A Comparison of Program Comprehension Strategies by Blind and Sighted Programmers

Ameer Armaly, Paige Rodeghero, and Collin McMillan, *Member, IEEE*

Abstract—Programmers who are blind use a screen reader to speak source code one word at a time, as though the code were text. This process of reading is in stark contrast to sighted programmers, who skim source code rapidly with their eyes. At present, it is not known whether the difference in these processes has effects on the program comprehension gained from reading code. These effects are important because they could reduce both the usefulness of accessibility tools and the generalizability of software engineering studies to persons with low vision. In this paper, we present an empirical study comparing the program comprehension of blind and sighted programmers. We found that both blind and sighted programmers prioritize reading method signatures over other areas of code. Both groups obtained an equal and high degree of comprehension, despite the different reading processes.

Index Terms—Program comprehension, accessibility technology, blindness

1 Introduction

Program comprehension is the task of interpreting software behavior [1], [2], [3], [4]. It usually involves reading source code in order to determine what that code does and how that code interacts with the software as a whole. Program comprehension is a critical component to software development, and is so common that it is essentially unavoidable [5], [6], [7]. Any impediment to program comprehension has the effect of reducing accessibility to software development.

Blindness is one of those impediments because it makes source code more difficult to read [8]. The state-of-the-practice for programmers who are blind is to navigate source code one line at a time using a screen reader.[1] The procedure is as follows: First, the programmer opens a source code file in an IDE and presses a key to read the line where the cursor is located, e.g., "float f equals five point two three semicolon." Next, the programmer uses the arrow keys to move the cursor one line up or down, and presses another key to begin reading that line.

This line-by-line strategy is feasible, but imposes a high cognitive load on the programmer—syntax details such as whitespace and block braces are often very difficult to follow, as Stefik et al. demonstrate in multiple studies [8], [9], [10]. At the same time, sighted programmers enjoy an advantage because of the ability to rapidly "skim" code [11]. Sighted programmers read a few keywords at a time from different areas of code, and use those keywords[2] to form a high-level understanding of that code, as has been shown in several studies including our own prior work [12], [13], [14]. The result is that programmers who are blind have utterly different mechanisms for reading source code as compared to their sighted counterparts.

What is not known is what effect that the differences in these mechanisms might have on program comprehension. It is often assumed that certain areas of code are particularly important for programmers. For example, function invocations have often been targeted by program comprehension tools—the idea being that if invocations are made more obvious, then program comprehension will become easier [15], [16], [17]. Invocations often represent discrete actions that together describe much of what a method does. But it is conceivable that invocations are emphasized because they are visually obvious in code. Likewise, areas that are harder to spot may be neglected. These effects are important to the generalizability of software development tools and, in particular, the design of accessibility tools for programmers who are blind. If blind programmers read different areas of code than sighted programmers, then tools designed for sighted programmers might not be sufficient for the blind community. Studies by Stefik et al. have hinted that this might be the case, as specialized programming languages were shown to be easier for blind programmers to learn [9].

In addition, a lack of studies of blind programmers in software engineering has contributed to a bias against the blind community in employment in software development [18]. In 2008, the Associated Press reported that blind programmers

1. Note that while blindness is defined by law in many countries, in practice many people with low vision but who are not blind require a screen reader.

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

Manuscript received 7 July 2016; revised 17 Feb. 2017; accepted 4 July 2017. Date of publication 19 July 2017; date of current version 21 Aug. 2018. (Corresponding author: Ameer Armaly.) Recommended for acceptance by R. DeLine. 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.2017.2729548

2. We use the same definition of keywords as the previous eye-tracking study [14]. Keywords are defined as any alphanumeric text including Java reserved words and developer-created identifiers. Keywords may be composed of sub-words separated by camelCase. Keywords do not include punctuation.

were a target of deliberate discrimination by employers, despite legal protections, due to confusion about how a blind person might integrate into team of sighted programmers [19]. Studies of how blind programmers comprehend code may help demystify the process of hiring the visually impaired.

In this paper,[3] we present a study comparing the process of program comprehension by blind programmers to the process followed by sighted programmers. We recruited blind programmers with various levels of professional experience to read source code and write brief summaries of that code. Using a modified screen reader, we recorded the words that the programmers asked the screen reader to speak, as well as the key strikes used for navigation around the code. Then, we compared the words that the blind programmers read to the words that sighted programmers looked at in an eye-tracking study we conducted in prior work [14]. We also conducted a quantitative comparison of the summaries produced by both groups to determine if one group produced better summaries than the other.

We found statistically-significant evidence that blind and sighted programmers both prioritized method signatures when reading code. Blind programmers read method invocations less than sighted programmers. Both groups focused less on control flow keywords than the rest of the method. There was no statistically-significant difference between the perceived quality of the summaries produced by either group.

These findings mean that the screen reader did not negatively affect the quality of program comprehension for purposes of code summarization. These findings have strong implications for the design of accessibility tools, as well as for advancing the integration of blind programmers in software development teams dominated by sighted persons.

2 The Problem

We target the following problem in software engineering literature: there are no studies that directly compare program comprehension by blind programmers to comprehension by sighted programmers. The lack of these studies causes problems for 1) designers of accessibility tools, 2) program comprehension researchers in general, and 3) blind programmers in the workplace. First, tool designers currently must make the assumption that blind programmers need the same information about source code as sighted programmers. This assumption may or may not be correct, given that blind programmers rely on a very different mechanism for reading code. It is possible that the mechanism affects the information that the programmers need. Second, researchers in program comprehension currently cannot generalize results of user studies and evaluations to blind programmers. At the same time, there is a common assumption that improved comprehension tool support will lead to improved comprehension - in other words, that the mechanism of reading code has an impact on the ability of the programmers to understand the software. A study of auditory versus visual intake of code could show the degree to which programmers are affected by the tools. Finally, it is our sincere hope that increased understanding of how blind persons understand code will help employers integrate the blind into the workplace.

3. A summary of this work was published as part of the ACM Student Research Competition at ICSE16 [20].

3 Background and Related Work

This section will cover background on existing studies of program comprehension both in general and for blind programmers, as well as our previous study of sighted programmers that we use as a basis for comparison.

3.1 Studies of Program Comprehension in Software Engineering

Biggerstaff et al. [21] define program comprehension as being able to "explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program." Subsequent program comprehension studies have examined the code comprehension process by examining what a participant reads as well as the external factors that could influence the program comprehension process. Ko et al. [2] tracked participants' activities during a programming task to determine how they familiarized themselves with a previously unfamiliar code base. The study found that developers spent approximately 35 percent of their time navigating in and between source files. LaToza et al. [1] conducted a series of interviews and surveys with professional programmers and found that programmers preferred to discuss code face to face rather than read it or read the accompanying API documentation. This means that programmers must regularly contend with interruptions that force them to reorient themselves when they return to reading code. Finally, Rodeghero et al. [14] conducted the eye tracking study to which we compared our results and found that programmers spend more time reading method signatures and method invocations over other code areas in the context of code summarization. In contrast, Kevic et al. [22] combined eye-tracking with interaction data and found that method signatures become much less important in the context of change tasks on a large software system. In that context, programmers focus primarily on method invocations and data flow.

