StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code

Catherine M. Baker, Lauren R. Milne, Richard E. Ladner
Computer Science & Engineering
University of Washington
{cmbaker, milnel2, ladner}@cs.washington.edu

ABSTRACT
It can be difficult for a blind developer to understand and navigate through a large amount of code quickly, as they are unable to skim as easily as their sighted counterparts. To help blind developers overcome this problem, we present StructJumper, an Eclipse plugin that creates a hierarchical tree based on the nesting structure of a Java class. The programmer can use the TreeView to get an overview of the code structure of the class (including all the methods and control flow statements) and can quickly switch between the TreeView and the Text Editor to get an idea of where they are within the nested structure. To evaluate StructJumper, we had seven blind programmers complete three tasks with and without our tool. We found that the users thought they would use StructJumper and there was a trend that they were faster completing the tasks with StructJumper.

Author Keywords
Blind Programmers; Screen Reader; Navigation; Accessibility; Code Structure

ACM Classification Keywords
H.5.2 [**Information Interface and Presentation**]: User Interfaces

INTRODUCTION
Computer programmers rely on the use of visual aids when programming [11], especially in an integrated development environment (IDE) such as Eclipse. These visual aids range from using different colors for syntax highlighting to using indentation within the code to indicate scope. The use of visual aids present difficulties for blind programmers, as they are unable to quickly access the same information available to sighted developers. In fact, blind developers have more difficulties navigating and understanding the structure of code than their sighted counterparts [6, 9, 11].

CHI 2015, April 18 - 23 2015, Seoul, Republic of Korea
Copyright 2015 ACM 978-1-4503-3145-6/15/04…$15.00
http://dx.doi.org/10.1145/2702123.2702589

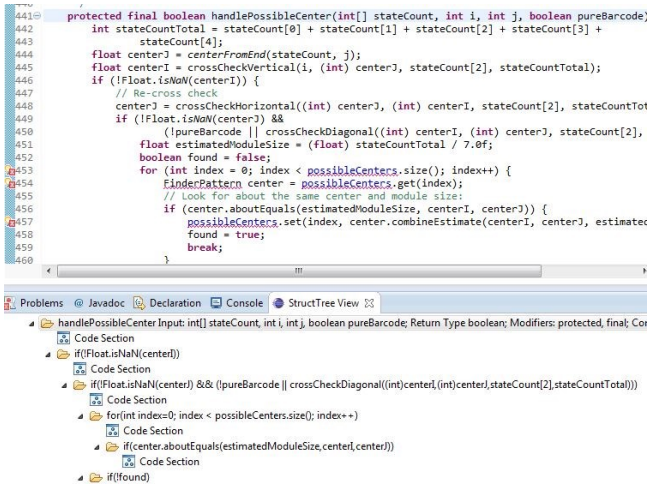

Figure 1. Screenshot of StructJumper with source code file on top and tree of nesting structure on bottom.

Screen readers only allow blind programmers to have access to a single line of code at a time. Therefore, to move around in the code, the programmers are limited to a few options: using the arrow keys to go through each line of code, using the outline or package explorer to navigate to a specific method and then navigating within the method line by line, or using a search mechanism. Despite these difficulties, the space of accessible developer tools and studying the practices of blind programmers is still a young field.

Smith et al. created a tool to allow blind programmers to navigate the hierarchical structure of a program, specifically the tree structure of files in the Eclipse IDE [9]. Our tool, StructJumper, expands on this work by creating a hierarchical tree of the nesting structure of a program (see Figures 1, 2 and 3) to allow users to both navigate within the program and gain an understanding of the structure of code within the program. We create one tree per Java file, and the root of each tree is an invisible node corresponding to the file. A node is a child to another node if the code of the child node is nested within the code of the parent node. Inner nodes represent classes, methods or statements, and leaf nodes are code sections without any changes in nesting. We include these code sections in the tree, because we want to allow users to easily switch between coding and finding where they are in the tree structure, so every line of code must be contained within a node on the tree.

StructJumper allows the user to quickly discover in which nested structure a particular line of code she is working on. She can do so by pressing a key in TreeView to jump to the node corresponding to the current location. Moreover, it allows the user to switch between being able to make edits within the code and gaining contextual information without losing her place. We present a prototype of our plugin for Eclipse for a single programming language (Java) and evaluate our tool in a user study with seven blind developers. We had participants perform a variety of code exploration tasks both with and without our tool.

With our evaluation, we aim to answer the following research questions:

(1) Does StructJumper make it easier for a blind programmer to navigate the code?

(2) Does StructJumper make it easier for a blind programmer to understand where they are within the code?

To evaluate StructJumper, we had seven blind programmers complete three tasks related to navigation and answer questions about the context of a line of code. We found that the users in the study thought our tool was useful for navigation and for understanding the structure of code.

Our contributions are:

(1) The StructJumper tool itself, available as a plugin to Eclipse.

(2) The results from our evaluation of our tool, which show that StructJumper is useful in helping blind developers navigate code and gain an understanding of which statements a line of code is nested within.

(3) Insights into how designers should create similar navigational tools for blind programmers.

