cluster for all others. K-medoids is an appropriate algorithm for our analysis because K-medoids is less sensitive to outliers than K-means or other strategies [73], [74], and our data contains a number of outliers due to eye fixations, visible in Fig. 6 as a tail extending to 20,000 milliseconds. Our cutoff point of 3,000 milliseconds is based on the cluster split point given by the K-medoids algorithm.

9.2.3 Statistical Analysis

To determine statistical significance in RQ_7 through RQ_{10}, we use a Wilcoxon signed-rank test [61] as in Section 4. This test is appropriate because our data are paired, and because we cannot guarantee that our data are normally distributed.

10 Keyword Analysis Results

In this section, we present the results of our analysis. We report our empirical evidence behind and answers to RQ_7, RQ_8, RQ_9, and RQ_{10}.

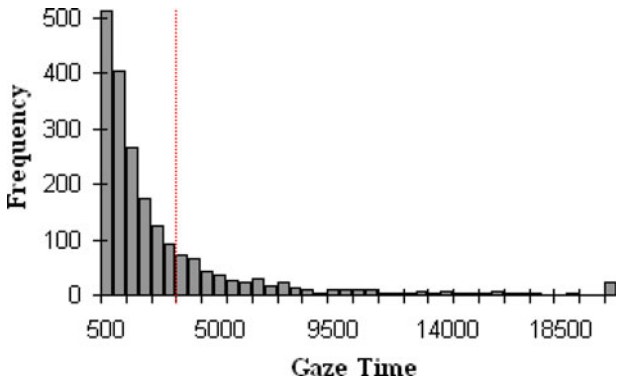


Fig. 6. Distribution of keywords for different gaze times. "Longer-viewed" keywords are those to the right of the 3,000 millisecond cutoff marked with a dashed red line.

10.1 RQ_7: Gaze Time and Keyword Usage

We found statistically significant evidence that there are fewer "longer-viewed" keywords contained in method summaries compared to all other keywords. On average, the percentage of "longer-viewed" keywords used in method summaries was 12 percent, while the percentage of all source code keywords used in method summaries was 16 percent. Note the differences in Fig. 7. This suggests that there is little to no correlation between the time spent reading a keyword and the likelihood of its use in a method summary.

The following describes the procedure we followed to draw this conclusion: Consider the statistical data in Table 6. We compared the percentages of "longer-viewed" keywords in summaries and overall keywords in summaries. To compute the "longer-viewed" percentage, we divided the number of "longer-viewed" keywords in summaries by the total number of "longer-viewed" keywords. Similarly, to compute the overall percentage, we divided the number of keywords in both the summaries and source code by the total number of source code keywords. We then posed H_{16} as follows:

H_{16} The percentage of "longer-viewed" keywords used in summaries, out of all "longer-viewed" keywords, is not significantly less than the percentage of all keywords used in summaries, out of all source code keywords.

Using the Wilcoxon test, we rejected the null hypotheses (see RQ_7 in Table 6). These results indicate that the programmers did not necessarily use keywords that they read for an extended period of time in their method summaries.

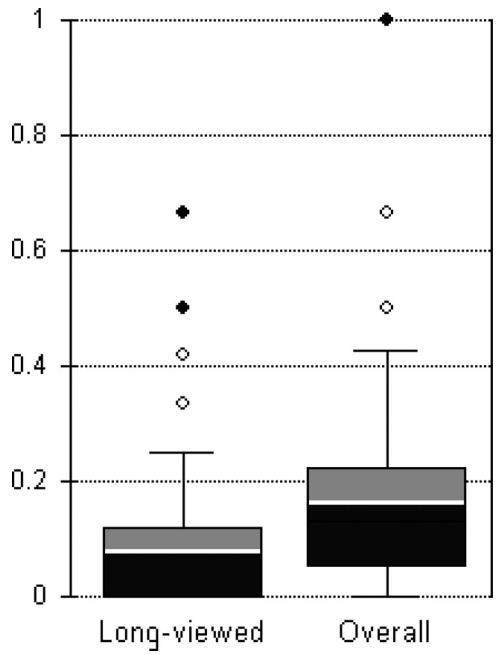


Fig. 7. Data for RQ_7. The box plot to the left represents the percentage of "longer-viewed" keywords used in summaries per participant per method. The box plot to the right represents the percentage of all keywords used in summaries per participant per method. The white line is the mean. The black box is the lower quartile and the gray box is the upper quartile. The thin line extends from the minimum to the maximum value, excluding outliers.