There are two classic theories of program comprehension: the top-down theory [23] and the bottom-up theory [24]. The top-down theory posits that programmers keep a running hypothesis of what the code does throughout the comprehension process. They form a new hypothesis when they encounter lines that contradict the old hypothesis. In contrast the bottom-up theory posits that programmers divide the code into "chunks" which are small groups of lines that can be easily understood. They combine these chunks into larger chunks until they have a complete understanding of the code. A third theory [25] posits that complete comprehension is impractical and unnecessary in the case of large programs. Instead, programmers develop understandings of how specific concepts are implemented on an as-needed basis. This theory is particularly relevant when a software system is under active development and any understandings can be invalidated in a matter of weeks.

3.2 Studies of Blindness in Software Engineering

To our knowledge, there are no extant studies directly targeting program comprehension involving professional programmers. There are however studies that target the blind programming experience more generally that can shed light on the experience of code comprehension. Albusays and Ludi [26] conducted an exploratory study aimed at ascertaining the challenges generally faced by blind programmers. Among the obstacles they highlight are code navigation and incomplete accessibility of development environments. Mealin and Murphy-hillmealin2012exploratory found that blind programmers relied on documentation to get a high-level overview of code. In cases where documentation was incomplete or nonexistent blind programmers reverted to the editor's find command. They also point out that blind programmers often underutilize their programming environments because they are unaware of all its features.

A rich body of literature describes accessibility tools designed for blind programmers, and many of these tools have been tested in diverse environments. The results of these tests shed light on how blind programmers understand code. These tools can broadly be categorized as one of the following: 1) specialized programming languages or environments, 2) specialized audio or tactile representations of programming "space", or 3) audio or tactile means of improving computing accessibility generally. Among specialized programming environments, Schweikhardt describes an APL Braille interface designed specifically for the syntax of APL [27]. Also, Sánchez et al. introduced the Audio Programming Language, a language that represents control flow, variables, etc., through different sounds [28]. The programming logic is organized to optimize the understanding of the language by listeners who have difficulty scrolling to arbitrary locations. The Audio Programming Language has been shown to be an effective teaching tool, but is not intended as a production language. The lack of a visual representation makes it difficult to interface with sighted programmers. In contrast, other researchers have noted that while reading code, programmers form mental models of the code which can be treated as spatial information [1]. This spatial information can be represented and navigated non-visually [29], [30], [31], for example, via sounds which represent different window action connections in Visual Basic [31] or a tree view that represents the hierarchical structure of a source file [32].

General efforts to improve computer accessibility have been underway since the 1960's [33], with a vast majority of effort focused on developing screen reading audio and tactile interfaces [34]. Several software development firms and associations have made efforts towards accessibility in software products that include these interfaces [35], [36]. Emacspeak [37] is a speech interface for the Emacs editor and accompanying programs including web browsers and programming environments. Emacspeak pioneered reading of the screen in a manner optimized for speech rather than simply reading the contents of the screen verbatim. Speakup [38] is a general-purpose screen reader notable for being built into the Linux kernel and consequently speaking even when higher-level operating system services crash.

3.3 Comparison Study of Sighted Programmers

The studies we present in the remaining sections of this paper are compared to an eye-tracking study we conducted of sighted, professional programmers, and published at ICSE 2014 [14]. In the study, we hired 10 professional programmers to read source code and write English summaries of that source code. While the programmers read the code, we used an eye-tracking apparatus to detect which keywords and statements that the programmers read, and for how long. We measured three metrics: gaze time, fixations and regressions. These metrics are taken from the literature on cognitive psychology [39], [40], [41]. The goal was to determine what patterns of eye movement that the programmers used when reading the code. It was well-known that programmers skim code by reading only certain keywords and statements [11], and our intent was to discover which keywords and statement that the programmers needed. Our preliminary result was that programmers tend to read keywords from the function signature area and function invocations more than keywords from other areas, including control flow.[4]

Using this result, we modified an existing source code summarization technique (Vector Space Summarization [42]) to weight keywords from function signatures and invocations more than other keywords. The original version weighted all keywords equally; the final score was based on how often the keyword appeared in the document. The input of the technique is a section of source code (e.g., a function) and the output is a list of the top-*n* keywords from that code that summarize its key functionality. For example, "record mp3 files." We then evaluated our modified summarization tool to the default tool. We recruited nine graduate student programmers to manually select the top-5 keywords for 24 Java methods. What we found was statistically significant improvement by our tool as compared to the default: the keywords from our approach were 20 percent more likely, on average, to overlap with top-5 lists selected by the programmers.

There were two key statistically significant findings from the earlier study, that are relevant to this paper. First, we found solid evidence that the programmers consistently prefer to read certain areas of code over others. We found that control flow, which has been suggested as critical to comprehension [2], [43], was not read as heavily as other code areas during summarization. This finding seems to confirm a recent study [44] 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 [11].

Second, we showed that the keywords that programmers read are actually the keywords that an independent set of programmers (e.g., *not* the same participants from the eye-tracking study) 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

4. Control flow keywords are those keywords that occur inside a control flow statement, but not the statement keyword itself.

read those sections more often because they were harder to understand. In an evaluation of the modified summarization tool, we confirmed that the sections of code that programmers read actually contain the keywords that should be included in summaries [14].

4 Blind Programmer Study

The first part of our research is a study of blind programmers to collect and analyze data for comparison to a previous eye-tracking study. This section describes how we adapt the previous methodology as closely as possible for the blind programmers. We cover our research questions, the methodology of the blind programmer study, and details for the environment of the study.

4.1 Research Questions

The goal of this study is to determine whether blind programmers read code differently than sighted programmers by adapting the methodology of the eye tracking study. To answer this question we pose three research questions related to the earlier study:

- RQ_1 Do programmers focus on a method's signature more than the method's body?
- RQ_2 Do programmers focus on a method's control flow more than the method's other areas?
- RQ_3 Do programmers focus on a method's invocations more than the method's other areas?

The rationale behind RQ_1 is that the eye tracking study found that programmers read a method signature more than the method body while summarizing. It is not known whether blind programmers focus on the method signature to the same degree when reading code through speech so we investigate it as part of this evaluation. The rationale behind RQ_2 is that the eye tracking study found that programmers spend less time on control flow keywords than other keywords despite related work that attaches a greater importance to control flow. It is not known whether blind programmers' use of speech affects the time spent on control flow keywords so we investigate it as part of this evaluation. Finally, the rationale behind RQ_3 is that the eye-tracking study found no evidence that programmers spent more time on method invocations than other parts of the method despite related work that attaches a greater importance to method invocations. It is not known if blind programmer's use of speech to read code affects the time spent examining method invocations so we investigate it as part of this evaluation.