BACKGROUND AND RELATED WORK

Practices and Challenges for Blind Programmers

The space of blind developer tools and the investigation of the blind developer programming practices is still a young field. Mealin et al. [6] interviewed eight blind developers and highlighted several practices employed by and challenges faced by blind developers. They found that despite issues with integrating screen readers with the complexities of integrated development environments (IDEs), five of the eight blind developers had used an IDE such as Eclipse or Visual Studio. However, the researchers found that blind developers rarely used and were not aware of the tools available to them within these complex IDEs [6]. This could be because it is more difficult for blind developers to explore the user interface of IDEs or the extra tools are not accessible. However, it could also be that the added complexity of the environment detracts more than the extra functionality adds in total usefulness. In our evaluation, the programmer is aware of the navigation tool and it is accessible, allowing us to explore the benefit provided: the ability to better understand the structure of the code.

Blind developers also indicated that they often had a difficult time getting an overview of the code. For example, skimming the code is not possible due to the linear nature of the screen reader. They tend to rely more heavily on API documentation and header files to get an idea of how the code is structured, although this is not useful if the code is not well-documented. Blind developers also use the find function to gain structural information by searching through common keywords (e.g. public, private, if etc.), which is often more time-consuming than skimming the code [6]. The researchers noted that none of the blind developers mentioned using other code navigation tools, such as moving the cursor back to the last edited text position. It was unclear whether this was because the developers did not know about these tools or because they did not find them useful. Blind developers found it difficult to search through the code to find variables or methods, especially since it requires moving the cursor to a specific line in order to have the screen reader read it aloud. Therefore, they could not easily alternate between editing the code and finding the piece of information that they need [6]. Based on the challenges presented in this paper, we decided to develop a better way to navigate through code and made the design choice to allow the user to switch between her location in the code and the nested tree structure without moving her cursor and losing her place.

Audio Based Programming Tools

There has been some prior work on creating accessible tools for blind developers. Stefik et al. created Sodbeans, a new programming IDE, which relies on audio cues to convey information such as complier errors or changing the values of variables while debugging. Sodbeans' auditory cues are built on three principles that we will also apply to the screen reader cues given by StructJumper: 1) they are short, 2) they are "browsable" (i.e. you can browse through the cues by only listening to the beginning of each cue), and 3) the important information comes first [12]. Another group created Audio Programming Language (APL), a new programming language specifically designed to help teach people who are blind how to program [8]. Stefik et al. used Sodbeans at a programming camp for blind high schoolers, but their evaluation focused on the efficacy of the curriculum and not Sodbeans itself [12]. Similarly, the focus of the evaluation of APL was on the ability of the student to learn programming concepts [8]. In contrast, as we evaluate StructJumper, we are trying to determine the effect of the nesting level navigation on experienced programmers who are already well-versed in programming concepts.

Another challenge mentioned by blind developers is using debugging tools, as most debugging tools do not work well with screen readers. Stefik et al. integrated an audio debugging tool in the Sodbeans IDE [12]. The same research group [10] also created a debugging tool for Microsoft's

Visual Studio IDE, which used sonification (non-speech audio) to aid developers. In a feasibility study, they found that blind developers were able to understand the sonification cues the majority of the time. Although not designed specifically for blind developers, Vickers et al. [13] added auditory cues to debugging tools by mapping the entry, exit and evaluation of program constructs (if, while, for, etc…) in Pascal to different musical cues. They found that sighted people learning programming found this useful to find bugs. Although the authors found that the audio cues were useful, they only used a small number of cues to map onto a small number of constructs. Other work suggests that auditory cues are difficult to understand and learn [7], so it seems likely that using audio cues would only be useful to convey short information, but not be useful to convey more complex information, so we chose not to rely on non-verbal audio cues for our navigation tool.

There has also been work on navigating through large projects in a non-visual manner. Smith et al. [9] wrote an Eclipse plugin to navigate the hierarchal structure of files in the Eclipse IDE and found that both blind programmers and sighted programmers who could not see the screen found it to be useful. We would like to expand on this idea by extending it to work on structures within the code, specifically to move between nesting levels.

Stefik et al. explored the use of audio cues to indicate the lexical scoping relationship between program statements [11]. These relationships were determined dynamically and the cues played when a change in scope was detected as the program executed. We created a similar static tool, which generates a tree that can be used to navigate through source code without running the program.

Navigation Aids for Screen Readers

Although there has been minimal research into tools for blind developers and ensuring that screen readers work well with IDEs, there have been useful research efforts on navigating web pages and textual documents with screen readers [1, 15]. In early work by Asakawa et al. [1] on developing an add-on to a screen reader that could read web pages, the researchers found that navigation was important to the design of the screen reader. Unlike navigation in IDEs, many controls allowed users to skip between links or lines on a page or skip directly to the first or last link.

With current standard screen readers, users can use controls to switch between header types (e.g. h2 and h3) and then skip from header to header on web pages. In a 2012 survey by WebAIM [14], 61% of 1782 respondents reported using headers as the main form of navigation when trying to find information on a lengthy web page, as opposed to using the find feature, navigate using links, landmarks or simply reading the page. Additionally, 82% of respondents found having different heading levels either useful or very useful when navigating a web page. However, in order to allow users to navigate using this structural information, webpage creators must provide it. The Web Content Accessibility Guidelines 2.0 [15] provides guidelines to help developers create accessible pages. The guidelines require that "Information, structure, and relationships conveyed through presentation can be programmatically determined or are available in text," and that "Headings and labels describe topic or purpose" [15]. Headers are particularly important for navigation and understanding content on a webpage.

