Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers

Paige Rodeghero, Collin McMillan, Paul W. McBurney,
Nigel Bosch, and Sidney D'Mello
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{prodeghe, cmc, pmcburne, pbosch1, sdmello}@nd.edu

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 and provide evidence to support the development of source code summarization systems.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*Productivity*

General Terms

Algorithms, Documentation

Keywords

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 [36, 31, 52]. However, studies of program comprehension consistently find that programmers would prefer to focus on small sections of code during software maintenance [52, 35, 36, 20], and "try to avoid" [45] comprehending the entire system. The result is that programmers skim source code (e.g., read only method signatures or important keywords) to save time [58]. 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 [33]. These summaries are popular, but have a tendency to be incomplete [15, 28] or outdated as code changes [21, 50].

As a result, automated source code summarization tools are emerging as viable techniques for generating summaries without human intervention [10, 39, 42, 54, 57, 63]. 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) [26]. 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 [55].

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 three-fold. First, we conduct an eye-tracking study of 10 Java programmers. During the study, the programmers read Java methods and wrote summaries for those methods. 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 tool that selects keywords from Java methods. Finally, we compared the keyword selections from our tool to the keywords selected using a state-of-the-art approach [26]. We found that our tool improved over the state-of-the-art when compared to keyword lists written by human evaluators.

ICSE '14, June 01 - 07 2014, Hyderabad, Andhra Pradesh, India

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 [26]. Moreno *et al.* [42] and Eddy *et al.* [18] 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 [56]. In prior work, they had identified different heuristics for statements within Java methods, to describe the key functionality of these methods [55]. 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. Holmes *et al.* and Ko *et al.* have observed that many of these studies target the strategies followed by programmers and the knowledge that they seek [31, 27]. For example, during maintenance tasks, some programmers follow a "systemic" strategy aimed at understanding how different parts of the code interact [30, 38]. In contrast, an "opportunistic" strategy aims to find only the section of code that is needed for a particular change [9, 32, 14, 35]. In either strategy, studies suggest that programmers form a mental model of the software [36], specifically including the relationships between different sections of source code [41, 34, 50, 51]. Documentation and note-taking is widely viewed as important because programmers tend to forget the knowledge they have gained about the software [1, 17]. At the same time, programmers are often interrupted with requests for help with understanding source code [23], and written notes assist developers in answering these questions [24].

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 [45]. 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 [58, 22]. 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 [25, 7]. Stylos *et al.* recommend highlighting summaries of code with references to dependent source code [60]. These recommendations corroborate work by Lethbridge *et al.* that found that documentation should provide a high-level understanding of source code [37]. 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 *et al.* performed one early study, and concluded that the process of reading source code is different from the process of reading prose [13]. Programmers alternate between comments and source code, and fixate on important sections, rather than read the entire document at a steady pace [13]. 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 [61]. In two separate studies Bednarik *et al.* 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 [6, 5]. 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 [47]. Eye-tracking has been used in studies of identifier style [49, 8] and UML diagram layout [46, 48], 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 [43]. Work that creates these descriptions includes work by Sridhara *et al.* [55, 57], Haiduc *et al.* [26], and Moreno *et al.* [42] as mentioned above. Earlier work includes approaches to explain failed tests [64], Java exceptions [11], change log messages [12], and systemic software evolution [29]. Studies of these techniques have shown that summarization is effective in comprehension [18] and traceability link recovery [3]. 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 (VSM), which is a classic natural language understanding technique [26]. 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 [18]. See Section 6.3 for an example of the output from this approach.

4. EYE-TRACKING STUDY DESIGN

This section describes our research questions, 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 (RQ) that we pose:

RQ_1 To what degree do programmers focus on the keywords that the VSM *tf/idf* technique [26, 18] extracts?

RQ_2 Do programmers focus on a method's **signature** more than the method's body?

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 [26, 18]. 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 [16, 31], while contradictory evidence suggests that other regions may be more valuable [2]. 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 [40, 50, 31, 60]. 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. Figure 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 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. From these logs, we calculated three types of eye-movement behavior to answer our research questions: gaze time, fixations, and regressions. Gaze time is the total number of milliseconds spent on a region of interest (ROI - e.g., a method, a keyword). Fixations are any locations that a participant viewed for more than 100 milliseconds. Regressions are locations that the participant read once, then read again after reading other locations. These three types of eye movements are widely used in the eye-tracking literature [44, 13, 61]. 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 7161 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.

Figure 1: Interface of the eye-tracking device.

4.2.3 Participants

We recruited 10 programmers to participate in our study. These programmers were employees of the Computing Research Center (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 programming experience ranged from 6 to 27 years, averaging 13.3 years.