4.2 Methodology

We designed a research methodology to replicate the eye-tracking study as closely as possible. We emailed a study packet to each participant consisting of a study application and a cursor tracking script. The study application presented a series of methods to the participant inside of a Notepad session and prompted them to submit their summaries and any comments in a separate window. The application terminated after one hour and took the participant to an exit survey. Note that we excluded one participant from statistical analysis and all subsequent analysis because the summaries he submitted indicated he did not make any attempt to effectively comprehend the methods.

```
method1.txt 2:43:34 1 1 line
method1.txt 2:43:39 1 1 word
method1.txt 2:43:41 1 1 word
method1.txt 2:43:41 10 1 word
method1.txt 2:43:43 10 2 line
method1.txt 2:43:43 10 1 line
```

Fig. 1. Example of data produced by the JAWS script.

4.2.1 Data Collection

We tracked the cursor by creating a script extension to the JAWS screen reader.[5] We were able to do this because all of our participants used JAWS as their main screen reader. This script activated automatically so there was no danger of the participant forgetting to enable it.

The script recorded every movement of the cursor within the Notepad main window and whether the participant was trying to move by line, word or character. These were the only units by which a participant could move. The script also recorded when the participant switched focus to or from the Notepad window. Participants used the voice settings with which they felt most comfortable. Voice settings include rate and pitch and are a matter of personal preference. They do not affect the behavior of the screen reader or its interactions with running applications. Fig. 1 shows a sample of the cursor movements that were logged by the JAWS script.

In the case of Fig. 1 the participant first reads line 1 by using the "say line" command. The participant then reads the current word using the "say word" command twice. The participant then moves to the second word using the "next word" command. The participant then uses the "next line" command to read line 2 and finally uses the "prior line" command to return to and read line 1.

We used the data collected by the script to measure two commonly used metrics: fixations and regressions. In the previous eye-tracking study we defined fixations as locations in the code where the participant spent more than 100 milliseconds. In our case we count every read request to a location as a fixation, e.g., a read request to a line would count as a fixation on all elements of that line. In order to be more precise we would need to track exactly what portions of a line the screen reader spoke. This is not possible in the JAWS scripting environment. Regressions are locations in the code that the participant read and then returned to in order to clarify a point of confusion.

After one hour the participant was taken to an exit survey. this survey was the same exit survey that was used at the end of the eye-tracking study. The survey asked the following questions:

- Please write below the programming languages you know and how many years of experience you have with each.
- What is your preferred programming language to work with?
- When reading someone elses code, how do you go about reading it so that you fully comprehend it?

5. http://www.freedomscientific.com/Products/Blindness/Jaws

Please order (15) how you would try to comprehend it: 1 being the first step you would take, 5 being the last
- How do you compose your documentation? Do you use any tools? Please Explain.
- Do you take notes when you write code? Please explain.
- Do you take notes when you read someone else's code? Please explain.
- Please write down any additional information that you would like to provide.

4.2.2 Subject Applications

We used the same 67 methods as the eye-tracking study. These methods were taken from a variety of application domains including XML parsing, text editing and multimedia. All methods were nontrivial, e.g., no get/set methods. We also used the same ordering: the first five methods were always the same but subsequent methods were chosen randomly. This guaranteed a degree of overlap for comparison.

4.2.3 Statistical Tests

We used the same statistical test as the eye tracking study: the Wilcoxon signed-rank test. We compared both fixations and regressions. This test is nonparametric and is suitable for our data because it does not assume a normal distribution.

4.3 Participants

We recruited twelve blind programmers to participate in the study. Seven were professional programmers, two were university lecturers and three were university students. They had an average of three years experience with Java and five years of general programming experience. All participants were legally blind and used a computer through a screen reader for all tasks including programming.

We recruited our participants through a combination of word of mouth and advertising on accessibility-related forums and email lists. The evaluation process took approximately eight months. Finding qualified participants was made more difficult by the fact that our cursor tracking scripts were confined to the JAWS screen reader. We had to follow up with some participants because they originally said that they did not have time to do the study. The first five participants received $100 for their time. Later we were able to increase the compensation to $200.

4.4 Threats to Validity

While we tried to faithfully adapt the eye-tracking study, our methodology does contain threats to validity. Since we used the same methods as the eye-tracking study, each method was no more than 22 lines long, due to a limitation of the screen size of the eye tracker [14]. We cannot claim that our results are generalizable to methods of arbitrary size. We have no way of measuring how attentive a participant is to a given read request. A participant could easily read a line, listen to a portion of it and move to the next line before the screen reader finishes reading the current line. This would likely happen when the participant is scrolling to a specific line. The screen reader does not offer a means to measure how much of a given read request was spoken before the next read request. For the purpose of statistical analysis we assume that the participant read the entire line and all of its components, but this is by no means true in all cases. We did not distinguish between regressions and revisits. Revisits are when a participant returns to a line of code because their interest has actually shifted. We attempted to mitigate this threat by limiting our study time to one hour and limiting the size of our methods to 22 lines or less in order to minimize the potential for a participant's interest to shift. It is possible that a more accurate tracking mechanism would yield different results.

The original eye-tracking study was conducted at the lab in person in a more formal setting whereas our participants ran the study in the environment of their choice such as their home or office. Participants used the JAWS settings they normally used when reading code such as voice rate or pitch. It is possible that placing the blind participants in the same lab setting with a consistent JAWS configuration would yield a different result. Finally, two of our participants expressed reservations about being able to effectively comprehend the code. One had only used Java in a classroom setting and had very little experience reading code written by others. Another had no recent programming experience since becoming a lecturer and another had very little familiarity with the Java standard library classes. It is possible that a more experienced group of participants would yield different results.

4.5 Reproducibility

To ensure the reproducibility of our work, we have made all raw data, analysis scripts, and statistical summary results available via an online appendix.[6]

5 Results of Blind Programmer Study

In this section we present our answer to research questions RQ_1-RQ_3 as well as our data and rationale of the answers.

5.1 RQ_1: Method Signatures

We found statistically-significant evidence that during summarization, blind programmers read a method's signature more heavily than the method's body. This is consistent with the findings of the eye-tracking study for sighted programmers. The programmers read the signatures in a greater proportion than the signatures' sizes. On average, the programmers spent 29 percent of their gaze time reading signatures even though signatures only averaged 12 percent of the methods' keywords.