Our plugin works similarly by designating certain parts of the code as key structural parts of the code (e.g. method and class declarations, control flow lines, etc.), which serve similar purposes as the headers on a website. Although there are tools available that can move between nesting levels in IDEs today, to the best of our knowledge no one has studied whether they are useful for navigation by blind developers, and they do not provide the ability to switch between navigation while still maintaining the current position of the cursor, which is useful for blind developers.

Code Navigation

While there have been many researchers who have looked into code navigation, many of the techniques would not be helpful for blind programmers. One of the most common techniques to help programmers is to make more information visible to provide the programmer better context such as providing a fisheye view of the code [5].

Other researchers have focused on allowing users to rearrange code in order to help them be able to navigate quicker [3,4]. While it may be beneficial for blind programmers to be able to rearrange their code, it does not help with the problem of navigating within those code sections.

STRUCTJUMPER DESIGN AND IMPLEMENTATION

We created StructJumper, a plugin for Eclipse. We chose to use Eclipse for a couple of reasons. Eclipse is a mainstream IDE that is used commonly by both blind and sighted programmers. As it is common for programmers to work in groups, having a common IDE is beneficial. Additionally, Eclipse is open source and has good support for creating and adding plugins.

For the plugin, we are combining two concepts that have already been used in software development. This first concept is turning code into a tree structure. This has been done with Abstract Syntax Trees (AST). We are using a simplified version of an AST, as we do not want to overwhelm the programmer with too much information. Grouping code together at a certain nesting level is not a new concept and is frequently done by visual cues (e.g. indentation or highlighting). We are just adjusting this method to be in a format accessible to blind programmers.

The plugin creates a hierarchical tree of the code based on nested structure (Figures 2 and 3). Nodes are broken into two categories: code sections and statements that precipitate a change in nesting. Code sections are sequential lines of code that are all within the same level of nesting. They can

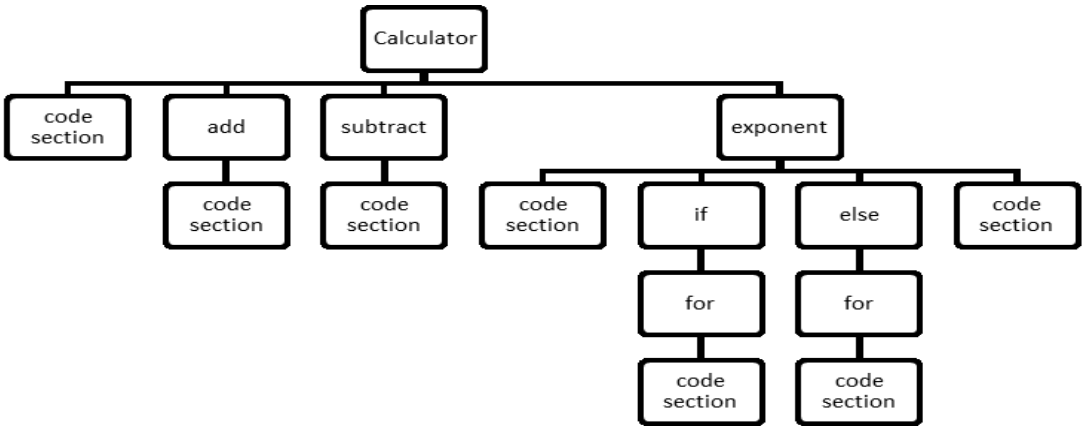

Figure 2. The tree created from the code in Figure 3. Code sections have no further nesting. Note in this image, the first code section corresponds to the code containing the member variable declared at the beginning of the Calculator class.

```
public class Calculator {
        private String display;
        public int add(int a, int b) {
                return a+b;
        }
        /**This method subtracts b from a*/
        public int subtract(int a, int b)  {
                return a-b;
        }
        public double exponent(int a, int b) {
                double answer = 1.0;
                if(b>=0) {
                        for(int i = 0; i<b; i++) {
                                answer = answer*a;
                        }
                }
                else{
                        for(int i = 0; i<b; i++) {
                                answer = answer/a;
                        }
                }
                return answer;
        }
}
```

Figure 3. This is the code of a simple calculator class, which is turned into the tree in Figure 2.

only be a leaf on a tree, not a parent to other nodes. A procedure call is not included as a separate node in the tree, but within the containing code section.

The tree is created in a separate window so that a programmer can use StructJumper both to navigate as well as to gain contextual information. This is further enabled by the key commands (Table 1). One of the major decisions we made was what should happen when you enter and exit StructJumper.

On entering, there were two options, (1) update the selected node to the one that represents the cursor location or (2) to leave it on the previously selected node. There are good arguments for either, but based on limitations of the screen reader, we decided to leave the cursor on the previously selected node (option 2).

