

Rochester Institute of Technology

RIT Digital Institutional Repository

Theses

11-2020

The Role of Sonification as a Code Navigation Aid: Improving Programming Structure Readability and Understandability For Non-Visual Users

Khaled L. Albusays
kla3145@rit.edu

Follow this and additional works at: <https://repository.rit.edu/theses>

Recommended Citation

Albusays, Khaled L., "The Role of Sonification as a Code Navigation Aid: Improving Programming Structure Readability and Understandability For Non-Visual Users" (2020). Thesis. Rochester Institute of Technology. Accessed from

This Dissertation is brought to you for free and open access by the RIT Libraries. For more information, please contact repository@rit.edu.

The Role of Sonification as a Code Navigation Aid:
Improving Programming Structure Readability and
Understandability For Non-Visual Users

by

Khaled L. Albusays

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences
Rochester Institute of Technology

November, 2020

The Role of Sonification as a Code Navigation Aid: Improving Programming Structure Readability and Understandability For Non-Visual Users

by
Khaled L. Albusays

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. Matt Huenerfauth
Dissertation Advisor

Date

Dr. Stephanie Ludi
Dissertation Committee Member

Date

Dr. Vicki L. Hanson
Dissertation Committee Member

Date

Dr. Kristen Shinohara
Dissertation Committee Member

Date

Dr. Jai Kang
Dissertation Defense Chairperson

Date

Certified by:

Dr. Pengcheng Shi
Ph.D. Program Director, Computing and Information Sciences

Date

The Role of Sonification as a Code Navigation Aid:
Improving Programming Structure Readability and
Understandability For Non-Visual Users

by

Khaled L. Albusays

Submitted to the

B. Thomas Golisano College of Computing and Information Sciences

Ph.D. Program in Computing and Information Sciences

in partial fulfillment of the requirements for the

Doctor of Philosophy Degree

at the Rochester Institute of Technology

Abstract

Integrated Development Environments (IDEs) play an important role in the workflow of many software developers, e.g. providing syntactic highlighting or other navigation aids to support the creation of lengthy codebases. Unfortunately, such complex visual information is difficult to convey with current screen-reader technologies, thereby creating barriers for programmers who are blind, who are nevertheless using IDEs.

This dissertation is focused on utilizing audio-based techniques to assist non-visual programmers when navigating through large amounts of code. Recently, audio generation techniques have seen major improvements in their capabilities to convey visually-based information to both sighted and non-visual users – making them a potential candidate for providing useful information, especially in places where information is visually structured. However, there

is little known about the usability of such techniques in software development. Therefore, we investigated whether audio-based techniques capable of providing useful information about the code structure to assist non-visual programmers. The major contributions in this dissertation are split into two major parts:

The first part of this dissertation explains our prior work that investigates the major challenges in software development faced by non-visual programmers, specifically code navigation difficulties. It also discusses areas of improvement where additional features could be developed in order to make the programming environment more accessible to non-visual programmers.

The second part of this dissertation focuses on studies aimed to evaluate the usability and efficacy of audio-based techniques for conveying the structure of the programming codebase, which was suggested by the stakeholders in Part I. Specifically, we investigated various sound effects, audio parameters, and different interaction techniques to determine whether these techniques could provide adequate support to assist non-visual programmers when navigating through lengthy codebases. In Part II, we discussed the methodological aspects of evaluating the above-mentioned techniques with the stakeholders and examine these techniques using an audio-based prototype that was designed to control audio timing, locations, and methods of interaction. A set of design guidelines are provided based on the evaluation described previously to suggest including an auditory-based feedback system in the programming environment in efforts to improve code structure readability and understandability for assisting non-visual programmers.

Acknowledgments

I would like to express my sincere gratitude to my co-advisors, Drs. Stephanie Ludi, and Matt Huenerfauth for their endless guidance, encouragement, and patience during my doctoral studies. It was a great pleasure to work under their supervision and learn from each one of them. I could not have imagined having a better co-advisors and co-mentors other than Drs. Stephanie Ludi, and Matt Huenerfauth.

I would also like to thank my dissertation committee members, Drs. Vicki Hanson and Kristen Shinohara for serving on my committee as well as their encouragement and insightful feedback.

Last but not least, I would like to thank Dr. Pengcheng Shi for his continuous support and valuable advice through my doctoral studies.

*To my father, Lafi, my first inspiration, whom I wish he is here today to
share this special moment with me.*

*To my mother, Aisha, my first teacher, whose love, prayers, and
encouragement makes me able to get such success and honor.*

*To my wife, Atheer, whose love and confidence is a constant source of
inspiration.*

And to my siblings, for their endless love, support, and encouragement.

Contents

List of Figures	xv
List of Tables	xix
1 Introduction	1
1.1 Introduction	1
1.2 Overview of This Dissertation	3
1.3 Research Questions	5
1.3.1 Part I: Understanding Requirements and Needs of Non-visual Programmers	6
1.3.2 Part II: Methodological details of Sonification to Aid Code Navigation For Non-Visual Programmers	7
1.4 Dissertation Organization	8
PART I: UNDERSTANDING REQUIREMENTS AND NEEDS OF NON-VISUAL PROGRAMMERS	10
PROLOGUE TO PART I	11

2	Background and Prior Work on Programming Challenges	13
2.1	Code Navigation	14
2.2	Programming Challenges	15
3	Programming Challenges	17
3.1	Introduction	17
3.2	Research Questions	18
3.3	Methodology	18
3.3.1	Survey Design	18
3.3.2	Sampling	19
3.3.3	Procedure and Response Rate	20
3.3.4	Participants	20
3.4	Results	21
3.4.1	RQ1a: Developer Background	21
3.4.2	RQ1a: Development Tools & Platforms	22
3.4.3	RQ1b: Assistive Technology	23
3.4.4	RQ1c: Open-Ended Responses	25
3.4.4.1	Limited Accessibility Aids in IDEs	25
3.4.4.2	Code Navigation	26
3.4.4.3	Diagrams	28
3.4.4.4	Debugging & User Interface Layout	29
3.4.4.5	Seeking Sighted Assistance	29
3.4.4.6	RQ1d: Workaround Techniques	30

3.5	Limitations	31
3.6	Conclusions	31
4	Code Navigation Difficulties	33
4.1	Introduction	33
4.2	Research Questions	34
4.3	Methodology	35
4.3.1	Interview Design	36
4.3.2	Participants	37
4.3.3	Procedure	38
4.3.4	Data Analysis	40
4.4	Results	42
4.4.1	RQ2a: Code Navigation Challenges	42
4.4.2	RQ2b: Tools in Software Development	48
4.4.2.1	Assistive Technologies	49
4.4.2.2	Development Languages & Tools	53
4.4.3	RQ2c: Programming Strategies	54
4.5	User Needs	60
4.6	Limitations	65
4.7	Conclusions	66
	EPILOGUE TO PART I	69

PART II: METHODOLOGICAL DETAILS OF SONIFICATION TO AID CODE

NAVIGATION FOR NON-VISUAL PROGRAMMERS 72

PROLOGUE TO PART II 73

5 Background and Prior Work on Audio Programming and Sonification 75

5.1 Sonification and Interaction in Programming 78

5.2 Accessing Mathematical Symbols via Audio 82

5.2.1 Conveying Depth of Brackets in Equations 82

5.2.2 Mathematical Formulas 83

5.2.3 Spatial Sounds in Mathematics 84

5.3 Converting Visual Graphs to Sound 85

5.4 Representing Menus or Outlines 86

5.4.1 Designing Audio Cues 87

5.4.2 Limits of Understandability of Non-Speech 89

5.5 Discussion of Prior Work 91

5.6 Limitations of Prior Work 93

6 Formative Study 94

6.1 Background and Introduction 94

6.2 Research Questions Investigated in this Chapter 97

6.3 Methodology 97

6.3.1 Stimuli Preparation 98

6.3.2	Recruitment and Participants	101
6.3.3	Procedure and Questions	102
6.4	Results	103
6.4.1	Timing of Audio Cues about Code Structure	104
6.4.2	Speech-Based vs. Non-Speech Audio Cues	105
6.4.3	Conveying Code Structure with Audio Properties	107
6.5	Discussion	109
6.6	Summary and Limitations	110
7	Experimental Study	112
7.1	Background and Introduction	112
7.2	Research Questions Investigated in this Chapter	114
7.3	Methodology	115
7.3.1	Stimuli Preparation and Prototype	116
7.3.2	Recruitment and Participants	123
7.3.3	Procedure and Questionnaire	124
7.4	Results	126
7.5	Discussion	132
7.6	Summary and Limitations	134
	EPILOGUE TO PART II	136
8	Limitations and Future Work	140
8.1	Limitations and Future Work	140

8.1.1	Part I: Limitations and Future Work	141
8.1.2	Part II: Limitations and Future Work	142
9	Summary and Contributions	145
9.1	Summary of the Contribution of This Research	146
9.1.1	Part I: Programming Challenges and Code Navigation Difficulties	146
9.1.2	Part II: Usability of Audio-based Techniques	147
9.2	Conclusion and Final Comment	150
	Bibliography	152
	Appendices	172
A	IRB Approval Forms	173
B	Survey Questionnaire	177
B.1	Survey Questionnaire	178
C	Interview Questionnaire	185
C.1	Interview Questionnaire	187
D	Formative Study Questionnaire	189
D.1	Example Script and Questions for Semi-Structured Interview, with Links to Audio Samples	192
D.1.1	Script for the Interviewer	194

E	Larger Study Questionnaire	201
E.1	Example Script and Questions for the Audio-based Interaction	
	Techniques Experiment Study	204
E.1.1	Script for the Interviewer	205
F	Supplementary Study Materials	217
F.1	Materials Description	217
F.2	To run the Prototypes	218

List of Figures

1.1	This figure shows IDE specific features which rely mostly on user vision.	3
1.2	In this figure, we demonstrate the type of nested code that participants are having difficult time understanding with current screen-reader technologies.	5
2.1	This figure shows an overview of the code bubble metaphor: (a) user opens a bubble through the search box, (b) bubble displayed, (c) users opens two or more bubbles side-by-side, (d) large set of bubbles with a (f) bubble references, (e) an overview is displayed in the panning bar, (g) hover preview less. [21] . . .	15
3.1	The number of participants in regard to visual acuity.	21
3.2	Software Development Tools Used by Participants in This Study.	23

3.3	In this figure, we show an exmaple of Braille Display device used by users with visual disabilities to access information on the computer display.	24
4.1	A participant using JAWS with an 80-cell Brilliant Braille Display, while programming in Java using the Eclipse Integrated Development Environment (IDE).	51
7.1	For-loops code sample.	116
7.2	While-loops code sample.	117
7.3	Mix of for-loops with one while-loop code sample.	117
7.4	In this figure, we show an example of how the for-loop portion will be time-stamped based on three different levels. To create the audio file annotation, it was necessary to listen to the computer voice recording while watching the codebase at the same time in order to track the computer voice recording, i.e., change a particular level to another level.	119
7.5	In this figure, we show an example of how inserting a sound effect into the computer audio recording with 10 milliseconds delay (pause) would result in changing the original audio recording time-frame. In this example, sound effects could be anything from the speech sound category, non-speech sound category or spatial of sound category.	121

7.6	This figure shows the process of inserting sound effect into the code sample recorded version using Audacity software.	121
7.7	Overview of the JavaScript function for the on-demand prototype.	123
7.8	Percentage distribution of participants' responses on the ease of using the three conditions (e.g., control, on-demand, and automatic).	127
7.9	Percentage distribution of participants' responses on the convenience of using the three conditions (e.g., control, on-demand, and automatic).	127
7.10	Percentage distribution of participants' responses on how helpful the three conditions (e.g., control, on-demand, and automatic) when working on a computer programming code.	128
7.11	Percentage distribution of participants' responses on the ease of completing a task conducted to evaluate the three conditions (e.g., control, on-demand, and automatic).	129
7.12	Percentage distribution of participants' responses on how frustrated they were when completing an evaluation task using the three conditions (e.g., control, on-demand, and automatic). . .	129
7.13	Percentage distribution of participants' responses on understanding the current location in the nested code (e.g., code sample) using the three conditions (e.g., control, on-demand, and automatic).	130

A.1	IRB Decision Form for “Understanding the major programming challenges in software development”	174
A.2	IRB Decision Form for “Interviews about code navigation difficulties”	175
A.3	IRB Decision Form for “Evaluating the usability of audio-based techniques”	176

List of Tables

3.1	Level of expertise in various programming languages	24
4.1	List of navigation difficulties and number of participants who mentioned each during interviews; the difficulties are sorted based on this number.	43
4.2	Number of participants in our study using various operating systems, assistive technologies, programming languages, and programming editors.	55
7.1	List of scale-based questions used in the larger study.	124

Chapter 1

Introduction

1.1 Introduction

An Integrated Development Environment (IDE) is software which integrates a text editor, file management, compiler, and other tools to promote an efficient workflow for modern computer programmers [30]. IDEs play an important role in the modern software development process, especially when creating lengthy codebases [45]. The text editors in these systems often include visual aids that use indentation to indicate scope level, different colors for syntax highlighting, and various other features to help programmers understand their code structure and navigate through it more easily (Figure 1.1) [7]. Unfortunately, such complex visual information is difficult to convey with current assistive technologies (e.g., screen-readers) [12, 52, 68], creating barriers for non-visual programmers [3, 47], who are nevertheless using IDEs, as we learned in a study

in Chapter 3.

To access the computer display, blind users tend to use screen-reader technologies [49], which were designed to present information linearly (one line at a time), assuming that the software or the website is designed to accommodate the screen-reader technology [16,19]. In the case of programming, screen-reader technology reads system menus, dialog boxes, tree views of code structure, as well as provides access to other system features [47].

Prior research on blind programmers has found that the information conveyed through visual metaphors in IDEs are often not conveyed by screen-readers [9], which creates challenges for blind programmers [48,75], putting them at a disadvantage when compared to their sighted peers [35,58]. In fact, programmers who are blind, who are using screen-reader technologies, have access to fewer advanced IDE features for quickly moving through large amounts of code, often forcing them to navigate code line-by-line or jump to different locations using “find/search” features [9]. Furthermore, blind programmers also have difficulty understanding structural relationships quickly [58,77,82], which prevent them from getting an overview of the entire programming code-base [59].

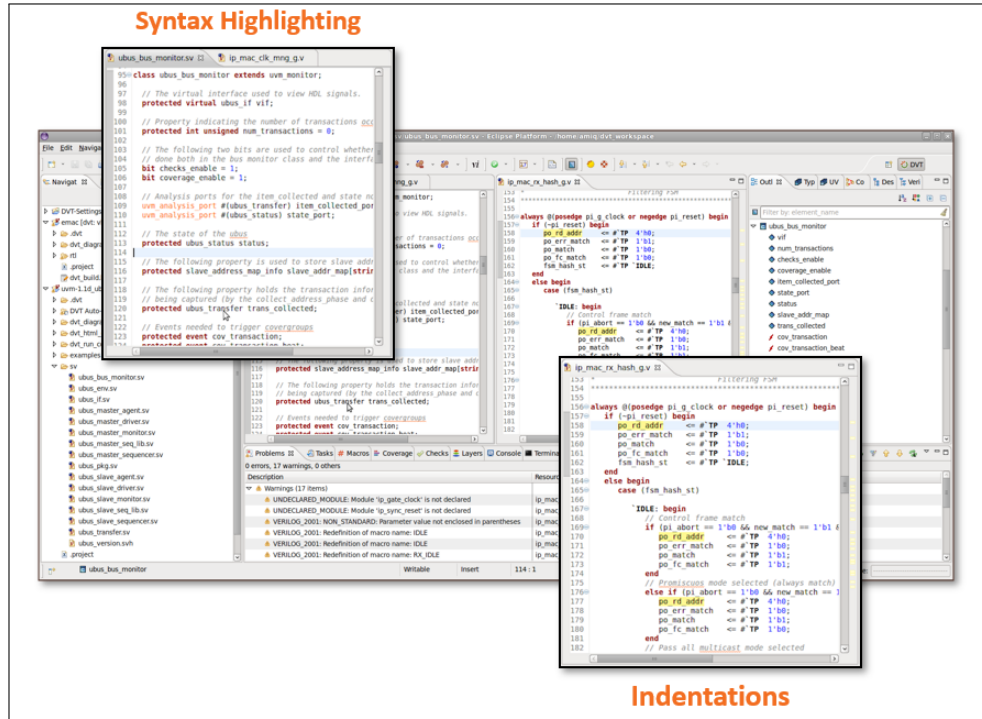


Figure 1.1: This figure shows IDE specific features which rely mostly on user vision.

1.2 Overview of This Dissertation

In this research, we investigated different ways to utilize audio-based techniques to assist non-visual programmers in understanding the structure of a programming codebase, mainly to ease code navigation. Specifically, we evaluated various sound effects (produced based on speech, non-speech, and spatial location), with modified parameters, and different interaction techniques to convey the structure of the programming codebase. This dissertation intends to answer the big research question:

How can we select specific audio interaction techniques, and best set their parameters, to best convey structural information about the programming codebase to assist non-visual users?

In our work [4,5], we explored a broad set of issues, and participants listed navigating through the code and understanding its structure as key concerns, especially nested code. By “nested code”, we refer to a code that performs a particular function and that is contained within another code that performs a border function, e.g., a loop within a loop, an inner loop within the body of an outer one (see Figure 1.2 for explanation). To understand how the code is nested, most blind programmers tend to go over the code many times until the entire code structure (nesting) is conveyed. This is due to the nature of the screen-reader technologies and how it making users feel isolated to only one line of code (or text) at a time. In this research, we examined different design dimensions in order to generate useful design guidelines for employing an auditory feedback system into the programming environment to assist non-visual programmers. Specifically, we investigated different audio-based techniques and whether they could convey the nested structure of code lines, e.g., depth of bracketing or level of indentation, as in nested loops.


```

i=0
while i<len(groups):
    j=0
    for TrainNames in groups[i]["TrainNames":
        k=0
        while k<len(TrainNames["schedule"]):
            print("[Station] " + str(groups[i]["StationName"]) + " -- [Train] " + str(
                TrainNames["name"]) + " -- [Arrival Time] " + str(
                TrainNames["schedule"][k]["ArivalTime"]))
            k+=1
        j += 1
    i += 1

```

Figure 1.2: In this figure, we demonstrate the type of nested code that participants are having difficult time understanding with current screen-reader technologies.

To answer this big research question, we evaluated a set of sound effects, audio parameters, and different audio interaction techniques to determine whether these techniques could help convey specific programming information to non-visual users. The evaluation process was based on different audio-based prototypes where participants have no control over cursor movement through the programming codebase, e.g., only limited interaction. We explain our evaluation process in Part II of this dissertation (see Chapter 6 and Chapter 7).

1.3 Research Questions

In this dissertation, we have five major research questions. The first two research questions (**RQ1-RQ2**) are mainly focused on domain understanding whereas the remaining research questions (**RQ3-RQ5**) are specifically focused on evaluating the use of audio-based techniques in programming environments. **RQ1** and **RQ2** questions have been investigated in Part I, whereas **RQ3-RQ5** are investigated in Part II. We present each one of these five research questions as follows:

1.3.1 Part I: Understanding Requirements and Needs of Non-visual Programmers

In this part, we present the first two research questions that focus on understanding the major programming challenges faced by non-visual users, especially code navigation challenges.

RQ1: In a survey-based study, what are the programming challenges that visually impaired programmers report facing, as well as workarounds or strategies to overcome these issues? The survey-based study of understanding the major programming challenges faced by non-visual programmers serves an important purpose in this dissertation. In this work, we identified a list of challenges where non-visual programmers urged the research community as well as the industry to propose additional improvements to overcome these challenges. In addition, the primary research problem discussed in this dissertation was based on the major findings from the survey-based study. We explained this work in Chapter 3.

RQ2: In an interview-based study, what are the code navigation difficulties that non-visual programmers report facing, as well as workarounds or strategies to overcome these issues? The interview-based study was conducted in order to understand the challenge of code navigation, which was indicated previously in our survey-based study. This work provided a list of additional improvements suggested

by the stakeholders in efforts to enhanced code navigation for non-visual programmers. We explain this work in Chapter 4.

1.3.2 Part II: Methodological details of Sonification to Aid Code Navigation For Non-Visual Programmers

In this part, we present the remaining three research questions aimed to investigate different methods for utilizing audio-based techniques in programming environments, in efforts to convey hierarchical nesting structure of code, mainly to assist non-visual programmers.

RQ3: In a formative interview study with a variety of audio examples, what forms of audio generation techniques and parameters do non-visual programmers express interest in? As discussed in Chapter 4, participants suggested the use of audio-based techniques to covey the hierarchical nesting structure of code. To understand the best approach, we investigated various audio cues based on different techniques in efforts to understand the suitable cues for conveying certain programming information about the code nesting structure. We explained this work in Chapter 6.

RQ4: When presented with an interactive audio prototype based on this prior formative study, do non-visual programmers prefer receiving this additional audio information about the structure of code, as compared to a control condition without such addi-

tional information? As explained in Chapter 6, we selected specific audio cues based on stakeholders' recommendations. These selected cues were examined using audio-based prototypes where participants interacted with different code samples. In this question, we look for participants' feedback and whether their prior recommendations remain the same. We explain this work in Chapter 7.

RQ5: When interacting with an audio prototype based on this prior formative study, do non-visual programmers have a preference between automatic level-crossing notifications or on-demand level indications?¹ Providing audio-based feedback about the code nesting requires some form of interaction between the user as well as the system. To understand the proper interaction, we investigated different interaction methods in efforts to understand the best approach to requested audio feedback. We explain this work in Chapter 7.

1.4 Dissertation Organization

This dissertation is structured into two major parts: The first part starts with Chapter 2 where we discuss prior work in field of programming accessibility and education. It also explains our methodological approach towards answering **RQ1** (Chapter 3) where we investigate the programming challenges more deeply via a survey-based study. In addition, we discuss our follow-up interview

¹Details of these interaction techniques are described in Chapter 7

study aimed to examine code navigation challenges more deeply, which address **RQ2** (Chapter 4).

The second part of this dissertation begins with Chapter 5, where we survey the most closely related prior work on audio-based techniques to increase the accessibility of programming for these users, to establish that little work has examined the issue of navigating the hierarchical structure of code and additional research is needed into how to convey indentation structure of individual lines of code in the context of the linear reading of code via screen-reader. In addition, we examine related research on using audio-based cues in settings that are analogous in some way, namely: conveying nesting structure in mathematical notation, conveying the relationships within graph structures, or representing navigation through nested menus or outlines. We also explain our methodological approach towards answering **RQ3** where we evaluated various audio-based cues and audio parameters in efforts to select some promising design options for the higher-fidelity prototype in the later study (Chapter 6). Furthermore, we also discuss our experimental study (Chapter 7) where we evaluated different audio-based interactions (e.g, on-demand and automatic) using different audio-based prototypes in efforts to answer **RQ4** and **RQ5**. In Chapter 8 we discuss the dissertation’s limitations where possible improvements could be conducted as future work. Finally, this dissertation will conclude in Chapter 9 by highlighting the dissertation’s major contributions and final comment about the research work presented herein.

PART I: UNDERSTANDING
REQUIREMENTS AND NEEDS OF
NON-VISUAL PROGRAMMERS

PROLOGUE TO PART I

In Part I, we will begin by discussing prior work related to software development challenges faced by programmers who are blind. Specifically, we will discuss some of the existing programming barriers in both computing education settings as well as the software industry, in addition to explaining some of the current design interventions for making programming environments more accessible to non-visual users.

Moreover, we will also explain our user-based studies aimed to understand the common programming challenges faced by non-visual users, mainly code navigation difficulties. Specifically, Part I of this dissertation discusses each one of the following research questions:

RQ1: In a survey-based study, what are the programming challenges that visually impaired programmers report facing, as well as workarounds or strategies to overcome these issues? (We examine **RQ1** in Chapter 3)

RQ2: In an interview-based study, what are the code navigation difficulties that non-visual programmers report facing, as well as workarounds or

strategies to overcome these issues? (We examine **RQ2** in Chapter 4)

Chapter 2

Background and Prior Work on Programming Challenges

Globally, the number of students entering the Computer Science discipline has increased over the past 10 years [35]; however, people with disabilities remain underrepresented in computing [62]. Students who are blind must overcome significant educational and technological barriers [8, 10, 28, 42, 72], including the heavy use of images and visual abstractions in classrooms; prior researchers have examined how the traditional curriculum in Computer Science has not been designed with assistive technologies in mind [46, 54, 75, 81, 82, 83]. While there has been significant prior research on investigating particular design interventions to benefit blind programmers, e.g. audio cues (Chapter 4), navigation aids (Section 2.1), there have been relatively few studies that

have explored the challenges faced by blind programmers more broadly.

2.1 Code Navigation

Several prior researchers, e.g. [33, 38, 66, 74, 76], have proposed interventions to help enhance code navigation for sighted and blind developers. By “code navigation,” we refer to the ability of blind programmers to understand lengthy codebases better and how each code statement is nested within the code, which results in enabling blind programmers to navigate code quicker [9].

Baker et al. [9] created an Eclipse plug-in called StructJumper that aimed to help screen reader users navigate through a large amount of code quickly. The tool was designed to create a hierarchical tree representation based on the codebase, which presents hierarchical tree-based information about the nesting structure of a Java class. In their tool, blind programmers used a `TreeView` feature to get an overview of the code structure. In addition, they could use a `Text Editor` feature to get an idea of where they are within the nested structure of the code. Thus, blind programmers could look up contextual information about their code without having to lose their position. For example, with the use of shortcut keys, blind developers could press a defined key to find which statement of the code he or she is working on. Such a technique allows a blind developer to quickly jump to the node corresponding to the current location. This approach was similar to that used by other researchers to recognize code in order to present a tree-like structure in a hierarchical tree representation [77].

Other researchers have examined technology interventions to improve code understanding for sighted developers: For example, researchers in [21] created a system where code is presented in “bubbles,” which are editable views of, e.g., specific methods or collections of variables; each bubble is in a different color (Figure 7.2). Of course, the heavy use of visual abstractions is not suitable for blind programmers; further study would be needed to determine whether this bubble metaphor could benefit non-visual users.

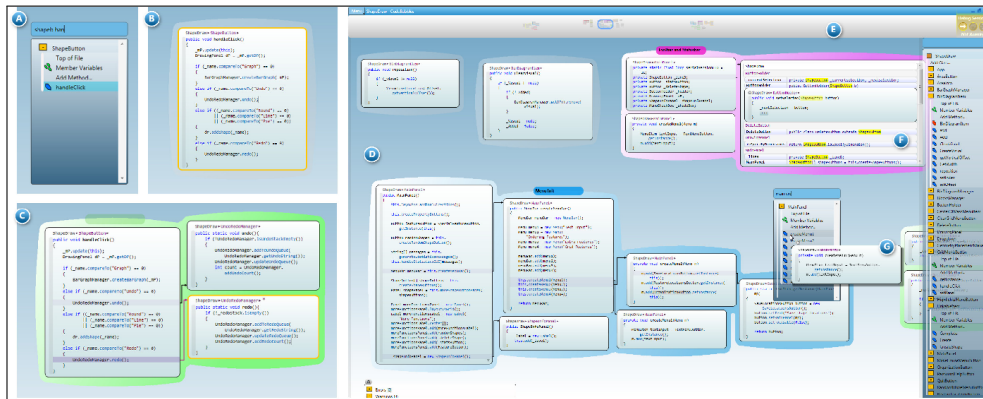


Figure 2.1: This figure shows an overview of the code bubble metaphor: (a) user opens a bubble through the search box, (b) bubble displayed, (c) users opens two or more bubbles side-by-side, (d) large set of bubbles with a (f) bubble references, (e) an overview is displayed in the panning bar, (g) hover preview less. [21]

2.2 Programming Challenges

While a variety of studies have been published focused on the design and evaluation of specific technology interventions to benefit blind programmers,

there have been relatively fewer empirical studies to explore and identify programming challenges. For example, Mealin and Murphy-Hill interviewed 8 participants [58], and Smith et al. [77] conducted an experiment with 12 participants to evaluate a code navigation plug-in. We discuss the Mealin and Murphy-Hill prior work below:

Mealin and Murphy-Hill conducted an interview study with eight experienced blind developers to highlight their programming difficulties [58], and they identified a number of challenges: First, they noticed that developers were not using the tools available within the IDEs. It was unclear from their study whether users were unaware of the tools offered within these IDEs, found the tools to be too complex, or if the tools were not easily accessible. Second, they found that many blind developers were using a temporary text buffer to store programming notes and to work in it. During the interviews, participants also mentioned challenges with debugging, inaccessible UML diagrams, code navigation, complexity of IDEs, and working in teams with sighted programmers. The authors discussed how blind developers use workarounds or other strategies to overcome the above-mentioned challenges.