We used a similar procedure as the eye-tracking study to derive our conclusion: We compared the fixation and regression count for the keywords in the signature to the keywords in the method body. We computed the fixation percentage by calculating the number of times the participant read the signature keywords for a given method. We then divided the percent of fixations that occurred on signatures by the percentage of keywords that are contained in method signatures to obtain the adjusted fixation count. For

6. http://www3.nd.edu/~aarmaly/papers/blindstudy

TABLE 1
Statistical Summary of the Results for RQ_1-RQ_3

RQ	H	Metric	Method Area	Samples	$\tilde{x}$	μ	Vari.	U	U_{expt}	U_{vari}	p
RQ_1	H_1	Fixations	Signature	155	2.488	2.381	1.691	11,838.000	6,045.000	313,330.000	< 0.0001
			Non-Sig.	155	0.797	0.812	0.031				
	H_2	Regress.	Signature	155	2.956	2.549	3.330	11,821.000	6,045.000	313,331.125	< 0.0001
			Non-Sig.	155	0.953	0.82665	0.0361				
RQ_2	H_3	Fixations	Ctrl. Flow	155	0.212	0.153	0.035	1.000	6,045.000	313,324.375	< 0.0001
			Non-Ctrl.	155	1.463	1.498	0.012				
	H_4	Regress.	Ctrl. Flow	155	0.286	0.281	0.062	12.000	6,045.000	313,007.125	< 0.0001
			Non-Ctrl.	155	1.461	1.505	0.019				
RQ_3	H_5	Fixations	Invocations	155	0.282	0.277	0.044	3.000	6,045.000	313,189.750	< 0.0001
			Non-Invc.	155	1.419	1.422	0.021				
	H_6	Regress.	Invocations	155	0.286	0.281	0.062	10.000	6,045.000	312,824.50	< 0.0001
			Non-Invc.	155	1.419	1.422	0.021				

Mann-Whitney 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.

regressions, we computed the percentage of each that occurred in the method signature and the body and divided them in the same way. We then posed two hypotheses (H_1 and H_2) as follows:

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

We tested these hypotheses using a Wilcoxon test. The results of this test are in Table 1. We rejected hypotheses H_1 and H_2. This means that programmers did look more often at method signatures than the method body and returned to the method signature more often than they returned to the method body.

5.2 RQ_2: Control Flow

We found statistically significant evidence that blind programmers read control flow keywords less than other keywords in a method. On average, programmers spent 7 percent of their time reading control flow keywords despite these keywords averaging 37 percent of method keywords. We defined control flow keywords as any keyword such as an identifier that occurred inside a control flow statement, but not the statement keyword itself. We computed the fixation count and regression count for control flow keywords the same way as for function signatures. Then we posed hypotheses H_3 and H_4:

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

Using the Wilcoxon test we rejected the null hypotheses, meaning that blind programmers red control flow keywords less than other method keywords.

5.3 RQ_3: Method Invocations

We found statistically significant evidence that blind programmers read method invocations less than other keywords in a method. We defined method invocation keywords as the name of the method and the names of any variables passed as parameters. The blind programmers spent 10 percent of their time reading invocation keywords despite those keywords averaging 37

H_n The difference between the adjusted [fixation/regression] metric for method invocations and all other keywords is not statistically significant.

Using the Wilcoxon test we rejected the null hypotheses, meaning that blind programmers read method invocations less than other method keywords.

5.4 Summary of Results

The main observation from these results is that we did find a significant increase in the proportion of reads and regressions for the method signatures (RQ_1). We also found a decrease in the proportions of reads and regressions for control flow and invocations (RQ_2, RQ_3). In other words, the participants did not read control flow keywords or method invocations as often as the rest of the code, nor did they feel the need to return to those areas any more than other areas for any sort of clarification of what the method did. In contrast, the participants read the method signatures more than the method bodies. They also returned to the method signatures more than they returned to other code areas.

This finding suggests that the screen reader imposes a top-down approach to reading code that differs from a sighted programmer's approach which allows for quickly jumping to the most important code areas. In other words, the programmers develop a hypothesis based on the method signature and alter that hypothesis as needed while reading through the code. The programmers return to certain lines of code when they need to remind themselves of the line's contents or when they need to resolve some perceived contradiction in their understanding of the code. If blind programmers are given the opportunity to skim the code's structure it could affect their approach to code comprehension. Still, it should be noted that the study participants returned more often to the method signature suggesting that they were more interested in resolving contradictions with the method signature than with other parts of the code. In the next section we will compare the blind and sighted groups further to find out if this difference impacts either groups ability to produce coherent code summaries.

6 Comparison of Blind and Sighted Programmers

In this section, we will compare the studies of the blind and sighted programmers. The blind study is a new contribution

of this paper (see Section 4), though the study of sighted programmers is from prior work [14].

6.1 Research Questions

Our objective is to determine whether the blind and sighted programmers prioritize different areas of code when reading that code. Therefore, we pose the following research questions:

- RQ_4 Did the blind programmers read the areas of source code at the same proportions as the sighted programmers?
- RQ_5 Can the Blind Group's emphasis of method signatures be attributed to the order in which the areas of source code are presented?

The rationale behind RQ_4 is that the blind and sighted programmers have vastly different mechanisms for reading the code, and these mechanisms may affect the degree to which the programmers prioritize various areas of code (we measured signatures, regions of control flow, and method invocations in all studies). In RQ_5, we aggregate the results of the blind group to determine if their emphasis of method signatures could be caused by the method signatures always being presented before other code areas.

While the analysis of the statistical results answers questions related to the physical process of reading code, it is also our objective to determine whether the process actually affects program comprehension. Therefore, we perform a quantitative study of the textual differences among the summaries that the programmers wrote. The rationale is that if the summaries are significantly different, then the programmers may be obtaining a significantly different understanding of the code. They may have varying ideas about what are the most important details in the code. This quantitative study is distinct from the qualitative study later in the paper, and covers the following five research questions:

- RQ_6 Is there a statistically-significant difference in the accuracy of summaries written by the blind programmers versus the sighted programmers?
- RQ_7 Is there a statistically-significant difference in the completeness of summaries written by the blind programmers versus the sighted programmers?
- RQ_8 Is there a statistically-significant difference in the conciseness of summaries written by the blind programmers versus the sighted programmers?
- RQ_9 Is there a statistically-significant difference in the confidence of summaries written by the blind programmers versus the sighted programmers?
- RQ_{10} Is there a statistically-significant difference in the overall quality of summaries written by the blind programmers versus the sighted programmers?

6.2 Methodology

The methodology we follow to answer RQ_4 is to compare each of the statistical hypothesis tests from each study for the various areas of code. For example, we compare the results for RQ_1 from this paper to RQ_2 in the sighted study because both questions relate to the method signatures in code—RQ_1 for the blind and RQ_2 for the sighted programmers.

Note that in the sighted study, "fixations" meant eye movements to a keyword, while "regressions" meant eye movements away from and then back to a keyword. But in the blind study, a fixation was a cursor location and read request to the screen reader, and a regression was a cursor movement away from and then back to a keyword, with an associated read request to that keyword (see Section 4.2.1). For the purposes of this comparison study, we assume that eye fixations are equivalent to cursor fixations and eye regressions are equivalent to cursor regressions.