Therefore, we found that the degree of correlation between long gaze times for a keyword and its use in a method summary is not significant.

10.2 RQ_8: "Stubborn" Keyword Prevalence

We found statistically significant evidence that keywords programmers used in their method summaries generally have higher *tf/idf* scores than keywords with high gaze times that programmers left out of their method summaries.

The following describes the procedure we followed to draw this conclusion: Consider the statistical data in Table 6. We compared the VSM *tf/idf* scores of keywords used in summaries, both "longer-viewed" and those with smaller gaze times, to the *tf/idf* scores of all of the unused "longer-viewed" keywords. We then posed H_{17} as follows:

H_{17} The *tf/idf* scores for keywords used in summaries are not significantly higher than the scores of "longer-viewed" keywords not used in summaries.

TABLE 6
Statistical Summary of the Results for RQ_7, RQ_8, and RQ_9

RQ	H	Approach	Samples	$\tilde{x}$	μ	Vari.	U	U_{expt}	U_{vari}	Z	Z_{crit}	p
RQ_7	H_{16}	Longer-viewed	98	0.075	0.022	0.0196	604	1,540.5	40,303.75	3.75	1.65	$<$1e-3
		Overall	98	0.138	0.095	0.0128						
RQ_8	H_{17}	Used	259	0.304	0.3034	0.0522	5,4427.5	34,390.5	3043,296.51	3.75	1.65	$<$1e-3
		Unused	379	0.137	0.13687	0.0136						
RQ_9	H_{18}	API	200	0.098	0.09794	0.003438	16,410.5	11,245	569,606.82	3.75	1.65	$<$1e-3
		Non-API	179	0.188	0.18753	0.022609						
RQ_{10}	H_{18}	SC∩PS	258	6.08	6.08	1.44	100,104	125,518	844,576.91	3.75	1.65	$<$1e-3
		SC∩LV	388	9.43	9.43	4.41						

Wilcoxon test Values are U, U_{expt}, and U_{vari}. Decision criteria are Z, Z_{crit}, and p. The "samples" are taken from the summaries written by participants.

TABLE 7
Table Showing the Average Lengths of Keywords Used in Various Places

SC	PS	SC∩PS	LV	SC∩LV
7.22	6.54	6.08	8.46	9.43

Source code (SC), Programmer summaries (PS), and "Longer-viewed" (LV) keywords are all represented here, as well as the intersection of source code and programmer summaries keywords and the intersection between source code and "longer-viewed" keywords. The averages of significant interest here are SC∩PS and SC∩LV.

Using the Wilcoxon test, we rejected the null hypotheses (see RQ_8 in Table 6). These results indicate that the keywords used in method summaries tended to have higher *tf/idf* scores than the "longer-viewed" keywords excluded from summaries, regardless of the length of their gaze times. This reconfirms the belief that VSM *tf/idf* scores are a reliable indicator for possible inclusion in summaries and reveals that, on average, the prevalence of "stubborn" keywords is relatively low.

10.3 RQ_9: "Stubborn" API Keywords

We found statistically significant evidence that keywords that can be considered "stubborn" tend not to be API calls.

The following describes the procedure we followed to draw this conclusion: Consider the statistical data in Table 6. We compared the VSM *tf/idf* scores of "longer-viewed" API keywords not used in summaries to the *tf/idf* scores of all of the "longer-viewed" non-API keywords also not used in summaries. We then posed H_{18} as follows:

H_{18} The tf/idf scores for non-API keywords are not significantly higher than the scores of Java API names for "longer-viewed" keywords.