Chapter 3

Programming Challenges

3.1 Introduction

In this chapter ¹, we present our initial user-based study aimed to understand the major challenges in software development faced by programmers who are blind. We performed this study in order to reveal the current programming challenges, workarounds or strategies (e.g., to overcome programming issues), and user needs (requested features) to enhance accessibility in current IDEs. We also performed this study to better characterize research problem for this dissertation research.

¹The work presented in this chapter is based on study published at the 9th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'16) with co-author Stephanie Ludi. For this study, this author was responsible for designing the study, conducting it, analyzing it, as well as serving as first author for the published paper, having authored the initial draft of the paper.

3.2 Research Questions

In this chapter, we intend to answer the first research question in our dissertation work, which has four sub-questions described below:

RQ1: In a survey-based study, what are the programming challenges that visually impaired programmers report facing, as well as workarounds or strategies to overcome these issues?

RQ1:.a What are the popular IDEs and programming languages that blind developers use?

RQ1:.b What assistive tools do blind developers use when programming?

RQ1:.c What are the difficulties faced by blind developers when developing code?

RQ1:.d How do blind developers use workarounds to solve programming challenges?

3.3 Methodology

In this section, we explain our methodological approach which was used in efforts to answer the previously mentioned research questions.

3.3.1 Survey Design

We created an initial survey and conducted a pilot test in order to validate our questionnaire. Our pilot testers were experienced developers (5+ years) with

no visual impairments. The survey design was modified upon their comments and feedback. For example, the survey wording was changed to more familiar terms as well as adding other common programming languages to multiple questions.

We estimated the completion time of the survey at 10 minutes. There were 15 questions in our survey, 11 multiple choice, 3 open-ended, and 1 Likert Scale (see Appendix B for the complete list of survey questions). We defined the survey questions to inquire about how programming was learned by the participants, their level of experience, visual acuity, visual perception, challenges, workarounds, assistive tools, and the type of development tools the participants used.

3.3.2 Sampling

In this work, we needed to find developers with visual impairments. As this is a subgroup of a limited and geographically dispersed population, we decided to use a snowball sampling technique. We decided to contact individuals who met the criteria and asked them to forward the survey to those who possess the necessary traits. The potential respondents were contacted via email invitations as well as posts in online groups of blind individuals on Google Hangouts, LinkedIn, and AppleVis (a community for blind and low-vision users of Apple's products).

3.3.3 Procedure and Response Rate

In order to eliminate geographic restrictions, we decided to set up an online survey through the Rochester Institute of Technology survey system. The system was designed to be accessible to screen-reader users. Participant response rates could not be calculated as we could only monitor the total number of responses submitted. The actual time for the survey completion could not be measured. The survey was open for more than two months.

3.3.4 Participants

The survey was taken by a total number of 69 participants, all of whom were blind developers. Nearly all 62 (89.86%) of the participants were male, 6 (0.09%) were female, and 1 participant decided not to answer. The mean age in our sample was 35.39 years, with a standard deviation of 13.55 years. The lowest age captured in our sample was 18 where the highest age of an individual was 68. Our survey sample showed variation in the visual acuity among the 69 respondents. About 29 (42.02%) of the participants were totally blind, followed by 25 (36.23%) who had light and shadow sensitivity, 12 had vision but needed corrective lenses (17.39%), 2 had macular degeneration (0.03%), and 1 was totally blind in one eye (0.01%) (see Figure 3.1). In regards to the visual perception of the 69 respondents, 43 had light perception, 26 had shadow perception, 22 had movement perception, and 16 had color perception.

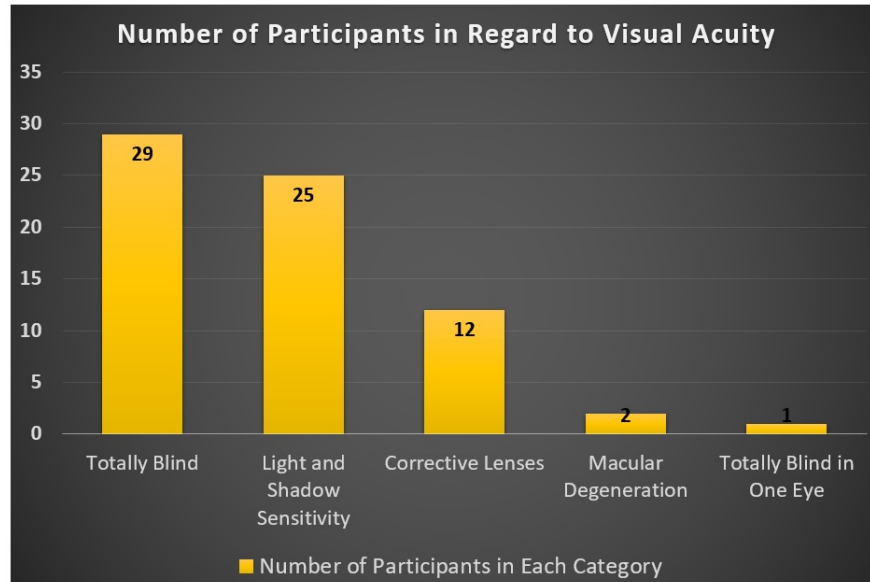


Figure 3.1: The number of participants in regard to visual acuity.

3.4 Results

The results section has been organized by the developers' background, tools used, assistive technology, and the challenges faced in software development.

3.4.1 RQ1a: Developer Background

Participants were asked to clarify the method used to learn programming. About 40 (57.97%) were self-taught, 28 (40.58%) attended schools, and 1 (0.01%) respondent did not answer the question.

We also asked participants to rate their levels of expertise in various programming languages. Table 3.1 shows respondents' experiences in various programming languages. The use of Python was expected as many undergraduate

computer science programs use Python. It also has gained wider popularity among many STEM disciplines. The use of Python itself is interesting given that blind developers can dynamically inspect and change their programs, since it is an interpreted language.

3.4.2 RQ1a: Development Tools & Platforms

We asked participants to indicate their development tools, development platforms, and the target platforms for their work. About 49 (71.01%) use the Windows environment to write code, where 15 (21.74%) use Mac OS X, and 14 (20.29%) use Linux. Less common environments included IBM Mainframe, Motorola, micro-controllers embedded C, and Unix. In regards to the target platforms, 39 (56.52%) of the respondents developed applications that run on Microsoft. 13 (18.84%) people developed applications for Linux, 10 (14.49%) for iOS, 7 (10.14%) for Mac OSX, and 5 (0.07%) for Android. In terms of the development tools used, the most preferred editor is Eclipse (31 people, or 44.92%), followed by Microsoft Visual Studio (28 people, or 40.58%), Xcode (17 people, or 24.64%), Emacs (3 people, or 0.04%), and Netbeans (2 people, or 0.02%) (see figure 3.2). Eclipse was expected due to its common adoption in undergraduate Computer Science programs, as well as it being open source.

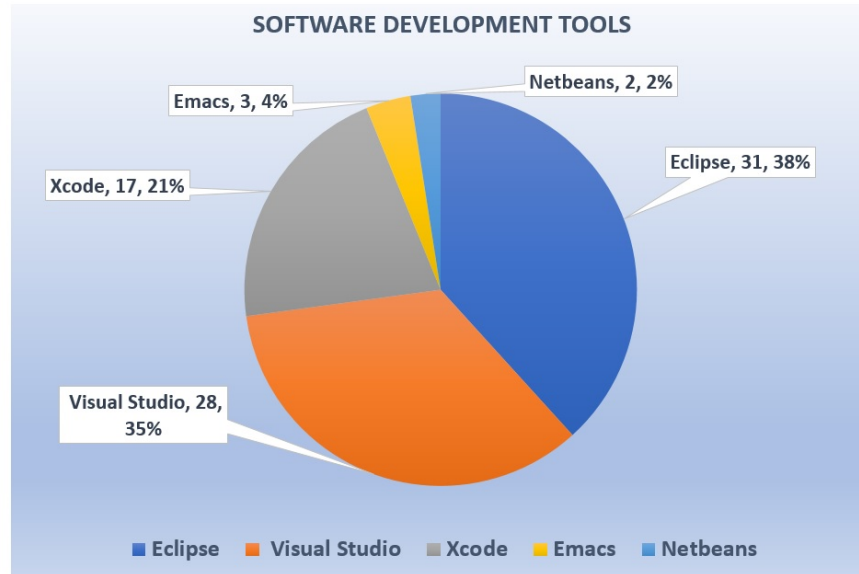


Figure 3.2: Software Development Tools Used by Participants in This Study.

3.4.3 RQ1b: Assistive Technology

The use of software and hardware-based assistive technology is integral to programming and related tasks. The use of a screen reader (e.g. VoiceOver, JAWS) is very common among participants, whereas several of the blind developers prefer to use refreshable Braille display when programming. A refreshable Braille display, a hardware device, translates a single line of text that is displayed on screen to a single line of Braille that can be read by touch (see Figure 3.3). Braille displays are very expensive and require the user to know Braille, which can be a limitation to blind developers.



Figure 3.3: In this figure, we show an example of Braille Display device used by users with visual disabilities to access information on the computer display.

Source: <http://www.hims-inc.com/product/braille-edge-40/>

In regards to the types of aids that respondents use for assistance with programming, all of the respondents indicated that they do utilize Screen Reader (69 people, or 100%). Braille Display is used by 30 people (43.48%), magnification software is used by 7 people (10.14%), and large fonts used by 6 people (8.70%).

Level of Expertise	List of Programming Languages									
	<i>Java</i>	<i>C</i>	<i>C#</i>	<i>C++</i>	<i>Objective-c</i>	<i>Python</i>	<i>Ruby</i>	<i>Perl</i>	<i>JavaScript</i>	<i>Php</i>
<i>None</i>	22	16	30	22	42	22	44	38	20	26
<i>Novice</i>	16	15	16	16	11	18	11	14	17	13
<i>Intermediate</i>	17	18	8	16	4	16	3	5	22	15
<i>Expert</i>	12	14	9	9	5	9	3	5	7	10

Table 3.1: Level of expertise in various programming languages

3.4.4 RQ1c: Open-Ended Responses

The survey contained three open-ended questions designed to elicit responses regarding the use of text-editors, challenges, and workarounds. We followed a qualitative methodology for our data analysis. Following an open-coding method [67], we analyzed open-ended questions based on their content using a set of codes that we developed to represent recurring ideas or problems raised by participants. We assigned codes to segments of text transcription or experimenter notes in our dataset.

Two researchers performed coding independently, reading and organizing the participants' transcripts. Afterwards, they met periodically to discuss code categories (e.g., limited accessibility aids in IDEs, code navigation, diagrams, debugging and user interface layout, seeking sighted assistance, and workaround techniques). In rare cases when coders disagreed, they held a meeting to reach an agreement and form a consensus coding. We generated a set of themes based on the number of times each issue was raised. For example, high occurrences indicate higher demand or importance. Themes were developed using affinity diagramming [15], which is a useful technique for organizing and analyzing large-scale qualitative data.

3.4.4.1 Limited Accessibility Aids in IDEs

Many participants (n=12) reported that accessibility in IDEs is poor and limited. Participants P16 and P53 indicated that the use of a text-editor is

necessary since IDEs are very complex environments. The following are some responses from participants:

“Using a text editor is completely necessary because accessibility for IDE’s is so poor.” (Participant 16)

“Accessibility issues in IDEs like visual studio.” (Participant 53)

While participant P5 indicated that certain parts of the IDEs are difficult to use due to unstable screen reader:

“Stability issues with the IDE’s and the screen readers. Certain parts of the IDE’s being more difficult to use than my sighted counterparts have to deal with.” (Participant 5)

It was clear that existing IDEs contain accessibility issues that prevent non-visual programmers from developing software comfortably. This is because the software industry designed their IDEs with visual representation, not knowing the existent of non-visual programmers and that their screen-reader technologies are only capable of conveying textual information to them.

3.4.4.2 Code Navigation

Writing code requires moving or navigating through it in order to revise it or to track down mistakes. Writing code requires moving or navigating through it in order to revise it or to track down errors. However, non-visual programmers rely mostly on arrow keys to move between code entities, such a technique is not beneficial due to the layout and the structure of the programming codebase.

Because of these difficulties, there is a need for an effective solution to overcome code navigation barriers for non-visual programmers.

In this section, we present testimony from our participants about different techniques to navigate through comfortably. For example, several participants (n=7) tried to overcome code navigation by using a scratchpad or simple editors. Here are some comments from these participants:

“I have another window open to serve as a scratchpad (notes to fix things, method/variable names, etc). Having that separate scratchpad allows me to avoid losing my place in the code if I need to go look something up.”
(Participant 1)

“Some text editors allows you to jump between the start and end of the block you are currently in.” (Participant 44)

Participant P36 uses a screen reader to listen to the code and Braille Display for more detailed information:

“I find that I listen to code with the screen reader audio, then if I want more detail, including punctuation, I use the Braille display.” (Participant 36)

As can be seen from these responses, participants vary in their techniques to navigate through software codebase. Some participants prefer using features in current IDEs where they can jump between code entities in order to avoid reading the entire code line-by-line. On the other hand, other participants enjoyed using braille display in conjunction with their screen reader in order

to get additional information that the screen-reader could not provide, which indicates that assistive technologies barriers, could be overcome by trying an alternative tool. Therefore, we believe detailed follow-up is needed to better understand how navigation occurs in different languages, environments, and with various skill levels.

3.4.4.3 Diagrams

Software developers need to be able to access various diagrams during the development process. Providing textual descriptions for diagrams in a timely manner is challenging. Several participants (n=5) discussed the problem of accessing UML diagrams and the need for UML assistive tools. Some of their comments include:

“It isn’t easy to diagram, I have to keep things in my head when I’m designing program flow.” (Participant 1)

Where other participant reported that diagrams does help show how certain things work before coding.

“It’s not possible to look at class diagrams to have a quick idea of how some stuff you did not code works.” (Participant 7)

As can be seen from participants response, this dependency on visual consents is not only affecting blind programmers in professional settings, but also in maintaining basic knowable about other people codebase, which clearly

indicates that visually structured information is difficult to convey by using current assistive technologies.

3.4.4.4 Debugging & User Interface Layout

Features in many IDEs include the support for debugging and also user interface layout. Respondents difficulties accomplishing both debugging and UI layout. Developers indicated the use of basic debugger utilities such as breakpoints, stepping through code, and print-f². Participants (n=8) also said that debugging tools are difficult to use. A sample of comments includes:

“The challenges I face more often concern interacting with errors and warnings and consulting documentation or tool tips.” (Participant 1)

“Debugging and interface design need visual development tools and they are not accessible and compatible with screen readers.” (Participant 12)

Although participants vary in their own debugging experiences, code debugging is considered a significant barrier to non-visual programmers, mostly because it is difficult to interpret software control flow while debugging.

3.4.4.5 Seeking Sighted Assistance

Many respondents (n=20) indicated the need to seek out help from sighted developers for certain tasks. Several respondents feel embarrassed when working with other sighted teammates. For example, participants P10 and P49

²This refers to a simplistic debugging technique of inserting print statements into code to see if lines are executed or to print values of variables.

rely heavily on the assistance of a sighted person to help them overcome some programming issues.

“Asking a sighted colleague for assistance.” (Participant 10)

“Many times I use the assistance of a sighted person.” (Participant 49)

3.4.4.6 RQ1d: Workaround Techniques

Because we did not want to miss information from participants about what was working well, in addition to asking about challenges, we also asked about successful strategies or workarounds. Many respondents (n=22) presented a myriad of workarounds for diverse development tasks. For example, P7 found an alternative way to access UML diagrams. However, they did not provide detailed information on the approach used. Other comments include:

“I have found alternative ways to access UML. A blind person to perform a software engineering job must know their access tech in side out.”
(Participant 7)

Participant P16 uses a text editor to overcome the complexity of IDEs.

“I have met the challenges by using a text editor to write code, attempting to run the code, and continuing to edit and revise until I achieve the result I want.” (Participant 16)

From these responses, we can see that participants were able to use alternative tools (workarounds) in order to write software code comfortably, which promote some unique ideas to enhance accessibility in modern IDEs.

3.5 Limitations

Although our study identified key accessibility issue for non-visual programmers, we recognized that our study had some unavoidable limitations. The study is limited by the snowball sampling technique, which resulted in uneven participant categories, e.g., participants vary in their visual acuity, type of assistive technologies, as well as their programming experiences. The technique was used in order to maximize the number of responses in the time allotted from a population that is challenging to recruit. In addition, the survey design of this study did not enable us to ask follow-up questions or observe the users while working on software code. For this reason, Chapter 3: Chapter 4 presents a follow-up interview and observation study.

3.6 Conclusions

In this chapter, we explored the major programming challenges faced by non-visual programmers. Our goal was to uncover some of the common programming challenges encountered by this user group, in addition, to understand their strategies or workarounds to overcome these programming challenges. In this work, some of the results were expected such as the lack of accessibility in IDEs as well as the use of a screen reader in programming environments. We were surprised to see those non-visual programmers use simple text editors as their preferred tools to write software codebase. It is not clear whether

non-visual programmers are unaware of the variety of features offered within the existing IDEs or find them difficult to use. For example, some participants indicated that Eclipse or Visual Studio was not accessible while other respondents did use these tools.

We have discussed several implications, but further investigation is needed to determine what our conclusions can be generalized. For example, the survey analysis indicated the difficulty of code navigation where non-visual programmers find it hard to navigate quickly through large amounts of code. The survey questions did not elicit sufficient details about our users' challenges for us to generate some user requirements. A further study is needed to illuminate this particular subject, and we should recruit users with more uniform programming experience to obtain more coherent results. In the next Chapter, we seek to conduct an observational and interview studies with blind developers in a remote setup (using Skype or Google Hangouts). Thus, the geographical distribution can be overcome while providing a representational sample of computer science students as well as professional software developers for the needed subject.

Chapter 4

Code Navigation Difficulties

4.1 Introduction

In this chapter ¹, we discuss our follow-up study aimed to understand the code navigation challenges faced by non-visual programmers. Our prior study (discussed in Chapter 3) explored a broad set of issues, and participants listed navigating through the code and understanding its structure as key concerns [4]. To address this issue, we conducted a observation and interview-based study to specifically investigate non-visual programmers navigate through large amounts of code, using their own preferred development tools while performing some of their common programming activities.

¹The work presented in this chapter is based on study published at the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS'17) with co-authors Stephanie Ludi and Matt Huenerfauth. For this study, this author was responsible for designing the study, conducting it, analyzing it, as well as serving as first author for the published paper, having authored the initial draft of the paper.

This chapter is structured as follows: Section 4.2 outlines our specific research questions to investigate how blind programmers navigate through a lengthy codebase, using their own preferred development tools and while performing common programming activities. Section 4.3 provides an overview of the methodology used in this paper to investigate the outlined research questions in Section 4.2. Section 4.4 explains our interview and observation results, and Section 4.7 summarizes our conclusions and future research directions.

4.2 Research Questions

In this chapter, we intend to answer the second research question in our dissertation work, which has three sub-questions described below:

RQ2: In an interview-based study, what are the code navigation difficulties that non-visual programmers report facing, as well as workarounds or strategies to overcome these issues?

RQ2:.a What difficulties do blind developers encounter when navigating through a codebase?

RQ2:.b What tools do they use in their development work?

RQ2:.c What workarounds or strategies do they use to overcome any code navigation barriers?

To preview for the reader, the study had three key findings:

1. Programming software (e.g., IDEs) did not meet participants' needs for code navigation; they regularly struggled when performing typical programming activities with these tools. Nevertheless, participants still preferred to use IDEs, even though they encountered these navigation difficulties.
2. Assistive technologies and specific accessibility features of some IDEs did not provide adequate support to enable users to navigate through code comfortably. Although some users were able to customize their assistive technology to better convey the information displayed by the IDE and trigger specific commands, the inefficiency of code navigation made participants feel a loss of control, and they often reported disorientation in the code.
3. Participants felt uncomfortable disclosing their programming needs (e.g., navigation difficulties) and their disability status to colleagues or researchers, which may prevent them from understanding the need to improve the accessibility of IDEs.

4.3 Methodology

As methodological inspiration, we have drawn upon the recent work of Szpiro et al., who conducted a study using contextual inquiry and qualitative data analysis to understand the challenges faced by people with low vision when

accessing computing devices [84]. Their goal was to uncover challenges and identify opportunities for researchers and industry to improve low vision accessibility tools, and we have similar aims in regard to code navigation for blind programmers. In this work, we used observations and semi-structured interviews with blind programmers to identify code navigation difficulties, the tools they used, and any workarounds they employ. This methodology will help us gain deeper insight, relative to our prior survey-based study in [4]. In addition to identifying future research opportunities, another goal was to involve blind programmers in the research and to gather firsthand comments and suggestions from these users.

4.3.1 Interview Design

Prior to the main study, we conducted pilot tests (mock interviews with five sighted programmers) to ensure that our semi-structured question plan, interview technique, and procedure were well-formed. As a result of these pilot tests, the interview questions and procedure were modified, e.g. the wording of some of the questions were changed to use terminology more familiar to this user group (see Appendix C for the complete list of interview questions).

The planned questions included five multiple-choice and 16 open-ended questions, which were grouped into several topics:

- **Demographics:** user characteristics such as age, gender, country, visual acuity, and level of expertise in software development.

- **Languages & Tools:** to identify a list of programming languages and programming tools (e.g., Eclipse, etc.).
- **Assistive Technologies:** to identify participants' preferences of assistive technologies such as screen readers or braille displays.
- **Development Style:** to capture and observe blind programmers' strategies when developing software, mainly to navigate code.
- **Navigation Difficulties:** to uncover navigation difficulties and how it impacted blind developers' performance.
- **Navigation Tools:** to investigate existing code navigation tools and how it helped overcome navigation difficulties.
- **Working in Teams:** to understand how blind programmers work in teams, mainly with sighted programmers.

4.3.2 Participants

We recruited participants using a private mailing list from previous studies (individuals had agreed to join this mailing list and had previously indicated an interest in participating in studies) and by posting advertisements on private groups (Google, LinkedIn, and AppleVis) for people who are blind. A total of 36 people responded. We conducted an initial screening interview over Skype and Google Hangouts to first determine the eligibility of the participants. To participate, individuals had to be an experienced developer (5+ years in

programming), 18 years or older, self-identified as fully blind, actively engaged in programming either as a job or a hobby, and a user of assistive technologies (e.g., screen reader or braille display). Eight respondents were excluded from the study due to the use of magnifiers and corrective lenses. (The focus of this study was on users of screen readers or braille displays.)

Afterward, we conducted the interview sessions with the remaining 28 blind programmers. Participants (all male) varied in age from 22 to 52 (mean = 29.68, SD = 6.59). Our sample showed variation in programming experience (lowest = 5 years, highest = 24 years) and employment status (e.g., retired, employed, unemployed, freelancer). Few unemployed participants are searching for job opportunities. All participants use screen readers, and 8 participants used braille displays (see Table 2). Participants were from five different countries: United States ($n = 22$), United Kingdom ($n = 3$), Australia ($n = 1$), India ($n = 1$), and the Netherlands ($n = 1$).

4.3.3 Procedure

The interview took place online via Skype and Google Hangouts per the participants' preference. Prior to the interview, participants were provided with informed consent documents and the interview questions (so they could familiarize themselves with the interview topic in advance). Each interview lasted approximately one hour and occurred during January to December 2016. Enrollment had not been fixed; rather, recruitment was discontinued (at 28)

after the researcher observed that no new issues were raised during the sessions conducted with participants 27 and 28. All participants were compensated for their time with \$50 Amazon gift cards.

The session began with the brief semi-structured interview (questions in section 4.1). This was followed by an observation period when we asked participants to engage in common programming activities. The participant transmitted their voice and their computer's audio output, and in addition, they transmitted the video image of what was displayed on their computer screen. The interviews and observations were recorded with the participants' prior approval, using Screencast-omatic software.

Since our goal was to understand code navigation difficulties and observe how participants deal with these navigation problems, we identified in advance a set of programming activities, which we requested the participant to perform during the observation:

- Conducting a programming walk through using any language or tool: We asked the participant to open some code that they had been editing recently as part of their professional work and to explain the code, giving a demonstration of its structure.
- Demonstrating for the researcher some code navigation difficulties they encounter frequently.
- Navigation walk through of some other programmers' codebase with which the participant had no prior knowledge.

- Demonstrating any strategies or workarounds that the participant uses to overcome navigation difficulties.
- Demonstrating any solutions or tools and how they helped.

We observed participants perform the above-mentioned programming tasks, and we occasionally interrupted them with questions. For each question, we encouraged participants to speak freely and openly (explaining that their feedback was very valuable and this research might benefit other programmers in the future) so that we can elicit more detailed answers. We did not insist that participants use specific programming languages or development tools, mainly because participants owned various platforms and had their own preferences. In a few cases, some participants did not wish to perform one of the asks or were unable to do so, and we did not insist in those cases. Our priority during the session was to elicit comments and impressions from the participants about code navigation difficulties that they encountered when performing these tasks, to capture information about: what assistive technologies that they use and why, how they used them, how they completed these activities, and how they felt when performing it.

4.3.4 Data Analysis

During the session, we captured the following data:

- Responses to closed-ended questions were recorded.

- Additional written notes were taken for open-ended responses, with particular focus on capturing direct quotations.
- Time-stamps were noted when an important issue was raised, to facilitate the researcher reviewing key portions of recordings.
- All notes were stored and duplicated for further analysis.

We followed a qualitative methodology for our data analysis. The data was managed and annotated using NVivo qualitative data analysis software. Following an open-coding method [67], we analyzed open-ended questions based on their content using a set of codes that we developed to represent recurring ideas or problems raised by participants. We assigned codes to segments of text transcription or experimenter notes in our dataset.

Two researchers performed coding independently, reading and organizing the participants' transcripts. Afterwards, they met periodically to discuss code categories (e.g., navigation challenges, assistive technologies, programming tools, workarounds, and user needs). In rare cases when coders disagreed (inter-rater reliability = 67%), they held a meeting to reach an agreement and form a consensus coding. We generated a set of themes based on the number of times each issue was raised. For example, high occurrences indicate higher demand or importance. Themes were developed using affinity diagramming [15], which is a useful technique for organizing and analyzing large-scale qualitative data.

4.4 Results

In this section, we describe key findings, illustrated with examples of our participants' behavior or comments from the interview sessions. Quotations are labeled with code numbers preceded by the letter P that represent individual participants (e.g., P1, P2, etc.). This section is organized based on the major themes that arose during our data analysis: code navigation challenges (Section 5.1), tools (e.g., assistive technologies, programming languages and tools) (Section 5.2), and strategies to overcome navigation difficulties (Section 5.3).

4.4.1 RQ2a: Code Navigation Challenges

In software development, programmers regularly use their sight to obtain information about their software codebase, which allows them to formulate an understanding of their code structure and navigate throughout the code. Blind programmers rely on other senses (e.g., hearing and touch) to acquire contextual and structural information about their software codebase. We observed our participants encountering several code navigation difficulties when performing various programming activities, and participants discussed this issue in their interview responses. We summarize a taxonomy of sub-types of navigation difficulties in table 4.1; next to each description, we provide the number of participants who mentioned each issue. The remainder of this section will summarize some key points, along with illustrative examples and quotations from participants.