To answer RQ_5, we calculated the normalized word read count for every line read by the blind group. The normalized word read count is taken by dividing the number of word read requests made on a line by the number of words in that line as perceived by the screen reader. The screen reader considers both the space character and the left paren character as delimiters of words. Since blind users typically read by line and read by word only when needing to clarify something about the line this allows us to determine whether the blind group emphasized method signatures because they were especially relevant or because they came first. We compared the normalized word read counts of the lines containing method signatures to the normalized word read counts for all other lines. We evaluated the significance of our results using the Mann-Whitney test because of the difference in size between the two groups.

To answer RQ_6 through RQ_{10}, we conducted a survey of fifteen graduate students. The survey presented the participants with a random method from the study followed by a randomly chosen summary of that method. The participant did not know whether the summary was produced by a blind or sighted programmer. The participant then answered five questions about that summary:

- Q_1 Independent of other factors, I feel that the summary is accurate.
- Q_2 The summary is missing important information, and that can hinder the understanding of the method.
- Q_3 The summary contains a lot of unnecessary information.
- Q_4 Based on the summary text, the author of the summary expresses doubts about the quality of the summary.
- Q_5 Please rate your impression of the overall quality of the summary.

The process repeated until the participant stopped themselves after twenty minutes. Possible answers for Q_1 through Q_4 were "strongly disagree", "disagree", "neutral", "agree" and "strongly agree." Possible answers for Q_5 were "very good", "good", "fair", "poor", "very poor" and "completely invalid." We excluded summaries marked as "completely invalid" from statistical analysis. We also excluded summaries of those methods that were not seen by at least one blind and one sighted programmer. Since Q_2 through Q_4 were negative questions we reversed the internal coding such that "strongly agree" was coded as one and "strongly disagree" was coded as five. This simplifies the interpretation of results since higher scores are always better.

6.3 Threats to Validity

Aside from threats to validity inherited due to the data collection in previous studies, there are four key threats to

TABLE 2
Statistical Summary of the Results for the Comparison Study

RQ	H	Metric	Pairs	Samples	$\tilde{x}$	μ	Vari.	U	U_{expt}	U_{vari}	p
RQ_5	H_7	count	sig. body	67 285	2.862 2.231	1.778 1.333	11.287 7.699	11,202.500	9,547.500	560,890.201	0.014
RQ_6	H_8	accuracy	blind sighted	34 34	3.735 3.941	4.000 4.000	1.898 1.451	534.500	578.000	5,982.985	0.574
RQ_7	H_9	completeness	blind sighted	34 34	2.941 3.382	2.000 4.000	2.178 2.486	486.000	578.000	6,224.537	0.244
RQ_8	H_{10}	conciseness	blind sighted	34 34	3.500 3.294	3.500 3.000	1.894 1.850	632.000	578.000	6,028.657	0.487
RQ_9	H_{11}	confidence	blind sighted	34 34	3.559 3.588	4.000 4.000	2.012 1.340	591.000	578.000	6,252.067	0.869
RQ_{10}	H_{12}	overall quality	blind sighted	34 34	3.176 3.294	3.000 3.000	1.301 1.305	558.500	578.000	6,218.955	0.805

Mann-Whitney test values are U, U_{expt}, and U_{vari}. Decision criteria are Z, Z_{crit}, and p. A "Sample" is one pair of summaries for one method. For pairs, Blind is represented by B and Sighted is represented by S.

validity to this part of our study. First, in the sighted study, we collected data for eye gaze time, fixations, and regressions. But because gazes are impossible for the blind group, we could only collect fixations and regressions based on the screen reader's cursor location. A second threat is that our study is limited to the three areas of code (signatures, control flow, and invocations) for which we have data from the blind and sighted groups. This threat is mitigated somewhat because these are three of the most important areas according to a large body of related work (see Section 3.2). However, it is possible that our conclusions might differ if we had data for more areas of code. A third threat to validity is our reliance of the screen reader's notion of what characters indicate the start of a new word. This has been observed to vary between applications and other circumstances, e.g., in some cases a comma is a distinct word where as in others it is not. It is possible that our conclusions might differ if we used a different set of word delimiters. The final threat to validity is the use of human experts to rate the summaries. Different experts might produce different results. The experts did not rate every summary produced by the two studies; a more complete analysis may yield different results.

7 Results of Comparison Study

In this section we answer research questions RQ_4 to RQ_{10} using data from the comparison study in the previous section. First we answer each question with statistical tests, then we provide our high-level interpretation of these results. Table 2 contains the details of our statistical tests.

7.1 RQ_7: Blind to Sighted Comparison

Put briefly, we found that the blind and sighted programmers both read the signatures more closely than other areas of code. But, the blind programmers tended to read all other parts of code equally, including control flow. That is different than the sighted programmers, who tended to ignore the control flow areas. Our rationale for this finding follows:

In the blind study from this paper, we found statistically significant evidence that the blind programmers focused the screen reader's cursor on the signature keywords more than keywords from other sections. We also found that they regressed to these keywords more than they regressed to other areas of the code. These results differ from the sighted programmers from the earlier paper, in which the fixations were significantly higher, but regressions were not. Another difference of blind and sighted programmers is that the blind programmers read method invocations less than the rest of the code. Sighted programmers read method invocations equally with the rest of the code. Both groups read control flow keywords less than the rest of the code.

7.2 RQ_5: Blind Code Areas by Word

We found statistically significant evidence that the normalized word read count was higher for method signatures than other parts of the code. We calculated the normalized word read count for all lines that the blind group read and divided them in two sets: one containing normalized word read counts for method signatures and the other containing normalized word read counts for all other lines. We evaluated the difference between the normalized word read count for both sets of lines using the Mann-Whitney test and posed hypothesis H_7:

H_{13} The difference between the normalized word read counts for method signatures and all other lines of code is not statistically significant.

As shown in Table 2 we rejected this hypothesis, meaning that the blind participants read method signatures by word more than other source code areas.

7.3 RQ_6: Summary Accuracy

We found statistically-significant evidence that the blind and sighted groups produced equally accurate summaries based on the ratings given by the human experts in the survey. This conclusion is based on hypothesis H_8:

H_8 The difference between the accuracy ratings of the summaries produced by the blind group and the sighted group is not statistically significant.

As shown in Table 2, we accepted the null hypothesis.

7.4 RQ_7: Summary Completeness

We found statistically-significant evidence that the blind and sighted groups produced equally accurate summaries based on the ratings given by the human experts in the survey. This conclusion is based on hypothesis H_9:

H_9 The difference between the completeness ratings of the summaries produced by the blind group and the sighted group is not statistically significant.

As shown in Table 2, we accepted the null hypothesis.

7.5 RQ_8: Summary Conciseness

We found statistically-significant evidence that the blind and sighted groups produced equally concise summaries based on the ratings given by the human experts in the survey. This conclusion is based on hypothesis H_{10}:

H_{10} The difference between the conciseness ratings of the summaries produced by the blind group and the sighted group is not statistically significant.

As shown in Table 2, we accepted the null hypothesis.

7.6 RQ_9: Summary Confidence

We found statistically-significant evidence that the blind and sighted groups produced equally confident summaries based on the ratings given by the human experts in the survey. This conclusion is based on hypothesis H_{11}:

H_{11} The difference between the confidence ratings of the summaries produced by the blind group and the sighted group is not statistically significant.