4.2.4 Statistical Tests

We compared gaze time, fixation, and regression counts using the Wilcoxon signed-rank test [62]. 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 T300 Eye-Tracker device for our study. The device has a resolution of 1920x1080 and a 300 Hz sampling rate. A technician was available at all times to monitor the equipment. The sampling rate was lowered to 120Hz to reduce computational load; such a sample rate is acceptable for simple ROI analyses like the ones conducted here.

4.2.6 Threats to Validity

Our methodology avoids different biases and technical limitations. 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. 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 when the participant repeatedly looks away from the screen. We were also limited in the size of the Java methods: the interface did support scrolling or navigation, and accuracy decreases as the font becomes smaller. 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.

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 Figure 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`. The overlap for the top five was 60% 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 ten keywords selected by the VSM *tf/idf* approach overlapped with the top ten 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 53 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 ten 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.

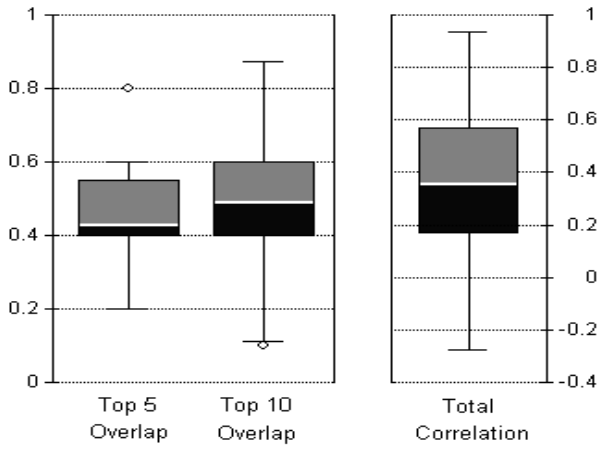


Figure 2: Data for RQ_1. Plots to the left show percentage overlap of VSM tf/idf top keywords to the top most-read keywords in terms of gaze time. Plot to the right shows Pearson correlation between VSM tf/idf values and gaze time for all keywords. The white line is the mean. The black box is the lower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

Table 1: Statistical summary of the results for RQ_2, RQ_3, and RQ_4. 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.

RQ	H	Metric	Method Area	Samples	$\tilde{x}$	μ	Vari.	T	T_{expt}	T_{vari}	Z	Z_{crit}	p
RQ_2	H_1	Gaze	Signature	95	1.061	1.784	4.237	2808	2280	72580	1.96	1.65	0.025
			Non-Sig.	95	0.996	0.933	0.054						
	H_2	Fixation	Signature	95	1.150	1.834	4.451	3008	2280	72580	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	2307	2280	72580	0.10	1.65	0.459
			Non-Sig.	95	1.014	0.978	0.032						
RQ_3	H_4	Gaze	Ctrl. Flow	111	0.781	0.924	0.392	1956	3108	115514	-3.389	1.96	0.001
			Non-Ctrl.	111	1.116	1.134	0.145						
	H_5	Fixation	Ctrl. Flow	111	0.834	0.938	0.274	2140	3108	115514	-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	1463	3108	115514	-4.840	1.96	<1e-3
			Non-Ctrl.	111	1.132	1.199	0.165						
RQ_4	H_7	Gaze	Invocations	106	0.968	1.069	0.778	2586	2836	100660	-0.786	1.96	0.432
			Non-Invc.	106	1.021	1.027	0.086						
	H_8	Fixation	Invocations	106	1.003	1.048	0.385	2720	2835	100660	-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	2391	2835	100660	-1.399	1.96	0.162
			Non-Invc.	106	1.028	1.045	0.065						

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% of their gaze time reading signatures, even though the signatures only averaged 12% 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% of his or her time reading a method's signature, and the signature contains 3 of the 20 keywords (15%) 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% of their time reading control flow keywords, even though these keywords averaged 37% of the keywords in the methods. To determine the significance of the 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.

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 [26], 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 is occurs in the a method invocation, and 0.9 if in a control flow statement (if a keyword occurrence is in both a control flow and invocation area, we count it as 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.