<i>Navigation Difficulties</i>	<i>No. of Participants</i>
Debugging: difficulty navigating through the code in the process of understanding a wrong output.	24
Line by Line: difficulty navigating through the code to locate specific information without having to go through the entire codebase linearly, line-by-line.	23
Indentation: unable to distinguish the level of white-space using a screen reader in indentation-based languages, e.g. Python.	22
Nesting: difficulty navigating through nested methods, loops, functions, or classes.	20
BackTrack: difficulty returning quickly to a specific line (in a lengthy codebase) when reviewing other code statements in various files.	18
Errors: difficulty quickly locating code errors while navigating through lengthy codebases.	14
Scope: difficulty understanding the scope level, e.g. while navigating deeply nested methods or loops.	14
Characters: difficulty perceiving certain characters, operators, and parentheses, e.g. missing some characters while coding.	10
Auto-complete: difficulty accessing the auto-complete feature due to incompatibility with the screen reader.	9
Relationship: unable to distinguish the relationship between code entities within a codebase, e.g. the relationship between a class and its sub-classes.	9
Line Numbers: difficulty accessing line numbers in the code editor as they were not designed to be readable by a screen reader, e.g. using PyCharm with VoiceOver.	7
Elements: unable to quickly locate a specific element within a given array, class, function or loop, e.g. locating values or variables.	5

Table 4.1: List of navigation difficulties and number of participants who mentioned each during interviews; the difficulties are sorted based on this number.

- **Debugging:** When a failure occurs in software, programmers must perform three main activities to correct the failure. First, they need to perform fault localization to identify the code statement responsible for the software failure. Second, they need to complete a fault understanding activity that involves understanding the origin of the software failure. Third, they must perform a fault correction activity, to determine the best way to remove the cause of the software failure. All three of these activities are commonly referred to as “debugging,” which is an essential skill in software development [65].

Our participants indicated that they understood the importance of debugging and how it helps to correct unwanted software behaviors. However, participants indicated that they tend to rely on simple debugging techniques, mostly because of the accessibility issues in current debugging tools (e.g., FindBugs, Firebug, etc.). For example, P4 examined several available debugging tools to find one that is compatible with their screen reader. He found that most debugging tools were not accessible as they were designed with vision in mind. Therefore, P4 and many other participants ($n = 19$) decided to rely on simple debugging techniques such as inserting print commands in the code or tracing:

“I rely on printf to fix code defects. I also tried to test different tools like FindBugs or Firebug, but they were not fully accessible to [my] screen reader.” (P4)

P26, on the other hand, discussed the difficulty of navigating through a lengthy codebase to find logic errors. He explained that debugging techniques, such as `printf`, may take longer as there is no clear indication where to find the problem that caused the software to behave incorrectly. While most participants relied on simple debugging techniques, some ($n = 9$) used advanced debugging tools:

“I was trained to use advanced debugging tools by my sighted colleagues even with the accessibility issues. I think the training helped me use them better.” (P8)

Although participants vary in their own debugging experiences, most participants mentioned that debugging is a significant barrier to blind programmers, mostly because it is difficult to interpret software control flow while debugging.

- **Line by Line:** We have discussed previously how vision helps software developers get an overview of the entire codebase. To get an overview of code, most of our participants ($n = 18$) indicated that they tend to go through a codebase line by line, mainly because screen readers encourage users to move through text in a linear fashion. P5, for example, explained a difficulty that they encountered when working with complex codebase:

“How to accomplish things in my complex code [is] frustrating. I need more time to understand each line and more time to remember what each code block is doing.” (P5)

While several participants discussed the difficulty of navigating linearly with a screen reader, some ($n = 8$) used other techniques, e.g. searching through the codebase using keywords, to avoid scrolling through the entire codebase line by line. While P13 and others enjoy using keywords, another participant (P20) indicated that keyword searching is time-consuming and often frustrating, because the same keyword might appear in several locations within the same codebase. P6 and a few others, on the other hand, agreed that keywords are very popular among blind developers to find a specific code statement. But considering that some cases where the same keyword is used twice or even more, blind programmers often need to review a few code statements before and after the keyword location to ensure that they have found the right line:

“Keywords [are] useful when you deal with the small code, but not a large one, especially when you try to find a variable that [has] been used several times in different locations. Which code block I am reviewing is hard to distinguish with keywords.” (P6)

- **Indentation:** Indentation-based languages (e.g., Python, Occam, etc.) use white-space indentation to delimit code blocks, instead of using keywords or curly braces. In these languages, an increase in indentation may indicate a new, deeper code block, and a decrease in indentation indicates the end of the code block. Python was the most commonly used programming language among our participants ($n = 18$), mainly because

of their job requirements. To navigate through an indentation-based language, most participants indicated that they tend to go through it block by block instead of line by line, mainly to avoid the verbalization of white-spaces (indents) using a screen reader. By “block by block,” we refer to instances when blind programmers wish to skip-over code blocks (e.g., in a loop, a function definition) to avoid reading one code statement at a time while browsing the entire codebase. For example, P10 explained that a screen reader will verbalize an indentation as a sequence of individual “space” characters, rather than a single indent of a particular length. When a screen reader user navigates through indentation based languages, the blind programmer will hear his or her screen reader verbalizing white-spaces as a single space (e.g., “space, space, space”) rather than a count (“three spaces”).

P21 explained how to overcome the white-space problem using a screen reader. The solution involves writing a custom script (a modification of the typical functionality of a screen reader for a particular application) that forces the screen reader to calculate white-spaces and verbalize it as a complete list of white-spaces:

“I found it useful to write script that forces my screen reader to calculate the white-spaces and then present it [to me]. I designed the script to say, for example, ‘four spaces’ instead of saying ‘space’ four different times.” (P21)

P3 explained a similar approach:

“Instead of listening to my own screen reader telling me all the spaces separately. I wrote [a] script to give me the level of indentation in my code.” (P3)

Although calculating white-spaces and verbalizing it helped several participants ($n = 5$), others ($n = 4$) found a braille display much more helpful in determining the level of white-spaces. For instance, P16 reported that a braille display provides valuable assistance in determining the level of white-spaces, through touch. Section 5.2 discusses how users mitigate this white-space issue by using a braille display in conjunction with their screen reader:

“I use a screen reader and [also] braille display with Python, it helps [me] feel the indentation in my code.” (P16)

4.4.2 RQ2b: Tools in Software Development

In this section, we discuss the participants’ behavior or experiences towards assistive technologies, programming languages, as well as development tools. We also describe each method and technique used by participants to perform various development activities. We presented each category with the actual number of users based on the participants’ use of each language or tool.

4.4.2.1 Assistive Technologies

Assistive technologies refer to any specialty hardware or software add-ons that were designed to increase the functional capabilities of people with disabilities. These assistive tools, whether developed by the industry or privately customized by the end users, provide freedom and independence to people with special needs to accomplish tasks that are difficult without getting help from those who are sighted. In this study, participants used two different forms of assistive technologies; screen readers and braille displays. A screen reader enables blind users to access the computer display by linearizing the presentation of information from the graphical user interface and verbalizing this information using a speech synthesizer (or transmitting this information to a braille display).

Participants described a variety of experiences performing common programming activities using their screen reader. For example, P2 prefers to use the Non-Visual Desktop Access (NVDA) screen reader when working with a Python codebase:

“I use NVDA because its free, made by a blind user, and helps me convert text into [a] Braille Display.” (P2)

P10 uses NVDA for programming activities, mainly because it allows for personal customization. He uses PyCharm to write Python applications, despite challenges in using this tool with his screen reader. P10 indicated that PyCharm is very complex platform, and it poses many programming problems:

“I like to use PyCharm to write python application, I modified NVDA script to ignore unwanted features and to help [me] reduce its complexity.”
(P10)

Although many participants ($n = 12$) decided to use NVDA for personal reasons or financial constraints, others ($n = 16$) preferred to use a different type of screen reader (see table 4.2). For instance, P13 uses JAWS with development software, mainly because it allows users to load specific scripts (customized modifications of its behavior) for each platform:

“JAWS provide me with great functionality. You can assign specific script to each application, it helps reduce the time I take to navigate through the entire application.” (P13)

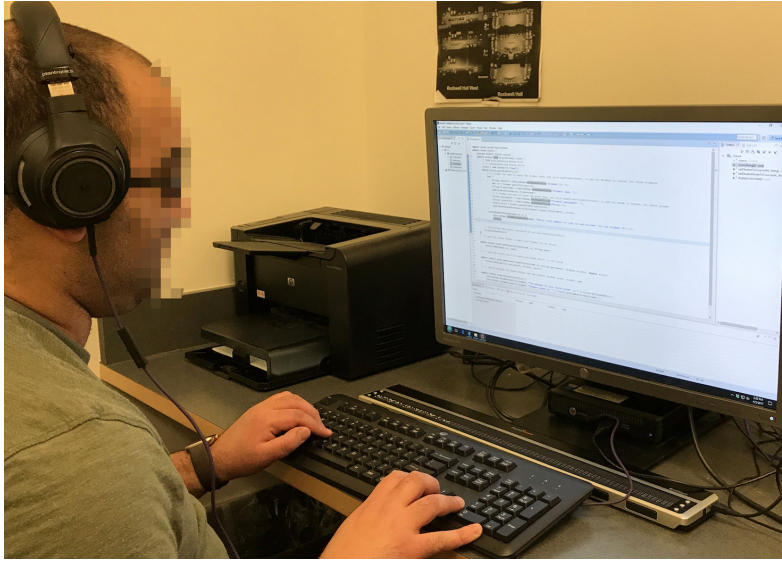


Figure 4.1: A participant using JAWS with an 80-cell Brilliant Braille Display, while programming in Java using the Eclipse Integrated Development Environment (IDE).

A refreshable braille display is an electro-mechanical device to translate information from the computer display into braille characters. It uses round-tipped pins in a flat surface that are raised through holes to convey information to blind users. These devices are available in different sizes (different number of characters that can be displayed in a line simultaneously, e.g., 18, 40, 80) based on the user's needs. In this paper, several participants ($n = 8$) indicated that they use a refreshable braille display with a screen reader to perform various programming activities (see figure 4.1). For example, P24 preferred to use a braille display when working with Python codebases, mainly to understand the level of indentation as its difficult to understand when using a screen reader alone:

“Braille display is much better than screen reader when it comes to detect-

ing indentation level. [The] screen reader will say 'space', 'space', 'space', etc. Which is too much to handle with complex code.” (P24)

Some participants ($n = 6$) explained that they preferred to use a refreshable braille display to navigate through a codebase because it was quicker than a screen reader. Others ($n = 2$) tend to use braille displays because it reduced their “hearing load,” i.e. the stress they experience from attending too much information conveyed on the audio channel in an interface. For example, P28 discussed how a screen reader creates significant hearing load when performing programming activities at work:

“I read texts and software code using braille display, [it] helps reduce [the] hearing load and makes me aware of the surrounding, especially in work settings.” (P28)

Other participants ($n = 2$) explained that they used a multi-line braille display. (Most braille displays present a single line of characters, but some are capable of presenting multiple rows of characters simultaneously.) Participants indicated that this device helped them to read several lines of code to get a better overview of the code structure, rather than using a screen reader or a singleline braille display, which presents information linearly:

“Navigating code [is] difficult with screen reader, you feel isolated to one line at a time, I use multi-line braille display which helps me read more than one line at a time.” (P16)

4.4.2.2 Development Languages & Tools

Our participants' knowledge and experience in programming languages and development tools varied. Some ($n = 15$) were proficient in more than one programming language, and others ($n = 13$) were experienced in a single language only. This variation was mainly due to their specific job requirements or constraints that are presented by the structure of the programming language.

Participants were asked to list the programming languages and tools that they use to develop software (see table 4.2). Our results showed that Python was the most used language among all participants. In fact, 18 participants ($n = 64\%$) indicated that they use Python to write software code for several reasons, including: its simplicity, its rising popularity, and the fact that it can be used as an interpreted language – thereby providing users with the ability to dynamically inspect and change their programming code. Although Python was the most used language among all participants, other participants ($n = 10$) preferred to use Java, again, mostly for job requirements. For example, P27 developed several applications that run on computers, smart cards, and cell phones for the company:

“I developed the company clients support application with other colleagues that was written in Java. [We] choose Java because [of] its well-written libraries. [We] use other languages as well, but mostly Java.” (P27)

In regard to development tools, all participants preferred to use simpler editors rather than current IDEs. Participants explained that simpler editors

(e.g., Notepad, Notepad++, Notepadqq, etc.) were popular due to their simplicity and flexibility with assistive technologies and programming languages. Notepad++, for example, was especially popular among users of the Windows operating system as it was available for free. P4 explained that his reason for using Notepad++ was due to its wide range of plugins, that helped facilitate writing software code. While some participants ($n = 7$) favored plug-in features to install tools that had previously been developed by the blind programming community, others ($n = 8$) find it useful to write their own plugins. For example, P15 worked with several blind programmers to develop a plug-in that allows screen reader users to navigate through auto-complete functionality, mainly to make it more accessible. Auto-complete is a common feature in most IDEs in which the system displays a pop-up menu of predictions of what the programmer is about to type next, based on the first few characters of the word they have typed. But this feature is not fully accessible to screen reader users, mainly because it appears on the screen as a pop-up which the screen reader does not recognize. Although most participants preferred to use simpler editors when performing various programming activities, all participants agreed that IDEs are necessary at times, despite accessibility problems.

4.4.3 RQ2c: Programming Strategies

In the midst of a discussion about navigation difficulties with our participants, it would have been easy for participants to forget to mention positive infor-

Operating Systems	#	Assistive Technology	#	Programming Languages	#	Programming Editors	#
Windows	23	NVDA	12	Python	18	Notepad++	18
Linux	8	JAWS	10	Java	10	PyCharm	16
Mac OS	4	ORCA	5	C++	10	Visual Studio	12
		VoiceOver	4	SQL	7	NetBeans	8
		LSR	3	C	6	Notepad	6
		Windows-Eyes	1	Swift	4	Notepadqq	5
		Braille Display	8	Ruby	3	Eclipse	4
				C#	2	Xcode	4
				Objective-C	2	CODA	4
				PHP	2	Atom	2
				Perl	1	IDLE	1
						TextMate	1
						Padre	1

Table 4.2: Number of participants in our study using various operating systems, assistive technologies, programming languages, and programming editors.

mation, such as navigation workarounds or strategies. For this reason, we specifically asked participants to demonstrate or explain some examples of these. Our participants discussed a myriad of strategies to overcome various programming challenges, mainly code navigation difficulties. Due to length constraints, this section will summarize some key points, along with illustrative examples and quotations from participants.

- **Simple Editors:** As discussed above most participants ($n = 26$) indicated that they rely on simple editors to write software code; we highlight here how several of our participants reported using simple editors in concert with IDEs – to overcome inaccessible features in existing IDEs. For example, several participants ($n = 8$) explained how they use simple

editors to record code errors, bugs status, and where variables located to enhance navigation. Other participants ($n = 7$) use them to avoid losing their current spot while reviewing other code statements. For example, P18 demonstrated how to use Notepad to navigate through a complex codebase that was written by other programmers:

“The code I am showing is large and long. I work with other programmers to maintain it and mostly to modify it. I use Notepad to record code errors while reviewing other statements for reference.”

(P18)

- **Custom Scripts:** Our participants expressed mixed feeling about the use of assistive technologies, mostly screen readers. In this work, most participants ($n = 19$) modified screen reader settings to match their own personal needs. Others ($n = 9$) wrote custom scripts to overcome many issues including programming difficulties. Participants explained that creating a custom script is not a perfect solution, yet it still provides an alternative method to solve some of the problems they experience when interacting with current IDEs. For example, P11 showed a script that was designed to force the screen reader to locate elements on the PyCharm (IDE) which was not fully accessible. P9, on the other hand, reported that his screen reader will not read line numbers on some of the IDEs, mainly because line numbers was not designed to be readable by a screen reader. Therefore, he wrote a custom script to force the screen

reader to read line numbers:

“I wrote many custom scripts to help do my job faster. My screen reader will not catch line numbers on some of the IDEs, so I coded [a] script to force my screen reader catch line numbers.” (P9)

- **Shortcut Keys:** as researchers, we were interested to know how blind programmers get a high-level overview of the entire codebase for navigation purposes. Screen readers navigate through codebase linearly, forcing the user to read the entire codebase one line at a time. To overcome this problem, several participants ($n = 9$) indicated that they use shortcut keys as a navigation strategy. For example, P4 relies on shortcut keys to locate specific code statements without scrolling through the entire codebase. Other participants ($n = 12$) use them to get structural information about their codebase. However, P1 argues that shortcut tools like find comment (to search for text strings) can help programmers find content in the codebase using keywords, but often a single keyword is not enough to jump through all the associated content (for e.g. in programming languages like Java and C++, jumping through all the functions in a code using a single search keyword can be ineffective as all related functions might not use those specific keywords).

However, P12 said that the use of shortcut keys was inefficient since it forces users to jump between code blocks, which is difficult for someone who is blind, especially for unfamiliar codebases:

“Depending on the language, the start of the block may not be easy to follow without reading through all lines. In cases like that, shortcut keys may not be [a] helpful strategy at all.” (P12)

P19, on the other hand, was annoyed that various IDEs make use of specific shortcut key combinations that are also used by his screen reader, leading to conflicting functions:

“I rely on shortcut keys to navigate through code, but there are overlapped keys between several applications. I had to write a custom script to control overlapped shortcut keys for me.” (P19)

- **Code Comments:** In software development, commenting involves placing different readable descriptions inside code blocks to detail the purpose of each block. Most blind developers rely on them to make code maintainable and debugging easier. Commenting is an important technique, especially when a project involves other programmers. In this work, most participants ($n = 16$) used commenting, not in the traditional manner (to make source code readable or document how a certain function works), but rather to overcome navigation barriers. For example, P3 used commenting to locate software bugs that need to be addressed immediately with other software programmers. Although some participants ($n = 6$) used commenting to locate code errors or bugs, others ($n = 9$) use it to highlight code statements that require further review:

“When modifying some of my code function, I use comments to locate them fast, especially while checking other statements so I can get back to them fast.” (P22)

- **Sighted Help:** Our participants indicated that seeking help from others, especially from those who are sighted, is avoided by many blind individuals in workplace settings, often due to embarrassment about the amount of time they take to accomplish certain tasks. Additionally, many participants indicated that they wanted to demonstrate that their visual loss had no impact on their ability to fulfill their job requirements. Although most participants ($n = 16$) tended to avoid seeking help from sighted co-workers, others ($n = 10$) found it necessary. For example, P25 seeks sighted help to get an overview of the entire codebase when a new implementation takes place. This helps reduce the amount of time a blind programmer takes to get an overview of the entire implementation. P7 agreed that requesting sighted help is understandable since blind programmers are unable to simply glance at codebase due to the linear nature of the screen reader:

“Reviewing another programmer’s code with a screen reader takes longer than someone who is not blind, I seek help sometimes to get [a] quick overview of the new implementation.” (P7)

P13 shares a similar opinion about the importance of requesting sighted help whenever needed:

“I enjoy working with sighted programmers, you always learn many tips.” (P13)

4.5 User Needs

As part of our interview, we also discussed with participants some possible future features that could be added to IDEs to improve their accessibility. In some cases, the participants requested features prior to being prompted. For all participants, we included a section in the interview in which we briefly described several possible future enhancements to IDEs – to gauge the interest our participants had in each option. Overall, 82% of our participants ($n = 23$) showed interest in using these various features (listed below), while 18% said that they might be willing to try them. The set of possible future enhancements to IDEs discussed during our interviews included the following:

- **Tree View:** Most participants expressed the need to have an alternative feature to navigate through codebase, mostly to avoid going through it line by line. For example, several participants suggested a hierarchical navigation feature in which codebase could be presented as a tree, mainly to hide code complexity. (This is in agreement with prior findings of Baker et al. [9].) In fact, 18 participants (64%) showed interest in using such a feature. Tree view (or tree list) is already available feature in some of the IDEs but is not fully accessible to screen readers:

“Going through code line by line is very difficult with [a] screen

reader, especially when you deal with complex software code. As blind programmers, we discuss many ideas about accessibility in programming. In fact, we thought to program [a] tool that presents the software code as tree instead of navigating through line code, which takes forever.” (P14)

Also, P27 showed the same interest:

“I would love to see a tool that shows code in a different way, not line by line.” (P27)

- **Auditory Feedback:** Several participants ($n = 7$) suggested that sounds should become a core integration component when interacting with programming activities, especially for blind programmers. For example, some participants ($n = 3$) indicated that sounds would help them monitor background processes in development tools while attending other tasks. One advantage is that auditory cues can help blind programmers split their attention between an immediate task and waiting for the result of some background process. Participants also suggested that sounds could be used to help provide additional information regarding syntax errors, invalid statements, and current location in code in order to reduce programming difficulties. (This is in agreement with findings of Vickers and Alty [86].) In fact, 19 participants were interested in using auditory feedback (68%), while 9 (32%) participants said that they would be willing to try it:

“It would be nice to have audio feedback when we make code mistakes. It will help locate errors while navigating through code or maybe highlight any syntax error.” (P8)

“The way how programming relies on visual representation is the major impact in almost all difficulties that we face as blind individuals. We need another way of programming, maybe with audio or something else as I can’t think of different way that could help us.” (P24)

- **Bookmarks or Tags:** Our participants described how they used comments to leave keywords at particular locations in their code, which they could then jump to more easily by using a search feature. Participants also reported that they tend to remove all of these comments before sharing their code with others, especially sighted people, mainly because they feel embarrassed. Several participants expressed the need to have a bookmark feature in which they could tag specific line of code and return to it later for further modification (without making use of comments and searching to accomplish this task). Participants cautioned that the bookmark feature should be designed to jump to a specific code statement, rather than to a specific line number (which may shift when additional code is inserted or deleted). Bookmark or Tag features are already available in Visual Studio and other IDEs, further investigation may indicate whether such tools are fully accessible and beneficial for

non-visual users. In fact, 24 participants (86%) showed interest in using bookmark feature, while 4 (14%) participants said that they might try it:

“I always wanted to build [a] tool that tags code for personal use. You could build it in [a] way that any line can be tagged either for private or public comments. You could also use shortcut keys to locate each tag to quickly find them. I guess I did not find the right time to develop it.” (P22)

- **Nesting & Scope Level:** Nested code is commonly used in software development where various programming logic structures are combined to one another (e.g., embedded within one another). Deeply nested code can pose challenges for blind users because it is harder to read. When nested code goes beyond three levels of indentation, it can be difficult to understand and navigate. To handle nested code, sighted programmers tend to use code folding in software editors. This feature allows them to collapse an entire code block (visually hide the full text of the code and replace it with a small visual placeholder instead), which allows programmers to have a better view of the surrounding code statements. Several of our participants also suggested that it would be valuable to have a scope and nesting level indicator feature. This would read aloud the current cursor location when a special shortcut key combination is pressed. We are not aware of any similar study or tool in this regard. In

fact, 19 (68%) of our participants were interested in having nesting and scope level indicators, while 9 (32%) participants said that they might try it:

“I find it difficult to know my location when working with nested code block. You can’t tell with a screen reader unless you read the entire block. I think a good solution is to have a tool that gives [me] the location and how deep I am within the nested code.” (P5)

- **Class Relationships:** In object-oriented programming, a class is used to describe one or more objects, mainly to serve as a template for creating various objects within a program. Each object is created from a single class – this one class could be used many times, mostly to instantiate multiple objects. It can be also used by software developers to isolate specific objects so that their internal variables or methods are not accessible from all parts of the program. This prevents the programmer from changing internal implementation details of some code, which might break other parts of the codebase. Programmers tend to use classes to help create more structured programs that can be easily modified. The inheritance relationships for classes can become complex, especially when there are multiple sub-classes that inherit all or some of the characteristics of the main class. To understand class relationships, sighted programmers often rely on diagrams (e.g., how components are interrelated). Diagrams can be difficult to understand by blind programmers. Our participants

expressed the need to have some method of conveying class relationship features, e.g. audio cues as they navigate through classes or sub-classes in order to provide an overview of classes in a codebase. We are not aware of any similar study or tool in this regard. In fact, 17 (61%) of our participants expressed interest in using a class relationship feature, while 11 (39%) participants said that they might use it:

“It would be interesting to have class relationship tools where you get instant feedback through audio. Maybe [by] pressing shortcut keys to get audio feedback whenever I need to know all the sub-classes of a class.” (P18)

4.6 Limitations

There were some limitations of our study: First, we only explored navigation difficulties encountered by experienced developers, who were totally blind, actively engaged in programming either as a job or hobby, and used assistive technologies to access the computer display (e.g., screen reader, braille display, or both). It was beyond our scope to study novice programmers or individuals with greater diversity in their visual acuity. A further investigation into such an important user group may reveal different findings. Secondly, while the qualitative design of this study allowed us to gather firsthand comments and experiences from our user group, and to discover new issues that arose, in future work, it may be important to follow up this study with a survey administered

to a larger group of participants, to verify some of our findings.

4.7 Conclusions

In this paper, we presented our exploratory study aimed at understanding code navigation challenges encountered by blind programmers when using various development tools. We illustrated and discussed our methodology for learning about code navigation difficulties from our participants: blind software developers. Our study offers a new perspective into the use of common development tools (e.g., Eclipse, NetBeans, etc.) alongside assistive technologies by developers who are blind. Most previous studies have based their findings on a small number of participants [9, 58, 77]. Our results arose from observing and interviewing a much larger sample, and our findings highlight various code navigation difficulties based on different programming languages and tools.

Our findings indicated that participants struggled to navigate through codebases using existing development software alongside assistive technologies (e.g., screen reader). Although accessibility tools provided benefits, they failed to give enough support for blind programmers to navigate through codebases quickly and comfortably. Since navigation options in IDEs are restricted to sighted users, blind programmers prefer simpler editors (e.g., Notepad, Notepad++). Participants explained and demonstrated how diverse programming environments, in combination with assistive technologies, lead to various challenges, often because these IDEs were designed without accessibility in

mind.

Most of our participants preferred to use a screen reader (despite its limitations) to write software code. Others found this difficult, and therefore, favored using a braille display instead. However, several of our participants indicated that they could not afford to purchase a braille display. While most IDEs were not fully accessible, blind programmers still rely on them to accomplish their work. Moreover, some blind programmers may seek sighted help for various reasons, mostly to access content that is not accessible with assistive technologies. Although some blind programmers seek sighted help, others prefer writing custom scripts to overcome many programming challenges. For example, several blind programmers wrote custom scripts to enhance navigation in indentation-based languages. Others wrote scripts for each IDEs,

The navigation challenges identified in this study illustrate the need for further research on improving the usability and accessibility of current IDEs. For example, participants showed interest in using a new forms of code navigation, e.g. using hierarchical navigation approaches. Participants also indicated a desire for bookmarks (or tags) features that would allow blind programmers to tag specific line of code and return to it later for further modification. They also expressed interest in scope and nesting level indicator, auditory additional feedback, and methods for conveying class relationships, which could make programming more accessible for these users.

Finally, while the participants in our study expressed interest in various

technology interventions to address their needs, it would be necessary in future work to conduct formal evaluations of the efficacy of such technology in studies with blind developers. In Part II, we explore some form of auditory feedback which could help convey important information while users are navigating through lengthy codebases. Several participants expressed interest in this technology. Participants also suggested that audio cues could be used in various other programming activities.

In summary, our findings indicated that navigating through hierarchical code (e.g., in python-based language) is a challenge for users with visual impairments. This is because Python-based language use white-space indentation to delimit code blocks, instead of using keywords or curly braces. In such a language, an increase in indentation may indicate a new, deeper code block, and a decrease in indentation indicates the end of the code block. To understand code indentation, users reported the use of custom scripts in order to calculate the white-space and verbalize it as a complete list of white-spaces. To overcome such a problem, participants suggested that audio cue should become a core integration component when interacting with programming activities, especially for determining the level of indentation. In this work, we provided future accessibility researchers a foundation for understanding the needs of blind programmers, which may support their work in creating and evaluating new technologies to address those needs.

Epilogue to Part I

In Part I, we have discussed some of the prior work related to software development challenges faced by non-visual programmers. Specifically, we have explained some of the existing programming barriers in both academia as well as the software industry, in addition to explaining some of the current design interventions for making programming environments more accessible to non-visual users. We also discussed our user-based studies aimed to understand the common programming challenges in software development, specifically code navigation difficulties. In summary, Part I of this dissertation has addressed the following research question:

RQ1: In a survey-based study, what are the programming challenges that visually impaired programmers report facing, as well as workarounds or strategies to overcome these issues?