Using the Wilcoxon test, we rejected the null hypotheses (see RQ_9 in Table 6). These results indicate that "stubborn" keywords are generally non-API keywords. Also, since the ratio, on average, of API to non-API keywords in each method was almost 1:1, we believe that a greater use of one type of keyword over another did not significantly affect the result.

10.4 RQ_{10}: Long Keywords

We found statistically significant evidence that long keywords, that were also "longer-viewed", are used more often in the source code than they are in programmer summaries. We interpret this finding as evidence that the length of the keywords could reduce their comprehensibility, as pointed out by Liblit et al. [63], and thus reduce the possibility that those words are used in summaries. Future summarization tools may adapt to this situation by making efforts to reduce the length and complexity of keywords extracted from source code, such as by using abbreviation detection and splitting.

The following describes the procedure we followed to draw this conclusion: Consider the statistical data in Table 6. We compared the lengths of the "longer-viewed" keywords and the keyword lengths in the programmer summaries. The averages of these lengths, and some relevant others, can be seen in Table 7. We then posed H_{19} as follows:

H_{19} The average length of keywords in source code, that also had long gaze times, is not significantly greater than the average length of keywords in summaries.

Using the Wilcoxon test, we rejected the null hypotheses (see RQ_{10} in Table 6). These results indicate that "longer-viewed" keywords in source code are generally much longer than keywords used in programmer summaries.

10.5 Summary of Keyword Analysis Results

We derive two main interpretations of our keyword analysis results. First, if a keyword is considered "longer-viewed", it is not guaranteed to be included in programmers' method summaries. We base this on the finding that there is no significant correlation between "longer-viewed" keywords and keywords used in summaries (H_{16}). Second, "stubborn" keywords are not prevalent enough to be the cause of the lack of "longer-viewed" keywords in summaries. This is based on our finding that keywords used in summaries have significantly higher VSM *tf/idf* scores than the keywords that were "longer-viewed" and left out of summaries (H_{17}). However, for those "stubborn" keywords that do exist, they mostly do not consist of commonly used Java API function calls. This is based on our finding that the *tf/idf* scores for "longer-viewed" non-API keywords are significantly higher than those for API keywords (H_{18}). Our conclusion is that, considering the results from our eye-tracking study, there must be further reasons beyond "stubborn" keywords that explains the lack of "longer-viewed" keywords in method summaries. One possible further explanation we suspected is the possible complexities caused by the length of the "long-viewed" keywords. We found that "longer-viewed" keywords tend to have more characters than keywords used in programmer summaries (H_{19}). This would mean that determining the complexities behind the length of these keywords could significantly help improve the process for generating summaries.

10.6 Qualitative Analysis

In order to determine reasons for why there is the discrepancy shown in RQ_7 between long gaze times and the use of keywords in programmers' method summaries, we manually scanned the summaries for patterns. We looked for any and all forms of "longer-viewed" keyword use. We discovered that many of the words in the summaries were related to the "longer-viewed" keywords, but the exact keyword was not used. Instead, the summary words were separated versions of the keywords, which were multi-word identifiers. Here, a separated version of a keyword is when most or all of the inner words are used. The following common examples of this phenomenon show how some programmers include keywords into a method summary without actually using it.

- getcolumnnameswithprefix
  "gets the column names with the given prefix"
- setvisible
  "then makes it visible"
- isclassbeingdefined
  "checks to see if the current class is being defined"

As can be seen, rather than using keywords directly, programmers used phrases that contained parts of the

keyword. This makes sense since many of the keywords are long combinations of common, everyday words. This also falls into the category of one of the complexities behind the length of many "longer-viewed" keywords, which we discovered to be important in RQ_{10}. This phenomenon would suggest that distinguishing a keyword as "longer-viewed" is still useful, but a different analysis technique is necessary to determine how these keywords are used in practice. Designing this analysis technique is beyond the scope of this paper, but points to a key area of our future work.