As shown in Table 2, we accepted the null hypothesis.

7.7 RQ_{10}: Summary Overall Quality

We found statistically-significant evidence that the blind and sighted groups produced summaries with equal overall quality based on the ratings given by the human experts in the survey. This conclusion is based on hypothesis H_{12}:

H_{12} The difference between the overall quality ratings of the summaries produced by the blind group and the sighted group is not statistically significant.

As shown in Table 2, we accepted the null hypothesis.

7.8 Interpretation

Our interpretation of the findings in this section is that 1) the use of a screen reader does affect the process through which programmers comprehend code, and 2) this affect does not hinder blind programmers from being as effective as sighted programmers in understanding code and conveying that understanding through summaries. The blind programmers (who all had extensive experience with the screen reader) read the method signatures more than other areas, as the sighted programmers did (RQ_4). The summaries produced by both groups varied in quality from programmer to programmer, but we did not observe a difference on any of the five measures of quality (RQ_6 through RQ_{10}). Still, the use of the screen reader presents unique challenges for code comprehension. We will explore these challenges in the next section.

8 Qualitative Analysis

In this section we present a qualitative analysis of our data in order to provide possible explanations for the statistical results.

8.1 Research Questions

The goal of this section is to find qualitative differences between the blind and sighted groups that can compliment the statistical results presented in the comparison study. To that end we pose the following two research questions:

RQ_{12} Is there qualitative evidence of a difference in the code comprehension of blind and sighted programmers?

RQ_{13} Is there qualitative evidence that blind programmers have more difficulties with Java syntax as compared to sighted programmers?

The rationale behind RQ_{12} is that the blind group is reading code through a different medium than the sighted group. The comparison study has shown that both groups produce summaries of equal quality but the fact that the blind group spent less time on method invocations suggests the possibility of a slight difference in their understandings of the code. The nature of that difference and any possible causes are unknown; it is possible this difference is completely incidental. We are not equipped to answer the question definitively but we explore it as part of this evaluation. The rationale behind RQ_{13} is that participants are being given code in Java. Java syntax does not resemble English syntax to the same degree that Python or Visual Basic syntax does. Syntax has already been shown to increase the learning curve for computer science students in general [45]. It is not known if the use of a screen reader makes Java syntax harder to comprehend so we investigate it as part of this evaluation.

8.2 Methodology

We examined the summaries submitted by the participants for length and level of detail. We also examined the comments for each method that each participant had the option to submit. Finally, we examined the responses each participant submitted in the exit survey. We used an affinity diagramming approach [46] to answer our research questions. Each research question was a top-level concern. We then grouped the summaries and comments into related concerns beneath each top-level concern. We then summarized the concerns presented for each related concern in order to coherently answer the research question. It is important to note that the participants were not informed of our research questions in order not to bias their summaries.

8.3 Results

8.3.1 RQ_{12}: Blind and Sighted Participants

We found that summaries varied within each group more than they varied between groups. We identified four major variations in the summaries: length and detail, reliance on context, domain specificity and the participant's interpretation of the term "summary." We observe all of these variations in both the blind and sighted groups.

We found that summaries varied in length and level of detail within each group. We found no difference between

the blind and sighted groups in this regard. Consider the following summaries of method seven produced by the sighted group:

- "Checks for access to media file, attempts to mount file's containing device if not already mounted. Warning message if no file specified."
- "Check if a file selected to play is ready."
- "This function first checks if there IS a file to play. If so, it checks if it is ready. If the file is not ready, it attempts to mount the device. If the mounting fails, it sends a "10" error code and notifies the "ObservationManager" (10 being "failed to mount" error). The result is then false. If there is no file to play, we send an error message (23) which would mean there is no file, and set the result to false. We then return result, which would mean that if true, the file is both there and ready to play (mounted). If false, error code 10 means it failed to mount, and error code 23 means no file exists."

Now consider the summaries for the same method that were submitted by the blind group:

- "Prepare to play file. Attempt is made to mount the device on error."
- "This method insures that a file is ready for playing. It tries to mount the device if the device is not already mounted."
- "In this method, we're checking a file before we play it. If any errors are caught, we are adding them to a debug log, as well as in some cases printing a warning message. Otherwise, the return Boolean is true, meaning that the file is ready to be played."

Both blind and sighted programmers found that certain methods depended heavily on other methods which they were not able to see. Consider these sighted group summaries from method two:

- "This looks like it is taking an integer and row, and building a dialog with it. It checks to make sure that there are some genres to add to this 'GenreSelectionDialog', if not it returns. It then uses a 'string builder' class add all of the genre names. It then deletes whatever is at the final index and seems to build out a UI with the information."
- "Overall, this method looks to take user input from a dialog to display a list of genre names on a button (up to 10). The input parameter looks like some kind of index to grab the correct 'wizard' which contains the genres to select from as well as the correct button . Below is a more detailed walkthrough. Sets up a UI dialog by taking the row parameter and building a list of selectable genres based on some pre-defined wizard. It then takes the selected genres from the dialog and exits the method if none were selected. Next, it limits the number of genres to 10 spots, loops through the selected list, and adds them to the "ambience" object. In that loop, it also creates a comma separated list of the genres using the "getName2" field. After the loop, it removes the final comma. It then sets the text of a JButton to that of the comma separated list. At the end, if the "ambiance" object has a non-empty name and contains more than 0 genres, we null out some problem field (maybe) and enable the JButton."
- "Looks like a piece of (crappy) UI code. It creates new dialog box and (apparently) tries to select items in dialog box from a collection of Genres from Ambience object. If selection doesn't work, quit Otherwise, create comma-separated string from names of selected Genres. Delete last comma. Last part is a bit confusing, because it's not clear what widgets and jbNew are. My guess is that widgets[row] [2] is a button, and this code sets button name to a comma-separated list of names of selected Genres generated earlier. Don't understand what the last if is doing."

Now consider these summaries from the blind group:

- "I'm not sure. I forgot what getColumnNamesWithPrefix does, so I want to look back at it but won't."
- "This method seems to check whether some data starts with a sequence, comment or string. I can't say much more than that."
- "Perhaps this is cleaning up the conf dictionary of configuration values prior to serializing them. All ambience conf keys seem to be stripped, and new values are added, probably to save a list of the active genres."