- In this question, our goal was to uncover some of the major programming challenges encountered by visually impaired users, in

addition, to understand their strategies or workarounds to overcome these programming challenges. To answer this questions, we conducted a survey-based study (discussed previously in Chapter 3) with 69 blind programmers where we explored a set of programming challenges as well as workarounds to overcome these issues. In this work, some of the results were expected such as the lack of accessibility in current IDEs as well as the difficulty of using screen-reader technologies with today’s software development. To overcome programming barriers, participants reported the use of alternative tools to understand code structure as well as seeking help from sighted co-workers whenever needed. In addition to using two different assistive technology (screen-reader and braille display) at the same time to uncover hidden information.

RQ2: In an interview-based study, what are the code navigation difficulties that non-visual programmers report facing, as well as workarounds or strategies to overcome these issues?

- In this question, our goal was to understand code navigation challenges encountered by non-visual programmers, especially when navigating through lengthy codebase. To answer this question, we conducted an interview-based study (discussed previously in Chapter 4) with 28 blind programmers where a set of code navigation difficulties were presented and discussed. In this work, we found

that blind developers felt overwhelmed when using existing IDEs (e.g., Eclipse, NetBeans, etc.), and therefore they preferred to use simpler editors to write software code comfortably (e.g., Notepad, Notepad++, etc.). Furthermore, participants discussed a list of code navigation difficulties as well as possible accessibility improvements where additional features could be developed in order to make the programming environment more accessible to non-visual programmers. For example, users indicated that deeply nested code can pose challenges for blind users because it is harder to read e.g., depth of bracketing or level of indentation, as in nested loops. When the nested code goes beyond three levels of indentation, it can be difficult to understand and navigate. To overcome this problem, participants suggested that it would be valuable to have audio-based feedback where the level of indentation can be determined via sound.

PART II: METHODOLOGICAL
DETAILS OF SONIFICATION TO
AID CODE NAVIGATION FOR
NON-VISUAL PROGRAMMERS

Prologue to Part II

In Part II, we will begin by examining the most closely related prior work on audio-based techniques to increase the accessibility of programming for these users, to establish that little work has examined the issue of navigating the hierarchical structure of code and additional research is needed into how to convey indentation structure of individual lines of code in the context of the linear reading of code via screen-reader. To broaden our focus, we consider related research on using audio-based cues in settings that are analogous in some way, namely: conveying nesting structure in mathematical notation, conveying the relationships within graph structures, or representing navigation through nested menus or outlines. We explain some of the existing audio-based design interventions used in these research areas in order to improve accessibility for non-visual users. Furthermore, we explain our formative study where participants discussed their personal preferences of various audio-based cues and sound properties, mainly to address our **RQ3**. In addition, we discuss our experimental study where we evaluate and compare three audio-

based interaction techniques using three code samples, specifically to address **RQ4** and **RQ5**. Specifically, Part II of this dissertation discusses each one of the following three research questions:

- RQ3:** In a formative interview study with a variety of audio examples, what forms of audio generation techniques and parameters do non-visual programmers express interest in? (We examine **RQ3** in Chapter 6)
- RQ4:** When presented with an interactive audio prototype based on this prior formative study, do non-visual programmers prefer receiving this additional audio information about the structure of code, as compared to a control condition without such additional information? (We examine **RQ4** in Chapter 7)
- RQ5:** When interacting with an audio prototype based on this prior formative study, do non-visual programmers have a preference between automatic level-crossing notifications or on-demand level indications? (We examine **RQ5** in Chapter 7)

Chapter 5

Background and Prior Work on Audio Programming and Sonification

In this Chapter, we examine the most closely related prior work on audio-based techniques to increase the accessibility of programming for these users, to establish that little work has examined the issue of navigating the hierarchical structure of code and additional research is needed into how to convey indentation structure of individual lines of code in the context of the linear reading of code via screen-reader. To broaden our focus, we consider related research on using audio-based cues in settings that are analogous in some way, namely: conveying nesting structure in mathematical notation, conveying

the relationships within graph structures, or representing navigation through nested menus or outlines.

As discussed previously (Chapter 4), we explained out interview-based study where we presented a set of issues, and participants indicated navigating through a nested code and understanding its structure as key concerns. In addition, participants suggested possible areas of improvement where additional features could be developed in order to make the programming environment more accessible to blind users, e.g., using an audio-based system to convey certain information to the end user. To better understand this, we investigated different audio-based techniques (in Chapters 6 & 7) and whether they could convey the nested structure of code lines, e.g., depth of bracketing or level of indention, as in nested loops.

Specifically, Chapters 6 & 7 discuss different design dimensions in order to generate useful design guidelines for including an audio-based feedback system into the programming environment. Can summarize these design dimensions as follows:

- **What to convey:** In this dimension, we needed to understand the type of information that blind programmers would like to know once they are navigating through nested codebase. For example, do stakeholders prefer to get an overview (the nested code depth) of the nested code, or understand the current location (where they are inside the code), or when the nested code starts and ends, etc. This is an important step

towards understanding the type of information that non-visual programmers would like to know when they are presented with a nested code, e.g. loops.

- **Audio Feedback:** In this dimension, we needed to understand the type of audio (speech, non-speech, etc.) that blind programmers would like to hear in order to understand the nested structure of code lines. For example, do stakeholders prefer to hear an audio feedback based on speech, non-speech or stereo spatial audio when they are interacting with the programming environment?
- **Audio Parameters:** In this dimension, we needed to understand the type of audio parameters that blind programmers would like to adjust once they hear the audio feedback. For example, do stakeholders prefer to hear sound with higher or lower pitch, when receiving an audio feedback? This is an important step considering that non-visual users have different preferences when it comes to audio since they have different settings for their screen-reader technologies and audio output.
- **Audio Interaction:** In this dimension, we needed to know the type of audio interaction techniques that blind programmers prefer to use in order to understand the nested structure of code lines. For example, do stakeholders prefer to receive an audio feedback on-demand (per request), or when they move between code levels (automatic), or with no background sound (baseline condition)?

Understanding stakeholders' preferences in the above-mentioned design dimensions are essential for determining the best audio-based techniques for conveying the hierarchical nesting structure of code to assist non-visual programmers. To achieve our goal, we needed to examine different methods and techniques in several research venues since the related work in our domain is quite limited. Thus, in this Chapter, we discuss some of the prior work that benefits from using audio-based techniques to overcome their research challenges, specifically visually-based applications for non-visual users:

5.1 Sonification and Interaction in Programming

Significant prior research has examined how to create audio-based accessibility tools for computing students with visual disabilities or other professional software developers [69, 79]. For instance, Sanchez and Aguayo 2005 developed a custom programming tool called Audio Programming Language (APL) aimed to help blind students write software code comfortably [70]. In their tool, users use keyboard keys to input a programming command which are dynamically presented in a circular command list. As a result, users would hear the input programming command feedback via TTS (Text-to-Speech)¹ engine. For example, users would use the keyboard to enter a variable and they would hear in return the command input as a speech synthesizer. In their tool, researchers goal was to alleviate the need to commit commands to memory, thus, enabling

¹Text-to-speech is a form of speech synthesis that converts text into spoken voice output.

novice blind programmer to focus their attention on the design process itself. In addition, researchers demonstrated that audio feedback could be used to convey certain programming information to non-visual users. However, their tool provided a limited set of commands (e.g., input, output, cycle, condition, and variable) making it difficult to scale.

Smith et al. [77] developed an Eclipse² plug-in to help non-sighted users understand code structure, to speed navigation through a codebase. The authors used keyboard inputs and speech/sound outputs of the hierarchical structure of the codebase to convey certain information to non-visual users. In this tool, users hear sound output based on speech to indicate which node is parent (beginning of function or loop) or sibling (code statements inside the main function). In addition, users hear a continuous, background clicking sound, with high frequency to indicate the size of subtree (how code is nested). In their work, authors performed a usability test using hyperbolic browser method that employs a fisheye technique [51]. The fisheye technique refers to zooming-in on a single node in a hierarchy tree structure, with the details of the ancestors and descendants presented in reduced detail. Such an approach helped researchers identify strategies that sighted developers tend to use while moving through familiar and unfamiliar trees. Based on this, the authors defined a set of user requirements for an accessible tree navigation system.

Similarly, Stefik et al. [82] created a tool called Sodbeans based on NetBeans IDE for Java programming, to help convey certain information to students who

²Eclipse is a popular IDE for Java programming.

are blind. Their developed tool includes a custom virtual machine, compiler, as well as a debugger. The tool used audible cues [22] based on speech so that blind students can learn programming concepts. For example, a user may wish to execute a code line “ $a = a + 1$ ”; in this case, the Sodbeans debugger would say “a to 5” (or another value). Similarly, the user may wish to execute an if statement such as if $a < b$ then end; the Sodbeans debugger would say either “if true” or “if false”. The tool used audible cues (using the word repeat over for and while, or cycle) [22] so that blind students can learn programming concepts. These cues were designed to be browsed in a hierarchical tree manner, to support navigation. In addition, blind students have a rich set of programming environments and tools that they can use beyond the use of Java (e.g., Java, PHP, Ruby). The tool was evaluated based on the students’ ability to master the programming concepts.

Various researchers, e.g. [78], have examined the potential of auditory cues to benefit programmers, including the potential benefits of non-speech audio for blind programmers: For instance, Vickers and Alty [86] developed a debugger called “CAITLIN,” which uses musical auditory cues in order to represent execution of computer program. The “CAITLIN” system was designed to map points of interest within the program to musical events. For example, with the use of “CAITLIN” system, novice programmers could locate code bugs by hearing a specific musical auditory cues to each type. In their work, researchers found that such audio cues helped novice programmers locate bugs.

Specifically, musical cues were found to be useful for conveying information to programmers during a debugging process.

Boardman et al. [18] developed a tool called “LISTEN” to investigate the use of sounds when analyzing various program behaviors; their goal was to instrument computer programs so that different audible sounds were mapped to different behaviors during the program execution. The “LISTEN” tool is focused on making code-to-audio mappings, which is not on the design of specific applications, i.e. programming environments for users with visual disabilities. In their tool, a user begins by editing the source program (code file) as well as a specification file, which is an Auralization Specification file, contains commands written in LSL (Linden Scripting Language). These commands are designed to specify the mapping of program behavior to sound.

Stefik et al. [80] investigated the use of audio cues to convey lexical scoping relationships in software code; different cues were played when a change in scope was detected. In their work, participants were given instructions about the study, including the type of sounds they will be hearing during the experiment, which is a set of auditory cues (based on speech) in different orders and contexts. When a user hears a sound, he or she will need to interpret the cues as a whole. Researchers would then ask participants questions like: Does this auditory cue indicate a loop or a selection statement? Does this cue indicate a scoping relationship or the number of iterations in a loop? in order to determine users understandability.

5.2 Accessing Mathematical Symbols via Audio

Although relatively few researchers have investigated the use of audio-based techniques for conveying certain information in programming software, additional researchers [13, 43, 44, 60] discussed various solutions to overcome visual barriers in teaching mathematics to non-visual students. For example, some researchers used speech and non-speech sounds in order to convey mathematical symbols (depth of bracket), whereas other researchers used prosodic aspects of spoken language or spatialized cues to convey the structure of an equation. We discuss these techniques in the following sections:

5.2.1 Conveying Depth of Brackets in Equations

Murphy et al. [60] created a prototype that uses audio cues (speech and non-speech sounds altogether) to convey mathematical symbols. In their work, researchers adjusted some audio parameters (e.g., speed, pitch, volume, etc.) for each audio sample in an effort to convey different audio meanings. For example, researchers modified pitch to indicate a different level of brackets in a given equation, using the higher pitch for a deeper level. It is reasonable to consider such a technique with non-visual programmers to determine whether adjusting audio parameters could help provide certain information about the code structure, i.e. we could convey how “deep” some line of code is in the structure of a nested loop.

5.2.2 Mathematical Formulas

While Murphy et al. [60] used adjusted audio parameters to convey mathematical brackets, Enda Bates and Dónal Fitzpatrick [14] used prosodic aspects of spoken language, in conjunction with a set of spatialized Earcons (a hierarchical progression of variable tones) [31] and Spearcons (spoken directions, compressed and sped up) [88, 89] in order to disambiguate the structure of mathematical formula.

In their work, researchers found that spatial sounds could help reduce the mental effort required by the users since sounds produced from different spatial locations are easier to distinguish, which Murphy et al. [60] had found in earlier work. However, researchers also found that auditory cues are more difficult to interpret than Earcons because their meaning may be easier for users to recognize, but they still difficult to represent structural mathematical constructs (e.g., parenthesis) since there is no obvious relationship between this abstract mathematical syntax and real-world sounds. However, researchers reported that Spearcons cues are an excellent way to indicate structural elements such as fractions, superscripts, and subscripts as they are less distracting than traditional lexical cues but still provide a description of the particular structural element involved to the users. Regarding non-speech, researchers found that non-speech sounds such as earcons can be used to represent a hierarchical structure such as nested parenthesis or menu items, but they will require additional cognitive processing from the user, which may distract the user from

processing the mathematical material.

Although auditory cues were difficult to interpret, prosody was found to be useful in conveying the structure of an equation and, significantly, required less effort from the end user than lexical cues [37]. In their work, researchers displayed the math expression using synthetic speech from left to right in front of the user, in addition to using prosodic cues to make the structure of the expression clear to the user. In addition, researchers also placed pauses between operators such as $+$, but only outside complex structures (e.g., fractions and parentheses) to help the user distinguish the audio differences. Researchers also used pitch to indicate the length of an expression so that users could distinguish the complexity of a math expression. The obstacle in such technique is that complex equation is difficult to determine based on prosody alone – suggesting that another form of delimiter is required to provide adequate information.

5.2.3 Spatial Sounds in Mathematics

Some researchers have examined spatial sounds to convey certain information to non-visual users [29, 32, 50, 50]. For example, Harling et al. [37] used spatial sounds in conjunction with different manual gestures (to ensure that a listener has the same speed and accuracy of control as a sighted person during manipulation) in order to design algebra manipulation tool for visually impaired mathematicians (students). In their work, researchers found that sounds generated in different spatial stereo locations were easier for users. This suggests

that when using a headphone to convey certain information based on spatial sound technique, the sound might appear to be between the ears, rather than outside the head. Researchers explained that as the sound source moves around the human body, around to 90 degrees to the vertical plane through the nose (opposite the ear) movement has to occur over 40 degrees to be detected. Due to the orientation of the human ears, front-back localization is much worse than localization in front of the listener – suggesting that the spatial sound technique is difficult to utilize and require specific equipment to achieve the overall purpose. Whereas in [34], researchers used spatial sound technique (discussed previously) to indicate whether a bracket in a math equation is open or closed by using the right or left ear.

5.3 Converting Visual Graphs to Sound

Although some researchers have examined audio-based techniques in enabling visually impaired users to access mathematical information, other researchers investigated various audio-based solutions to make visually-based graphs fully accessible to non-visual users [85]. This type of research is considered in our prior work analysis because there is similarity between graph structures and relationships in code.

However, in Cohen et al. [26], researchers suggested that increasing volume could be used to communicate different events, e.g., representing the distance from the central axis of an edge, and depicted by variation in saturation. In

addition, researchers indicated that speech sound was used to announce the name of the element, and possibly to give a brief description – suggesting that speech could be used to give non-visual programmers an overview of the entire codebase structure, e.g., an overview of loop [27]. Researchers also used shortcut keys to provide detailed information whenever needed, which could be adapted to inform non-visual programmers about their current location within the codebase. Authors also used different tones with a variation of pitch and loudness to guide non-visual users through the entire graph – indicating that repeating tones, or pitch cannot be used again in order to make the audio feedback meaningful [87].

5.4 Representing Menus or Outlines

Significant prior research has examined the use of audio-based techniques to make mobile menus or outlines more accessible to non-visual users [73]. This type of research is considered in our prior work analysis because there is nesting in menu hierarchies, like nested code in programming. For example, Pavani Yalla and Bruce Walker [91] conducted a study where they outline design guidelines for designing an auditory menu for mobile devices. In their work, researchers discussed different type of menu structure and the rules to move from one item to another. They suggested that audio feedback should be designed based on the importance of content, feedback, as well as consistency. For example, the sound of a focus movement through the menu might be a short

beep, where the sound for a menu selection might be a short melody consisting of three different notes. In addition, researchers found that non-speech sounds were a useful technique in giving the user proper feedback during an interaction with the mobile device menu, e.g., simple beeps can be coordinated with key presses to confirm the press. This technique could be adopted in our research work where blind programmers hear short audio feedback about the current location when they are moving through the codebase (e.g., 1 beep indicates level 1 in a nested code).

Similarly, other researchers [41] investigated the use of the Spindex technique (Auditory Index Based on Speech Sounds), which is a non-speech cue based on the pronunciation of the first letter presented in each menu item, e.g., Spindex cue for “Privacy” would be the sound “Pe” or “P” depending on the spoken sound of the letter “P,” the first letter of the word “Privacy.” In their work, researchers found that Spindex in conjunction with text-to-speech outperformed text-to-speech only when users navigated through a mobile menu, which indicated that Spindex technique was able to speed up the navigation process while moving through a long list of items [39,40].

5.4.1 Designing Audio Cues

Moreover, other researchers investigated the design of Earcon cues in their research [11,17,25,36,57]. For example, Brewster et al. [24] found that sounds might be difficult to distinguish by non-visual users when they played next

to each other. Researchers explained that a gap between each sound should be used in order to help users understand when one sound finishes and the other starts – suggesting a delay of 0.1 seconds is adequate for the users to understand, thus, recognition rates should be sufficient.

Moreover, Pavani Yalla and Bruce Walker [92] found that Earcon could be used to provide navigational feedback in hierarchical menus – suggesting that each sub-menu in the hierarchy would play a different sound in the background in order to assist users in identifying movements through different sub-menu. For example, if users moved through the hierarchical structure, the background Earcons would add an extra sound for each extra sub-menu that is traversed. In addition, pitch polarity would change as the user scrolls throughout menu items, which means pitch increases as the user scrolls down the menu, and a decrease as the user scrolls up the menu. However, researchers found that increasing of pitch was “distracting” and “annoying” as users scrolled down through menu, and the pitch became increasingly higher, which suggests that users may prefer higher pitch than lower pitch. Such a technique may make non-visual users feel disoriented or lost in the hierarchical structure when moving through complex menus. To overcome such a problem, the overall structure of the menu or the user location should be conveyed in advance to provide contextual information about the menu structure [64].

Furthermore, Pavani Yalla and Bruce Walker [92] found that adjusting specific audio parameters (e.g., pitch) could help provide useful information to

the non-visual users. For example, pitch of the first tone would correspond to the location of an item, and the pitch of the second tone would correspond to the pitch of the very first or very last item – depending on whether the user is scrolling up or down. This means the second tone would act as a reference, so if the pitch gap between the two consecutive tones is large, the user knows that he or she has to scroll for a long time before reaching the top or bottom of the menu. In case the pitch gap is small, it means that there are only a few items left before the end of the menu. Adjusting audio parameters was also helpful in [60] when conveying mathematics symbols to non-visual students.

5.4.2 Limits of Understandability of Non-Speech

Some researchers discussed the importance of having prior knowledge of non-speech cues and how it could help improve the accuracy of cue recognition [13, 23, 53]. By “prior knowledge,” the authors refer to the involvement of stakeholders throughout the entire design process of non-speech cues. In their work, researchers argued that non-speech cues are completely abstract: the sound has no relation to the objects it represents. They require some form of training in advance – especially for novice users as they might find it obscure to associate each auditory cue with its proper meaning. This suggests that the mapping between the sounds and the event or object they represent are important. If the mapping is hard to determine, the meaning will be ambiguous. To overcome such a problem, we would need to follow a participatory design

approach [71] by involving stakeholders through the design process.

Moreover, other researchers discussed how natural conflict may occur between audio cues when users are presented with two similar concepts within an interface [61]. In their work, researchers explained that sound is identified either as an object (e.g., camera, printer, door, etc.) or action (e.g., closing, locking, etc.) - indicating that sound should be separated either as an object or an action.

Furthermore, some researchers [13, 34, 90] have examined the limits of understandability in audio cues. In [63], researchers compared two different audio-based techniques for navigating a menu, Earcons (a hierarchical progression of variable tones) and Spearcons (spoken directions, compressed and sped up) [90]. In their work, researchers found that Spearcons helped non-visual users navigate through cell phones quickly and comfortably. Similarly, other researchers also indicated that Spearcons (when words are sped up by 70%) recognition rate was higher for visually impaired users compared to their sighted counterparts (this is because non-visual users are used to work with fast spoken words, e.g., screen-reader). Researchers also found that short cues were more effective than more complicated cues, for conveying certain types of information.

5.5 Discussion of Prior Work

As discussed previously, our goal was to investigate whether audio-based techniques could provide adequate support to assist non-visual programmers understand the hierarchical nesting structure of code. In this dissertation work, four major design dimensions are investigated to generate useful design guidelines for employing an audio-based feedback system in the programming environment, mainly to aid non-visual programmers. We review these design dimensions again below:

- **What to convey:** What type of information do stakeholders prefer to know about their nested codebase?
- **Audio Feedback:** What kind of audio feedback (speech, non-speech, etc.) do stakeholders prefer to use to convey the nested codebase?
- **Audio Parameters:** What type of audio properties (pitch, volume, etc.) do stakeholders prefer to adjust when they presented with an audio feedback?
- **Audio Interaction:** What kind of audio interaction techniques (on-demand, automatically, etc.) do stakeholders prefer to use in order to hear an audio feedback?

This prior work analysis has revealed key concepts which we could adopt in this research work: First, we learned that speech sounds were found to be

helpful in giving the users an overview of structurally-based information (math equations, mobile menus, etc.). Second, we learned that non-speech sounds were found to be useful in giving the users feedback during an interaction with the system – suggesting that users like to be informed about their interaction process via a brief sound (very short). Third, we learned that adjusting sound properties was useful in providing the users with different audio meanings, e.g., modifying pitch to indicate a different level of brackets in a math equation. Forth, we learned that non-speech sounds require prior knowledge for the users to understand, especially novice users. Fifth, we learned that non-speech sounds were found to be difficult to use when conveying complex math equations – suggesting that speech sounds should be used instead. Finally, we learned that spatial sounds were found to be helpful in providing non-visual users with important feedback about math equations, e.g., using the right or left ear to indicate whenever a bracket is open or closed. However, additional investigation is needed prior to adopting any one of these techniques in this research work since we are dealing with a different research problem, i.e., conveying the nested structure of code lines. Therefore, we discuss our research studies aimed to evaluate the usability and efficacy of these audio-based techniques for conveying the hierarchical nesting structure of code to assist non-visual programmers in Chapters 6 & 7.

5.6 Limitations of Prior Work

In summary, audio-based techniques were found to be useful for enabling non-visual users to access variety of visually-based information. However, none of the above-mentioned prior work had examined this systematically in order to convey the hierarchical nesting structure of code to aid non-visual programmers, mainly to ease code navigation. Thus, we needed to investigate this particular issue in order to find out whether audio-based techniques could assist non-visual programmers while navigating through large amounts of code.

Chapter 6

Formative Study

6.1 Background and Introduction

During software development, programmers often use integrated development environments (IDEs) or other text-editors to write code, which may include some hierarchical structure, such as nested blocks of code or loops. In addition to whitespace, e.g. some number of tabs or spaces at the beginning of each line of code, text editors or IDEs may also include additional visual cues to convey this code structure efficiently, e.g. level indication or syntax highlighting. To access the computer display, persons who are blind typically use screen-reader software, a form of assistive technology that converts the text, images, and other visual content into synthesized speech or Braille output, depending upon the user's preference. Although screen-reader technologies provide essential access to computer systems for blind users, prior research has

revealed limitations in this technology when a blind user is reading or writing software code [4, 5, 9, 58]. Given the complex visual information in many IDEs, e.g. color-coding or other visual indicators of code structure, non-visual computer programmers who use screen-reader technologies do not yet have access equivalent to their sighted peers [5, 75, 80].

Prior research on computer-programming accessibility has found that blind programmers have access to limited advanced features for enabling a user to move quickly through a large codebase; these limitations force blind programmers to navigate code linearly, one line at a time, or jump between code blocks using “find/search” features [5, 9, 85]. Specifically, prior work has found that non-visual programmers have difficulty understanding the hierarchical nesting structure of code [9, 58, 80]. As a result, researchers have investigated various approaches for enabling faster navigation and a better comprehension of code structure among non-visual programmers [77, 86]. Although recent research has investigated different methods for enabling faster code navigation [9, 77], there has been a lack of research on the usability of audio-based techniques in this context.

For this reason, we investigated whether audio-based techniques could provide support to non-visual programmers while navigating through a nested codebase through a formative interview-based study. In this study, 12 blind programmers indicated their preferences among various forms of verbal (e.g. “Level 1”) and non-verbal (e.g. beeps) sonification for conveying code indenta-

tion when using a screen-reader. The primary goal of this research work was determining empirically whether these users prefer using specific audio-based cues about code structure, mainly to form the design of our experimental study, in Chapter 7. Rather than select details of this prototype arbitrarily, we began with a formative interview-based study, in which non-visual programmers discussed what information they wanted to know as they read nested code, listened to samples with different audio notification types (e.g., speech, non-speech, etc.) with variations in properties (e.g., stereo, pitch, speed, duration, etc.), and discussed their preferences among these options.

The contribution of our work is empirical: We identified the preferences of non-visual programmers in regard to various dimensions of how audio information could supplement understanding of code indentation during the screen-reader-based reading of software code. The study findings established the design of our experimental study where we investigated different audio-based interaction techniques using audio-based prototypes.

This Chapter is structured as follows: Section 6.2 outlines our research questions aimed to investigate the usability of audio-based techniques to convey the hierarchical nesting structure of code. Section 6.3 provides an overview of the formative study methodology, used to answer RQ3. Section 6.4 discusses the study’s overall major findings, and Section 6.6 summarizes our conclusions and future work directions.

6.2 Research Questions Investigated in this Chapter

In this chapter, we address the following research question:

RQ3: In a formative interview study with a variety of audio examples, what forms of audio generation techniques and parameters do non-visual programmers express interest in?

We present the evaluation of our RQ3 through an interview-based study where we collected subjective preferences from 12 participants about various sound effects (e.g., speech, non-speech, stereo, etc.) for conveying the hierarchical nesting structure of code. This smaller study was not enough to show statistically significant differences regarding users' choices of various sound effects and properties. The goal of this smaller study was to understand user preferences of sound effects and provide some answers to RQ3 so that we were not making arbitrary choices about our audio interaction prototypes in phase 2, e.g., larger study (Chapter 7).

6.3 Methodology

During this one-to-one interview and audio prototype evaluation, we presented participants ($n = 12$) with various sound effects with various audio properties. The audio samples included variations in whether the cue was based on

speech (e.g. a voice saying “Level 2”), non-speech sounds (e.g. multiple beep sounds), or left-to-right stereo position of the screen-reader speech audio to convey indentation structure of code (e.g. with the code spatially positioned more the right if indented further). In addition, audio samples of all of these types were generated with variations in audio properties (e.g., pitch, speed, duration). The formative study was planned as a preliminary study, in support of our subsequent larger study, in Chapter 7, which had a larger number of participants. So, the goal of presenting a wide variety of audio cues during this interview study was to formatively explore the design space of sound effects and their properties for conveying code structure. Our aim was not conclusively identifying the best possible sound options, but rather to eliminate any sound effects that users found difficult to perceive or understand. Our goal was to use this study to formatively select a reasonable set of sound options for the prototype to be deployed in our subsequent experimental study, e.g. so that the later study could determine if users prefer such a prototype to a control condition.

6.3.1 Stimuli Preparation

To prepare the speech-based audio stimuli, the JAWS speech synthesis (text-to-speech) software [1] was used to produced brief recordings of a voice saying, e.g. “Level 1,” “Level 2,” etc. In JAWS, we used the default profile (e.g. Eloquence) with a 57-speed rate, 100 volume, 65 pitch, 20 pitch for upper case,

and -20 for the spell rating. Non-speech audio samples were selected from the Freesound open-source library [18] and included, e.g. beeps, bell sounds, pure tones, and other alert sounds. Next, an audio recording was made of the Audacity voice [6] reading aloud a short sample of Python code, so that a recording of an audio cue could be spliced into this recording of the voice, to simulate a sound effect being played while the user is listening to some code being read aloud. Code samples were played using the Python programming language in this study, given the popularity of this language. In addition, this selection was made based on the unique syntax of the language, which uses whitespace indentation at the beginning of lines to convey the nesting structure of lines of code, rather than using curly braces or “begin/end” keywords for blocks. In Python, programmers must indent code properly in order to convey this structure. In our prior work [5], we found that blind programmers were frustrated by listening to screen readers convey whitespaces by reading aloud each space at the beginning of a line of code, e.g. as “Space, Space, Space, Space” or as a count (“four spaces”).