11 Discussion

Our paper advances the state-of-the-art in three key directions. First, we contribute to the program comprehension literature with empirical evidence of programmer behavior during source code summarization. We recorded the eye movements of 10 professional programmers while they read and summarized several Java methods from six different applications. We have made all raw and processed data available via an online appendix (see Section 4.2.7) to promote independent research. At the same time, we have analyzed these data and found that the programmers constantly preferred to read certain areas of code over others. We found that control flow, which has been suggested as critical to comprehension during data-flow visualization [57] and software maintenance [2], was not read as heavily as other code areas during summarization. Method signatures and invocations were focused on more-often. This finding seems to confirm a recent study [6] that programmers avoid reading code details whenever possible. In contrast, the programmers seek out high-level information by reading keywords from areas that the programmers view as likely to contain such information [7]. Our study sheds light on the viewing patterns that programmers perform during summarization, in addition to the areas of code they view.

Second, we show that the keywords that programmers read are actually the keywords that an independent set of programmers felt were important. Our eye-tracking study provided evidence that programmers read certain areas of code, but that evidence alone is not sufficient to conclude that keywords from those areas should be included in source code summaries—it is possible that the programmers read those sections more often because they were harder to understand. The tool we presented in Section 6 is designed to study this problem. It is based on a state-of-the-art approach [19] for extracting summary keywords from code, except that our tool favors the keywords from the sections of source code that programmers read during the eye-tracking study.

In an evaluation of this tool, we found that an independent set of programmers preferred the keywords from our tool as compared to the state-of-the-art tool. This finding confirms that the sections of code that programmers read actually contain the keywords that should be included in summaries.

Third, we reveal that there are further steps that need to be taken in order to create usable summaries. Although we have seen that certain keywords are more important for summarization to programmers, we cannot be sure that they would use those keywords in method summaries. We analyzed the written summaries of the 10 professional programmers to determine exactly what keywords they used. At first glance, it appears that programmers do not necessarily favor keywords from certain sections over others when writing method summaries. However, we discovered that a major reason for this phenomenon is that keywords in code are not very usable as words in an English paragraph. Many summaries contained phrases using pieces of the important keywords, but written in a shorter, more readable way. This discovery shows that the important keywords found using our methods should be indirectly included in method summaries after pre-processing them to make them into English phrases.

An additional benefit of this work is the improvement of existing source code summarization tools. While we have demonstrated one approach, progressive improvements to other techniques may be possible based on this work. Different source code summarization tools have generated natural language summaries, instead of keyword lists [13], [14], [15], [16], [17]. These approaches have largely been built using assumptions about what programmers need in summaries, or from program comprehension studies of tasks other than summarization. Our work can assist code summarization research by providing a guide to the sections of code that should be targeted for summarization. At the same time, our work may assist in creating metrics for source code comment quality [75] by providing evidence about which sections of the code the comments should reflect.

12 Future Work

Although we have completed this study, we do not believe that everything that we could explore or discover has been covered here. Therefore, there is a plethora of possible future work that we would like to mention in order to keep the discussion open. First, a helpful addition to this work would be to include an analysis of the effects of long, multi-word identifiers on the gaze times, fixations, and regressions during a method. This was shown to be significant during our exploration of stubborn keywords. One could also include this addition to the tool we created and evaluated during this study to explore if the weights could be modified based on keyword length in order to compensate for these effects. Second, we believe that a logical next step to this work would be to explore if the results found with these professional programmers also applies to programming novices. An identical, but separate novice study could be conducted and a statistical comparison could be made to judge the effects of programming experience. Third, since we limited our study to general summaries written by the average developer, an exploration of the effects of different types of developers could produce interesting, possibly incomparable results. This exploration could include a comparison between the eye-tracking results from several differing types, including testers, security experts, database experts, etc. Fourth, we used complete, well-formed code, all written in English and all without any internal comments. However, this is also not the most realistic scenario to be expected, especially in an open source environment. Therefore, we believe an important future work would be to explore the effects of incomplete, semi-documented, and/or non-English code could have on the results we have found here. Last, we believe that simply delving deeper into the work we have presented here could be possible future work. Looking more into stubborn

keywords, especially if multi-word identifiers are a focus, could produce better conclusions than we could provide in this study. Also, more fine-grained development of our small tool, such as creating a regression model for weights rather than hand-picking them, could be beneficial. In general, we believe that further exploration into this area is important for the several new techniques and tools to come.