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

6.3 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 Figure 3. In this example, where VSM *tf/idf* increments each weight a fixed amount of each occurence 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 6902 different methods in the project. But it occurs twice in the `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 occurs 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;
}
```

Figure 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 [26]. 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* [26]), 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 [19, 4], and we follow the procedure recommended by Fagin *et al.* [19]: 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 [53]. 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. 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-5 keyword lists that contain, on average, 76% of the keywords that programmers selected during the study. In other words, almost 4 out of 5 of the keywords extracted by Eye_C were also selected by human experts. In contrast 67% of the keywords, just over 3 of 5, 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% of the 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% from the default VSM *tf/idf* approach (67%) to our best-performing approach, Eye_C (76%).

Figure 4(a) 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.

Table 3: Data for RQ_5. Overlap for top-5 lists.

	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

Table 4: Data for RQ_6. K_{min} for top-5 lists.

	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

This finding is strong evidence that some areas of code should be prioritized over other areas for summarization. The programmers in our study preferred the keywords from our approach by a statistically-significant margin. We expand on the implications of these findings in Section 9.

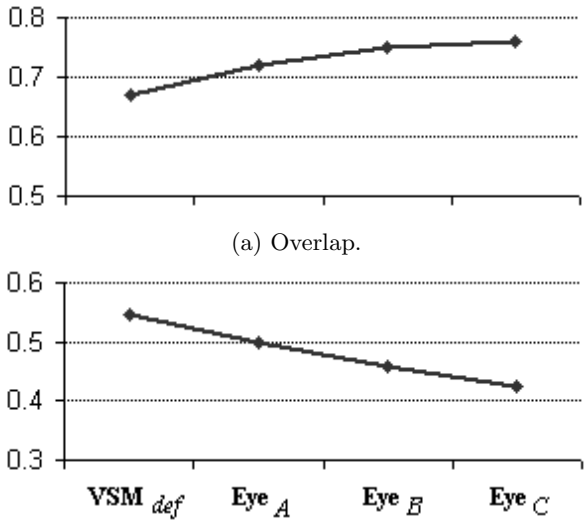
8.2 RQ_6: Keyword List Order

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 key-



(a) Overlap.

(b) K_{min}.

Figure 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.

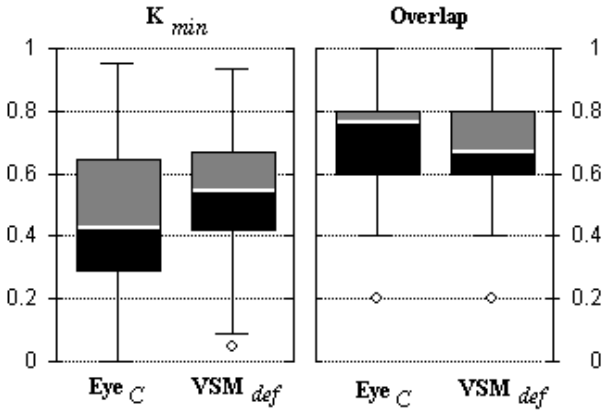


Figure 5: Boxplots comparing VSM_{def} with Eye_C, the best performing configuration of our approach. Each datapoint in each boxplot represents one K_{min} or Overlap value. That value is calculated between two keyword lists: one generated by the given approach for a method, and one written by a participant 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}.

word 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% 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 Figure 4(b) 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 for keywords based on code area; it also experienced the highest level of performance of any approach tested here.

9. DISCUSSION

Our paper advances the state-of-the-art in two 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 [16, 31], 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 [45] 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 [58]. 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 [26] 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 (e.g., *not* the same participants from the eye-tracking study) 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.

An additional benefit of this work is the improvement of existing source code summarization tools. While we have

Table 5: Statistical summary of the results for RQ_5 and RQ_6. Mann-Whitney test values are U, U_{expt}, and U_{vari}. Decision criteria are Z, Z_{crit}, and p. Testing procedure is identical to Table 1.

RQ	Metric	H	Approach	Samples	$\tilde{x}$	μ	Vari.	T	T_{expt}	T_{vari}	Z	Z_{crit}	p
RQ_5	Overlap	H_{10}	Eye_A	170	0.800	0.719	0.030	6725	3698	234964	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	9092	5175	331605	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	9628	5727	360607	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	3519	6878	406567	-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	3028	7222	412696	-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	2971	7254	412950	-6.664	1.96	<1e-3
			VSM_{def}	170	0.533	0.545	0.032						

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 [10, 39, 42, 54, 57]. 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 targetted for summarization. At the same time, our work may assist in creating metrics for source code comment quality [59] by providing evidence about which sections of the code the comments should reflect.

Although this work contributes important evidence and advances the state-of-the-art in source code summarization, there are some limitations which we aim to address in future work. Figure 5 shows a comparison between Eye_C, the best-performing configuration of our approach, and VSM_{def}. The average values are improved for both K_{min} and overlap, but both our approach and VSM_{def} perform poorly on certain methods (visible as a substantial tail in the boxplots in Figure 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. An area of future work is to study these methods to determine how our approach might handle them differently.