Some of the audio properties of the non-speech cues were adjusted in order to generate variations of each sound effect so that users could be asked which they prefer. In addition, some samples were generated with alternative levels of an audio property to investigate conveying code structure using variations in the property, e.g. increasing the pitch level in a beep sound to convey the depth of the nested loop. Similar variations in sound cues were generated to

investigate repetitions of audio sounds (e.g. multiple beeps indicating different levels of nesting), changes in volume, etc. We summarize the investigated audio parameters as follows:

1. **Spatialization of sound:** refers to a sound processed to give the listener the impression of a sound source within a three-dimensional environment.
2. **Pitch:** refers to the frequency of the voice or sound at which it vibrates, the higher the frequency the higher the note, and by extension the lower the frequency the lower the note.
3. **Volume:** refers to the amount of space occupied by a three-dimensional object or region of space – louder sounds correspond to higher pressure.
4. **Duration:** refers to the amount of time or a particular time interval, e.g., how long a sound lasts.
5. **Continuity:** refers to the continuity of a particular sound, e.g., some sounds may last a few seconds, while other sounds may continue to sound without stopping.
6. **Speed:** refers to the speed of sound waves in air, e.g., the distance travelled per unit time by a sound wave as it propagates through an elastic medium.

Timing of the audio sounds relative to the speech reading aloud the Python code was also investigated by generating samples in which the audio overlapped

with the speech, or with variations in the time delay between the speech and the sound effect if presented serially. To ensure that the audio samples were played at high quality on the participant's side of the videoconferencing interview, the complete list of samples was provided to participants via a web page that the participant opened on their local computer during the interview, so that recordings could be triggered during the interview and played on the participant's local computer.

6.3.2 Recruitment and Participants

Non-visual programmers were recruited through advertisements on mailing lists (e.g., NFB, program-l, etc.) and online groups (e.g., Google, LinkedIn) for people with visual disabilities. The criteria for participating in the study was that individuals had to be 18 years or older, self-identify as totally blind, with at least two years of programming experience, know the Python programming language, and use a screen-reader. Participants were compensated with a \$40 Amazon gift card for the 70-minute interview. In this study, we interviewed 12 participants (11 males, 1 female) with ages from 23 to 41 years (mean = 32.75, SD = 6.14). Our sample showed some variation among participants in their programming experience (from 2 to 22 years' experience) and employment status (e.g., student, employed, and freelancer). All participants used screen-reader technologies, and only 5 used Braille displays with their screen-readers. Participants were from seven different countries: United States ($n = 6$), and 1

participant from each one of the following: Canada, Bulgaria, Ukraine, Italy, Netherlands, and India.

6.3.3 Procedure and Questions

The interview study occurred virtually via Skype and Google Hangouts per participants' preferences. Prior to the interview, participants answered a screening questionnaire to confirm eligibility and gather demographic data. Participants were provided with an informed consent document prior to this IRB-approved study. Each interview lasted approximately 70 minutes. At the beginning of each interview, the premise for the interview was explained, i.e. gauging users' interest in and preferences for a tool that used audio cues to convey the hierarchical nesting structure of code. Interleaved with interview questions, participants listened to various audio cue samples, and for each, participants were asked to share their opinion via open-ended interview questions about the usability and efficacy of each. At the end of the interview, participants responded to questions about their interest in various audio alternatives, and they suggested additional sound options for conveying code structure. The goal of this formative study was to explore as many sound effects as possible via an efficient form of prototyping, to select some promising design options for the higher-fidelity prototype in the later study (Chapter 7).

To analyze data, answers to open-ended questions were noted, with additional written notes taken for participants with particular focus on capturing

direct quotations. We used Microsoft word audio recording feature to record participants' responses during the interview session, and transcripts were corrected by the researcher who reviewed a recording. We followed an open-coding method, to represent ideas or issues raised by participants. Codes were assigned to the transcript of participants' responses and to experimenter notes. Two researchers performed coding independently, and they met after an initial round of coding to produce a unified set of codes. After independently re-coding the data, the two annotators held a meeting to finalize a consensus coding. Based on all coded segments, an affinity diagramming procedure was used to develop a set of themes, which form the basis of our results below.

6.4 Results

This study investigated our research question RQ3, about what forms of audio generation techniques and parameters non-visual programmers would express interest in. Participants discussed the timing of audio cues about the indentation level or structure of a codebase (Subsection 6.4.1), the type of audio feedback they would prefer for conveying code nesting (Subsection 6.4.2), and the type of audio properties stakeholders would prefer to be modified in order to convey various nesting levels (Subsection 6.4.3).

6.4.1 Timing of Audio Cues about Code Structure

In regard to whether to convey the nesting depth of individual lines or groups of lines, participants expressed interest in knowing when each level starts and ends (multiple lines of identical depth). For example, P10 indicated that:

“when I’m skimming the code, I can see the line but I’m not able to see how deep that line is” and wanted “the sound to play before the line.”
(P10)

Similarly, P12 said;

“I do like the idea, knowing how deep it is, that gives an idea of where exactly I am.” (P12)

Some were specifically interested in knowing information about the depth level as code became increasingly nested, e.g. with P9 sharing;

“I think definitely anything that can help to understand the deeper nesting of levels would be beneficial.” (P9)

Participants also commented on the timing of the audio cue, relative to the screen-reader speech, to prevent causing confusion or distraction. Participants did not want audio cues to be played simultaneously with the screen-reader speech; they preferred audio cues to occur in-between lines of code, with a buffer of 10 milliseconds of silence before and after the sound effect (after listening to various timing options), to help listeners distinguish the audio feedback from the screen-reader speech. To elaborate, several participants discussed their thoughts:

“First, it should be the [feedback], after that is the gap. and after that the screen reader... There should be a gap between the two as fast as possible. Also, if I decide to move fast to the next one it should stop the last one of the previous sound or announcements.” (P3)

“Up to 10 milliseconds, yeah well so when you press the down arrow you should immediately hear what’s happening, I mean the speech or non-speech cues from the plugin.” (P4)

“I prefer a very small gap significantly very small one in these files or the cues starting and then speech starting after that.” (P10)

6.4.2 Speech-Based vs. Non-Speech Audio Cues

Participants listened to a variety of sound samples, with some based upon computer-synthesized speech, and others based on a library of non-speech sounds. Overall, participants preferred non-speech audio, rather than speech-based audio messages, which they worried could be confused with the lines of code being read or interfere with their attention as they listen to code. In regard to non-speech cues, participants were asked to share their thoughts about various samples (e.g., beep, bell, musical tones, woodpecker, etc.) as a strategy for conveying the depth of the nested loop. Participants expressed their interest in the “beep” sound and decided to use it over other samples, with several participants explaining:

“I would say just the beep because it’s short and it’s very quickly.” (P5)

“The short one which is beeps, it would be enough for me because it is not something hard to detect and to it is very short and easy to recognize.”
(P7)

“Having to hear the woodpeckers, I can now imagine that would get very great annoying very quickly; so, the beep sound is more settled and less harsh.” (P10)

The speech-based samples included variations such as “you are in level 1,” “level 1,” “L 1,” “1 indent,” “deeper,” and others which indicated the specific integer level of nesting, as well as relative information about whether a line of code is more/less indented than the previous line. Although users preferred non-speech cues (as discussed above), when asked to select from only among the speech-based cues, most preferred a short message, but they did not want to consist of only a number. As P3 explained, if it were only a:

“number pronounced, it might be mistaken for... part of the text or the code.” (P3)

Similarly, P6 commented:

“Level 1 kinds of tells you like heading level, you know, and it makes a mental map to let you [know] where you are, instead of just 1, 2, 3. So, the word ‘level’ actually reminds you [of] the hierarchy.” (P6)

When asked to consider samples of audio that contained a longer spoken message, e.g. “You are in Level 1,” users were concerned that it would take too

much time to listen to such messages, as speed is a premium in their interaction with the computing using the screen reader. For instance, P9 explained:

“So, I definitely do not want to listen to something [that] say; you are in level 2 before I get to hear the line of code. So if it was going to be speech, it would definitely need to be [a] brief and preferably [with] a higher speed. [that] would be good.” (P9)

6.4.3 Conveying Code Structure with Audio Properties

Participants listened to a variety of sound samples with variations in audio properties, e.g. pitch, volume, speed, duration, repetition, stereo, etc. Participants reported that varying some audio properties led to sounds that were annoying when listening to synthesized speech reading code aloud. For instance, in regard to whether the pitch of sounds should be varied to convey the depth of code, participants were skeptical of this concept, as they found this concept difficult to use since it increases memory load. For example, P8 commented:

“If I’m listening to different kinds of pitch, I need to internally in my head map that backs into what number of indentation is that; so, there’s like one extra step that I have to do in my head to figure out the level of indentation.” (P8)

Further, when considering changing audio properties of speech-based audio cues, participants were concerned that the speech could become unintelligible, as P12 explained:

“I am used to having to listen [to] something normal pitch. It is important to me that I could control it. The pitch level definitely, especially with NVDA, it becomes very difficult to understand, and because of the nature of the synthesis voice. It defiantly impossible to understand anything in [a] higher pitch and sped up..” (P12)

In regard to stereo left-to-right spatial audio cues, participants disliked the concept as a whole, as it would require a particular set-up of speakers or headphones. Users in workplace settings preferred not to wear two headphones, as they would be less aware of their surroundings. As P2 commented:

“It seems a little bit redundant. Also, usually, when I’m reviewing code in work and using headphones, I’m only using one [headphone], so that I can still maintain knowledge of what’s going on around me. So, I would be unlikely to use 2 headphones at the same time and get that stereo sound.” (P2)

Participant P7 explained that such a technique requires higher mental load, which should be avoided:

“It requires more attention because my attention should be mostly focused on the project, or of what I am doing and maybe I listen to someone else or thinking or something or listening to disrepute the speech sounds if I had to be focused also in the three dimensions or two dimensions it would require higher mental loud..” (P7)

In regard to adjusting speed, mainly to cut audio length, several participants reported that the duration cut should not exceed 50%, or increase the

speed, as it may cause the meaning of the audio to be difficult to understand. As P4 commented:

“I believe some people may find it difficult if you go beyond 50%.” (P4)

Although participants disliked the idea of stereo left-to-right spatial audio-based cues, other participants reported positive experience when it comes to hearing repetitive beeps, as a strategy for conveying the code indentation. For example, P8 and P10 indicated that repetitive beeps are far easier to use and remember when compared to pitch changes, as an indication for nesting level changes:

“I would like [the] second where you do one beep or two beeps or three beeps as opposed to playing with [the] pitch because then I don’t need to map pitch to the indentation.” (P8)

“[I] can identify something beep, beep, beep, and count those beeps. But pitch can be so hard, something that there is a certain frequency that they can hear . . . I think beep is the best I cannot really think of anything like a better recommendation.” (P10)

6.5 Discussion

The purpose of conducting this formative study had been to inform our choices for the audio interaction prototypes in our experimental study (Chapter 7) so that its design would not be arbitrary. We, therefore, collected responses

about the type of audio feedback, audio timing, and sound properties participants preferred for receiving assistive information about the hierarchical nesting structure of code. In our study, participants indicated an interest in receiving supportive information in the form of non-speech audio (“beeps”) with repetitions of the sound which indicates the number of levels of indentations of some code. In addition, participants recommended the use of time-intervals of silence before and after these audio cues (e.g., 10 milliseconds), to avoid interfering with the speech-reader voice. Furthermore, participants disagreed whether this assistive information should be conveyed automatically when the indentation level changed, or if it should be provided upon demand, e.g., per the user request.

6.6 Summary and Limitations

Although the benefits of using audio-based techniques for assisting non-visual users have been studied in prior work, conveying the indentation level via such cues had been relatively under-studied. In this work, we conducted a formative study where we investigated whether non-visual programmers prefer to receive assistive information about code indentation through audio-based cues. This study provided greater empirical evidence about the need for utilizing an audio-based feedback system in programming environments, and it identifies an opportunity for the research community as well as the software engineering field to address these needs. This research work contributes to

the literature by providing firsthand detailed information from non-visual programmers about the potential of employing audio-based feedback within programming environments.

In this research work, there were several limitations that we would like to mention: Our interview study was too underpowered to allow us to make any statistically significant claims in regard to RQ3; this study had been intended as a formative investigation into the design space, to guide the creation of our prototype for the subsequent experimental study. Secondly, this work has focused on adult computer programmers, but the specific needs of children or students who are learning to the program may differ.

In conclusion, we have identified that non-visual programmers preferred some audio-based cues (e.g., “beeps” and “Level 1”) in our formative study, yet these findings must be investigated in a further empirical study with interactive prototypes where participants have some form of interaction with computer programming code sample.

Chapter 7

Experimental Study

7.1 Background and Introduction

Based on the preliminary findings of our formative study in Chapter 6, we conduct an experimental study with an interactive audio prototype. In this large experimental study, 21 blind programmers indicated their satisfaction with various forms of code-indentation sonification.

For this study, we developed an interactive audio prototype, based on the (previously discussed in 6) formative findings, to address our research question (listed below) as to whether users would actually prefer to receive such audio information when they engage in a software code reading task. In the formative study participants had expressed interest in both automatic notification when moving from one level of indentation to another, as well as on-demand information about how indented their current line of code was

upon request. Thus, we implemented an interactive audio prototype that read aloud, with screen-reader synthesized speech, samples of Python code with nested loops, with three versions: a) automatic interaction where sound effects are played automatically without user involvement, b) on-demand interaction where the user presses a specific keystroke to receive audio feedback, c) and the code read aloud without any audio feedback conveying the code indentation (control condition). Participants' responded to scalar questions about their preferences, and they provided open-ended feedback. Participants preferred receiving audio feedback (both automatic and on-demand preferred to the control condition), but no significant difference was observed when comparing the automatic and on-demand conditions.

The contribution of our work is empirical: Our experimental study provides evidence that non-visual users prefer receiving this supplemental audio information, as compared to a control condition without this additional information. As a minor contribution, we disseminate our audio prototype code, along with our code examples and question items as a supplemental electronic file, to enable future replication of our work or comparison to alternative techniques (see Appendix F).

This Chapter is structured as follows: Section 7.2 outlines our research questions aimed to investigate the usability and efficacy of audio-based techniques to convey the hierarchical nesting structure of code. Section 7.3 provides an overview of the experimental study methodology, used to answer RQ4 and

RQ5. Section 7.4 discusses the study’s overall major findings, and Section 7.6 summarizes our conclusions and future work directions.

7.2 Research Questions Investigated in this Chapter

In this work, we investigated RQ4 and RQ5: RQ4 asked whether non-visual programmers, when actually given an opportunity to use an interactive prototype, would prefer receiving audio information about the indentation of code, as compared to a control condition (simulating the experience of listening to code with a screen reader voice without such cues). RQ5 asked whether users, now that they could interact with a system, preferred to receive notifications: automatically (i.e. when the level of indentation changes) or on-demand (i.e. if the user presses a button while listening to a line of code to hear an audio cue indicating its indentation level). The rationale for investigating both of these questions in a separate study from our formative study (Chapter 6) is to enable us to create a reasonable prototype to investigate whether users actually prefer this type of information (RQ3). Further, the non-interactive nature of the audio samples played in our initial interview-based study did not enable us to investigate this issue of initiative in RQ5. Therefore, we investigated the following research questions in our dissertation work:

RQ4: When presented with an interactive audio prototype based on this prior

formative study, do non-visual programmers prefer receiving this additional audio information about the structure of code, as compared to a control condition without such additional information?

RQ5: When interacting with an audio prototype based on this prior formative study, do non-visual programmers have a preference between automatic level-crossing notifications or on-demand level indications?

7.3 Methodology

In this experimental study, 21 non-visual programmers used an interactive audio prototype with a synthesized voice reading Python code, with all participants trying three versions of the prototype: with automatic level-crossing notifications, with on-demand level indications, or no feedback - with this final case being a control condition. In our within-subjects design, each participant was able to try all three versions of the prototype. In addition, we prepared a set of three Python code examples which participants could explore (each engineered to have a similar level of code complexity and nesting, as discussed below). We, therefore, used a Greco-Latin schedule for rotating the order of presentation of each of the three prototype versions, along with the assignment of each code sample to one of the prototype conditions.

7.3.1 Stimuli Preparation and Prototype

To prepare the Python code stimuli, we wrote a short Python program with three nested loops (three levels in depth), which printed the contents of a nested data structure. After preparing an initial code sample, two additional code samples were written with identical levels of complexity, but with slight variations in the thematic topic of the code and the style of a loop: Example 1 included for-loops and was on the topic of sports team scores, example two used while-loops and was on the topic of movie/cinema reviews, and example 3 included a mix of for-loops with one while-loop and was on the topic of train schedules (see Figures 7.1, 7.2 & 7.3). To ensure a similar complexity of all code examples, the cyclomatic complexity metric [56] was calculated for each, to ensure that each sample had identical metric scores.

```
for group in groups:
    for games in group["games"]:
        i=0
        for result in games["result"]:
            i=i+1
            print("[Game] -- "+str(group["team"])+ " [VS] "+str(games["team"])+ " -- [Score] "+str(result["score"]))
```

Figure 7.1: For-loops code sample.


```

i=0
while i<len(groups):
    j=0
    while j<len(groups[i]["MovieNames"]):
        k=0
        while k<len(groups[i]["MovieNames"][j]["Reviews"]):
            print("[Reviewer] " + str(groups[i]["ReviewerName"]) + " -- [Movie] " + str(
                groups[i]["MovieNames"][j]["name"]) + " -- [Rate] " + str(
                    groups[i]["MovieNames"][j]["Reviews"][k]["score"]))
            k+=1
        j += 1
    i += 1

```

Figure 7.2: While-loops code sample.

```

i=0
while i<len(groups):
    j=0
    for TrainNames in groups[i]["TrainNames"]:
        k=0
        while k<len(TrainNames["schedule"]):
            print("[Station] " + str(groups[i]["StationName"]) + " -- [Train] " + str(
                TrainNames["name"]) + " -- [Arrival Time] " + str(
                    TrainNames["schedule"][k]["ArivalTime"]))
            k+=1
        j += 1
    i += 1

```

Figure 7.3: Mix of for-loops with one while-loop code sample.

Each code sample printed a specific JSON data structure, which had identical structure, but with variable names on different topics: i) team scores, ii) movie/cinema reviews, iii) and train schedules. We describe them as follows:

- **Team Scores:** The game scores data structure is written using JSON data structure. It consists of `groups` which is an array of three objects. Each object in the game scores data structure contains four attributes. The names of these four attributes are `team`, `role`, `teamID`, and `games`. The data type of these four attributes are string, integer, and array of object. The `games` attribute is an array of three objects. Each object in `games` attribute contains two attributes, `team` and `result`. The data

type of these attributes are string and an array of object. The **result** attribute is an array of one single object called **score**. The data type for the **score** object is string.

- **Movie Reviews:** The movie rating data structure is written using JSON data structure. It consists of **groups** which is an array of three objects. Each object in the movie rating data structure contains four attributes. The names of these four attributes are **ReviewerName**, **role**, **ReviewerID**, and **MovieNames**. The data type of these four attributes are string, integer, and array of object. The **MovieNames** attribute is an array of three objects. Each object in **MovieNames** attribute contains two attributes, **name** and **reviews**. The data type of these attributes are string and an array of object. The **reviews** attribute is an array of one single object called **score**. The data type for the **score** object is string
- **Train Schedules:** The train schedule data structure is written using JSON data structure. It consists of **groups** which is an array of three objects. Each object in the train schedule data structure contains four attributes. The names of these four attributes are **StationName**, **role**, **StationID**, and **TrainNames**. The data type of these four attributes are string, integer, and array of object. The **TrainNames** attribute is an array of three objects. Each object in **TrainNames** attribute contains two attributes, **name** and **schedule**. The data type of these attributes are string and an array of object. The **schedule** attribute is an array of

one single object called `ArivalTime`. The data type for the `ArivalTime` object is string.

To produce our interactive audio prototype, we created an audio recording of each code example read aloud by the JAWS screen-reader [1]. In addition, to assist with prototype creation (described below), a researcher annotated a timeline for each audio recording, to note when each line of code began or ended, and to note the indentation level of each line of code (see Figure 7.4). For our study, we needed to produce three different versions of our audio prototype, and details of which are explained below:

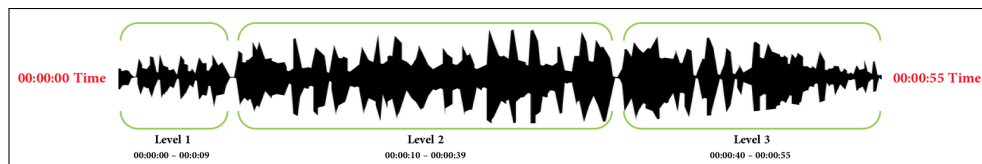


Figure 7.4: In this figure, we show an example of how the for-loop portion will be time-stamped based on three different levels. To create the audio file annotation, it was necessary to listen to the computer voice recording while watching the codebase at the same time in order to track the computer voice recording, i.e., change a particular level to another level.

1. For the “control” condition prototype, we wanted to create a simulation of the experience of listening to a screen reader as it read aloud some code, without any additional audio cues. In this case, our prototype merely consisted of a web page that contained an embedded audio player, that allowed the participant to play the entire audio recording, as many times as they wish.

2. For the “automatic” notifications prototype, the original audio recording was edited, so that we could splice additional time into the recording at each of the between-lines-of-code boundaries at which the indentation level of the code changed. At each of these depth-change boundaries, we inserted a sound effect consisting of a “beep” sound, based on the sound effect beep available at Freesound [2]. This beep was repeated a number of times, to indicate the level of indentation of the code, such that a line of code that was not indented at all would receive 0 beeps. When entering a block of code at a different level of indentation than the previous line of code, the beep would be played a specific number of times, to indicate the level of indentation, e.g. with two beeps to indicate code indented two levels (see Figures 7.6 & 7.5). As discussed above, the selection of a beep sound was based on feedback and suggestions of participants in our earlier formative study (Chapter 6). To enable clear differentiation from the screen-reader speech of the Python code, an interval of silence of duration 10 milliseconds was used to buffer the sound effect from the adjacent speech in the audio recording (also based on our formative study). In this “automatic” version of our prototype, participants only heard the computer voice recording, occasionally interleaved with repetitive beeps to indicate depth changes in the code indentation. As in the control, participants were simply presented with a web page with embedded audio played that enabled the user to play the recording.

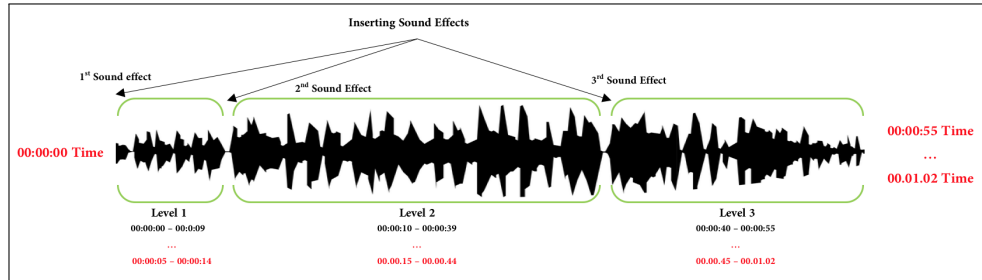


Figure 7.5: In this figure, we show an example of how inserting a sound effect into the computer audio recording with 10 milliseconds delay (pause) would result in changing the original audio recording time-frame. In this example, sound effects could be anything from the speech sound category, non-speech sound category or spatial of sound category.

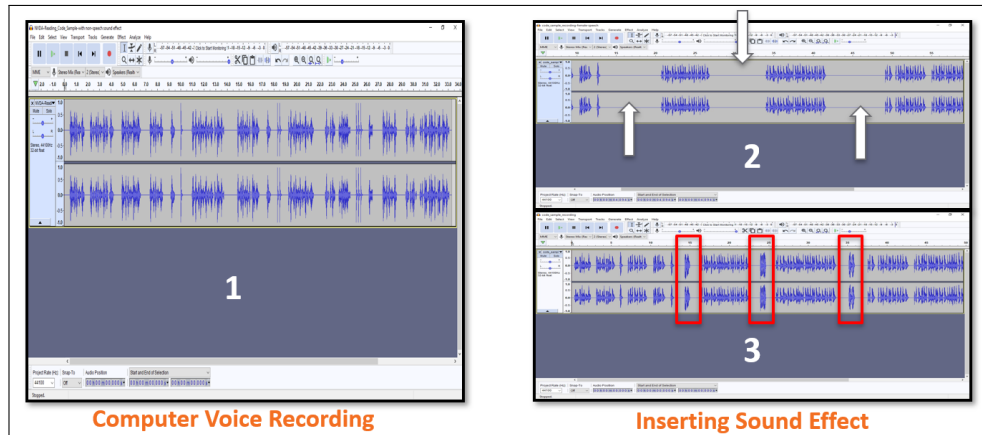


Figure 7.6: This figure shows the process of inserting sound effect into the code sample recorded version using Audacity software.

3. For the “on-demand” level interaction prototype, the “control” version of the prototype was augmented to provide a more interactive experience for users. Specifically, a JavaScript function was implemented to control the audio player such that the user could press the space bar as the audio recording was playing, which would have the following effect: the audio

playback of the code was momentarily paused, a sound played (a series of repeated beeps, identical to those used in the “automatic” condition), and then the audio playback immediately resumes. An interval of silence of duration 10 milliseconds occurred immediately before and after the sound effect. As in the automatic condition, the number of beeps indicated the level of depth of indentation of the line of code, in this case, to indicate the depth of the line of code that was currently being read aloud. The aforementioned timeline annotation created by a researcher to indicate the indentation level of the line of code being read at each moment of the audio recording was used in the implementation of this JavaScript function (Figure 7.7). To demonstrate how this tool would work: (1) user presses the play button, (2) audio file starts to play, (3) user presses the spacebar to request the current location in the codebase, (4) JavaScript function checks computer voice recording time-frame and then provide the corresponding audio feedback (based on the current location in the codebase), (5) audio file pauses for a moment (so that the participant could differentiate the audio feedback (sound effect) from the computer voice recording), (6) sound effect starts to play, (7) user receives the audio feedback corresponding to the current location, and (8) current location is now determined. In this tool, the user would have the freedom to repeat this process many times in order to understand the current location during different levels in the codebase. Our goal is to understand

whether non-visual programmers prefer to hear sound effects played to them on-demand whenever they need to know their current location, to answer the question, “where am I right now?”



```

<script>
var x = document.getElementById("myAudio");
var toggle=0;

timing_data=[];
d3.csv("timing.csv", function(data) {
  console.log(data);
  timing_data=data;
});

document.body.onkeyup = function(e) {
  if(e.keyCode == 32){
    if(toggle==0){
      toggle=1;
      x.play();
    }
    else{
      toggle=0;
      x.pause();
      for (var i = 0; i <= timing_data.length - 1; i++) {
        console.log(timing_data[i]['start'], timing_data[i]['end'], Math.ceil(x.currentTime))
        var startTime=timing_data[i]['start'].split(':');
        var endTime=timing_data[i]['end'].split(':');
        var startMS=(parseInt(startTime[0])*3600+parseInt(startTime[1])*60+parseInt(startTime[2]))*1000+parseInt(startTime[3]);
        var endMS=(parseInt(endTime[0])*3600+parseInt(endTime[1])*60+parseInt(endTime[2]))*1000+parseInt(endTime[3]);
        console.log(startMS,endMS);
        console.log(x.currentTime*1000);
        if(startMS<=parseInt(x.currentTime*1000) && endMS>parseInt(x.currentTime*1000)) {
          console.log("paused");
        }
      }
    }
  }
}

```

Figure 7.7: Overview of the JavaScript function for the on-demand prototype.