13 Conclusion

We have presented an eye-tracking study of programmers during source code summarization, a tool for selecting keywords based on the findings of the eye-tracking study, and an evaluation of that tool. We have explored six research questions aimed at understanding how programmers read, comprehend, and summarize source code. We showed how programmers read method signatures more-closely than method invocations, and invocations more-closely than control flow. These findings led us to design and build a tool for extracting keywords from source code. Our tool outperformed a state-of-the-art tool during a study with an independent set of expert programmers. The superior performance of our tool reinforces the results from our eye-tracking study: not only did the programmers read keywords from some sections of source code more-closely than others during summarization, but they also tended to use those keywords in their own summaries. We also observed that programmers sometimes did not use these exact keywords, but instead used variations of or sub-words included in these keywords.

Acknowledgments

The authors thank and acknowledge the 10 Research Programmers at the Center for Research Computing at the University of Notre Dame for participating in our eye-tracking study. They also thank the nine graduate students and programmers who participated in our follow-up study of our approach. They also sincerely thank Nigel Bosch and Sidney D'Mello for their contributions to previous editions of this paper.

References

[1] T. D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," in *Proc. 28th Int. Conf. Softw. Eng.*, New York, NY, USA, 2006, pp. 492–501.

[2] 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 Trans. Softw. Eng.*, vol. 32, no. 12, pp. 971–987, Dec. 2006.

[3] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An examination of software engineering work practices," in *Proc. Conf. Centre Adv. Studies Collaborative Res.*, 1997, p. 21.

[4] A. Lakhotia, "Understanding someone else's code: Analysis of experiences," *J. Syst. Softw.*, vol. 23, no. 3, pp. 269–275.

[5] R. K. Fjeldstad and W. T. Hamlen, "Application program maintenance study: Report to our respondents," in *Proc. GUIDE 48*, Apr. 1983, , pp. 13–30.

[6] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 255–265.

[7] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2009, pp. 157–166.

[8] D. Kramer, "API documentation from source code comments: A case study of Javadoc," in *Proc. 17th Annu. Int. Conf. Comput. Documentation*, New York, NY, USA, 1999, pp. 147–153.

[9] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. 23rd Annu. Int. Conf. Des. Commun.: Documenting Designing Pervasive Inf.*, New York, NY, USA, 2005, pp. 68–75.

[10] M. Kajko-Mattsson, "A survey of documentation practice within corrective maintenance," *Empirical Softw. Eng.*, vol. 10, no. 1, pp. 31–55, Jan. 2005.

[11] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? On the relation between source code and comment changes," in *Proc. 14th Working Conf. Reverse Eng.*, Washington, DC, USA, 2007, pp. 70–79.

[12] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Trans. Softw. Eng.*, vol. 34, no. 4, pp. 434–451, Jul. 2008.

[13] H. Burden and R. Heldal, "Natural language generation from class diagrams," in *Proc. 8th Int. Workshop Model-Driven Eng., Verification Validation*, New York, NY, USA, 2011, pp. 8:1–8:8.

[14] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: Approach for unsupervised bug report summarization," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, New York, NY, USA, 2012, pp. 11:1–11:11.

[15] L. Moreno, J. Aponte, S. Giriprasad, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes," in *Proc. 21st Int. Conf. Program Comprehension*, 2013, pp. 23–32.

[16] G. Sridhara, "Automatic generation of descriptive summary comments for methods in object-oriented programs," Ph.D. dissertation, Univ. of Delaware, Newark, DE, USA, Jan. 2012.

[17] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Generating parameter comments and integrating with method summaries," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, Washington, DC, USA, 2011, pp. 71–80.