On exiting there were also two options. (1) update the cursor location to that of the selected node or (2) leave it at its previous location. Work by Mealin [6] shows that being able to get information (such as a method name) while leaving the cursor in the current location is valuable to programmers and that many blind programmers create workarounds to gain this functionality by using an text buffer. Therefore, we allowed for both options to be possible with 'E' updating the cursor location and 'Ctrl+F7' leaving the cursor in the previous location.

In Eclipse, the Package Explorer window has a similar layout, but only includes class, fields and methods. As we wanted to make sure that our tool used the same mental model as all the other parts of Eclipse, we used that model for our tool as opposed to one similar to Figure 2. While this was inspiration for the layout and key selection choice, we made one change in how the arrow keys work. In the Package Explorer, if a user presses the down arrow key when the selection was on "TreeParent" (Figure 4) then the selection moves to "children." To avoid moving the selection onto a further nested statement without the programmer being aware, the programmer has to use the right arrow to move to a child. If the user presses the down arrow, the selection would move from "TreeParent" to the next item at the same nesting level, "ViewContentProvider."

When the user is in navigation mode, the screen reader reads relevant and unique cues (such as method names) first and then provides the rest of the information, so that users can quickly navigate and skim through code as suggested by Stefik et al [12]. In order to present the most important information first, we reordered the presentation of several lines of code that are in the tree.

For a method, we first present the name, followed by the input, return type, and then any modifiers or annotations. For a class declaration, we first present the name, then what it extends and implements, followed by the modifiers.

Figure 4. This is an example of a portion of what the Package Explorer in Eclipse would show.

When necessary, keywords are inserted to distinguish the end of one type of item and the start of the other. For instance the keyword "return type" would be added in between the input and return type to more easily distinguish the difference. Other lines of code are read as is, as they already have the important keywords first. For example, the subtract method in Figure 3 would be read "*subtract, Input: int a, int b, Return type: int Modifiers: public Comments: /**This method subtracts b from a*/.*" The plugin is written in Java, and currently only parses Java code.

EXPERIMENT DESIGN

To evaluate StructJumper, seven blind programmers completed three tasks, while using StructJumper and without the tool. After they had completed the tasks, we asked them questions about their experience.

Participants

We conducted the study with seven blind programmers, one of whom was female. Participants were recruited using emails lists and our contacts. The average age of the participants was 24.1 (SD = 4.9). The programmers had an average of 7.8 years of programming experience (SD = 3.9), with a minimum experience of 3.5 years. Additionally, the participants had an average of 2.8 years of experience with Eclipse (SD = 2.6) and 3.8 years of experience with Java (SD = 2.8). The minimum experience for Java and Eclipse was 0.5 years.

Table 1. A table of the keyboard shortcuts that can be used to navigate in the tree and the code editor.

Key	Action
Ctrl+F7	Switches between tree and editor and leaves the cursor/selected node at previous location (Eclipse Built-In Command)
Left Arrow	Go to parent
Right Arrow	Go to first child
Up Arrow	Go to previous sibling
Down Arrow	Go to next sibling
C	Go to node representing cursor location
T	Go to top of the tree
U	Update the tree
E	Switches to the editor and updates the cursor location to that of the current node

Set-Up

The study was conducted remotely, allowing each participant to use their own computer set-up. The participants used a variety of screen readers including JAWS, NVDA, and Window Eyes. By conducting the study remotely, the participants were able to use the settings in which they are comfortable, such as talking speed or amount of punctuation to speak. Using the screen sharing abilities of Skype and Google Hangouts, the researchers were able to watch and record as the participants completed the tasks and track their progress.

Procedure

Before the study, the participants were asked to fill out demographic information and install StructJumper. Participants were not given access to the code until minutes before we started the study.

The study was divided into three parts, completing a series of tasks with our tool, completing a series of tasks without our tool and the post-session interview. There were two different code bases and each code base had three tasks that were similar to each other. The participants completed the tasks on one code base using our tool and one code base without our tool. The order of the code bases and whether they used our tool first or second was counterbalanced.

As common advice to improve programming skills is to read other people's code, we selected two trending repositories from GitHub. The code bases were selected as they each had a long file (600-800 lines of code), which was well commented, on which navigation would not be a trivial task. The two repositories chosen were the ZXing[1] repository which scans QR codes and the other was MPAndroidChart[2] which creates charts and graphs for Android applications. The ZXing file selected is code that searchs the image for FinderPatterns which are markers in the corners of QR codes. The MPAndroidChart file is the code that creates Pie Charts in Android applications.

As the users were unfamiliar with the StructJumper, the users were first given a short tutorial on how to use the tool. They were given a description of the tree created and an overview of the key commands that could be used with the tool. Then, they could practice using the tool on a toy code base that did a variety of matrix calculations. Once they felt

[1] https://github.com/zxing/zxing
[2] https://github.com/PhilJay/MPAndroidChart

familiar with the tool, they were asked to follow a series of directions to check if they knew each of the key commands.

Before completing the tasks, users were given a chance to familiarize themselves with the code. The participants could spend up to 15 minutes becoming familiar with the code. Users were given the option to use StructJumper in order to familiarize themselves as well as their own methods.