7.3.2 Recruitment and Participants

Non-visual programmers were recruited through advertisements on mailing lists (e.g., NFB, program-l, etc.) and online groups (e.g., Google, LinkedIn) for people with visual disabilities. The criteria for participating in the study was that individuals had to be 18 years or older, self-identify as totally blind, with at least two years of programming experience, know the Python programming language, and use a screen-reader. Participants were compensated with a \$40 Amazon gift card for the 70-minute interview. This experimental study included 21 participants (19 males, 2 females), who had ages ranging from

Questions	Scale
Q1. This system was easy to use. Q2. This system was convenient to use. Q3. This system was helpful for your programming tasks.	5-point Likert Scale
Q4. Rate how easy you found the task to complete. Q5. Rate how frustrating you found the task to complete. Q6. Rate whether you felt you had a good idea where you were in the code.	7-point Scale

Table 7.1: List of scale-based questions used in the larger study.

21 to 64 years (mean = 35, SD = 12.46). There was variation in the level of programming experience (lowest = 2 yrs., highest = 49 yrs.) and employment status (e.g., student, employed, unemployed, and freelancer). All participants used screen-reader technologies, and 6 used Braille displays alongside their screen-readers. Participants were from seven different countries: United States (12), India (4), and 1 participant from each one of the following countries: Canada, Netherlands, Georgia, South Africa, and Indonesia.

7.3.3 Procedure and Questionnaire

The study occurred remotely using Skype, Zoom or Google Hangouts, per each participant's preference. At the beginning of the session, participants were informed that they would interact with three audio interaction prototypes, with each explained briefly immediately prior to its use. As discussed above, a Greco-Latin schedule was used in this within-subjects study to assign participants to individual schedules, so that the sequence of presentation could be rotated and the prototype conditions (control, automatic, on-demand) could be rotated in their assignment to the three code samples (sports team scores, movie/cinema

reviews, and train schedules). During the study, participants interacted with all three prototypes. After each interaction, participants were asked to perform the following tasks:

1. Could you explain to me, in three sentences, what this code does?
2. Could you explain to me, in three sentences, what is the code output?

The above-mentioned tasks were asked so that we could measure participants' understanding of each code sample. After using each, participants answered a set of questions, designed to measure the usability and efficacy of each prototype, as summarized in table 7.1. Most instruments had been used in prior studies with blind programmers, especially studies that investigated issues of code navigation. Questions Q1, Q2, and Q3 [55] were 5-point Likert-items, which had previously been used by Bragdon et al. in [20]. Questions Q4, Q5, and Q6 required a response on a seven-point scale, and they had been previously used by Baker et al. [7]. After each prototype, participants were invited to share any open-ended feedback about the prototype they had just used. At the end of the session, participants were again invited to share open-ended feedback about the prototypes, thoughts about their experiences with understanding the indentation structure of code, or to suggest other ideas or improvements to the prototypes they had experienced.

7.4 Results

For each question in Table 1, we collected responses from 21 blind programmers on the three prototypes discussed previously, which are referred to as “Control,” “On-Demand,” and “Automatic” in Figures 1 through 6, which indicate significant differences with asterisks as follows: *** $p < 0.0001$, ** $p < 0.001$, * $p < 0.01$, or N.S. not significant.

Figure 7.8 compares responses when participants are asked about the ease of the system for all conditions (represented on the Y-axis of the chart). After scaling 5-point Likert responses to integer (e.g., “Strongly Disagree” = 1, “Disagree” = 2, etc.), a Friedman test indicated a significant difference ($\chi^2 = 5.991$, $p\text{-value} = p < 0.0001$), and post-hoc pairwise comparison using Wilcoxon Signed Rank tests with Bonferroni corrections indicated significant pairwise differences among the conditions pairs when compared with the baseline: control vs. automatic ($p\text{-value} = 8.770\text{E-}05$), control vs. on-demand ($p\text{-value} = p < 0.0001$).

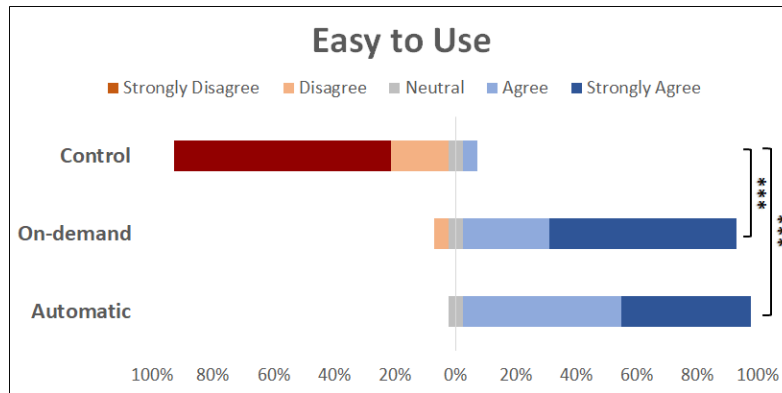


Figure 7.8: Percentage distribution of participants' responses on the ease of using the three conditions (e.g., control, on-demand, and automatic).

Figure 7.9 displays the participants' responses about how convenient the system was (e.g., using 5-point Likert), a Friedman test revealed a significant difference ($\chi^2 = 5.991$, $p\text{-value} = p < 0.0001$) across all conditions, and post-hoc comparison revealed significant differences between: control vs. automatic ($p\text{-value} < 0.00001$), control vs. on-demand ($p\text{-value} = 0.000247$). Figure 7.9 displays the participants' responses:

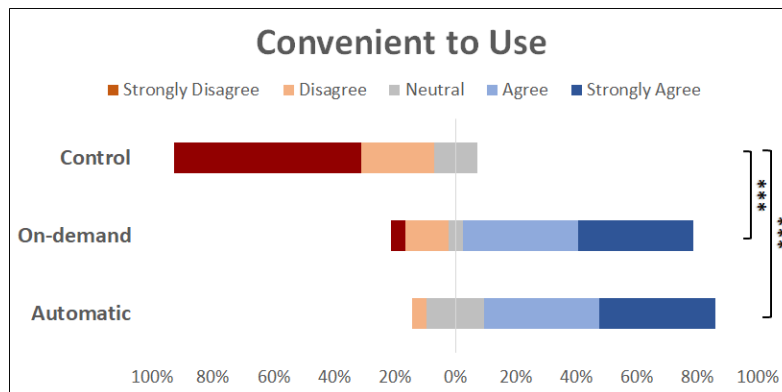


Figure 7.9: Percentage distribution of participants' responses on the convenience of using the three conditions (e.g., control, on-demand, and automatic).

Figure 7.10 displays participants' responses to "this system was helpful for your programming tasks". A Friedman test showed a significant difference ($\chi^2 = 5.991$, $p\text{-value} = p < 0.0001$), and post-hoc pairwise comparison using Wilcoxon Signed Rank tests indicated significant pairwise differences between: control vs. automatic ($p\text{-value} < 0.00001$), and control vs. on-demand ($p\text{-value} < 0.00001$).

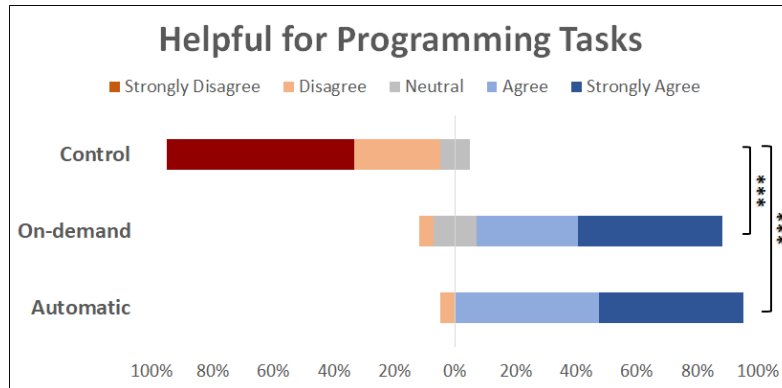


Figure 7.10: Percentage distribution of participants' responses on how helpful the three conditions (e.g., control, on-demand, and automatic) when working on a computer programming code.

Figure 7.11 displays the participants' responses for how easy the task was to compete in all three conditions (using 7-point scalar), a Friedman test indicated a significant difference ($\chi^2 = 5.991$, $p\text{-value} = 0.000320$), and post-hoc pairwise comparison using Wilcoxon Signed Rank tests indicated pairwise differences between: control vs. automatic ($p\text{-value} = 0.000211$), and control vs. on-demand ($p\text{-value} = 0.000227$):

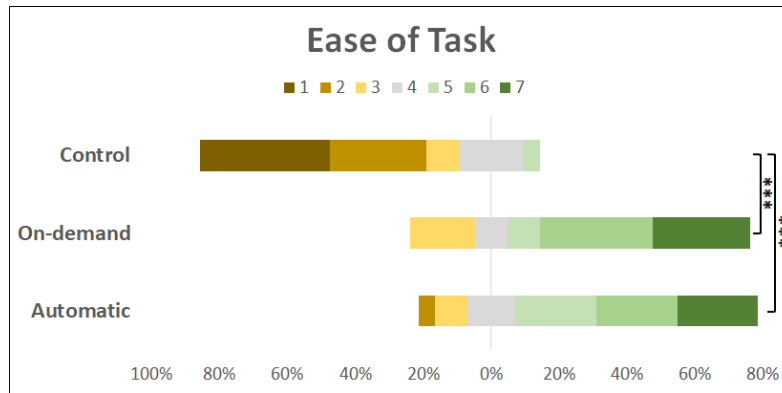


Figure 7.11: Percentage distribution of participants' responses on the ease of completing a task conducted to evaluate the three conditions (e.g., control, on-demand, and automatic).

Figure 7.12 displays responses to “rate how frustrating you found the task to complete.” A Friedman test indicated a significant difference ($\chi^2 = 5.991$, p -value = 0.000324), and post-hoc tests revealed pairwise difference for: control vs. automatic (p -value = 0.000448), and control vs. on-demand (p -value = 0.00020)

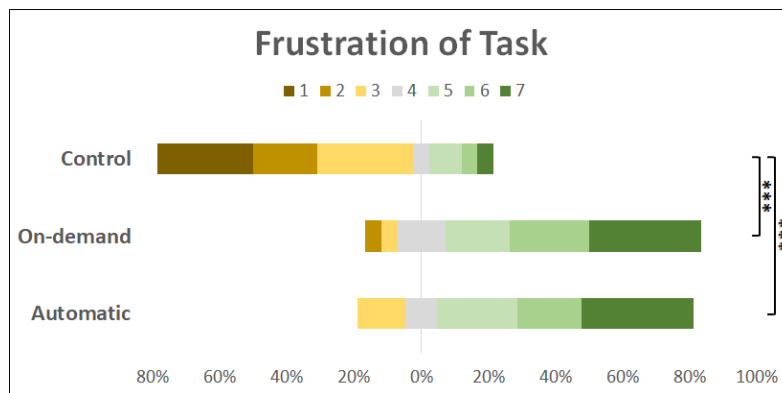


Figure 7.12: Percentage distribution of participants' responses on how frustrated they were when completing an evaluation task using the three conditions (e.g., control, on-demand, and automatic).

Figure 7.13 displays participants’ responses as to whether they had a good idea where they were in the code, with a Friedman test indicating a significant difference ($\chi^2 = 5.991$, $p\text{-value} < 0.00001$) among all conditions, and post-hoc pairwise comparison indicating significant pairwise differences between the following pairs: control vs. automatic ($p\text{-value} = 0.000110$), and control vs. on-demand ($p\text{-value} = 0.000131$)

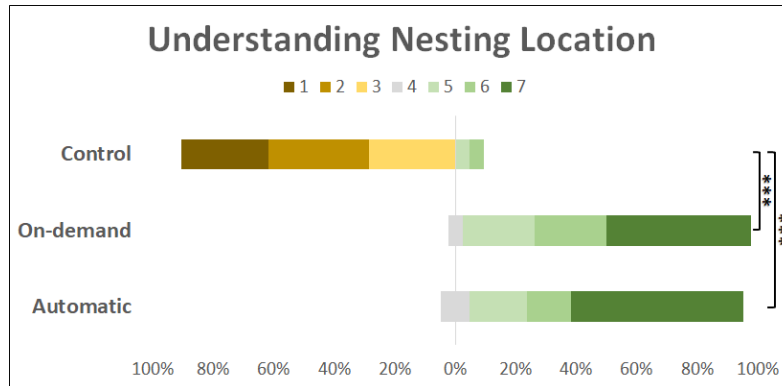


Figure 7.13: Percentage distribution of participants’ responses on understanding the current location in the nested code (e.g., code sample) using the three conditions (e.g., control, on-demand, and automatic).

Some open-ended feedback comments at the end of the study are summarized below, for each of the three prototypes, but additional quotations from participants are included in the Discussion (Section 7.5). When discussing the “control” condition, participants had mostly negative reactions and indicated that it was difficult to understand the code indentation. As P8 explained:

“What I dislike about it [no feedback], not being able to get the information

that I need on-demand and basically the ability to not go wherever the start of the code, [it] always you remember to figure out how far you are nested in the code.” (P8)

When discussing the “automatic” condition, participants had positive feedback about the repetitive-beeps audio cue and indicated that it assisted them in understanding the indentation level. P7 explained:

“What I like about it is when you are sifting through the code with arrow keys it automatically tells you whether there is an indent or not and you can find out whether you have invaded [a] mistake whenever you are sifting through the code in the function.” (P7)

Similarly, P18 said:

“[What] I liked about the automatic, when going through the various loops, you know I could tell if I was at a level one indentation, level 2 indentation, or level three. I mean it was there, no doubt as to where you know in the code and what level I was at. I mean that was perfect.” (P18)

When discussing the “on-demand” condition, participants had similarly positive feedback, indicating that they liked the instant feedback about the indentation level. Participant P18 discussed how:

“On-demand, you can figure out where you are in the code, at your own speed, I mean if you don’t want to know what in the indentation level, you’re right, you’re not being interrupted by a beep tone so that part I did like.” (P18)

P21 also discussed the benefit of using the on-demand prototype:

“So, what I like about it is that it’s on-demand, so you get the information only when you request it and you need it. if you attached to a good key that doesn’t conflict with other text editing commands you might let say it’s control shift or something like that would be fantastic information about your position of the code and I’ll be on your terms.” (P21)

7.5 Discussion

In this experimental study, 21 blind programmers interacted with three different prototype conditions, with more positive subjective responses for the automatic and on-demand prototypes, as are compared to the control condition. Our findings for RQ4 indicated that non-visual programmers prefer receiving this additional audio information about the structure of code, as compared to a control condition without such additional information – thereby suggesting the usability of audio-based techniques for conveying code indentation for blind computer programmers. In open-ended comments, participants indicated that these techniques helped them in understanding their location in the code, and they found the prototypes easy to use and did not require prior knowledge to use them. For instance, P7 explained:

“If no feedback is reported to them then they have no way of navigating through the code like navigate their minds and you know, [to] get exactly where they are.” (P7)

P17 preferred the on-demand condition, saying :

“On-demand was easy to use, I know we wouldn’t be using a space bar, but you know it’s very easy and very quick to get where you are in terms of indentation level.” (P17)

In regard to RQ5, post-hoc pairwise testing did not reveal any significant differences in participant responses between the automatic and on-demand conditions. In open-ended comments, participants mentioned some trade-offs between these two conditions. For instance, some participants indicated that the on-demand prototype would be useful for debugging code, as it helps navigate quicker and find code errors. Participants P2, P5, and P12 shared their thoughts in this regard:

“Debugging would be easy with the case where we can directly jump on that particular level instead of checking indentation every time.” (P2)

“I would use it for debugging because when I’m stepping through Python code sometimes, I need to see what I’ve done if the code doesn’t work and then you have to really understand indentation.” (P5)

“The automatic beep sounds can be used in all the places, even while I am writing the code or while skimming through code or even debugging.” (P12)

On the other hand, several participants indicated that automatic-based feedback would be useful for skimming through a large amount of code, especially code written by someone else, which the user is reviewing. Participants

preferred continuous feedback while moving through nested loops. P19 and P21 discussed their thoughts:

“The beeping definitely helps with skimming, without it becomes slower unless you have features in the editor jumps into different blocks.” (P19)

“Sometimes it’s challenging to skim through the code with a screen reader. So, it easy to miss something that way, but I do sometimes, if I really need a really high-level understanding of what’s going on, in that case, I would prefer to use the automatic option. Because if I have to constantly be pressing a key on each line, I can lose some of the benefits of skimming the code.” (P21)

These comments suggest that each of these two methods may be useful in different contexts, specifically with on-demand for debugging and automatic for skimming code. While these comments from participants were suggestive, further study would be needed to investigate if there are indeed advantages of each technique in these contexts.

7.6 Summary and Limitations

In this work, we evaluated the usability and efficacy of audio-based interaction techniques in efforts to understand whether non-visual programmers preferred these audio-based interactions over the baseline condition (e.g., no feedback). This study contributes to the literature by providing detailed information from a user-based study with a relatively large number of participants (given the

specific user group of non-visual programmers) about the potential of audio-based feedback within programming environments. As an additional minor contribution of our work, we have demonstrated an experimental methodology that can be utilized by future researchers who wish to investigate methods for conveying code structure for non-visual programmers, and we have disseminated our experimental prototype in the electronic supplementary file shared with dissertation.

In this work, there were several limitations: First, as the prototypes in the experimental study were audio-based prototypes with pre-recorded audio streams, participants did not have immediate interaction with code examples, so that they could step through the code line-by-line or interactively navigate throughout the code. Therefore, a further experimental study using a text-buffer-based prototype, which enables the user to move through the code and edit the text, would be needed to determine whether the findings from this study would generalize to more realistic environments. Second, the study findings were based on experienced programmers, another study with novice or students' participants may draw different conclusions.

Epilogue to Part II

In part II, we surveyed prior work on the experience of non-visual programmers to establish that these users currently face some challenges in reading software code when using screen-reader technologies, especially in regard to the issue of understanding the nesting indentation structure of code. In our analysis, we examined the most closely related prior work on audio-based techniques to increase the accessibility of programming for these users, to establish that little work has been done to investigate the issue of navigating the hierarchical structure of code. Additional research is needed into how to convey indentation structure of individual lines of code in the context of the linear reading of code via screen-reader. To broaden our focus, we considered related research on using audio-based cues in settings that are analogous in some way, namely: conveying nesting structure in mathematical notation, conveying the relationships within graph structures, or representing navigation through nested menus or outlines. The prior work analysis has suggested various audio-based strategies for conveying this type of information to users, which has motivated the

specific types of audio cues explored in our research work (Chapters 6 & 7). Furthermore, we discussed our user-based studies (6 & 7) where we evaluated the usability and efficacy of audio-based techniques to convey the hierarchical nesting structure of code to assist non-visual programmers. In summary, Part II of this dissertation work has addressed the following research questions:

RQ3: In a formative interview study with a variety of audio examples, what forms of audio generation techniques and parameters do non-visual programmers express interest in?

- The above-mentioned research question was investigated in Chapter 6 where we recruited 12 non-visual programmers to evaluate various audio cues for conveying the hierarchical nesting structure of code to assist non-visual programmers. The research findings indicated that participants do prefer to receive audio-based feedback when compared to the baseline condition, e.g., only the screen-reader reading the codebase. Participants provided positive responses in regard to the use of audio-based techniques – suggesting a further investigation where users have some form of interaction with the codebase. Overall, participants indicated an interest in these techniques in efforts to enhance code understanding, specifically code navigation.

RQ4: When presented with an interactive audio prototype based on this prior

formative study, do non-visual programmers prefer receiving this additional audio information about the structure of code, as compared to a control condition without such additional information?

- As reported in Chapter 6, participants were asked to evaluate a set of audio cues without the ability to interact with the codebase, e.g., participants' responses were based on listening only. To ensure the formative study outcomes, in regard to the selected audio cues, we conducted a follow-up study with 21 participants where we investigated users' preferences via an audio-based prototypes, mainly to address the RQ listed above (Chapter 7). Our findings indicated that participants were interested in such feedback, thereby confirming the formative study (Chapter 6) outcomes when compared to the baseline condition, e.g., no feedback.

RQ5: When interacting with an audio prototype based on this prior formative study, do non-visual programmers have a preference between automatic level-crossing notifications or on-demand level indications?

- In the experimental study, our primary investigation was to examine different audio interaction techniques (e.g., on-demand and automatic) and whether participants prefer specific interaction when compared to the baseline condition (Chapter 7). The study findings revealed that participants preferred such interaction to understand

the indentation in Python-based code over the baseline condition, e.g., no feedback. In addition, our findings did not reveal significant differences between on-demand and automatic interactions since participants liked both interactions for several reasons. While participants enjoyed both techniques, further investigation using a fully interactive prototype may reveal why such findings emerged.

Chapter 8

Limitations and Future Work

This chapter discusses limitations, which we highlight in two major parts. As future work, we provided several opportunities where additional research could be conducted to address these limitations which could improve accessibility in software-based environments.

8.1 Limitations and Future Work

In Part I, we explain the limitations of two user-based studies that were conducted to investigate the major challenges in software development faced by non-visual programmers, specifically code navigation difficulties. In addition, we discuss possible future work enhancements where possible features could be created in order to make the programming environment more accessible to non-visual users. In Part II, we explain the limitations of two user-based

studies that were conducted to evaluate the usability and efficacy of audio-based techniques for conveying the structure of the programming codebase, which was suggested by the stakeholders in Part I. Specifically, the two studies were conducted to investigate various sound effects, properties, and several interaction techniques to determine whether these techniques capable of providing adequate support to assist non-visual programmers in regard to code understanding and navigation. As future work, we highlight possible avenues for potential future research which could help make the software-based environments more accessible to non-visual programmers:

8.1.1 Part I: Limitations and Future Work

This Subsection explains the limitations of the two user-based studies that were conducted in efforts to understand the major challenges in software development. We discuss these limitations as follows:

- In the survey-based study (Chapter 3), we followed a snowball sampling technique, which resulted in uneven participant categories, e.g., participants vary in their visual acuity, assistive technologies, and programming experiences. We followed this approach in order to maximize the number of responses in the time allotted from a population that is difficult to recruit. In addition, the survey design of this study did not allow us to ask follow-up questions or observe the users while performing some programming tasks in regard to certain programming issues. Thus, future

work is needed to explore specific issues (e.g., UML Diagrams, Debugging, etc.) in detail with a well-defined user profile, which could help reveal interesting findings.

- In the interview-based study (Chapter 4), there were several limitations: First, we only explored code navigation difficulties with experienced developers, who were totally blind, actively engaged in programming, and used assistive technologies to access the computer display (e.g., screen reader, braille display, or both). It was beyond our scope to study students, novice programmers, or individuals with greater diversity in their visual acuity. A further investigation with these important user groups may reveal different findings. Second, while the qualitative design of this study allowed us to gather firsthand comments from our user group, and to discover new issues that arose, in future work, it may be important to follow up this study with a survey administered to a larger group of participants, to verify some of our findings.

8.1.2 Part II: Limitations and Future Work

This Subsection explains the limitations of the two user-based studies that were conducted in efforts to evaluate the usability and efficacy of audio-based techniques in software developments. We discuss these limitations as follows:

- In the formative-based study (Chapter 6), our work was too underpowered to allow us to make any statistically significant claims in regard

to **RQ3**; this work had been intended as a formative investigation into the design space, to guide the creation of our audio-based prototypes where we investigated several interaction techniques. After identifying that participants preferred at least some prototypes in our study, we believe that future researchers and accessibility designers would benefit from a rigorous investigation into design variations of such prototypes, with larger participants and more statistical power.

- In the experimental study (Chapter 7), the prototypes in our work were only audio-based prototypes with pre-recorded audio streams, participants did not have immediate interaction with code examples so that they could step through the code line-by-line or interactively navigate throughout the code. In future work, we would like to investigate the generalizability of our results using text-buffer prototypes, including the implementation of plug-ins for integrated development environments, so that participants could use their own screen-reader and computer, with industry-standard text-editing environments, to investigate this design space. A further investigation while using a text-buffer-based prototype, which enables the user to move through the code and edit the text, would be needed to determine whether the findings from this experimental study would generalize to more realistic environments. Such a study would also enable us to investigate a wider variety of software code, with various languages, and various levels of complexity. In addition, this research

work has focused on adult computer programmers, but the specific needs of children or students who are learning to the program may indicate otherwise. Continuing this line of research may lead to tools that will enable greater participation in computing education or professional careers among non-visual programmers. It would also enable research into which types of tool users prefer when they are writing, editing, or debugging code.

Chapter 9

Summary and Contributions

This dissertation has presented several research studies that address the programming challenges faced by non-visual programmers, mainly the challenge of code navigation. The discussed issues mostly stemmed from the fact that screen-reader technologies were designed to present information in a linear fashion, e.g., one line at a time. In addition, we also presented research studies that evaluated the usability and efficacy of audio-based techniques for conveying the hierarchical nesting structure of code to assist non-visual programmers. In this chapter, we present the major contributions of this dissertation in two different parts as well as our final comments that summarize the overall dissertation objectives.

9.1 Summary of the Contribution of This Research

The research studies presented in this dissertation are organized into two major parts. We summarize the contributions of each as follows:

9.1.1 Part I: Programming Challenges and Code Navigation Difficulties

Part I of this dissertation investigated the major programming challenges of blind computer programmers, to guide the selection of more specific interventions to be explored in later phases of the dissertation. As, the selected research problem (e.g., code navigation) was understudied, further investigation was needed so that we could propose some novel solutions to overcome such a problem. We summarize the major contributions of Part I as follows:

1. **Empirical Contribution:** In Chapter 3, we presented the major findings that emerged from conducting a survey-based study where participants highlighted and discussed briefly their challenges in software-based environments. In this study, some of the findings were expected such as the lack of accessibility in current IDEs as well as the difficulty of using screen-reader technologies with today’s software development. To overcome programming barriers, participants reported the use of alternative tools to understand code structure as well as seeking help from sighted co-workers. In addition to using two different assistive technology (screen-reader and braille display) at the same time to uncover hidden

information.

2. **Empirical Contribution:** In Chapter 4, we presented the major findings that emerged from conducting an interview-based study where we examined the issue of code navigation. These issues were discussed in detail in efforts to illustrate the issue that has a higher demand or importance among participants. In this study, we found that blind developers felt overwhelmed when using existing IDEs (e.g., Eclipse, NetBeans, etc.), and therefore they preferred to use simpler editors to write software code comfortably (e.g., Notepad, Notepad++, etc.). Furthermore, participants discussed a list of code navigation difficulties as well as possible accessibility improvements where additional features could be developed in order to make the programming environment more accessible to non-visual programmers.

9.1.2 Part II: Usability of Audio-based Techniques

Part II of this dissertation investigated the usability and efficacy of audio-based techniques in software-based environments and whether such techniques could provide adequate support to aid non-visual programmers when navigating through the hierarchical nesting structure of code. Through user-based studies with non-visual programmers, we gathered positive responses on the usability of such techniques when compared to the baseline condition, e.g., only screen-reader with no additional feedback. We summarize the major contributions of

Part II as follows:

1. **Methodological Contribution:** In Chapter 6, the questions-types, as well as the empirical result, could be used to aid future researchers when investigating other design aspects or parameters in audio-based techniques, or when evaluating various approaches for conveying the code indentation in Python-based language via different settings or configurations, with our results as a potential baseline.
2. **Empirical Contribution:** In Chapter 6, with the goal of investigating the usability and efficacy of audio-based techniques, this research study provided evidence of users' preferences of various audio cues, with participants reporting subjectively higher scores when compared with the baseline condition, e.g., only the screen-reader without any additional feedback. As indicated previously, the formative study provided empirical results in regard to the type of information participants would like to know, the type of audio feedback participants would like to hear, the placement of audio feedback, how audio should be timed, and the design of such cues.
3. **Empirical Contribution:** In Chapter 7, to evaluate different interaction techniques (e.g., on-demand and automatic), our experimental study revealed that screen-reader technologies do not provide adequate support when used in software-based environments – suggesting the need for accessibility improvements in order to convey the nesting structure

of code as it currently prevents users from such an important feedback. We also identified an opportunity for the research community as well as the software engineering industry to address those needs. This work contributes to the literature by providing detailed information from a relatively large number of participants (given the specific user group of non-visual programmers) about the proper interaction method that users preferred to receive information (based on audio cues) about code indentation.