[18] A. T. T. Ying and M. P. Robillard, "Code fragment summarization," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, New York, NY, USA, 2013, pp. 655–658.

[19] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Proc. 17th Working Conf. Reverse Eng.*, Washington, DC, USA, 2010, pp. 35–44.

[20] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, New York, NY, USA, 2010, pp. 43–52.

[21] B. Eddy, J. Robinson, N. Kraft, and J. Carver, "Evaluating source code summarization techniques: Replication and expansion," in *Proc. 21st Int. Conf. Program Comprehension*, 2013, pp. 13–22.

[22] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," in *Proc. 33rd Int. Conf. Softw. Eng.*, New York, NY, USA, 2011, pp. 101–110.

[23] R. Holmes and R. J. Walker, "Systematizing pragmatic software reuse," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 4, pp. 20:1–20:44, Feb. 2013.

[24] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *J. Vis. Lang. Comput.*, vol. 16, no. 12, pp. 41–84, 2005.

[25] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "Mental models and software maintenance," *J. Syst. Softw.*, vol. 7, no. 4, pp. 341–355, 1987.

[26] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: Integrating web search into the development environment," in *Proc. 28th Int. Conf. Human Factors Comput. Syst.*, New York, NY, USA, 2010, pp. 513–522.

[27] G. Kotonya, S. Lock, and J. Mariani, "Opportunistic reuse: Lessons from scrapheap software development," in *Proc. 11th Int. Symp. Component-Based Softw. Eng.*, Berlin, Germany, 2008, pp. 302–309.

[28] J. W. Davison, D. M. Mancl, and W. F. Opdyke, "Understanding and addressing the essential costs of evolving systems," *Bell Labs Tech. J.*, vol. 5, pp. 44–54, 2000.

[29] S. Mirghasemi, J. J. Barton, and C. Petitpierre, "Querypoint: Moving backwards on wrong values in the buggy execution," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, New York, NY, USA, 2011, pp. 436–439.

[30] J.-P. Krämer, J. Kurz, T. Karrer, and J. Borchers, "Blaze," in *Proc. Int. Conf. Softw. Eng.*, Piscataway, NJ, USA, 2012, pp. 1457–1458.

[31] S. E. Sim, C. L. A. Clarke, and R. C. Holt, "Archetypal source code searches: A survey of software developers and maintainers," in *Proc. 6th Int. Workshop Program Comprehension*, Washington, DC, USA, 1998, p. 180.

[32] E. M. Altmann, "Near-term memory in programming: A Simulation-based analysis," *Int. J. Human-Comput. Studies*, vol. 54, no. 2, pp. 189–210, 2001.

[33] C. Douce, "Long term comprehension of software systems: A methodology for study," *Proc. Psychol. Programm. Interest Group*, 2001, pp. 147–159.

[34] V. M. González and G. Mark, "'Constant, constant, multi-tasking craziness': Managing multiple working spheres," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, New York, NY, USA, 2004, pp. 113–120.

[35] A. Guzzi, "Documenting and sharing knowledge about code," in *Proc. Int. Conf. Softw. Eng.*, Piscataway, NJ, USA, 2012, pp. 1535–1538.

[36] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A survey," in *Proc. ACM Symp. Document Eng.*, New York, NY, USA, 2002, pp. 26–33.

[37] G. Gweon, L. Bergman, V. Castelli, and R. K. E. Bellamy, "Evaluating an automated tool to assist evolutionary document generation," in *Proc. IEEE Symp. Vis. Lang. Human-Centric Comput.*, Washington, DC, USA, 2007, pp. 243–248.

[38] L. Bergman, V. Castelli, T. Lau, and D. Oblinger, "Docwizards: A system for authoring follow-me documentation wizards," in *Proc. 18th Annu. ACM Symp. User Interface Softw. Technol.*, New York, NY, USA, 2005, pp. 191–200.

[39] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *Proc. Vis. Lang. Human-Centric Comput.*, Washington, DC, USA, 2006, pp. 195–202.

[40] T. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE Softw.*, vol. 20, no. 6, pp. 35–39, Nov./Dec. 2003.