Once they were familiar with the code, the participants completed three tasks. The tasks were selected to get at the two goals of the tool: improving understanding of nesting information and improving navigation within code. There were two navigation tasks, which had non-obvious so that the users would need to navigate more to determine the correct answer. There was one context task, which was nested deeply so that it was possible to either miss a condition or to add a condition to their answer. The tasks were similar for each code base. For the StructJumper tasks, participants were asked to use the tool, but did not have to use it exclusively.

Mealin [6] mentioned that search was a technique that some blind programmers use to navigate in the code. To simulate exploring code in which you may or may not know keywords to look for, we phrased the two feature location tasks different. One of the feature location task's answer could be found by using search on the keywords included in the question (referred to as the *With Keywords* task). The other feature location task purposefully did not use keywords that would allow the answer to be found using search (referred to as the *Without Keywords* task). In this way, we can investigate the differences in the use of StructJumper when search is effective or is not effective for navigation.

The third task, regarding which conditions were necessary for a line to execute, involved a deeply nested line of code (referred to as the *Conditions* task*)*. In both code bases, the line was within three if statements or for loops and there was at least one other if statement on the same level as an if statement necessary for the line to execute.

The three tasks for the ZXing code were:

1. *With Keywords:* Find the location in the code where we skip more than the normal number of rows of the image in our search for finders patterns
2. *Without Keywords:* Find the location in the code where after we have found all the potential finder patterns, we determine which are most likely the actual finder patterns
3. *Conditions:* What are the conditions necessary for line 463 to execute?

The three tasks for the MPAndroidChart code were:

1. *With Keywords:* Find the location in the code where the text for each slice of the pie chart is added
2. *Without Keywords:* Find the location in the code where the size of all the chart slices are determined
3. *Conditions:* What are the conditions necessary for line 300 to execute?

The tasks were timed and the answers were recorded for later analysis. The task time started when the investigator had finished reading the questions and ended when the participant had stated their answer and stopped looking through the code.

The answers were evaluated on a 3 point scale. Participants received 3 points on the feature location tasks if they found the correct location in the code, 2 points if they found a similar or related section of code, 1 point if it was loosely related or similar, and 0 points if it was not at all related or similar to the correct answer.

For the *conditions* tasks, the participants were awarded full points if they correctly identified all the conditions and did not add any conditions. If any conditions were missing or erroneously added, a point was subtracted per condition. If a participant had more than 3 errors, they were just given a 0. It was not possible to get negative points.

Once they had completed the three tasks, the participants were asked to rate their experience completing the tasks on a seven point semantically anchored scale. They were asked:

1. How easy the tasks were to complete: 1 – Very Hard to 7 – Very Easy
2. How frustrating the tasks were to complete: 1 – Very Frustrating to 7 – Not at all Frustrating
3. How well they knew where they were in the code while completing the tasks: 1 – No idea where they were in the code to 7 – Always knew where they were in the code

After both sets of tasks had been completed, the participants were asked to reflect on the differences in their experience completing the tasks both with and without StructJumper.

Design and Analysis

We used a 2x2 within-subjects factorial design with factors of the code base and whether or not StructJumper was used. Each participant completed three tasks for each code base. We presented the tasks in the same order for each code base, but the order was counterbalanced for each participant using a Latin square. Participants completed a total of 6 tasks for a total of 42 tasks completed altogether.

While analyzing completion task completion time, we used a mixed-effects model analysis of variance with a fixed effect of *Tool*, with *Participant* modeled as a random effect. For the semantically anchored scale data, we looked at the descriptive statistics.

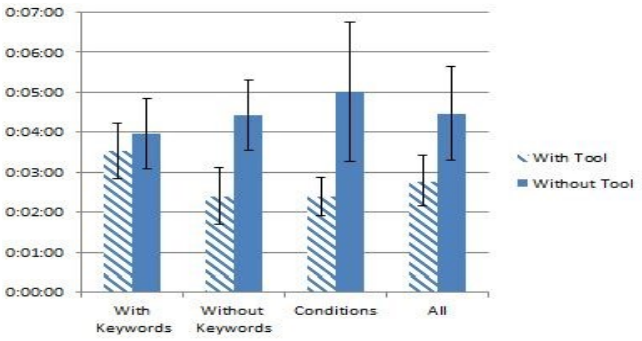

Figure 5. This chart shows the average completion time that it took participants to complete the three tasks broken down by type. The bars represent the standard error.

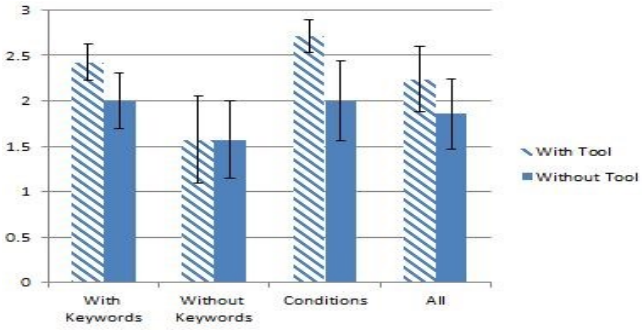

Figure 6. This chart shows the average score for all participants on the three tasks broken down by task. The bars represent the standard error.