4. **Empirical Contribution:** In Chapter 7, we examined different interaction techniques for providing supportive information about the nesting structure of code, based on users' recommendations and suggestions from the formative-based study. Our findings indicated that participants preferred both methods (on-demand and automatic) over the baseline condition, no audio feedback. This work contributes to the literature by providing detailed information about the design aspects for utilizing such techniques in software-based environments, which could be used by future researchers who wish to investigate such methods in various aims. Finally, this study has demonstrated an audio simulation methodology for investigating audio-based interventions with screen-reader users in a programming context, e.g., depth of bracketing or level of indentation, as in nested loops. As part of this contribution, we have disseminated our experimental prototypes as an open-source for future researchers in order

to improve accessibility in a software-based environment for non-visual users (see Appendix F).

9.2 Conclusion and Final Comment

In this dissertation, we have presented and discussed the major findings that emerged from conducting several research studies aimed to enhance accessibility in software-based environments. The dissertation's entire work was organized into two major parts. In Part I, we have presented two user-based studies where we investigated the software development accessibility issues. The first study was conducted to uncover the major programming challenges faced by non-visual programmers. Whereas the second study was conducted to better understand the issue of code navigation, which was revealed previously in the first study. Our findings indicated that participants were interested in using supportive tools that use audio as the primary method of interaction – suggesting a further investigation into the design space of such techniques.

In Part II, we have presented and explained the major findings that emerged from conducting two user-based studies where we evaluated the usability and efficacy of audio-based techniques in software-based environments. The first study was conducted to evaluate various audio cues and parameters in efforts to eliminate unwanted settings and configurations – suggesting specific cues to be utilized for the higher-fidelity prototype in the experimental study. While the second study was conducted in efforts to evaluate different interaction

techniques (no feedback, on-demand, and automatic) through audio-based prototyping. In this study, our findings concluded that participants enjoyed both on-demand and automatic techniques over the baseline condition, e.g., no-feedback – indicating that participants want both techniques for various reasons.

In conclusion, the dissertation work presented herein discusses the need for improving accessibility features in programming environments in order to aid non-visual programmers. It shows the proper approach towards utilizing audio-based techniques in efforts to benefit programmers who are blind from the widespread features that are eliminated due to existing barriers. This dissertation major findings as well as disseminated resources will be useful for future researchers to further investigate this important research problem – with our design guidelines as the potential baseline.

Bibliography

- [1] 2020a. Freedom Scientific. (2020). <https://www.freedomscientific.com/> (pages 98, 119).
- [2] 2020b. FreeSound. (2020). <https://freesound.org/> (page 120).
- [3] Nusaibah M Al-Ratta and Hend S Al-Khalifa. 2013. Teaching programming for blinds: A review. In *Information and Communication Technology and Accessibility (ICTA), 2013 Fourth International Conference on*. IEEE, 1–5. (page 1).
- [4] Khaled Albusays and Stephanie Ludi. 2016. Eliciting Programming Challenges Faced by Developers with Visual Impairments: Exploratory Study. In *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE '16)*. ACM, New York, NY, USA, 82–85. DOI:<http://dx.doi.org/10.1145/2897586.2897616> (pages 4, 33, 36, 95).

- [5] Khaled Albusays, Stephanie Ludi, and Matt Huenerfauth. 2017. Interviews and Observation of Blind Software Developers at Work to Understand Code Navigation Challenges. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17)*. ACM, New York, NY, USA, 91–100. DOI:<http://dx.doi.org/10.1145/3132525.3132550> (pages 4, 95, 99).
- [6] Audacity. 2018. Audacity is a multi-track audio editor and recorder for Windows, Mac OS X, GNU/Linux and other operating systems. (2018). <https://www.audacityteam.org/> (page 99).
- [7] Catherine Marie Baker. 2017. *Understanding and Improving Blind Students' Access to Visual Information in Computer Science Education*. Ph.D. Dissertation. (pages 1, 125).
- [8] Catherine M. Baker, Cynthia L. Bennett, and Richard E. Ladner. 2019. Educational Experiences of Blind Programmers. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 759–765. DOI:<http://dx.doi.org/10.1145/3287324.3287410> (page 13).
- [9] Catherine M. Baker, Lauren R. Milne, and Richard E. Ladner. 2015. StructJumper: A Tool to Help Blind Programmers Navigate and Understand the Structure of Code. In *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI '15)*. ACM, New York, NY,

- USA, 3043–3052. DOI:<http://dx.doi.org/10.1145/2702123.2702589> (pages 2, 14, 60, 66, 95).
- [10] Suzanne P. Balik, Sean P. Mealin, Matthias F. Stallmann, Robert D. Rodman, Michelle L. Glatz, and Veronica J. Sigler. 2014. Including Blind People in Computing through Access to Graphs. In *Proceedings of the 16th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '14)*. Association for Computing Machinery, New York, NY, USA, 91–98. DOI:<http://dx.doi.org/10.1145/2661334.2661364> (page 13).
- [11] Woodrow Barfield, Craig Rosenberg, and Gerald Levasseur. 1991. The use of icons, earcons, and commands in the design of an online hierarchical menu. *IEEE Transactions on Professional Communication* 34, 2 (1991), 101–108. (page 87).
- [12] Kitch Barnicle. 2000. Usability Testing with Screen Reading Technology in a Windows Environment. In *Proceedings on the 2000 Conference on Universal Usability (CUU '00)*. ACM, New York, NY, USA, 102–109. DOI: <http://dx.doi.org/10.1145/355460.355543> (page 1).
- [13] Enda Bates and Dónal Fitzpatrick. 2010a. Spoken Mathematics Using Prosody, Earcons and Spearcons. In *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 407–414. DOI:http://dx.doi.org/10.1007/978-3-642-14100-3_61 (pages 82, 89, 90).

- [14] Enda Bates and Dónal Fitzpatrick. 2010b. Spoken mathematics using prosody, earcons and spearcons. In *International Conference on Computers for Handicapped Persons*. Springer, 407–414. (page 83).
- [15] Hugh Beyer and Karen Holtzblatt. 1997. *Contextual design: defining customer-centered systems*. Elsevier. (pages 25, 41).
- [16] Jeffrey P. Bigham, Irene Lin, and Saiph Savage. 2017. The Effects of "Not Knowing What You Don'T Know" on Web Accessibility for Blind Web Users. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17)*. ACM, New York, NY, USA, 101–109. DOI:<http://dx.doi.org/10.1145/3132525.3132533> (page 2).
- [17] Meera M Blattner, Denise A Sumikawa, and Robert M Greenberg. 1989. Earcons and icons: Their structure and common design principles. *Human-Computer Interaction* 4, 1 (1989), 11–44. (page 87).
- [18] David B Boardman, Geoffrey Greene, Vivek Khandelwal, and Aditya P Mathur. 1995. Listen: A tool to investigate the use of sound for the analysis of program behavior. In *Computer Software and Applications Conference, 1995. COMPSAC 95. Proceedings., Nineteenth Annual International*. IEEE, 184–189. (page 81).
- [19] Yevgen Borodin, Jeffrey P. Bigham, Glenn Dausch, and I. V. Ramakrishnan. 2010. More Than Meets the Eye: A Survey of Screen-reader Browsing

- Strategies. In *Proceedings of the 2010 International Cross Disciplinary Conference on Web Accessibility (W4A) (W4A '10)*. ACM, New York, NY, USA, Article 13, 10 pages. DOI:<http://dx.doi.org/10.1145/1805986.1806005> (page 2).
- [20] Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. 2010a. Code Bubbles: Rethinking the User Interface Paradigm of Integrated Development Environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. Association for Computing Machinery, New York, NY, USA, 455–464. DOI:<http://dx.doi.org/10.1145/1806799.1806866> (page 125).
- [21] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr. 2010b. Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*. ACM, New York, NY, USA, 2503–2512. DOI:<http://dx.doi.org/10.1145/1753326.1753706> (pages xv, 15).
- [22] Stephen A. Brewster. 1998. Using Nonspeech Sounds to Provide Navigation Cues. *ACM Trans. Comput.-Hum. Interact.* 5, 3 (Sept. 1998), 224–259. DOI:<http://dx.doi.org/10.1145/292834.292839> (page 80).

- [23] Stephen A Brewster. 2002. Non-speech auditory output. *The human-computer interaction handbook* (2002), 220–239. (page 89).
- [24] Stephen A Brewster, Peter C Wright, and Alastair DN Edwards. 1994. A detailed investigation into the effectiveness of earcons. In *Santa Fe Institute Studies in the Sciences of Complexity-proceedings Volume-*, Vol. 18. Addison-Wesley Publishing Co, 471–471. (page 87).
- [25] Stephen A. Brewster, Peter C. Wright, and Alistair D. N. Edwards. 1993. An Evaluation of Earcons for Use in Auditory Human-Computer Interfaces. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI '93)*. Association for Computing Machinery, New York, NY, USA, 222–227. DOI:<http://dx.doi.org/10.1145/169059.169179> (page 87).
- [26] Robert F. Cohen, Arthur Meacham, and Joelle Skaff. 2006. Teaching Graphs to Visually Impaired Students Using an Active Auditory Interface. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '06)*. ACM, New York, NY, USA, 279–282. DOI:<http://dx.doi.org/10.1145/1121341.1121428> (page 85).
- [27] Robert F. Cohen, Rui Yu, Arthur Meacham, and Joelle Skaff. 2005. PLUMB: Displaying Graphs to the Blind Using an Active Auditory Interface. In *Proceedings of the 7th International ACM SIGACCESS Conference on Computers and Accessibility (Assets '05)*. ACM, New York, NY,

- USA, 182–183. DOI:<http://dx.doi.org/10.1145/1090785.1090820> (page 86).
- [28] Richard Connelly. 2010. Lessons and Tools from Teaching a Blind Student. *J. Comput. Sci. Coll.* 25, 6 (June 2010), 34–39. <http://dl.acm.org/citation.cfm?id=1791129.1791137> (page 13).
- [29] Kai Crispian, Wolfgang Würz, and Gerhard Weber. 1994. Using spatial audio for the enhanced presentation of synthesised speech within screen-readers for blind computer users. In *International Conference on Computers for Handicapped Persons*. Springer, 144–153. (page 84).
- [30] Ajantha Dahanayake. Integrated Development Environments (IDEs). *Wiley Encyclopedia of Management* (????). (page 1).
- [31] Tilman Dingler, Jeffrey Lindsay, and Bruce N Walker. 2008. Learnability of sound cues for environmental features: Auditory icons, earcons, spearcons, and speech. International Community for Auditory Display. (page 83).
- [32] Hilko Donker, Palle Klante, and Peter Gorny. 2002. The Design of Auditory User Interfaces for Blind Users. In *Proceedings of the Second Nordic Conference on Human-computer Interaction (NordiCHI '02)*. ACM, New York, NY, USA, 149–156. DOI:<http://dx.doi.org/10.1145/572020.572038> (page 84).

- [33] Olutayo Falase, Alexa F. Siu, and Sean Follmer. 2019. Tactile Code Skimmer: A Tool to Help Blind Programmers Feel the Structure of Code. In *The 21st International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '19)*. Association for Computing Machinery, New York, NY, USA, 536–538. DOI:<http://dx.doi.org/10.1145/3308561.3354616> (page 14).
- [34] Colin Fitzsimons, Emma Murphy, Catherine Mulwa, and Donal Fitzpatrick. 2016. SpatialMaths: a Library for Conveying Content and Structure of Equations. (2016). (pages 85, 90).
- [35] Joan M. Francioni and Ann C. Smith. 2002. Computer Science Accessibility for Students with Visual Disabilities. *SIGCSE Bull.* 34, 1 (Feb. 2002), 91–95. DOI:<http://dx.doi.org/10.1145/563517.563372> (pages 2, 13).
- [36] John CK Hankinson and Alistair DN Edwards. 1999. Designing earcons with musical grammars. *ACM SIGCAPH Computers and the Physically Handicapped* 65 (1999), 16–20. (page 87).
- [37] Philip A Harling, Robert Stevens, and Alistair Edwards. 1995. Math-grasp: The design of an algebra manipulation tool for visually disabled mathematicians using spatial-sound and manual gestures. *HCI Group, University of York, UK* (1995). (page 84).

- [38] Austin Z. Henley and Scott D. Fleming. 2014. The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '14)*. ACM, New York, NY, USA, 2511–2520. DOI:<http://dx.doi.org/10.1145/2556288.2557073> (page 14).
- [39] Myounghoon Jeon and Bruce N Walker. 2009. “Spindex”: Accelerated initial speech sounds improve navigation performance in auditory menus. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Vol. 53. SAGE Publications Sage CA: Los Angeles, CA, 1081–1085. (page 87).
- [40] Myounghoon Jeon and Bruce N Walker. 2011. Spindex (speech index) improves auditory menu acceptance and navigation performance. *ACM Transactions on Accessible Computing (TACCESS)* 3, 3 (2011), 10. (page 87).
- [41] Myounghoon Jeon, Bruce N. Walker, and Abhishek Srivastava. 2012. “Spindex” (Speech Index) Enhances Menus on Touch Screen Devices with Tapping, Wheeling, and Flicking. *ACM Trans. Comput.-Hum. Interact.* 19, 2, Article 14 (July 2012), 27 pages. DOI:<http://dx.doi.org/10.1145/2240156.2240162> (page 87).

- [42] Shaun K. Kane and Jeffrey P. Bigham. 2014. Tracking @stemxcomet: Teaching Programming to Blind Students via 3D Printing, Crisis Management, and Twitter. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education (SIGCSE '14)*. Association for Computing Machinery, New York, NY, USA, 247–252. DOI:<http://dx.doi.org/10.1145/2538862.2538975> (page 13).
- [43] Arthur Karshmer. 2007. Access to mathematics by blind students: A global problem. (2007). (page 82).
- [44] Arthur I Karshmer and Chris Bledsoe. 2002. Access to mathematics by blind students. In *International Conference on Computers for Handicapped Persons*. Springer, 471–476. (page 82).
- [45] Mik Kersten and Gail C. Murphy. 2006. Using Task Context to Improve Programmer Productivity. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, USA, 1–11. DOI: <http://dx.doi.org/10.1145/1181775.1181777> (page 1).
- [46] Mario Konecki. 2012. A new approach towards visual programming for the blinds. In *MIPRO, 2012 Proceedings of the 35th International Convention*. IEEE, 935–940. (page 13).

- [47] Mario Konecki, Robert Kudelić, and Danijel Radošević. 2010. Challenges of the blind programmers. In *Central European Conference on Information and Intelligent Systems*. (pages 1, 2).
- [48] Mario Konecki, Alen Lovrenčić, and Robert Kudelić. 2011. Making programming accessible to the blinds. In *MIPRO, 2011 Proceedings of the 34th International Convention*. IEEE, 820–824. (page 2).
- [49] Ivan Kopecek and A Jergová. 1997. Programming and visually impaired people. In *Proceedings of the XV. World Computer Congress, ICCHP*, Vol. 98. 365–372. (page 2).
- [50] SH Kurniawan, A Sporka, V Nemec, and P Slavik. 2004. Design and user evaluation of a spatial audio system for blind users. In *Proceedings of The 5th International Conference on Disability, Virtual Reality and Associated Technologies, ICDVRAT 2004*. 20–22. (page 84).
- [51] John Lamping, Ramana Rao, and Peter Pirolli. 1995. A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '95)*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 401–408. DOI:<http://dx.doi.org/10.1145/223904.223956> (page 79).
- [52] Jonathan Lazar, Aaron Allen, Jason Kleinman, and Chris Malarkey. 2007. What Frustrates Screen Reader Users on the Web: A Study of 100 Blind

- Users. *International Journal of Human-Computer Interaction* 22, 3 (may 2007), 247–269. DOI:<http://dx.doi.org/10.1080/10447310709336964> (page 1).
- [53] Paul A Lucas. 1994. An evaluation of the communicative ability of auditory icons and earcons. Georgia Institute of Technology. (page 89).
- [54] Stephanie Ludi, Jamie Simpson, and Wil Merchant. 2016. Exploration of the Use of Auditory Cues in Code Comprehension and Navigation for Individuals with Visual Impairments in a Visual Programming Environment. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '16)*. ACM, New York, NY, USA, 279–280. DOI:<http://dx.doi.org/10.1145/2982142.2982206> (page 13).
- [55] Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE '15)*. IEEE Press, 260–270. DOI:<http://dx.doi.org/10.1109/ASE.2015.42> (page 125).
- [56] Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering* 4 (1976), 308–320. (page 116).

- [57] David K. McGookin and Stephen A. Brewster. 2004. Understanding Concurrent Earcons: Applying Auditory Scene Analysis Principles to Concurrent Earcon Recognition. *ACM Trans. Appl. Percept.* 1, 2 (Oct. 2004), 130–155. DOI:<http://dx.doi.org/10.1145/1024083.1024087> (page 87).
- [58] Sean Mealin and Emerson Murphy-Hill. 2012. An exploratory study of blind software developers. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*. IEEE, 71–74. (pages 2, 16, 66, 95).
- [59] Lauren R. Milne, Catherine M. Baker, and Richard E. Ladner. 2017. Blocks4All Demonstration: A Blocks-Based Programming Environment for Blind Children. In *Proceedings of the 19th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '17)*. ACM, New York, NY, USA, 313–314. DOI:<http://dx.doi.org/10.1145/3132525.3134774> (page 2).
- [60] Emma Murphy, Enda Bates, and Dónal Fitzpatrick. 2010. Designing Auditory Cues to Enhance Spoken Mathematics for Visually Impaired Users. In *Proceedings of the 12th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '10)*. ACM, New York, NY, USA, 75–82. DOI:<http://dx.doi.org/10.1145/1878803.1878819> (pages 82, 83, 89).

- [61] Elizabeth D. Mynatt. 1994. Designing with Auditory Icons: How Well Do We Identify Auditory Cues?. In *Conference Companion on Human Factors in Computing Systems (CHI '94)*. ACM, New York, NY, USA, 269–270. DOI:<http://dx.doi.org/10.1145/259963.260483> (page 90).
- [62] National Center for Science National Science Foundation and Engineering. 2017. Women, Minorities, and Persons with Disabilities in Science and Engineering. (2017). www.nsf.gov/statistics/wmpd/ (page 13).
- [63] Dianne K Palladino and Bruce N Walker. 2007. Learning rates for auditory menus enhanced with spearcons versus earcons. Georgia Institute of Technology. (page 90).
- [64] Dianne K Palladino and Bruce N Walker. 2008. Navigation efficiency of two dimensional auditory menus using spearcon enhancements. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, Vol. 52. SAGE Publications Sage CA: Los Angeles, CA, 1262–1266. (page 88).
- [65] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 199–209. DOI:<http://dx.doi.org/10.1145/2001420.2001445> (page 44).

- [66] Venkatesh Potluri, Priyan Vaithilingam, Suresh Iyengar, Y. Vidya, Manohar Swaminathan, and Gopal Srinivasa. 2018. CodeTalk: Improving Programming Environment Accessibility for Visually Impaired Developers. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems (CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–11. DOI:<http://dx.doi.org/10.1145/3173574.3174192> (page 14).
- [67] Johnny Saldaña. 2015. *The coding manual for qualitative researchers*. Sage. (pages 25, 41).
- [68] Jaime Sanchez and Fernando Aguayo. 2004. Listen what I do: blind learners programming through audio. *Memorias TISE* (2004), 120–124. (page 1).
- [69] Jaime Sánchez and Fernando Aguayo. 2005. Blind Learners Programming Through Audio. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems (CHI EA '05)*. ACM, New York, NY, USA, 1769–1772. DOI:<http://dx.doi.org/10.1145/1056808.1057018> (page 78).
- [70] Jaime Sánchez and Fernando Aguayo. 2006. APL: audio programming language for blind learners. In *International Conference on Computers for Handicapped Persons*. Springer, 1334–1341. (page 78).
- [71] Douglas Schuler and Aki Namioka. 1993. *Participatory design: Principles and practices*. CRC Press. (page 90).

- [72] Waltraud Schweikhardt. 1982. A Programming Environment for Blind APL-Programmers. *SIGAPL APL Quote Quad* 13, 1 (July 1982), 325–331. DOI:<http://dx.doi.org/10.1145/390006.802260> (page 13).
- [73] Andrew Sears and Ben Shneiderman. 1994. Split Menus: Effectively Using Selection Frequency to Organize Menus. *ACM Trans. Comput.-Hum. Interact.* 1, 1 (March 1994), 27–51. DOI:<http://dx.doi.org/10.1145/174630.174632> (page 86).
- [74] Kaitlin Duck Sherwood. 2008. *Path exploration during code navigation*. Ph.D. Dissertation. University of British Columbia. (page 14).
- [75] Robert M. Siegfried. 2006. Visual Programming and the Blind: The Challenge and the Opportunity. *SIGCSE Bull.* 38, 1 (March 2006), 275–278. DOI:<http://dx.doi.org/10.1145/1124706.1121427> (pages 2, 13, 95).
- [76] Janice Singer, Robert Elves, and M-A Storey. 2005. Navtracks: Supporting navigation in software maintenance. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*. IEEE, 325–334. (page 14).
- [77] Ann C. Smith, Justin S. Cook, Joan M. Francioni, Asif Hossain, Mohd Anwar, and M. Fayezur Rahman. 2003. Nonvisual Tool for Navigating Hierarchical Structures. *SIGACCESS Access. Comput.* 77-78 (Sept. 2003),

- 133–139. DOI:<http://dx.doi.org/10.1145/1029014.1028654> (pages 2, 14, 16, 66, 79, 95).
- [78] Ann C. Smith, Joan M. Francioni, and Sam D. Matzek. 2000. A Java Programming Tool for Students with Visual Disabilities. In *Proceedings of the Fourth International ACM Conference on Assistive Technologies (Assets '00)*. ACM, New York, NY, USA, 142–148. DOI:<http://dx.doi.org/10.1145/354324.354356> (page 80).
- [79] Andreas Stefik, Roger Alexander, Robert Patterson, and Jonathan Brown. 2007. WAD: A feasibility study using the wicked audio debugger. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*. IEEE, 69–80. (page 78).
- [80] Andreas Stefik, Christopher Hundhausen, and Robert Patterson. 2011a. An empirical investigation into the design of auditory cues to enhance computer program comprehension. *International Journal of Human-Computer Studies* 69, 12 (2011), 820–838. (pages 81, 95).
- [81] Andreas Stefik, Richard E. Ladner, William Allee, and Sean Mealin. 2019. Computer Science Principles for Teachers of Blind and Visually Impaired Students. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE '19)*. Association for Computing Machinery, New York, NY, USA, 766–772. DOI: <http://dx.doi.org/10.1145/3287324.3287453> (page 13).

- [82] Andreas M. Stefik, Christopher Hundhausen, and Derrick Smith. 2011b. 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 (SIGCSE '11)*. ACM, New York, NY, USA, 571–576. DOI:<http://dx.doi.org/10.1145/1953163.1953323> (pages 2, 13, 79).
- [83] Andreas M. Stefik, Christopher Hundhausen, and Derrick Smith. 2011c. 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 (SIGCSE '11)*. Association for Computing Machinery, New York, NY, USA, 571–576. DOI:<http://dx.doi.org/10.1145/1953163.1953323> (page 13).
- [84] Sarit Felicia Anais Szpiro, Shafeka Hashash, Yuhang Zhao, and Shiri Azenkot. 2016. How People with Low Vision Access Computing Devices: Understanding Challenges and Opportunities. In *Proceedings of the 18th International ACM SIGACCESS Conference on Computers and Accessibility (ASSETS '16)*. ACM, New York, NY, USA, 171–180. DOI:<http://dx.doi.org/10.1145/2982142.2982168> (page 36).
- [85] Brianna J. Tomlinson, Jared Batterman, Yee Chieh Chew, Ashley Henry, and Bruce N. Walker. 2016. Exploring Auditory Graphing Software in the Classroom: The Effect of Auditory Graphs on the Classroom Environment.

- ACM Trans. Access. Comput.* 9, 1, Article 3 (Nov. 2016), 27 pages. DOI: <http://dx.doi.org/10.1145/2994606> (pages 85, 95).
- [86] Paul Vickers and James L Alty. 2002. When bugs sing. *Interacting with Computers* 14, 6 (2002), 793–819. (pages 61, 80, 95).
- [87] Bruce N Walker and Joshua T Cothran. 2003. Sonification Sandbox: A graphical toolkit for auditory graphs. Georgia Institute of Technology. (page 86).
- [88] Bruce N Walker and Anya Kogan. 2009. Spearcon performance and preference for auditory menus on a mobile phone. In *International Conference on Universal Access in Human-Computer Interaction*. Springer, 445–454. (page 83).
- [89] Bruce N Walker, Jeffrey Lindsay, Amanda Nance, Yoko Nakano, Dianne K Palladino, Tilman Dingler, and Myounghoon Jeon. 2013. Spearcons (speech-based earcons) improve navigation performance in advanced auditory menus. *Human Factors* 55, 1 (2013), 157–182. (page 83).
- [90] Bruce N Walker, Amanda Nance, and Jeffrey Lindsay. 2006. Spearcons: Speech-based earcons improve navigation performance in auditory menus. Georgia Institute of Technology. (page 90).
- [91] Pavani Yalla and Bruce N Walker. 2007. *Advanced auditory menus*. Technical Report. Georgia Institute of Technology. (page 86).

- [92] Pavani Yalla and Bruce N. Walker. 2008. Advanced Auditory Menus: Design and Evaluation of Auditory Scroll Bars. In *Proceedings of the 10th International ACM SIGACCESS Conference on Computers and Accessibility (Assets '08)*. ACM, New York, NY, USA, 105–112. DOI: <http://dx.doi.org/10.1145/1414471.1414492> (page 88).

Appendices

Appendix A

IRB Approval Forms

All of the studies presented in this dissertation has been approved by the Institutional Review Board (IRB). We provide the IRBs documents for the four research projects below:

- *Understanding the major programming challenges in software development:* This IRB covers the survey-based study presented in Part I of this dissertation.
- *Interviews about code navigation difficulties:* This IRB covers the interview-based study presented in Part I of this dissertation.
- *Evaluating the usability of audio-based techniques:* This IRB covers the formative study as well as the experimental study presented in Part II of this dissertation.

	Rochester Institute of Technology RIT Institutional Review Board for the Protection of Human Subjects in Research 141 Lomb Memorial Drive Rochester, New York 14623-5604 Phone: 585-475-7673 Fax: 585-475-7990 Email: hmfsrs@rit.edu
---	--

Form C
IRB Decision Form

TO: Khaled Albusays
FROM: RIT Institutional Review Board
DATE: November 5, 2015
RE: Decision of the RIT Institutional Review Board

Project Title – Understanding major the differences and challenges between novice and expert developers who are visually impaired

The Institutional Review Board (IRB) has taken the following action on your project named above.

☒ Exempt 46.101 (b) (2)

Now that your project is approved, you may proceed as you described in the Form A.

You are required to submit to the IRB any:


- **Proposed** modifications and wait for approval before implementing them,
- Unanticipated risks, and
- Actual injury to human subjects.

Signature Hidden for Security and Privacy

Heather Foti, MPH
 Associate Director
 Office of Human Subjects Research

Revised 10-18-06

Figure A.1: IRB Decision Form for “Understanding the major programming challenges in software development”.

<div style="text-align: center;">  </div> <div style="text-align: center; margin-top: 20px;"> Form C IRB Decision Form FWA# 00000731 </div>	<div style="text-align: right;"> Rochester Institute of Technology RIT Institutional Review Board for the Protection of Human Subjects in Research 141 Lomb Memorial Drive Rochester, New York 14623-5604 Phone: 585-475-7673 Fax: 585-475-7990 Email: hmfsrs@rit.edu </div>
---	--

TO: Khaled Albusays
FROM: RIT Institutional Review Board
DATE: March 6, 2018
RE: Decision of the RIT Institutional Review Board

Project Title – Interviews about Sonification of Structured Programming Code for Blind Users

The Institutional Review Board (IRB) has taken the following action on your project named above.

☒ Exempt 46.101 (b) (2)

Now that your project is approved, you may proceed as you described in the Form A.

You are required to submit to the IRB any:

- **Proposed** modifications and wait for approval before implementing them,
- Unanticipated risks, and
- Actual injury to human subjects.