We observed that some participants in both groups were unfamiliar with the problem domain of a particular method. This could cloud the issue if the method author assumes a certain level of familiarity with the problem domain. We found evidence of this effect in summaries of several methods from the sighted group:

- "This returns a polygon whose shape is defined by by an affinetransform (not sure what that is) and a flatting path iterator (also unsure). It loops through this flattening path iterator which returns a set of coordinates (seems like 6 dimensional space) and adds that point to the polygon. The polygon is then returned."
- "opens up a stream to a reader object that can handle xml parsing. Not very familiar with Stacked/Pushback readers"

Now consider these summaries from the blind group:

- "This method appears to flatten some shape into a polygon."
- "This method does some calculations on a given set of values and returns the hash code for these"

Finally, We observed a difference in the participants' conception of the level of detail required for a summary. Some participants wrote their summaries in the style of Javadoc or other code documentation tools where the summary for a method is supported by a separate explanation of each parameter, the return value and any exceptions that might be raised. Natural-sounding grammar is optional. In contrast other participants produced complete summaries that documented only the most important parts of the method. These summaries tended to have natural-sounding grammar. Both groups exhibited this variation.

8.3.2 RQ_{13}: Language Syntax

We found no qualitative evidence that blind programmers had difficulty comprehending Java syntax as compared to sighted programmers. As shown in Section 7.8 the blind participants produced summaries that of equal quality to the sighted group. This similarity suggests that the blind participants were able to adapt to the complexities of Java syntax to a degree comparable to their sighted counterparts. One way the blind participants adapted to the syntax was to read the most complex lines by word. In examining the data for RQ_{10} (see Section 6) we found that many of the lines with the highest normalized word read count were complex. Consider the following examples:

- `keys.addAll(Arrays.asList (getColumnNamesWithPrefix(key, field.getName()+"_")));`
- `if ((prop = isKeyword( _currentReadPos, len)) != null)`

Each of these lines had a high normalized word read count, meaning that each line was read by word to a high degree relative to the number of words it contained. Reading by word is uncommon in the context of source code because words generally contain very little useful information compared to lines. Each of these lines does more than one task which serves to make it more complex. Each of these lines could be considered bad style during a code review precisely because they take more effort to comprehend. This suggests that blind programmers value the same stylistic choices as sighted programmers such as short methods, meaningful variable names, short lines of code, etc.

9 Conclusion

Our conclusion is that blind and sighted programmers comprehend code in similar ways and that both groups are equally capable of producing coherent summaries of that code. We have presented a cursor-tracking study of blind programmers during source code summarization aimed at determining whether blind and sighted programmers focus on the same parts of a method when attempting to comprehend it. We compared our results to a previously conducted eye-tracking study of sighted programmers through both quantitative and qualitative analyses. We found that the blind and sighted groups both spent more time reading method signatures than other parts of the code. This underscores the previously-established importance of coherent method signatures to code comprehension. We also found that both blind and sighted programmers spent less time on control flow than other code areas. The only difference that we found between the two groups is that blind programmers spent less time reading method invocations than did the sighted programmers.

Our results suggest that blind programmers read and comprehend code much like their sighted counterparts. Tools and techniques to speed up code comprehension on the part of sighted programmers are likely to be equally applicable to blind programmers provided any user interfaces are accessible to screen readers. Blind programmers can produce summaries that are as useful as those produced by sighted programmers, ignoring factors such as familiarity with the problem domain or third-party libraries. We hope that this study along with other relevant research will demonstrate to potential employers that blind programmers can be easily integrated into the workplace in much the same way as sighted programmers.

The blind group examined certain lines more closely than others because of the complexity of those lines as well as the complexity of the resulting syntax. This suggests that certain programming languages and coding styles are less suited for introducing blind students to programming because the syntax creates an otherwise-avoidable distraction. C-family languages such as Java make extensive use of parentheses and semicolons which are always spoken by the screen reader and sound unnatural to the blind novice. Variable naming conventions such as Hungarian notation or separating words with underscores can complicate matters further. Languages such as Python whose syntax is more natural-sounding are likely to present lower barriers to entry for the blind. The Camel Case naming convention also presents a lower barrier to entry because most screen readers intelligently separate the words in a Camel Case variable name. Note that these observations are specifically aimed at lowering the barrier to entry for the blind into computer science. Blind programmers are as capable as sighted programmers of adapting to a variety of syntax variations and naming conventions once they are familiar with the concepts involved in programming. Reducing the barriers of entry for the blind will yield more particiapnts for accessibility studies. Some may even choose to research topics in accessibility that are informed by their experience using available accessibility tools.

One area of future research is to further explore the impacts of syntax. If the ease with which blind novice programmers acclimate to different language syntax families can be quantified it would provide further guidance on the best way to increase the participation of the blind in computer science. Moreover our finding that blind programmers tend to break down complex lines of code into words for greater clarity suggests that read counts and eye fixations correlate to the cyclomatic complexity of a given method. Another area of future research is to explore the effectiveness of alternative representations of code such as call graphs. If call graphs could be made accessible to the blind and their effectiveness as a code comprehension aid could be quantified it would further illuminate the similarities or differences in the way blind and sighted programmers mentally model a piece of code. Finally, we did not address the impact of other reading methods such as a braille display on code comprehension. A braille display shows a line of braille using metal pins ranging in width from twelve to eighty characters. Users can use a braille display in conjunction with speech output or as an alternative when it is necessary to mute the screen reader. Braille displays are expensive and uncommon except in Europe and other places where the government will subsidize the cost.

Acknowledgments

The authors would like to thank the several participants in our user studies for their careful attention to detail. In addition, the authors strongly thank the anonymous referees for

greatly improving the manuscript. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1313583. This work is supported in part by the US National Science Foundation CCF-1452959 and CNS-1510329 grants. Any opinions, findings, and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

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.*, 2006, pp. 492–501. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134355

[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. [Online]. Available: http://dx.doi.org/10.1109/TSE.2006.116

[3] A. von Mayrhauser and A. Vans, "Program comprehension during software maintenance and evolution," *IEEE Comput.*, vol. 28, no. 8, pp. 44–55, Aug. 1995.

[4] M. Harman and S. Danicic, "Amorphous program slicing," in *Proc. 5th Int. Workshop Program Comprehension*, 1997, pp. 70–79.

[5] T. A. Corbi, "Program understanding: Challenge for the 1990's," *IBM Syst. J.*, vol. 28, no. 2, pp. 294–306, Jun. 1989. [Online]. Available: http://dx.doi.org/10.1147/sj.282.0294

[6] 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, no. 2, pp. 44–54, Apr.–Jun. 2000.

[7] 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, Art. no. 21. [Online]. Available: http://dl.acm.org/citation.cfm?id=782010.782031

[8] A. M. Stefik, C. Hundhausen, and D. Smith, "On the design of an educational infrastructure for the blind and visually impaired in computer science," in *Proc. 42nd ACM Tech. Symp. Comput. Sci. Edu.*, 2011, pp. 571–576. [Online]. Available: http://doi.acm.org/10.1145/1953163.1953323

[9] A. Stefik, A. Haywood, S. Mansoor, B. Dunda, and D. Garcia, "Sodbeans," in *Proc. IEEE 17th Int. Conf. Program Comprehension*, May 2009, pp. 293–294.

[10] A. Stefik, S. Siebert, K. Slattery, and M. Stefik, "Toward intuitive programming languages," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, Jun. 2011, pp. 213–214.