RESULTS

To analyze the results, we used task completion times, task scores, and reported semantically anchored scale values. We found that participants were faster using StructJumper and had a better experience while using our tool as they were less frustrated and were more aware of where they were in the code.

Task Completion Time

Participants completed all of the tasks in an average time of 3 minutes and 38 seconds. Participants were faster with StructJumper (mean = 2m 47s, SD = 1m 41s) than without (mean = 4m 28s, SD = 3m 7s). While *Tool* did not have a statistically significant effect on *Time*, there was a trend in this direction ($F(1,6) = 5.783$, $p = .053$). The largest difference in time for the tasks came on the *conditions* task (see Figure 5) where participants performed faster with StructJumper (mean = 2m 24s, SD = 1m 16s) than without (mean = 5m 1s, SD = 4m 34s).

Task Score

The average score for participants was 2.0 (SD = .99). There was no significant effect of tool on score ($F(1,6) = 1.038$, n.s.). The average score with StructJumper was 2.2 (SD = .94), and 1.9 (SD = 1.01) without. The largest difference in task score was on the third task, (Figure 6), where participants received higher scores using StructJumper (mean = 2.7, SD = .49) than without (mean = 2.0, SD = 1.15). It is not surprising that participants performed better on this task, as providing the nesting context was one of the main goals for StructJumper. When participants did the completion task without the tool, two participants missed at least one condition and one participant added an extra condition, and one participant made both of these mistakes. With StructJumper, only two participants made errors, where one missed a condition and one added a condition.

On the feature identification tasks of *With Keywords* and *Without Keywords,* many of the participants found related or similar sections of code. Full credit was achieved on 10 out of the 28 feature identification tasks.

For example, one of the tasks asked the participants to find the section of code where the code skipped more than the normal number of rows in the search through the image. For this task, many participants found sections of code that determined how many rows of the image to skip instead of where the actual skip happened.

Another task asked participants to find where the most likely finder patterns were identified from all the potential finder patterns that were found. One participant found a section of code that identified if a single section of the image is likely a finder pattern.

Participants may have found related sections of code as opposed to the correct answer because our tasks purposefully contained as little information about the correct code section as possible. We made this decision in attempt to mimic a real life situation where the programmer is searching for a certain feature or action in a newer or unfamiliar code base. This choice may have added difficulty for users to locate the correct section of code, and thus caused users to find a section that is similar or related.

Participant Experience

We gathered insights on participant experience by asking three semantically anchored scale questions after each set of three tasks. We asked about how easy they found the tasks to complete, how frustrated they felt completing the tasks and how well they knew where they were in the code.

For how easy they found the tasks to complete, there was no difference in the average. The average score was 4.1 for both. However, the standard deviation was larger with the tool, than without the tool (SD = .83 vs. SD = 1.64).

When asked about their frustration when completing the tasks, the average was higher with the tool. For this semantically anchored scale, a higher number was less frustrating. The average was 4.3 without the tool (SD = 1.67) and 5.0 with the tool (SD = 1.4).

The largest difference however, was when the participants were asked about how well they knew where they were in the code (see Figure 7). The average was higher with the tool on this question as well. Once again, a higher number

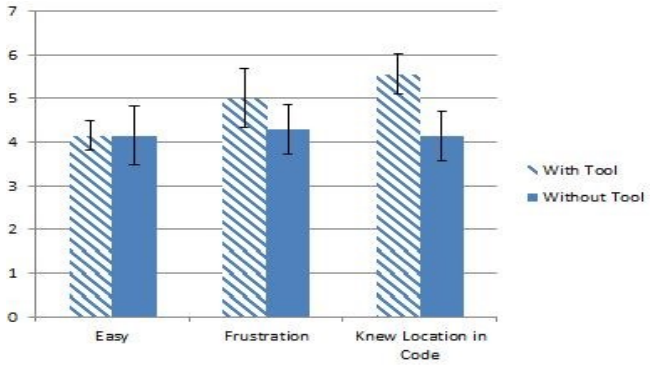

Figure 7. This chart show the average score for the participants for the semantically anchored questions. A higher value is better for all three questions. The bars represent the standard error.

is better as it meant they knew more often where they were in the code (1 was they never knew where they were and 7 was they always knew where they were). The average without the tool was 4.1 (SD = 1.12) and the average with the tool was 5.6 (SD = 1.40).

One thing that stood out about the responses for the question about how well they knew where they were in the code was the number of people who felt like they always knew where they were (response of 7) or almost always knew where they were (response of 6). Without the tool, nobody indicated they always knew where they were and only one person indicated that almost always knew where they were. Conversely, with the tool, two people always knew where they were and another three people almost always knew where they were.

QUALITATIVE RESULTS

After they had completed the sets of tasks with and without StructJumper, the participants were asked to reflect on completing the tasks themselves, their ability to complete the tasks, their knowledge of where they were in the code, and their understanding the code. Participants were asked if they would use StructJumper, and if so, for what types of tasks. Five of the participants indicated that they would use the tool.

To analyze the qualitative results, we looked at the interviews for concrete examples provided by the participants on how StructJumper changed the experience of completing the tasks and for examples on how the participants indicated that they would use the tool. Themes that were brought up by multiple participants were included in the results.