Another area of future work is to expand the set of keywords that our approach is able to extract. The evaluation of our approach focused on the top-5 keywords extracted from Java methods. The limit of 5 was selected due to practical concerns (see Section 7.2). However, summaries may consist of other words which support the meaning of the summary. Natural language summaries, in particular, would include numerous supporting words in complete sentences. It is an area of future work to determine the appropriate strategies for finding and organizing these words. It is important to understand these strategies because the meaning of the keywords extracted from the code may differ slightly (or be modified) depending on the surrounding keywords.

10. 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.

Acknowledgments

We 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. We also thank the nine graduate students and programmers who participated in our follow-up study of our approach. The fourth and fifth authors were supported by the National Science Foundation (NSF) (HCC 0834847 and DRL 1235958). Any opinions, findings and conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the NSF.

11. REFERENCES

[1] E. M. Altmann. Near-term memory in programming: a simulation-based analysis. *International Journal of Human-Computer Studies*, 54(2):189 – 210, 2001.

[2] K. Anjaneyulu and J. Anderson. The advantages of data flow diagrams for beginning programming. In C. Frasson, G. Gauthier, and G. McCalla, editors, *Intelligent Tutoring Systems*, volume 608 of *Lecture Notes in Computer Science*, pages 585–592. Springer Berlin Heidelberg, 1992.

[3] J. Aponte and A. Marcus. Improving traceability link recovery methods through software artifact summarization. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '11, pages 46–49, New York, NY, USA, 2011. ACM.

[4] M. S. Bansal and D. Fernández-Baca. Computing distances between partial rankings. *Inf. Process. Lett.*, 109(4):238–241, Jan. 2009.

[5] R. Bednarik and M. Tukiainen. An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, ETRA '06, pages 125–132, New York, NY, USA, 2006. ACM.

[6] R. Bednarik and M. Tukiainen. Temporal eye-tracking data: evolution of debugging strategies with multiple representations. In *Proceedings of the 2008 symposium on Eye tracking research & applications*, ETRA '08, pages 99–102, New York, NY, USA, 2008.

[7] L. Bergman, V. Castelli, T. Lau, and D. Oblinger. Docwizards: a system for authoring follow-me documentation wizards. In *Proceedings of the 18th annual ACM symposium on User interface software and technology*, UIST '05, pages 191–200, New York, NY, USA, 2005. ACM.

[8] 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. Engg.*, 18(2):219–276, Apr. 2013.

[9] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: integrating web search into the development environment. In *Proceedings of the 28th international conference on Human factors in computing systems*, CHI '10, pages 513–522, New York, NY, USA, 2010. ACM.

[10] H. Burden and R. Heldal. Natural language generation from class diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa, pages 8:1–8:8, New York, NY, USA, 2011. ACM.

[11] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software*

testing and analysis, ISSTA '08, pages 273–282.

[12] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 33–42.

[13] M. E. Crosby and J. Stelovsky. How do we read algorithms? a case study. *Computer*, 23(1):24–35, Jan.

[14] J. W. Davison, D. M. Mancl, and W. F. Opdyke. Understanding and addressing the essential costs of evolving systems. *Bell Labs Technical Journal*, pages 44–54, 2000.

[15] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, SIGDOC '05, pages 68–75, New York, NY, USA, 2005.

[16] D. Dearman, A. Cox, and M. Fisher. Adding control-flow to a visual data-flow representation. In *Proceedings of the 13th International Workshop on Program Comprehension*, IWPC '05, pages 297–306, Washington, DC, USA, 2005. IEEE Computer Society.

[17] C. Douce. Long term comprehension of software systems: A methodology for study. *Proc. Psychology of Programming Interest Group*, 2001.

[18] B. Eddy, J. Robinson, N. Kraft, and J. Carver. Evaluating source code summarization techniques: Replication and expansion. In *Proceedings of the 21st International Conference on Program Comprehension*, ICPC '13, 2013.

[19] R. Fagin, R. Kumar, and D. Sivakumar. Comparing top k lists. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 28–36, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[20] R. K. Fjeldstad and W. T. Hamlen. Application Program Maintenance Study: Report to Our Respondents. In *Proceedings GUIDE 48*, Apr. 1983.

[21] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 70–79, Washington, DC, USA, 2007.

[22] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng '02, pages 26–33, New York, NY, USA, 2002. ACM.

[23] V. M. González and G. Mark. "constant, constant, multi-tasking craziness": managing multiple working spheres. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 113–120, New York, NY, USA, 2004. ACM.

[24] A. Guzzi. Documenting and sharing knowledge about code. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1535–1538, Piscataway, NJ, USA, 2012. IEEE Press.