[41] M. E. Crosby and J. Stelovsky, "How do we read algorithms? A case study," *IEEE Comput.*, vol. 23, no. 1, pp. 24–35, Jan. 1990.

[42] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proc. Symp. Eye Tracking Res. Appl.*, New York, NY, USA, 2006, pp. 133–140.

[43] R. Bednarik and M. Tukiainen, "Temporal eye-tracking data: Evolution of debugging strategies with multiple representations," in *Proc. Symp. Eye Tracking Res. & Appl.*, New York, NY, USA, 2008, pp. 99–102.

[44] R. Bednarik and M. Tukiainen, "An eye-tracking methodology for characterizing program comprehension processes," in *Proc. Symp. Eye Tracking Res. Appl.*, New York, NY, USA, 2006, pp. 125–132.

[45] N. Ali, Z. Sharafl, Y. Gueheneuc, and G. Antoniol, "An empirical study on requirements traceability using eye-tracking," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, Sept 2012, pp. 191–200.

[46] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *Proc. Symp. Eye Tracking Res. Appl.*, New York, NY, USA, 2012, pp. 381–384.

[47] B. Sharif and J. I. Maletic, "An eye tracking study on camelcase and under_score identifier styles," in *Proc. IEEE 18th Int. Conf. Program Comprehension*, Washington, DC, USA, 2010, pp. 196–205.

[48] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif, "The impact of identifier style on effort and comprehension," *Empirical Softw. Eng.*, vol. 18, no. 2, pp. 219–276, Apr. 2013.

[49] B. Sharif, "Empirical assessment of UML class diagram layouts based on architectural importance," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, Washington, DC, USA, 2011, pp. 544–549.

[50] B. Sharif and J. I. Maletic, "The effects of layout on detecting the role of design patterns," in *Proc. 23rd IEEE Conf. Softw. Eng. Edu. Training*, Washington, DC, USA, 2010, pp. 41–48.

[51] G. C. Murphy, "Lightweight structural summarization as an aid to software evolution," Ph.D. dissertation, University of Washington, Seattle, WA, USA, Jul. 1996.

[52] S. Zhang, C. Zhang, and M. D. Ernst, "Automated documentation inference to explain failed tests," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, Washington, DC, USA, 2011, pp. 63–72.

[53] R. P. Buse and W. R. Weimer, "Automatic documentation inference for exceptions," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 273–282.

[54] R. P. Buse and W. R. Weimer, "Automatically documenting program changes," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2010, pp. 33–42.

[55] M. Kim, D. Notkin, D. Grossman, and G. Wilson, "Identifying and summarizing systematic code changes via rule inference," *IEEE Trans. Softw. Eng.*, vol. 39, no. 1, pp. 45–62, Jan. 2013.

[56] J. Aponte and A. Marcus, "Improving traceability link recovery methods through software artifact summarization," in *Proc. 6th Int. Workshop Traceability Emerging Forms Softw. Eng.*, New York, NY, USA, 2011, pp. 46–49.

[57] D. Dearman, A. Cox, and M. Fisher, "Adding control-flow to a visual data-flow representation," in *Proc. 13th Int. Workshop Program Comprehension*, Washington, DC, USA, 2005, pp. 297–306.

[58] K. Anjaneyulu and J. Anderson, "The advantages of data flow diagrams for beginning programming," in *Proc. 2nd Int. Conf. Intell. Tutoring Syst.*, 1992, vol. 608, pp. 585–592.

[59] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: Finding relevant functions and their usage," in *Proc. 33rd Int. Conf. Softw. Eng.*, New York, NY, USA, 2011, pp. 111–120.

[60] K. Rayner, A. Pollatsek, and E. D. Reichle, "Eye movements in reading: Models and data," *Behavioral Brain Sci.*, vol. 26, pp. 507–518, 2003.

[61] D. A. Wolfe and M. Hollander, "Nonparametric statistical methods," New York, 1973.