Quicker and Easier

As found in our results, there was a trend that StructJumper may have reduced the task completion time. In fact, six participants said that felt that they were able to navigate through the code faster and easier:

It's much easier than reading code. It's far more efficient because I'm reading relevant information. I don't have to read the complete code.

They mentioned that they were able to find the relevant information from the tool and skip between methods. When they were deep within a method, they were able to get information about where they were more quickly than without StructJumper.

Better Understanding of the Layout

Six participants indicated that the tool helped them with their understanding of how the code was laid out and how the statements were related to each other. Participants also felt that it helped them get a broad overview of the code.

This was particularly seen in the *conditions* task. As one participant was asked to complete the *conditions* task without StructJumper, they said:

I don't know how to see that in Eclipse. For that matter, in fact, I don't know how to see it in any of the IDEs I've worked in. Short of reading the code using brute force.

Another participant mentioned as they started the *conditions* task that this was one case where they would definitely want to use the tool as it made it a lot easier. This was also a common type of task that participants mentioned in the interviews that that the tool would be helpful for.

Participants also indicated that StructJumper was useful for gaining a broad understanding of the code. One participant indicated that they would be likely to use this to skim the code a few times before reading the code line by line when first introduced to a large, new code base.

Lack of Cues

Two participants mentioned that StructJumper was helpful when there is a lack of cues to indicate how far a statement is nested in the code. There were two cases of this mentioned by participants.

For instance, one participant mentioned Python. The participant mentioned that in languages like Java, there are cues, such as curly braces, to indicate the start and end of a nesting level. These cues make knowing how far nesting a statement is possible. Without these cues, the participant indicated that it is much harder to know where the statement was. Therefore, expanding this concept to a language like Python could potentially have a large impact on a blind programmer.

Another participant indicated a similar sentiment about the lack of cues. In order to have the screen reader speak cues such as the braces, the settings of the screen reader need to be set to speak all the punctuation. However, the participant said:

I really hate changing the punctuation verbosity of my JAWS, on my screen reader, to most or all. So it's always set at none. So I don't even know braces and stuff like that.

And it's just that since I read through the code so many times and I understand it so well that I can figure out, ok well here has to be a brace or here is where the indentation changes. So I never use a feature ever to actually notify me. It's really annoying. So that one thing that this tool really helps with. I know, ok, well this condition, it's within the other condition, within that loop. So it kind of helps that way.

As participants were completing the tasks, we saw some change the verbosity of the punctuation level as they were completing the tasks as they may only use that level of punctuation verbosity for programming and it may be the case that others would prefer not to have to have the screen reader read all the punctuation all the time.

Change in Focus

One of the prompts we asked the participants to reflect on was whether or not the tool affected their ability to understand the code. A few brought up that it removed the number of things that they would have to focus on as they were going through the code, which may make it easier to understand. For example, some participants mentioned that it allowed them to focus less on the little details of how to navigate or keep track of where they are in the code. One participant said:

You know, the navigation part, you know, without it, I was more focusing on that probably. How do I navigate, how do I get to the next thing, what keywords can I use to easily jump to where it needs to go? So, you know, without having to do that, you know, maybe I was focusing more on understanding what the code actually does.

Another participant indicated that they would use the tool when they were trying to understand the code. They said that it was useful when:

Trying to keep track of what level you're on basically… If I'm reading through code and trying to remember how many right braces you have remaining, how the different conditionals are related to each other.

It allowed them to see the relationship between blocks of code more easily. For instance, one participant indicated that is made it easier for them to know which conditionals a statement in the code it was nested under.

Unfamiliar Code

Multiple participants indicated that this tool would be more helpful for unfamiliar code than for code they know well. Many indicated that this was because, for code in which they are familiar, they already know the keywords they can use to jump to a section of code or what statements are under which conditionals. StructJumper aims to help provide this information, so it helps them learn when they first see a code base.

In familiar code, the biggest benefit this tool may add is the ability to navigate more quickly to that line, as mentioned by one participant. The tool can be used to skip lines and the user can quickly skip through code as they know each where the section is nested. This could be better than a keyword search as frequently the keywords may not be unique to that section of code only, and far better than moving through code line by line.

Programming Use

There were some participants who mentioned use cases for the tool that were more related to coding and finding errors in code. Some participants mentioned that this might be good at finding errors such as improperly matched braces. Another area that a participant suggested it might be useful to use it to skim over the logic for errors, such as looking at what the switch cases are to determine why the correct case is not getting called.

One participant mentioned that in his work, he might receive feedback from customers about the functionality of the application. He could then use the tool to navigate to a section of code where the functionality in question is and then fix the problem or make the requested change.

The researchers are interested in investigating the use of this tool for programming by conducting follow-up studies.

DISCUSSION

We believe the lack of significant effect of the tool on time comes from some variation in the participants. For 8 of the 21 tasks, participants were slower with our tool than without. Of the 8 slower tasks, 2 were on C*onditions* tasks, 3 were on the *With Keywords* task and 3 were on the *Without Keywords* tasks.