Signature Hidden for Security and Privacy

Heather Foti, MPH
 Associate Director
 Office of Human Subjects Research

Revised 08.17.2017

Figure A.2: IRB Decision Form for “Interviews about code navigation difficulties”.


	Rochester Institute of Technology RIT Institutional Review Board for the Protection of Human Subjects in Research 141 Lomb Memorial Drive Rochester, New York 14623-5604 Phone: 585-475-7673 Fax: 585-475-7990 Email: hmfsrs@rit.edu
 Form C IRB Decision Form FWA# 00000731	
 TO: Khaled Albusays, Matt Huenerfauth	
FROM: RIT Institutional Review Board	
DATE: July 10, 2018	
RE: Decision of the RIT Institutional Review Board	
Project Title – Interviews about Audio-based Prototype of Structured Programming Code for Blind Users	
The Institutional Review Board (IRB) has taken the following action on your project named above.	
<input checked="" type="checkbox"/> Exempt <u>46.101 (b) (2)</u>	
Now that your project is approved, you may proceed as you described in the Form A.	
You are required to submit to the IRB any:	
<ul style="list-style-type: none">• Proposed modifications and wait for approval before implementing them,• Unanticipated risks, and• Actual injury to human subjects.	
<div style="background-color: black; color: white; padding: 5px; margin: 10px auto; width: 200px;">Signature Hidden for Security and Privacy</div> <div>Heather Foti, MPH Associate Director Office of Human Subjects Research</div>	
Revised 08.17.2017	

Figure A.3: IRB Decision Form for “Evaluating the usability of audio-based techniques”.

Appendix B

Survey Questionnaire

This appendix presents our survey questionnaire aimed to understand the major programming challenges encountered by visually impaired programmers, which was used to conduct the survey-based study in Chapter 3.

Title: Eliciting Programming Challenges Faced by Developers with Visual Impairments: Exploratory Study

Informed Consent: Thank you for accepting to be a part of this important survey, seeking to elicit programming challenges for individuals with visual impairments. We are researchers from Rochester Institute of Technology conducting a research on visually impaired developers. The purpose of the study is to address challenges blind developers face while programming. You must be 18 years or older to participate in our survey. We would appreciate your help by taking 10 to 15 minutes to complete this survey. The incentive for

participation is that each participant will be entered in a raffle for an Amazon gift card. All information collected will be used only for our research and will be kept confidential. Please submit your survey by clicking on the Submit button at the bottom of this page. If you have any questions or concerns, please do not hesitate to contact us at: kla3145@rit.edu

Are you 18 years or older?

☐ Yes

☐ No (*Not allowed to participate in this Survey, thus, survey will ends with a thank you message!*)

B.1 Survey Questionnaire

1. Please indicate your age?

2. What is your gender?

☐ Male

☐ Female

☐ Prefer not to say

3. What best describe your visual acuity?

- ☐ Vision but corrective lenses have extremely little ability to help (less than 20/200)
- ☐ Tunnel vision where part of the visual area are absent
- ☐ Macular degeneration where part of the visual area are absent
- ☐ Light/Shadow sensitivity but unable to distinguish objects
- ☐ Total blindness
- ☐ Other: _____

4. Do you have any visual perception?

- ☐ Light
- ☐ Shadows
- ☐ Colors
- ☐ Movement
- ☐ Other: _____

5. Which of the following assistive aids you use when programming?

You may select more than one.

- ☐ Screen reader, Example: Voice over
- ☐ Braille display
- ☐ Large fonts
- ☐ Other: _____

6. How did you learn your first programming language?

- ☐ Self-Taught
- ☐ School
- ☐ Other: _____

7. Please specify your development platform?

- ☐ Windows
- ☐ Mac OS X
- ☐ Linux
- ☐ Other: _____

8. What platform do you develop for?

- ☐ Windows
- ☐ Mac OS X
- ☐ Linux
- ☐ Windows Phone
- ☐ Android
- ☐ iOS
- ☐ Web
- ☐ Other: _____

9. What compilers/IDEs (Integrated development environment) do you use?

☐ Eclipse

☐ Netbeans

☐ Xcode

☐ Visual Studio

☐ Other: _____

10. Do you use an editor in addition to the IDE when you program?

If yes, please specify why.

11. For each one of the following programming languages, select your level of expertise.

Indicate your level of experience in the following table

Languages / Experience	None	Novice	Intermediate	Expert
Java	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
C	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
C#	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
C++	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Objective-C	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Python	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Ruby	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Perl	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
JavaScript	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
PHP	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

12. Which of the following programming languages do you use most?

- ☐ Java
- ☐ C
- ☐ C#
- ☐ C++
- ☐ Objective-C
- ☐ Python
- ☐ Ruby
- ☐ Perl
- ☐ JavaScript
- ☐ PHP
- ☐ Other: _____

13. How many hours per week do you write code?

14. What are the challenge(s) that you face when programming?

15. Describe solution(s) that you found for the challenge(s)?

16. Are willing to participate in our future study?

☐ Yes

☐ No

17. Please provide your email address to be entered in a raffle for an Amazon gift card and / or participate in our future study.

END OF THE SURVEY QUESTIONNAIRE!

Thank you very much for taking your time to participate in this survey

Appendix C

Interview Questionnaire

This appendix presents our interview questionnaire aimed to understand code navigation difficulties encountered by non-visual programmers, which was used to conduct the interview-based study in Chapter 4.

Title: Interviews and Observation of Blind Software Developers at Work to Understand Code Navigation Challenges

Informed Consent: The research involves a study of visually impaired developers and the major issues related to programming. The purpose of the study is to address the major differences and challenges between novice and expert developers. You will answer a set of questions posed by the researcher if you participate in this interview. The interview will take approximately 45 minutes to 1 hour.

If you participate in the interview, you will receive a questionnaire as an inter-

view, which will take 45 minutes to 1 hour after reviewing the consent form. This study will be held at the researcher workplace without inconveniencing the participant space. In case the participant could not make it, an online interview will be held via Skype or Google hangouts. The researcher however, will film the participant computer screen for the purpose of observation some tasks. The researcher will collect your personal data such as age, gender, contact information, and user experience for the study. There are no expected risks, harms, inconveniences and discomforts to you as the subject. The incentive for participation is that each participant will be entered in a raffle for an Amazon gift card. The findings from this research will enable us to make critical decisions about major differences and challenges for novice and expert developers who are visually impaired. The same applies to the society. There is no financial effort required from you at any point in the research. The researcher will be the only person handling the information you provide in the research. Once the data is analyzed, the researcher will store all the sheets of paper and recording until the approval of the paper. When the research report is complete, the researcher will destroy all the data collecting and storage material. The report will be evaluated by a committee for educational purposes and later availed to the public via online libraries but not personal information will be deducible from the report.

Taking part in this study is voluntary; you do not have to participate, and you can withdraw from the research at any point you wish. There is no penalty

or any benefit loss for any choice you make. You can contact me through my phone number: (785) 498-9095 or email me at: kla3145@rit.edu

If you feel that you have a question about your rights or any adverse event, you can contact the HSRO Associate Director through email at:hmfsrcs@rit.edu

Please write your full name below and sign if you agree to be part of this study:

Name:_____

Date:_____

Signature:_____

Interview Instruction: The interview is going to be face-to-face interview unless the participant could not make it to the interviewer location. In such a case, we will conduct this interview via the Internet using Skype or Google hangouts.

C.1 Interview Questionnaire

1. Describe the process that you use to program?

(a) Demonstrate simple programming task to me?

2. What kind of tools you use in order to help you program?

3. As blind developer, what kind of challenges do you face?

4. Describe challenge(s) that you found solution(s) for and what were the solutions you found?

5. Is code navigation a challenge for you, why or why not?

(a) Code navigation limitation?

6. If you face issues with inaccessible tools while programming or collaborating with other developers, how do you overcome this issue?

7. Do you work individually or with teams?

(a) If you work with team, please explain your process and tools that you find useful?

(b) And if your team member sighted or visually impaired?

END OF THE INTERVIEW QUESTIONNAIRE!

Thank you very much for taking your time to participate in this study!

Appendix D

Formative Study

Questionnaire

This appendix presents our Sonification interview questionnaire aimed to investigate whether audio-based techniques capable of assisting non-visual programmers to convey the structure of the programming codebase, which would be used to conduct the audio-based prototype (Sonification interview) study in Chapter 6.

Title: Interviews about Sonification of Structured Programming Code for Non-visual Users

Investigator: Khaled L. Abusays, Ph.D. student in Computing and Information Sciences, Rochester Institute of Technology

Faculty Supervisor: Matt Huenerfauth, Professor, Department of Informa-

tion Sciences and Technologies, Rochester Institute of Technology

Thank you for taking time to participate in this research study. The below information is helpful to decide whether to proceed further or not.

Nature and Purpose of the Project: The goal of this research project is to learn how to improve the experience of computer programmers who are blind. We are studying whether various auditory cues (e.g. sound effects) may help programmers who are using a screen reader when writing computer code.

Explanation of Procedures: This study will not take more than 70 minutes to complete. Today, you will be asked some questions about possible ways someone could interact with different sound effects in a programming environment. I will also ask you some other questions about evaluating various audio samples: some consist of computer generated speech, and other are non-speech sound effects that vary in their pitch, loudness, or other properties, e.g. left-to-right stereo. We are interested in learning how you would interact with software that included these sounds as well as your opinions. Please be honest with your feedback. All interviews will be audio recorded for further analysis.

Potential Discomfort and Risks: You will speak with the interviewer and listen to some sound effect recordings; the potential risks are minimal. Also, you may request a break at any time.

Potential Benefits: You will not receive any direct benefits for participating in this study. The study will be used to help direct future design decisions of

the interface experience.

Costs/Reimbursements: After the 70-minute study, you will be compensated for your time with either: \$40 cash (if you are meeting the investigator in person for this study) or a \$40 Amazon gift card (if you are participating remotely) which will be transmitted to you by email within one week of your interview appointment. In this case, please ensure the interviewer has your correct email address.

Confidentiality: Every effort will be made by the investigator to keep your research records and other personal information confidential, except as may be required by court order or law. Access to the research records may be provided to the authorized representatives of Rochester Institute of Technology, including members of the Institutional Review Board (IRB), a committee which reviews and approves all research involving human subjects.

Withdrawal from the Project: Participation in this research project is voluntary. You may decide not to participate or to leave the study at any time. Choosing to leave or deciding to not participate the study will not result in any penalty or loss of benefits to which you are entitled, nor harm your relationship with the university.

Whom to Contact with Questions: This research project has been reviewed by the Rochester Institute of Technology Institutional Review Board. If you have any questions related to your rights as a research participant, you may contact: Heather Foti, Associate Director, Office of Human Subjects Re-

search, Phone Number: 585-475-7673, Email: hmfsrs@rit.edu

If you have any questions about the conduct of this research project or think that something is unusual or unexpected happening or psychological discomforts, any injuries, you may contact: Dr. Matt Huenerfauth, Professor, Department of Information Sciences and Technologies, Rochester Institute of Technology, Departmental Phone Number: 585-475-7924, Email: matt.huenerfauth@rit.edu

D.1 Example Script and Questions for Semi-Structured Interview, with Links to Audio Samples

Interview Method:

The interview questions presented herein is a semi-structured interview. The interview is going to be faceto-face unless the participant could not make it to the investigator location. In such a case, we will conduct this study online via telephone or Skype or Google Hangouts per the participant preference.

Audio Files:

In addition to asking the participant some questions about their experiences as a computer programmer who is blind, the investigator will also play some sample sound effects for the participant to consider. The goal of this project is to consider how sounds could communicate information to programmers who are listening to a screen reader (computer voice) that is reading computer

programming code aloud. In this interview, we will ask participants to provide their feedback about possible ways to interact various sound effects into the programming environment. The goal is to generate useful user requirements that could help us implement a system that uses auditory cues to help non-visual users understand the structure of the codebase. So, several of these sound effect audio files contain recordings of a computer voice reading some computer code, along with some sound effects (e.g. beeps at different pitch) playing in concert with the speech audio recording. In order for the IRB to understand the nature of the sound effects that may be played in this study, some examples have been produced and posted at the following locations: The list of sound effects used herein is a mixture of speech sounds, non-speech sounds, and spatialization of sounds (stereo left-right differences). There is no fixed list of sound effects that will be played during this study – since the investigator may invent new variations (e.g. beeps at different pitches) based on the feedback of the first few participants. Thus, the specific repertoire of sound effects that may be played to participants may evolve during the study, following a traditional "participatory design" approach, common in the field of human-computer interaction, wherein the stakeholders are actively involved in the design process to help ensure the result meets their needs and is usable. Thus, if a participant in an interview early in the study suggests a particular sound effect that might work well, then the investigator may include a sound effect like this in the subsequent interviews, to determine whether participants

like this sound effect.

D.1.1 Script for the Interviewer

Please note that in order for the interview session to feel natural and comfortable for the participant, the interviewer will not actually read verbatim this specific script to the participant, which would sound awkward and stilted. Instead, the interviewer will converse in a comfortable and fluent manner with the participant – but will follow this arrangement of topics in a semi-structured manner. Thus, the IRB may consider this script as a typical example of the nature of questions and conversation topics that will arise during this interview.

Conformation/Approval:

Thank you for coming to the interview today, I am going to read aloud the informed consent prior to conducting the interview study. We cannot start the interview without getting your confirmation/approval on the informed consent.

Interview Questions:

Thank you for your confirmation/approval. In this interview, I want you to please to start imagining yourself working with some computer programming code using your favorite screen-reader technology and text-editor software. Your screenreader will read some code to you, and sometimes it is going to make additional sound effects to inform you how the codebase is indented. For instance, if text is "nested" inside other regions of text, such as in the case of

a for-loop that is nested within another for-loop. These sound effects could be speech sounds (human voice), non-speech sounds (e.g., bell), or spatialization of sounds (stereo left-right differences). For example, in a speech sound, the sound effect will be based on the human voice that informs you how your code-base is nested. For example, the human voice might say something like level 1, level 2, level 3, one, two, three, etc. It might also say how your codebase is intended in different ways. Here are some examples for you:

SPEECH SOUNDS EXAMPLES

Now, after listening to some of the speech audio samples. I want to ask which one of these audio samples sound better? Do you like the one says Level 1, Level 2, Level 3? or 1 Indent, 2 Indents, 3 Indents? Or maybe the shortest version 1, 2, and 3.

Do you like to be indicated about each level? Do you think receiving short audio feedback that tells you when you are inside or outside the nested code-base?

Which one in your opinion could be used to indicate the nesting level?

Do you have any suggestions or ideas to improve these speech sounds list?

Do you have any other word phrases that could be used to include in our speech sound list?

Based on the participant responses, additional questions could be added to solicit more explanation or feedback.

Before we move out of the speech sounds category. Let me tell you about one

of the common techniques that researchers use to make speech sounds shorter and quicker. This technique is known as Spearcon, where the entire phrase is compressed and sped up.

Let us think about the speech examples presented to you earlier. Now, imagine yourself listening to these speech audio examples in a way that they are completely compressed and sped up. Here are some examples for you to listen:

SPEARCON SOUNDS EXAMPLES

What you think about this approach? Do you find it useful? Do you like it? Do you think we should use it to make speech sounds shorter and quicker? Do you think screen-reader users can understand them or they are very difficult to recall?

Do you think this technique requires some level of prior training?

Do you have any suggestions or ideas about this technique or other techniques?

Based on the participant responses, additional questions could be asked to solicit more explanation or feedback.

Thanks for your comments about the speech sounds category.

Now, we will move to the non-speech sounds category where we will have an exciting conversation.

First, I want you to please imaging previous scenario, but this time with non-speech sounds.

Non-speech sounds could be based on various musical notes or tones that could translate how your codebase is indented.

These non-speech sounds might include sounds like a bell, waterdrops, or a sound of woodpecker pecking a tree, etc. Here are some examples for you to listen:

NON-SPEECH SOUNDS EXAMPLES

Now, which one the sound affects you heard better? Do you like the bell one, or the waterdrops, or the woodpecker sound? Which one in your opinion could be used to indicate how the codebase is indented?

Also, I want you to know that in the non-speech category, I am using different techniques to play with some of the sound dimension. For example, I am using different level of pitch, volume, speed, duration, etc. to indicate how the codebase is intended. Here are some examples for you to listen:

NON-SPEECH SOUNDS EXAMPLES

Now, what you think about changing sound dimensions. Which one do you think we should use and how? Which one of these dimensions is not important and why?

Do you have any suggestions or ideas for these sound dimensions?

Do you have any other non-speech sound that we could use to include in our non-speech sound list?

Based on the participant responses, additional questions could be asked to solicit more explanation or feedback.

Excellent.

Now, another way to provide audio feedback is to play a specific sound with

different pitches, e.g., the bell sound.

For example, if you are entering level 1, the sound will be played with a higher pitch to indicate that you are in level 1, and if you are entering level 2 or 3, the sound will be played in a different level of pitch.

Think about the space around you, if the sound was played with a higher pitch that indicates your left side, medium pitch indicates your middle side, and low pitch indicates your right side. The level of pitches will be based on how your codebase is indented, e.g., lowest pitch means level 1, lower pitch means level 2, low pitch means level 3, high pitch means level 4, higher pitch means level 5, highest pitch means level 6, and so on.

Please let me play you some examples so that you can get a sense of this technique:

SPATIALIZATION SOUNDS EXAMPLES

Now, what you think about this technique? Do you like it? Do you think this technique is easy to convey by screen-reader users? Do you think we should use it in our auditory cues system?

Do you have any suggestions or ideas about this technique?

Based on the participant responses, additional questions could be asked to solicit more explanation or feedback.

Alright, now I need to ask few questions about the possibility of mixing speech sounds with non-speech sounds. Do you like this idea? Do you think it will be useful? Allow me to play some examples so that you can get sense of that.

MIXED SOUNDS EXAMPLES

Do you like it? What is your opinion about using such a technique?

Based on the participant responses, additional questions could be asked to solicit more explanation or feedback.

Now, I need to ask you few questions before we end our interview study.

I want to know if you prefer to hear these sounds effect automatically while you move throughout the codebase? Or maybe on your wish by using specific short key to request the feedback? Or maybe as a background sound while you are working with your codebase? The only difference between automated feedback and background noise is that automated only occurs when you move from one level to another while the background noise played to you all the time with low volume.

Let me play some examples so that you can get sense of these techniques.

AUTOMATED SOUNDS EXAMPLES

Now, after you heard the automated example. What you think about this technique? Do you find it better and useful or you wish to have more control over the audio feedback?

Based on the participant responses, additional questions could be asked to solicit more explanation or feedback

END OF THE INTERVIEW QUESTIONS!

Thank you very much for taking your time to participate with us in our study.

*We sincerely appreciate your taking time to provide your comments and
feedback.*

Appendix E

Larger Study Questionnaire

This appendix presents our audio-based interaction techniques questionnaire aimed to investigate whether non-visual programmers prefer a specific type of interaction for conveying the structure of the programming code.

Title: Interviews about Sonification of Structured Programming Code for Non-visual Users

Investigator: Khaled L. Abusays, Ph.D. student in Computing and Information Sciences, Rochester Institute of Technology

Faculty Supervisor: Matt Huenerfauth, Professor, School of Information (iSchool), Rochester Institute of Technology

Thank you for taking some time to participate in this research study. The below information is helpful to decide whether you would like to proceed further with this study or not.

Nature and Purpose of the Project: The goal of this research project is to learn how to improve the experience of computer programmers who are blind. We are studying whether various auditory cues (e.g. sound effects) may help programmers who are using a screen reader when writing computer code. Specifically, we are investigating three different audio interaction techniques: 1) automatic level-crossing notifications, 2) on-demand level indications, and 3) no feedback. The goal is to find the best way to provide audio feedback in order to help non-visual users understand the structure of the programming codebase.

Explanation of Procedures: This study will not take more than 70 minutes to complete. Today, you will be asked some questions about possible ways to provide audio feedback (sound effects) in order to help someone, understand how code is nested. I will also ask you some other questions about evaluating various audio samples: some consist of computer-generated speech, and other are non-speech sound effects. We are interested in learning the best technique (automatic, on-demand, or no feedback) to provide sound effects to help someone who is blind understand the structure of the code. Please be honest with your feedback. All interviews will be audio recorded for further analysis.

Potential Discomfort and Risks: You will speak with the interviewer and listen to some sound effect recordings; the potential risks are minimal. Also, you may request a break at any time.

Potential Benefits: You will not receive any direct benefits for participating

in this study. The study will be used to help direct future design decisions of the interface experience.

Costs/Reimbursements: After the 70-minute study, you will be compensated for your time with either: \$40 cash (if you are meeting the investigator in person for this study) or a \$40 Amazon gift card (if you are participating remotely) which will be transmitted to you by email within one week of your interview appointment. In this case, please ensure the interviewer has your correct email address.

Confidentiality: Every effort will be made by the investigator to keep your research records and other personal information confidential, except as may be required by court order or law. Access to the research records may be provided to the authorized representatives of Rochester Institute of Technology, including members of the Institutional Review Board (IRB), a committee which reviews and approves all research involving human subjects.

Withdrawal from the Project: Participation in this research project is voluntary. You may decide not to participate or to leave the study at any time. Choosing to leave or deciding to not participate the study will not result in any penalty or loss of benefits to which you are entitled, nor harm your relationship with the university.

Whom to Contact with Questions: This research project has been reviewed by the Rochester Institute of Technology Institutional Review Board. If you have any questions related to your rights as a research participant, you

may contact: Heather Foti, Associate Director, Office of Human Subjects Research, Phone Number: 585-475-7673, Email: hmfsrs@rit.edu

If you have any questions about the conduct of this research project or think that something is unusual or unexpected happening or psychological discomforts, any injuries, you may contact: Dr. Matt Huenerfauth, Professor, Department of Information Sciences and Technologies, Rochester Institute of Technology, Departmental Phone Number: 585-475-7924, Email: matt.huenerfauth@rit.edu

E.1 Example Script and Questions for the Audio-based Interaction Techniques Experiment Study

Interview Method:

The interview questions presented herein is a semi-structured interview. The interview is going to be face-to-face unless the participant could not make it to the investigator location. In such a case, the study will be conducted online via telephone or Skype or Google Hangouts per the participant preference.

Prototypes:

In this study, we will be evaluating three different audio interaction techniques (automatic level-crossing notifications, on-demand level indications, and no feedback interaction). The goal is to find the best way to provide useful information to screen-reader users (computer voice) that is reading computer

programming code aloud. In this interview, we will be asking participants to provide their feedback about the above-mentioned interaction techniques. The goal is to generate useful user requirements that could help us implement a system that uses auditory cues to help non-visual users understand the structure of the codebase. In this experiment, there are audio files that contain recordings of a computer voice reading some computer code, along with some sound effects based on speech cues (e.g., level 1, level 2, etc.) and none-speech cues (e.g., repetitive beeps). The list of sound effects used herein is a mixture of speech sounds and non-speech sounds. The list of sound effects was generated during an early stage from a previous study. Participants were recruited to provide their feedback on various sound effects. Only the top requested sound effects were carried in this project. In this current study, we are only investigating different ways to provide audio feedback (sound effects) to non-visual programmers in order to provide useful feedback about the structure of the programming codebase.

E.1.1 Script for the Interviewer

Conformation/Approval: Thank you for coming to the interview today, I provided the informed consent prior to the interview via email. We cannot start the interview without getting your confirmation/approval on the informed consent. Do I have your approval to start the interview process?

Interview Questions:

1. Code Understanding:

(a) On a scale from 1 to 7, rate how easy you found the task to complete, with one being very difficult, and seven being very easy 2.

(b) On a scale from 1 to 7, rate how frustrating you found the task to complete, with one being very frustrating, and seven being not at all frustrating

Very frustrating □—□—□—□—□—□—□ Not at all frustrating

Not at all likely ☐—☐—☐—☐—☐—☐—☐—☐—☐ Extremely
likely

Post-experiment Questionnaire:

5. Rating questionnaire:

- (a) On a scale of 1 to 10, how would you rate the automatic feedback system?

1 ☐—☐—☐—☐—☐—☐—☐—☐—☐—☐ 10

- (b) On a scale of 1 to 10, how would you rate the on-demand feedback system

1 ☐—☐—☐—☐—☐—☐—☐—☐—☐—☐ 10

- (c) On a scale of 1 to 10, how would you rate the no feedback system?

1 ☐—☐—☐—☐—☐—☐—☐—☐—☐—☐ 10

6. Open-ended Questionnaire

- (a) Could you explain to me, in a few sentences, what do you like and dislike (positive and negative aspects) about the automatic feedback system?

- (b) Could you explain to me, in a few sentences, what do you like and dislike (positive and negative aspects) about the on-demand feedback system?

- (c) Could you explain to me, in a few sentences, what do you like and dislike (positive and negative aspects) about the no feedback system?

- (d) What do you think would be the impact of a tool like this (automatic/on-demand feedback) being available to the public?
- i. Who do you think would benefit the most?

Semi-structured Interviews Questions:

7. Automatic-based feedback prototype questions:

- (a) Do you like to be indicated one time about each level?

-
- (b) Do you like the idea of providing sound effects using this type of interaction?

- (c) Would you use this type of interaction? For what?

- (d) Reflect on how the experience of navigating through the code was different with the tool than without the tool:

- i. How did the tool affect your ability to complete the tasks?

- ii. How did the tool affect your ability to know where you were in the code?

iii. How did the tool affect your ability to understand the code?

iv. How did it change how you do your initial skimming or orient yourself;

(e) Do you have any suggestions or ideas to improve this type of interaction?

(f) Do you like the timing of this type of interaction?

- (g) In this type of interaction, there is 10 millisecond delay between computer voice recording (reading code line) and each sound effect.

- i. Do you think 10 millisecond delay is long or short for you to notice the difference?

- ii. Did you find it easy to distinguish each sound effect as well as the computer voice recording?

- (h) Do you have any other comment about this type of interaction?

8. On-demand-based feedback prototype questions:

- (a) Do you like to be indicated about each level whenever you want?

- (b) Do you like the idea of providing sound effects using this type of interaction?

- i. Sound effects will be played per your request.

- (c) Would you use this type of interaction? For what?

- (d) Reflect on how the experience of navigating through the code was different with the tool than without the tool:

i. How did the tool affect your ability to complete the tasks?

ii. How did the tool affect your ability to know where you were in the code?

iii. How did the tool affect your ability to understand the code?

iv. How did it change how you do your initial skimming or orient yourself;

(e) Do you have any suggestions or ideas to improve this type of interaction?

(f) Do you like the timing of this type of interaction?

(g) Did you find it easy to distinguish each sound effect as well as the computer voice recording?

(h) Do you have any other comment about this type of interaction?

9. No-feedback prototype questions:

(a) Do you like the idea of using just your screen-reader where no audio feedback is played to you?

(b) Do you have any suggestions or ideas to improve it?

(c) Do you have any other comment about this type of interaction?

END OF THE INTERVIEW QUESTIONS!

Thank you very much for taking your time to participate with us in our study.

*We sincerely appreciate your taking time to provide your comments and
feedback.*

Appendix F

Supplementary Study

Materials

This appendix presents the experimental study materials which were used to conduct the larger study. This work was conducted to evaluate the usability and efficacy of audio-based techniques and whether users have preferences for various forms of audio-based cues to help convey the hierarchical nesting structure of code.

F.1 Materials Description

The supplementary study materials can be downloaded [here](#). There are three main folders.

1. Audacity Project

- Folder 1: Computer-recording; it has the original recordings for the larger study experiment.
- Folder 2: Cues; it has two small files, speech, and non-speech, each one contains the original cues.
- Folder 3: Interaction; it has three small files; automatic, no feedback, and on-demand. Each one has its recordings with the original files

2. Code samples

- Contains the three code samples (written in python language) that were used in the larger study.

3. Sites

- Each one of the below folders contains several small folders and files; cues, recordings, HTML files, timing CSV files, experiment plan order list as a text-file, and d3 Javascript file.
 - Folder 1: Plan A
 - Folder 2: Plan B
 - Folder 3: Plan C

F.2 To run the Prototypes

Follow the next steps to run the study prototypes:

1. Upload the sites main folder to a server or a localhost.
2. Navigate to the index.html path, based on the site folder location either on your server or localhost.
3. The index.html page will display three options, each one will guide you to the three prototypes, with each in a different order.