[62] B. Walters, T. Shaffer, B. Sharif, and H. Kagdi, "Capturing software traceability links from developers' eye gazes," in *Proc. 22nd Int. Conf. Program Comprehension*, New York, NY, USA, 2014, pp. 201–204.

[63] B. Liblit, A. Begel, and E. Sweeser, "Cognitive perspectives on the role of naming in computer programs," in *Proc. 18th Annu. Psychol. Programm. Workshop*, Sep. 2006.

[64] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," in *Proc. 14th Annu. ACM-SIAM Symp. Discrete Algorithms*, Philadelphia, PA, USA, 2003, pp. 28–36.

[65] M. S. Bansal and D. Fernández-Baca, "Computing distances between partial rankings," *Inf. Process. Lett.*, vol. 109, no. 4, pp. 238–241, Jan. 2009.

[66] M. D. Smucker, J. Allan, and B. Carterette, "A comparison of statistical significance tests for information retrieval evaluation," in *Proc. 16th ACM Conf. Inf. Knowl. Manage.*, 2007, pp. 623–632.

[67] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards mining replacement queries for hard-to-retrieve traces," in *Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng.*, New York, NY, USA, 2010, pp. 245–254.

[68] X. Zou, R. Settimi, and J. Cleland-Huang, "Improving automated requirements trace retrieval: A study of term-based enhancement methods," *Empirical Softw. Eng.*, vol. 15, no. 2, pp. 119–146, Apr. 2010.

[69] D. Hou and D. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping api methods for code completion," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, Sep. 2011, pp. 233–242.

[70] J. M. Daughtry III and J. M. Carroll, "Perceived self-efficacy and APIs," Programming Interest Group, p. 42.

[71] C. McMillan, D. Poshyvanyk, and M. Grechanik, "Recommending source code examples via API call usages and documentation," in *Proc. 2nd Int. Workshop Recommendation Syst. Softw. Eng.*, New York, NY, USA, 2010, pp. 21–25.

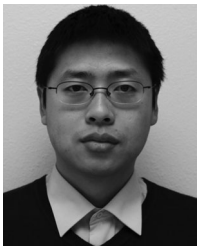
[72] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1069–1087, Sep./Oct. 2011.

[73] L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: An Introduction to Cluster Analysis*. New York, NY, USA: Wiley, 1990.

[74] H.-S. Park and C.-H. Jun, "A simple and fast algorithm for k-medoids clustering," *Expert Syst. Appl.*, vol. 36, no. 2, pp. 3336–3341, Mar. 2009.

[75] D. Steidl, B. Hummel, and E. Juergens, "Quality analysis of source code comments," in *Proc. 21st Int. Conf. Program Comprehension*, 2013, pp. 83–92.

Paige Rodeghero is a Computer Science PhD student at the University of Notre Dame working under Dr. Collin McMillan. She completed her Bachelor's Degree at Ball State University with a major in Computer Science and a minor in Dance Performance. Her current research focuses primarily on program comprehension and source code summarization.



Cheng Liu received his PhD in Physics from the University of Illinois at Urbana-Champaign in 2011. He then joined the Center for Research Computing (CRC) at the University of Notre Dame as a research programmer after graduation. In 2015, he became a computational scientist at the CRC and also a research assistant professor in the Department of Psychology at Notre Dame. Dr. Lius research interests include data analytics, software engineering, cognitive computing, and applications of statistical modeling in psychological and educational measurement.

Paul W. McBurney is a PhD student in Computer Science at the University of Notre Dame working under Dr. Collin McMillan. He received his Bachelor's Degree and Master's Degree in Computer Science from West Virginia University. His research interest is Software Engineering with a focus on automatic documentation and program comprehension.

Collin McMillan is an Assistant Professor at the University of Notre Dame. He completed his PhD in 2012 at the College of William and Mary, focusing on source code search and traceability technologies for program reuse and comprehension. Since joining Notre Dame, his work has focused on source code summarization and efficient reuse of executable code. Dr. McMillan's work has been recognized with the National Science Foundation's CAREER award.