We attribute this variation to a variety of reasons. In one case, the participants only found one of the conditions without the tool and with the tool found all three and spent more time double-checking them. Two of the participants that were slower on some tasks with the tool did not use anything like the package explorer or outline. They were efficient with a line-by-line navigation approach. All but one other of the participants used either the package explorer or the outline and may be more accustomed to using a similar tool.

For some participants, it was not clear what made them slower with the tool than without the tool. It may be that they were more efficient with their current method. It could also be that they just found the navigation task that they received while using the tool more difficult.

One of the limitations of this work is that it only looks at navigation within unfamiliar code. As programmers will spend much of their time navigating through code that they have been working on and is familiar to them, this study does not look at one of the main use cases. We cannot know if the benefits we saw in this study will carry over to navigation in familiar code.

FUTURE WORK

The evaluation of StructJumper focused only on navigational tasks as that is its main purpose, but it would be interesting to learn how participants use this tool to perform different coding tasks as well. Some participants mentioned its use in debugging to try and find logical errors as well as navigation to find problem sections of code. In addition, the researchers could learn by having StructJumper used in the wild. This would allow us to look at how its use might change with familiar programs versus unfamiliar programs.

Additionally, it would be interesting to research whether this tool is beneficial to sighted programmers and investigate how their use of the tool varies from blind programmers. Much of the benefit of the tool is overcoming the difficulties that screen readers have in jumping many lines of code, which can be done more easily with sight and visual clues like indentation. However, there may be benefits for sighted programmers to only look at the included lines of code and ignore other code sections as they are collapsed on the tree.

CONCLUSION

We created StructJumper, an Eclipse plugin that allows blind programmers to quickly navigate through the code and see the how specific statements are nested within the code. We ran a user study with seven blind programmers and found that there is a trend that the tool has an effect on the time it took users to complete the tasks.

We also found that participants were positive about the tool and that they would be interested in continuing to use the tool. The participants found it quicker to navigate through the code and, thought that StructJumper provided valuable information about the conditionals that apply to a line of code.

ACKNOWLEDGMENTS

This material is based upon work supported by the NSF Graduate Research Fellowship under Grant Nos. DGE-0718124 and DGE-1256082 and NSF grant IIS-1116051.

REFERENCES

[1] Asakawa, C. and Itoh, T. 1998. User interface of a Home Page Reader. In *Proc. of ACM conference on Assistive technologies* Assets '98. ACM, New York, NY, USA, 149-156.

[2] Austin, J. 2013. Want a Great Scientific Career? Choose Computer Science. From http://sciencecareers.sciencemag.org/career_magazine/previous_issues/articles/2013_03_25/caredit.a1300053.

[3] Bragdon, A., Zeleznik, R., Reiss, S.P., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F. and LaViola, J.J. Jr.. 2010. Code bubbles: a working set-based interface for code understanding and maintenance. In *Proceedings of CHI '10*. ACM, New York, NY, USA, (April 2010), 2503-2512.

[4] Henley, A.Z. and Fleming, S.D. 2014. The patchworks code editor: toward faster navigation with less code arranging and fewer navigation mistakes. In *Proc. of CHI '14*. ACM, (April 2014), 2511-2520.

[5] Jakobsen, M.R. and Hornbaek, K. 2006. Evaluating a Fisheye View of Source Code. In *Proc. CHI '06,* ACM Press, (April 2006), 377-386.

[6] Mealin, S., Murphy-Hill, E. 2012. An exploratory study of blind software developers. *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, (September 2012), 71-74.

[7] Palladino, D. and Walker, B. 2007. Learning rates for auditory menus enhanced with spearcons versus earcons. In *Proc. of International Conference on Auditory Display* (Montreal, Canada June 26-29, 2007) ICAD '07.

[8] Sánchez, J., Aguayo, F. 2005. Blind learners programming through audio. In *CHI'05 extended abstracts on Human factors in computing systems*, ACM, (April 2005), 1769-1772.

[9] Smith, A., Cook, J., Francioni, J., Hossain, A., Anwar, M., and Rahman, M. 2003. Nonvisual tool for navigating hierarchical structures. In *SIGACCESS Access. Comput.*77-78 (September 2003), 133-139.

[10] Stefik, A., Alexander, R., Patterson, and Brown, J. 2007. WAD: A Feasibility study using the Wicked Audio Debugger. In *Proceedings of the 15th IEEE International Conference on Program Comprehension* (ICPC '07). IEEE Computer Society, Washington, DC, USA, 69-80.

[11] Stefik, A., Hundhausen, C., and Patterson, R. 2011. An empirical investigation into the design of auditory cues to enhance computer program comprehension. In *International Journal of Human-Computer Studies*, 69 12, (December 2011), 820-838.

[12] Stefik, A., Hundhausen, C., and Smith, D. 2011. On the design of an educational infrastructure for the blind and visually impaired in computer science. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, ACM, (March 2011), 571-576.

[13] Vickers, P. and Alty, J. 2002 When Bugs Sing. In *Interacting with Computers*, 14 (6). 793-819. ISSN 0953-5438

[14] WebAIM, 2012. Screen Reader User Survey #4 Results. From http://webaim.org/projects/screenreadersurvey4/.

[15] W3C, 2008. Web Content Accessibility Guidelines (WCAG) 2.0, From http://www.w3.org/TR/WCAG20/