[25] G. Gweon, L. Bergman, V. Castelli, and R. K. E. Bellamy. Evaluating an automated tool to assist evolutionary document generation. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '07, pages 243–248, Washington, DC, USA, 2007.

[26] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 35–44, Washington, DC, USA, 2010. IEEE Computer Society.

[27] R. Holmes and R. J. Walker. Systematizing pragmatic software reuse. *ACM Trans. Softw. Eng. Methodol.*, 21(4):20:1–20:44, Feb. 2013.

[28] M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Softw. Engg.*, 10(1):31–55, Jan. 2005.

[29] M. Kim, D. Notkin, D. Grossman, and G. Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45 –62, Jan. 2013.

[30] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16(1âĂŞ2):41 – 84, 2005.

[31] 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.*, 32(12):971–987, Dec. 2006.

[32] G. Kotonya, S. Lock, and J. Mariani. Opportunistic reuse: Lessons from scrapheap software development. In *Proceedings of the 11th International Symposium on Component-Based Software Engineering*, CBSE '08, pages 302–309, Berlin, Heidelberg, 2008.

[33] D. Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, SIGDOC '99, pages 147–153, New York, NY, USA, 1999. ACM.

[34] J.-P. Krämer, J. Kurz, T. Karrer, and J. Borchers. Blaze. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1457–1458, Piscataway, NJ, USA, 2012. IEEE Press.

[35] A. Lakhotia. Understanding someone else's code: analysis of experiences. *J. Syst. Softw.*, 23(3):269–275.

[36] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[37] T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: the state of the practice. *Software, IEEE*, 20(6):35–39, 2003.

[38] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. *Journal of Systems and Software*, 7(4):341 – 355, 1987.

[39] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 11:1–11:11, New York, NY, USA, 2012. ACM.

[40] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International*

Conference on Software Engineering, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.

[41] S. Mirghasemi, J. J. Barton, and C. Petitpierre. Querypoint: moving backwards on wrong values in the buggy execution. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 436–439, New York, NY, USA.

[42] L. Moreno, J. Aponte, S. Giriprasad, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the 21st International Conference on Program Comprehension*, ICPC '13.

[43] G. C. Murphy. *Lightweight structural summarization as an aid to software evolution*. PhD thesis, University of Washington, July 1996.

[44] K. Rayner, A. Pollatsek, and E. D. Reichle. Eye movements in reading: Models and data. *Behavioral and Brain Sciences*, 26:507–518, 7 2003.

[45] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 255–265.

[46] B. Sharif. Empirical assessment of uml class diagram layouts based on architectural importance. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 544–549, Washington, DC, USA, 2011.

[47] B. Sharif, M. Falcone, and J. I. Maletic. An eye-tracking study on the role of scan time in finding source code defects. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, ETRA '12, pages 381–384, New York, NY, USA, 2012. ACM.

[48] B. Sharif and J. I. Maletic. The effects of layout on detecting the role of design patterns. In *Proceedings of the 2010 23rd IEEE Conference on Software Engineering Education and Training*, CSEET '10, pages 41–48, Washington, DC, USA, 2010.

[49] B. Sharif and J. I. Maletic. An eye tracking study on camelcase and under_score identifier styles. In *Proceedings of the 2010 IEEE 18th International Conference on Program Comprehension*, ICPC '10, pages 196–205, Washington, DC, USA, 2010.

[50] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451.

[51] S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension*, IWPC '98, pages 180–, Washington, DC, USA, 1998.

[52] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '97, pages 21–. IBM Press, 1997.

[53] M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. In *CIKM*, pages 623–632, 2007.

[54] G. Sridhara. *Automatic Generation of Descriptive Summary Comments for Methods in Object-oriented Programs*. PhD thesis, University of Delaware, Jan.

[55] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 43–52, New York, NY, USA, 2010. ACM.

[56] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 101–110, New York, NY, USA, 2011.

[57] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*, ICPC '11, pages 71–80, Washington, DC, USA, 2011. IEEE Computer Society.

[58] J. Starke, C. Luce, and J. Sillito. Searching and skimming: An exploratory study. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 157–166, 2009.

[59] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension*, ICPC '13, 2013.

[60] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing*, VLHCC '06, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society.

[61] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, ETRA '06, pages 133–140, New York, NY, USA, 2006. ACM.

[62] D. A. Wolfe and M. Hollander. Nonparametric statistical methods. *Nonparametric statistical methods*.

[63] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 655–658, New York, NY, USA, 2013. ACM.

[64] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 63–72, Washington, DC, USA, 2011. IEEE Computer Society.