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

[12] R. Bednarik and M. Tukiainen, "An eye-tracking methodology for characterizing program comprehension processes," in *Proc. Symp. Eye Tracking Res. Appl.*, 2006, pp. 125–132. [Online]. Available: http://doi.acm.org/10.1145/1117309.1117356

[13] 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.*, 2012, pp. 381–384. [Online]. Available: http://doi.acm.org/10.1145/2168556.2168642

[14] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 390–401.

[15] C. McMillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodology*, vol. 22, no. 4, pp. 37:1–37:30, Oct. 2013. [Online]. Available: http://doi.acm.org/10.1145/2522920.2522930

[16] J. Stylos and B. A. Myers, "Mica: A web-search tool for finding API components and examples," in *Proc. Visual Languages Human-Centric Comput.*, 2006, pp. 195–202. [Online]. Available: http://dx.doi.org/10.1109/VLHCC.2006.32

[17] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 157–166. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882316

[18] S. Alexander, "Blind programmers face an uncertain future," in *ComputerWorld*, vol. 32, pp. 86–87, Nov. 1998.

[19] D. Crary, "Employer bias thwarts many blind workers," Associated Press, New York, NY, USA, May 2008.

[20] A. Armaly and C. McMillan, "An empirical study of blindness and program comprehension," in *Proc. 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 683–685.

[21] T. J. Biggerstaff, B. G. Mitbander, and D. Webster, "The concept assignment problem in program understanding," in *Proc. 15th Int. Conf. Softw. Eng.*, 1993, pp. 482–498. [Online]. Available: http://dl.acm.org/citation.cfm?id=257572.257679

[22] K. Kevic, B. Walters, T. Shaffer, B. Sharif, D. Shepherd, and T. Fritz, "Eye gaze and interaction contexts for change tasks–observations and potential," *J. Syst. Softw.*, vol. 128, pp. 252–266, 2017.

[23] R. Brooks, "Towards a theory of the cognitive processes in computer programming," *Int. J. Man-Mach. Studies*, vol. 9, no. 6, pp. 737–751, 1977.

[24] S. Letovsky, "Cognitive processes in program comprehension," *J. Syst. Softw.*, vol. 7, no. 4, pp. 325–339, 1987.

[25] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proc. 10th Int. Workshop Program Comprehension*, 2002, pp. 271–278.

[26] K. Albusays and S. Ludi, "Eliciting programming challenges faced by developers with visual impairments: Exploratory study," in *Proc. 9th Int. Workshop Cooperative Human Aspects Softw. Eng.*, 2016, pp. 82–85.

[27] W. Schweikhardt, "A programming environment for blind APL-programmers," in *Proc. Int. Conf. APL*, 1982, pp. 325–331. [Online]. Available: http://doi.acm.org/10.1145/800071.802260

[28] J. Sánchez and F. Aguayo, "Blind learners programming through audio," in *Proc. CHI Extended Abstracts Human Factors Comput. Syst.*, 2005, pp. 1769–1772. [Online]. Available: http://doi.acm.org/10.1145/1056808.1057018

[29] P. Blenkhorn and D. Evans, "Using speech and touch to enable blind people to access schematic diagrams," *J. Netw. Comput. Appl.*, vol. 21, no. 1, pp. 17–29, 1998. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1084804598900601

[30] V. Khambadkar and E. Folmer, "GIST: A gestural interface for remote nonvisual spatial perception," in *Proc. 26th Annu. ACM Symp. User Interface Softw. Technol.*, 2013, pp. 301–310. [Online]. Available: http://doi.acm.org/10.1145/2501988.2502047

[31] K. G. Franqueiro and R. M. Siegfried, "Designing a scripting language to help the blind program visually," in *Proc. 8th Int. ACM SIGACCESS Conf. Comput. Accessibility*, 2006, pp. 241–242. [Online]. Available: http://doi.acm.org/10.1145/1168987.1169035

[32] C. M. Baker, L. R. Milne, and R. E. Ladner, "StructJumper: A tool to help blind programmers navigate and understand the structure of code," in *Proc. 33rd Annu. ACM Conf. Human Factors Comput. Syst.*, 2015, pp. 3043–3052.

[33] B. Hodson, "Sixties ushers in program to train blind programmers," *ComputerWorld*, Jun. 2004.

[34] E. P. Glinert and B. W. York, "Computers and people with disabilities," *Commun. ACM*, vol. 35, no. 5, pp. 32–35, May 1992. [Online]. Available: http://doi.acm.org/10.1145/129875.129876

[35] Microsoft, "Windows XP accessibility," Jan. 2014. [Online]. Available: http://support.microsoft.com/kb/308978

[36] K. Dahlke, 2014. [Online]. Available: http://leb.net/blinux/

[37] T. Raman, "Emacspeak—A speech interface," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 1996, pp. 66–71.

[38] A. Armaly, "Making Linux accessible for the visually impaired with speakup," *Linux J.*, vol. 2005, no. 140, 2005, Art. no. 6.

[39] M. E. Crosby and J. Stelovsky, "How do we read algorithms? a case study," *IEEE Comput.*, vol. 23, no. 1, pp. 25–35, Jan. 1990.

[40] K. Rayner, A. Pollatsek, and E. D. Reichle, "Eye movements in reading: Models and data," *Behavioral Brain Sci.*, vol. 26, no. 4, pp. 507–518, 2003.

[41] 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.*, 2006, pp. 133–140.

[42] 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.*, 2010, pp. 35–44. [Online]. Available: http://dx.doi.org/10.1109/WCRE.2010.13

[43] D. Dearman, A. Cox, and M. Fisher, "Adding control-flow to a visual data-flow representation," in *Proc. 13th Int. Workshop Program Comprehension*, 2005, pp. 297–306. [Online]. Available: http://dx.doi.org/10.1109/WPC.2005.5
[44] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 255–265. [Online]. Available: http://dl.acm.org/citation.cfm?id=2337223.2337254
[45] A. Stefik and S. Siebert, "An empirical investigation into programming language syntax," *ACM Trans. Comput. Edu.*, vol. 13, no. 4, 2013, Art. no. 19.
[46] R. Scupin, "The KJ method: A technique for analyzing data derived from japanese ethnology," *Human Organization*, vol. 56, no. 2, pp. 233–237, 1997.

Ameer Armaly is a blind student working toward the PhD degree at the University of Notre Dame advised by Dr. Collin McMillan. His research is in software engineering with a focus on code reuse, feature localization, and program comprehension of blind programmers.

Paige Rodeghero is working toward the PhD degree at the University of Notre Dame advised by Dr. Collin McMillan. Her research interests include source code summarization and program comprehension.

Collin McMillan received the PhD degree from the College of William & Mary, in 2012, focusing on source code search and traceability technologies for program reuse and comprehension. He is an assistant professor with the University of Notre Dame. Since joining Notre Dame, his work has focused on source code summarization. His work has been recognized with multiple best paper and distinguished paper awards, and the NSF CAREER award. He is a member of the IEEE.